TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl für Betriebssysteme und Systemarchitektur

The Contextual Map

Detecting and exploiting Affinity between contextual Information in context-aware mobile Environments

Robert Schmohl

Vollständiger Abdruck der von der Fakultät für Informatik
der Technischen Universität München
zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. B. Brügge, Ph.D.
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. U. Baumgarten
	2.	Univ.-Prof. Dr. J. Schlichter

Die Dissertation wurde am 01.07.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.10.2010 angenommen.

Abstract

Context-aware computing generally focuses on abstracting the situation of individual entities, such as persons, places and objects, making this information available for further computational exploitation. Those resulting entities' contexts allow a wide spectrum of application cases in various domains, foremost in mobile computing and internet applications. With contexts from multiple entities available, the degree of alikeness of those contexts poses an interesting piece of information. For this purpose, we propose a context model capable of easily identifying affinities among contexts. This multi-dimensional context model is inspired by geographical map models, which are generally applicable for geographical proximity management. We have discovered that geographical proximity can be leveraged to contextual proximity depicting the alikeness of different contexts. Hence, our goal is to apply proximity detection methods from the location-aware computing domain on context-aware computing. The context model has been named the *contextual map*, representing an entity's context by a set of multiple contextual attributes in a multi-dimensional vector. The representation of entities' contexts as multi-dimensional points in Euclidean space allows the application of location-based proximity detection in order to identify affinities between contexts that encompass far more contextual information than just location. This work presents the concept of the contextual map and discusses its prototypic application in identifying large clusters of similar contexts that aim to facilitate the adaptation mechanisms of context-aware systems. We especially emphasize the utilization of proximity and separation detection on general non-location contexts, hence enabling to dynamically monitor their affinity to each other.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Context-aware computing focuses on utilizing any information describing the current *context* of an entity. An entity is an abstract term that may represent a place, a person, an object, etc. Hence, context is the information that is comprised of the *situation* of such an entity, derived from its current surrounding. So basically, an entity's context is an abstract description of its physical and logical (non-physical) surrounding, captured by context sensors. The physical surrounding depicts the physically measurable environment of an entity, such as its location, environmental conditions and influences, physical movement, velocity and so on. An entity's logical surrounding is described by all non-physical characteristics encircling the entity, e.g. relationships of all kinds (e.g. to other entities), the entity's local attributes, etc. Together, the physical and logical characteristics of an entity and its surrounding define its situation, and thus, its own unique *context*.

While the physical aspects of context are acquired directly by hardware sensors, such as motes or mobile devices, other context sensors provide additional information from databases, the infrastructure hosting mobile devices and sensors, etc. to supplement the entity's contextual information on the logical (non-physical) plane [36, 42]. The decisive purpose of context-aware systems is to utilize the acquired information to make it available to a ubiquitous environment of entities and - finally - users roaming in that environment. This procedure allows the pervasive utilization of contexts belonging to diverse entities, i.e. places, persons, objects. The utilization of contextual information enables context-aware systems to automatically adapt applications to their users' preferences. At this, the utilization of an entity's single context is equally important as the utilization of meshed context consisting of multiple contexts belonging to different entities.

This work focuses on the contexts of individual entities. To store and exploit such contextual information, context-aware systems utilize sophisticated context models. Those models represent the context captured from the real-world in a way suitable for further processing, such as identifying contextual coherences and inferring new context. An important aspect in context-aware computing is the identification of *similar* contexts, i.e. entities sharing similar situations. It allows to identify and evaluate relationships among entities, which is applicable for a wide spectrum of higher-level application cases.

1.1 Problem Statement

As stated, context-aware systems aim at making individual context information ubiquitously available. With each entity possessing its situational context, context-aware systems are usually expected to adapt to each of the entities' context individually. E.g., a user visiting a smart exhibition wishes to receive information about exhibits on his PDA. However, only a small subset of exhibits is suitable to the visitor's situation: only those that are located in his current hall and fitting his personal interests. Hence, the context-aware exhibition system needs to adapt to each visitor individually. With an increasing number of entities and an increasing level of adaptation complexity, the context-aware system's context adaptation mechanism gets increasingly elaborate. Let us keep in mind that a context-aware adaptation process may include large numbers of context-attributes from many highly heterogeneous context sources, clearly extending our fairly simple exhibition example. Following this observation, context-dependent adaptation mechanisms do not scale well under the discussed circumstances.

Instead of regarding individual entities, efforts can be significantly reduced by addressing groups of entities, which share similar context, i.e. which are in similar situations. Hence, a context-aware system needs to uniquely adapt to such sets of entities with similar contexts instead of adapting to each individual one. Recalling the exhibition example, identifying groups of visitors in adjacent exhibition halls with similar interests yields the definition of entity sets in which all entities (visitors) have contexts (situations) similar to each other. The context-aware exhibition system may restrict to adapting to those visitor groups providing each group with the same information. In summary, we aim at identifying *contextual similarity* among entities, which possess similar situations, i.e. contexts, in order to determine groups of entities sharing similar context.

The deduction of a set of similar contexts may be labeled as a *contextual similarity query*. Given a reference context, such a query outputs all entities whose context is similar to the reference context. Since the reference context serves as the center point for identifying contextually similar entities, it is to be regarded as the query point for the contextual similarity query. The applicability is straight-forward: with a query entity given, many application cases require knowledge about entities with similar situations, i.e. with similar contexts.

Following this argumentation, the examination of contextual similarity implies the deduction of *super-situations* that include the contexts of entities, which are similar to each other. Such super-situations correspond to the union of situations belonging to sets of entities with similar contexts. Thus, a super-situation defines the contextual realm denoting a unified contextual representation of those entities, so that this contextual realm abstractly depicts the union of their situations. Hence, the context adaptation process of a context-sensitive system is simplified, since it needs to adapt to the super-situation only while addressing all included contexts as well.

The scope of this work is the definition of mechanisms to enable the application of contextual similarity and the deduction of sets encompassing similar contexts. In particular, we cover the following problems:

- How to formally describe contextual similarity and how to define an according similarity metric?
- How to determine similar contexts based on the contextual similarity metric?
- How to reasonably exploit contextual similarity?

With those questions tackled, we derive a foundation for handling entities with similar contexts, i.e. entities in similar situations. This applies for both similarity among individual entity contexts as well as for groups of contextually similar entities.

1.2 Approach in brief

Our approach to illuminate the aspect of contextual similarity is inspired by the principles of managing geographical proximity in the domain of location-aware systems. In order to introduce our approach to the reader, we therefore regard an entity's unique piece of contextual information: its *location*. Although generally utilized by context-aware systems, location-awareness has its own dedicated application domain. Location-aware systems employ map-based models, which are primarily designed to manage locations and associated information. Hence, the trivial approach to identify geographical proximity between entities is to locate them and check the distance between them against a proximity threshold, which defines entities to be proximate at all.

Although exhibiting a high degree of specialization, location models yield principles, which can be applied to conventional context-aware-computing as well. We have surveyed the current research in the domain of location-based services and combined the presented approaches with mechanisms of general context-aware computing. We have discovered that the principle of *proximity* in the location domain can be applied to face certain aspects of context-awareness, too. The geographical proximity in the location domain expresses that entities are physically close to each other. By projecting this setting on more general contexts, *contextual proximity* may express that contexts are alike or affine, hence "close" to each other. Thus, we use the term contextual proximity synonymously with expressing contextual similarity from this point forward.

For example, one can imagine a weather station's context expressing its measured environmental conditions. Hence, an Acapulco-based weather station's context may be quite "close" to the context of a station in Rio de Janeiro since both are located in tropical climate zones with similar weather conditions. On the contrary, the Acapulco station's context may be "distant" to a weather station in Eastern Russia's Wladiwostok, which records frosty climate all-year long. Hence, the contextual proximity between the stations in Acapulco and Rio is high in terms of being contextually "close" to each other, whereas Acapulco and Wladiwostok are contextually distant depicting low contextual proximity. Concluding, we use contextual proximity to depict the similarity between the contextual information of weather stations. Geographical proximity does not have any influence on this approach (the stations in Acapulco and Rio are contextually proximate despite being thousands of kilometers apart). Nevertheless, we borrow the principles of geographical proximity to determine contextually similar entities.

Recalling the introductory discussion from the previous section 1.1, an entity's context is comprised of its current situation. Thus, the first step in determining contextual proximity is to decompose the situations of individual entities into contextual attributes so that each attribute represents a particular part of the entity situation. Subsequently, those contextual attributes are quantified accordingly and mapped into a multi-dimensional Cartesian map model - the *Contextual Map*. As a Cartesian map, the contextual map is spanned by multiple dimensions. The key idea of this model is to assign each context attribute to exactly one of the contextual map's dimensions. In detail we proceed as follows:

1. *Typification:* An entities context is typified, meaning that it is decomposed into contextual attributes.

2. *Quantification:* The attributes are assessed with values depicting the current contextual situation of the entity.

3. *Mapping:* The context attributes are mapped into the contextual map by assigning each attribute to one of the map's dimensions.

Concluding, an entity's situational contexts is represented as a multi-dimensional point (or location vector, respectively) in the contextual map model, because all dimensions represent all contextual attributes equally, and thus the complete entity context or situation, respectively.

In this setting, this model allows the identification of contextual proximity according to the Euclidean distance between those points representing individual contexts. The distance between contexts in the contextual map determines their degree of alikeness. At this point, the analogy between proximity management in location-aware computing and contextual proximity in the context map becomes illuminated. Such as geographical proximity detection requires the location of entities given by coordinates in real-world 3-space, contextual proximity management requires the "location" of the entities' contexts in the contextual map given by the quantifications of their contextual attributes. In terms of location-awareness, the Euclidean distance between entities expresses the degree of geographical proximity. In the contextual map, Euclidean distance denotes the degree of contextual alikeness between entities. Since this context model represents contextual information in a multi-dimensional geographic model, it allows the application of mechanisms known from location-aware computing on non-location contextual information. Figure 1.1 sketches the idea behind the contextual map and its contextual proximity determination.

It remains to be clarified how proximity thresholds are defined in the contextual map. Concerning geographical proximity, such a threshold is usually scalar defining a distance when entities are to be regarded as proximate. However, since the similarity among contexts may encompass many contextual attributes from complex environments, we need a

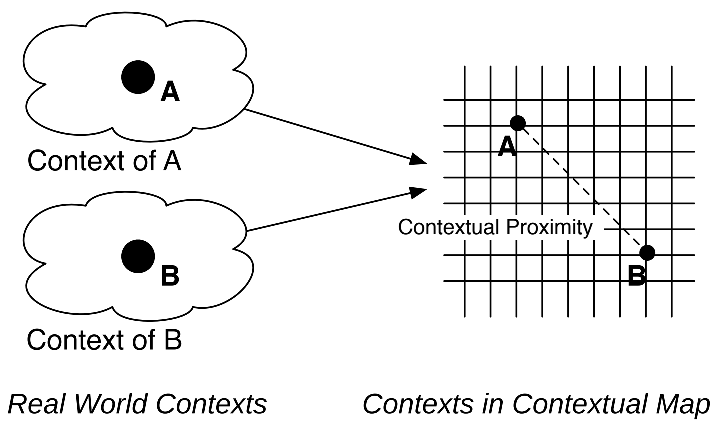


Figure 1.1: Contexts in the Contextual Map

more sophisticated data structure defining contextual proximity. In our approach, we define the notion of *context boundaries* as the degree of alikeness between different contexts. Those data structures texture all context attributes relevant for contextual proximity detection. Since contextual information is highly heterogeneous and situation-dependent, contextual boundaries are highly application-specific, as the upcoming discussions in subsequent sections will show. We have identified general principles of proximity and separation detection [61] to employ such context boundaries as a metric for determining affinity between contexts. More precisely, we monitor dynamic affinities between pieces of context according to their changing Euclidean distances to each other in the contextual map. Those distances allow us to determine, whether context boundaries have been crossed, hence whether contexts have become more affine (converging) or less affine (separating).

Based on the contextual map model, sets of contexts that encompass similar (i.e. proximate) contexts can be deduced. The definition of such context sets ranges from the identification of contexts similar to a specific query context to the identification of large context clusters, which are spawned by numerous contexts, which are all similar to each other. Finally, such a context set can be deduced to a situation definition, which encompasses the contexts of the underlying context set. As stated in the previous section 1.1, that context set allows context-aware systems to adapt to that situation only, while serving all corresponding entities of the underlying context set.

1.3 Problem Decomposition

In the previous sections 1.1 and 1.2 we have discussed the motivation of determining contextual proximity as well as sketching our basic approach to address the problem. In this section we further decompose the problem into multiple blocks in order to illuminate the problem definition. We conduct this process by defining a top-level workflow describing how contexts of entities are applicable for use cases employing contextual similarity. Figure 1.2 illustrates the workflow, which is described in the following.

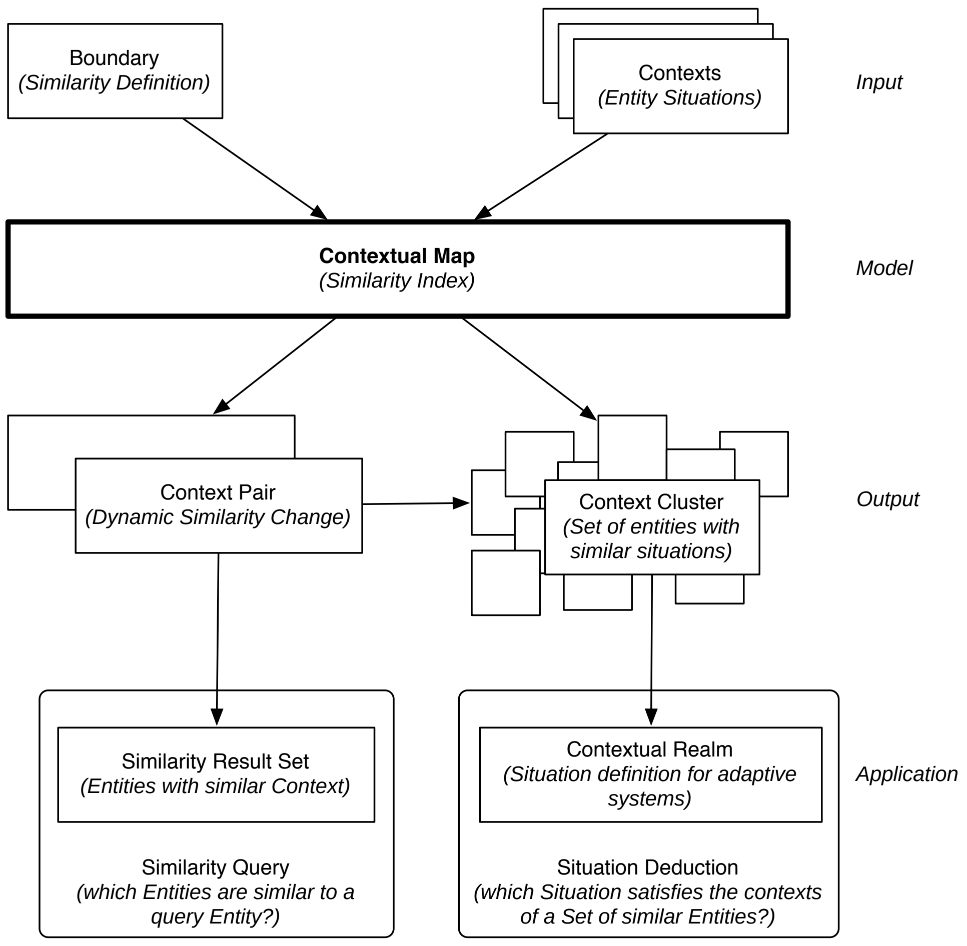

Boundary
(Similarity Definition)
Contexts
(Entity Situations)
Input
Contextual Map
(Similarity Index)
Model
Context Pair
(Dynamic Similarity Change)
Context Cluster
(Set of entities with similar situations)
Output
Similarity Result Set
(Entities with similar Context)
Similarity Query
(which Entities are similar to a query Entity?)
Contextual Realm
(Situation definition for adaptive systems)
Situation Deduction
(which Situation satisfies the contexts of a Set of similar Entities?)
Application

Figure 1.2: Applicability of the Contextual Map

As the figure shows, the data flow is divided into four layers describing which *input* is provided to the contextual map *model*, what *output* is generated by applying similarity detection operations on the model, and how this output is utilized *application*-specifically. In more detail, the workflow is to be interpreted as follows:

1. *Input*: In order to perform contextual similarity detection at all we need two parameters:

   - Entities that are to be regarded. For this purpose, the contexts of potentially many entities are captured and mapped to the contextual map.
   - A threshold defining proximity. As described above in section 1.2, we employ our notion of contextual boundaries to define the degree of contextual similarity.

   The input part - as denoted in figure 1.2 - includes the definition of mapping techniques describing how the contexts of entities have to be abstracted into the contextual map's Euclidean space. This context mapping requires to be conducted without semantic loss of the situational context of the represented entity. Additionally, this part requires a thorough definition of the contextual boundary data structure. It defines the desired degree of proximity and is mostly application-specific (meaning that the use case for which contextual proximity handling is actually utilized specifies the similarity degree, see subsequent tag point 4)

2. *Model:* After completing context mapping, the contextual map stores each entity's context as a multi-dimensional point or location vector, respectively[1], denoting all of the respective entity's contextual attributes. The basic operation conducted on the context map is the identification of contexts that are similar to each other. Thus, the model part particularly includes the architectural design of the context model and the design of efficient data structures, which enable fast querying of contexts. Monitoring of dynamic similarity changes among contexts also falls into the model block. Those changes occur due to constantly changing situations of the entities.

3. *Output:* The similarity detection operation performed on the contextual map yields similar contexts. This output can be divided into two possible result sets:

    - the basic result is a *pair of contexts* that are similar to each other, i.e. which are close to each other in regard to the Euclidean distance between each other checked against a defined context boundary. This result usually originates due to a dynamic similarity change, i.e. contexts have recently become similar in a sufficiently high degree so that becoming relevant to a contextual boundary. The event of *becoming similar* is of pivotal importance here, since recurring results of the same context pairs with the same similarity degree hold much fewer information.
    - A more sophisticated result set extracted from the contextual map is the *context cluster*. It includes similar contexts where every included context is similar to at least one other context. As a prerequisite, a context cluster requires to determine pairs of similar contexts, since decomposing a cluster results in context pairs as well. In summary, context pairs form the building blocks of a context cluster, as illustrated by an according arrow in figure 1.2.

4. *Application:* Inferring from the output of the similarity operations conducted on the contextual map, we have identified two application cases:

    - A *Similarity Query* yielding a *Similarity Result Set* including all contexts or entities similar to a given a query context or query entity, respectively. The similarity query relies on the identification of similar context pairs, so that each context returned by the query is similar to the query context (thus, each context and the query context denoting a context pair). The similarity query answers the basic question:

[1]location vectors are a representation of points in Euclidean space

"Which entities are similar to a particular query entity?"

- The second possible application is the *Contextual Realm* interpreted from context clusters. Such a context cluster describes a "super situation", which a context-aware system may solely adapt to while addressing the contexts of all entities represented in the contextual realm. Thus, the realm is spanned by the contexts belonging to the cluster. Since contexts denote entity situations, and since the clustered contexts are similar to each other, a coherent realm is depicted by the context cluster. The question to answer in this application case is:

  "Which (super-)situation satisfies the contexts of a set of similar entities?"

Again, figure 1.2 illustrates the decomposition of contextual proximity handling into multiple problems aligned into a causally dependent workflow. It is to be noted that the denotation of the elements is complemented by an annotation (italic, in parenthesis) depicting what they actually represent. E.g. on the input layer, contexts actually represent the situations of entities and context boundaries define the similarity.

With the workflow above described, the problem statement of this work is outlined. In the subsequent chapters we give an overview about affected research domains before approaching the problems identified in this section.

1.4 Structural Overview

This work explores the application possibilities of the contextual map model. Chapter 2 enumerates the work related to our project and delivers a detailed overview of the research domains related to the contextual map. It especially concerns the areas of context- and location-awareness. With this discussion given, we introduce our contextual map model in chapter 3. Chapter 4 explains how the contextual map is employed to monitor affinity between contexts and how this is related to contextual boundaries. With the basic mechanisms defined, the subsequent chapter 5 focuses on utilizing contextual proximity. It is dedicated to the contextual similarity query and it describes the conceptual mechanisms of detecting clusters of similar contexts in the contextual map. Subsequently, chapter 6 discusses system-specific aspects that concern the implementation of the contextual map model. This particularly concerns the application of the contextual map in distributed environment. The design and implementation of this concept is documented in chapters 7 and 8, respectively. Closing, chapter 9 concludes the contextual map documentation proposing some additional interesting application cases for the contextual map and giving an outlook on the enhancements of the approaches presented here.

Chapter 2

Background and Research Domains

In this chapter we introduce the research domains and related work that converge with our research. First of all, in section 2.1, we present an overview on related research by sketching the most important work conducted by other research groups. Subsequently, sections 2.2, 2.3 and 2.4 summarize the domain of context-aware computing, which is the decisive cornerstone of our work. Sections 2.5 and 2.6 present a brief overview of location-aware systems and location-based services, especially discussing principles of proximity detection in 2-space, which forms our starting point for inferring contextual proximity. Section 2.7 illuminates another issue, which influences both the application of context- and location-aware systems: the heterogeneity of sources and technology. The concluding section 2.8 explains, how the contextual map fits into the research domains described in this chapter.

2.1 Overview on related Work

This section aims at sampling related work out of the contextual map's research scope to provide an according overview. It basically represents a list of publications that have influenced the conceptualization of the contextual map. An in-depth discussion about the related work presented in this section is provided in the rest of the chapter in sections 2.2 through 2.7.

2.1.1 Context-Awareness

Prior to the conceptualization of the contextual map, we have surveyed the domain of context-aware computing aiming at identifying common characteristics in order to generate a generalized view on the domain. We have analyzed research results depicted by various groups active in the domain as enumerated subsequently. The works in [22] and [95] provide surveys about context-aware computing from different viewing angles. Additionally, Strang et al. provide a thorough overview about context models [95] whereas Christopoulou et al. present a generally applicable context reasoning process [36]. In addition, we have analyzed publications about several context-aware system implementations [26, 29, 40, 67, 100] extracting additional important aspects to augment our survey. The detailed results of this survey, which includes the derivation of a generalized perspective on the context-aware computing domain, is discussed in the subsequent sections 2.3 and 2.4.

Regarding context models, the contextual map model is quite specialized in nature due to its narrow focus on contextual affinities. The unquestionably most widespread context models are based on ontologies [95]. For this reason, the majority of researchers put their focus on this type of context models [36, 37, 51, 56]. The contextual map can be regarded supplemental to such well proven context models in the context-aware computing domain, in order to augment those with contextual similarity handling.

2.1.2 Location-Awareness and Proximity Detection

With the conceptual map, we present a novel concept of a context model exploiting principles of the location-aware computing domain for determining *contextual proximity*. An important cornerstone of our work is depicted by the principles of geographical proximity and separation detection Amir et al. [20] and Küpper and Treu [61] have elaborately examined this issue. As a particular aspect of location-awareness, geographical proximity detection may be reduced to checking the distance between two mobile entities against a defined proximity threshold to declare them as proximate or not. It subdivides into two important problems:

- First, given a query entity, which of potentially many other entities are eligible for calculating the distance to the query entity? Distance calculation is elaborate. So obviously, calculating the distance from the query entity to *every* other entity in the system is not feasible. Bononi et al. give a short insight into the problem [31]. Nearest-neighbor queries [86, 94] and nearest-surrounder queries [64, 63] examine a queried entity's neighborhood and thus, narrow the set of entities, which may be proximate to that query entity. Hence, such result set of entities is eligible for distance calculation and thus, proximity detection. Further, locations can be indexed in efficient indices yielding location indices. Lots of research has been done on efficient multi-dimensional indices [30, 28, 62, 101, 25, 58, 66], which allow to approach the proximity detection issue with the help of range queries. A range query performed on such an index fetches elements from a particular part of the index. A range query conducted on a location index equals fetching all entities in a particular given area (or space).
- The second important aspect concerning proximity detection is keeping entity locations current. Constant tracking of *all* entities' locations is inefficient and due to generally costly location updates not feasible. Amir et al. [20] present principles to minimize the amount of location updates while keeping the locations as current as possible. Küpper and Treu [61] have augmented those principles and focus on optimizing the accuracy of the detection mechanism while minimizing the amount of location updates committed in a distributed environment. Their work represents

an important cornerstone for the contextual proximity mechanisms developed in our work.

The mechanisms for enabling proximity and separation detection allow the definition and utilization of context boundaries when employed with the contextual map model. Context boundaries define the *distance* of contextual information relevance crossing a specified threshold. A similar scenario is given by Roman et al. [85] where contextual affinity in ad-hoc network environments in examined.

2.1.3 Related Approaches

An approach, which is similar to our work, is depicted by the theory of context spaces introduced by Padovitz et al. [75]. As the contextual map, context spaces aim at the multi-dimensional representation of contextual attributes. Multi-dimensional context spaces are partitioned into regions denoting bounds of specific contextual situations. Concrete context states mapped from real world entities are then associated to such context regions according to their proximity in the context space. The authors of [75] employ their context space model to maintain stability of contexts in context-aware systems by evaluating and refining uncertain contextual data, which is represented in proximity of each other in the context space [77, 78, 76]. Predefined multi-dimensional regions in the context space are defined by ranges of contextual attribute values, hence defining context situations, which are considered stable. Those regions are used to map uncertain situations to stable and known situations.

2.2 Context: Definition and Properties

This section introduces the reader to the context-aware computing domain by examining the term *context* in more detail. We begin with the discussion about the definition of *context*, so that we can agree on a distinct definition of an ambiguous term. Based on this discussion, we proceed with arguing about how to model and employ contextual information from entities in the real world in the subsequent sections 2.3 and 2.4, which focus on context modeling and on the utilization of context models.

2.2.1 Context Definition

An early definition of context has been presented in 1994 by Schilit et al. [88] who regard context as a limited amount of information covering a person's proximate environment. They have identified the main aspects of context being the subject's location, its relationship to other subjects and its nearby resources. This early context definition implies that the notion of *context* has evolved from the notion of *location* by enriching it with auxiliary information. Later on, in 1999, Schmidt et al. [89] have contributed further work about location evolving to context, decoupling the notion of context semantically from location. The most notable and broadly accepted definition of context was postulated by Abowd, Dey et al. [18]. Context has been coined as follows:

Context is any information used to characterize the situation of an entity, where an entity can be a person, place or physical or computational object

Many research groups have defined context in accordance to their respectively specific focus of research. None of them, however, provides a definition, which differs significantly from the one quoted above. It is to be noted that the definition equalizes context and situation, as we have argued earlier already. Most of the research groups in the context-awareness domain agree that context is a set of the associated situations [36, 22]. Henricksen et al. [49] regard context employed in an application and thus describe context as the circumstances or situation in which an application's computing tasks takes place.

A significant aspect in this discussion is the information that actually comprises context [24, 29, 42, 77, 76]. Context, as defined above, is composed of pieces of individual information. That individual *contextual information* must be acquired from suitable sources [22], such as sensors, databases and users. After acquiring and refining elementary context data, it is put together yielding semantic context. Those sources are highly heterogeneous, since context can be comprised of all kinds of information. In summary, context comprises of contextual information which is retrieved from heterogeneous sources.

Context is the physical and logical surrounding of a physical device. While the physical neighborhood is captured by sensors of the device or the infrastructure, the logical context information surrounding the entity is mostly derived from known context or profiled by the user himself. A device's awareness of context may further be categorized as either *direct*, denoting the context is captured by the device's sensors, or *indirect*, or indirect in case context implied (e.g. by the infrastructure) [42].

Context awareness is the ability of capturing and processing contexts [22], enabling the provision of context-relevant information for entities [18]. Hence, a context-aware computing environment adapts the information or services it provides to entities' by deriving the entities' needs from their surrounding contexts [49]. In other words, a context-aware system responds to its environment [88, 18] by continuously sensing the entities' contexts, i.e. their physical and computational environments [81]. The design of context-aware systems is subject of the subsequent section 2.4 and will be discussed in depth there.

2.2.2 Properties of Context

Contextual information and full entity contexts excel a few characteristics, which are strongly related to ubiquitous computing environments. E.g., those characteristics particularly include the following:

- *Temporal characteristics:* Context may be composed of both static and dynamic contextual information. Static context data is invariant over its lifetime whereas dynamic data is subject to continuous change [49]. E.g., when regarding context of people, birthdays and gender pose static contextual information, because they never change. On the other hand, locations, personal preferences, relationships, etc. pose dynamic contextual data, since all of the latter is dynamic in nature, i.e. it may change arbitrarily.

- *Context is highly dynamic:* As stated, context depicts an entity's situational information as well as its physical and logical surrounding, all originating from the entity's environment. Since pervasive environment are constantly changing, the resultant context information and context itself is highly dynamic [88, 22]. Context-aware systems need to react accordingly.

- *Imperfection of contextual information:* Context is comprised of an abundant amount of information. However, none of it can be regarded as accurate if regarded right after being acquired [49, 78, 29]. Faulty sensor readings, wrong inferences about context and other errors lead to the falsification of contextual information. This has a crucial impact on the quality of contexts. Hence, contextual errors must be identified and handled accordingly (and corrected if possible).

- *Heterogeneity:* Heterogeneity has a decisive impact on context-awareness since pervasive computing environments are packed with heterogeneous hardware and eventually heterogeneous context sources [49, 39]. This especially applies for a wide spectrum of different sensors utilized in pervasive environments. Context-aware systems have to address the heterogeneity issue by proper interpretation and loss-free generalization of heterogeneous data (further discussion follows in section 2.4.4).

- *Interrelation of contexts:* A single entity's context is usually related to multiple other entities' contexts [49]. For one, relationships among contexts can be straightforward, such as friendships among persons or communication links among devices. However, there are also less obvious relations among contexts, for example: if they are identified by derivation rules. An individual entity's context may even be derived from other entities' contexts if rules for deriving context are clearly defined by inter-context relationships.

2.2.3 Context Classification

Closing the context definition section, we briefly sketch our interpretation on how to classify contextual information. We have identified four aspects to base the classification on:

- Entity type
- Type of context acquisition
- Context source type
- Contextual information type

Figure 2.1 shows how those aspects are related to each other in regard to the context capturing procedure.

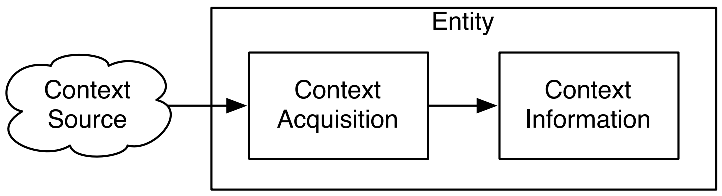


Figure 2.1: Context Classification Aspects

Entity Type

This aspect focuses on the type of entity, which contextual information belongs to. E.g., the context of a restaurant in downtown San Francisco is totally different than the context of a student at Oxford University. Obviously, both are totally different entities (place vs. person) with totally different context characteristics. This motivates us to distinguish contextual data according to their corresponding entity type. Earlier, we have already mentioned the typification, which is defined as follows [18]:

- *Place:* depicting a stationary entity at a certain location, e.g. the Times Square in New York City.
- *Person:* represented by a human being, presumably a user of context-sensitive services.
- *Physical Object:* a spatial existing object, e.g. a mobile phone.
- *Computational Object:* can be any abstract entity assigned with or being involved in a computational task, e.g. an object in memory (object-oriented programming), a data structure, a mobile phone, a stationary terminal, a user involved in a context-aware service, etc. As it can be seen, a computational object may well be a physical object.

Context Source Type

As stated before, contextual information is acquired from context sources. Regarding figure 2.1, the context sources form the initial point in the context capturing process and depict *what* context is acquired. We can categorize the context sources despite their extremely heterogeneous nature:

- *Physical Environment:* As stated, the physical environment is a major context source for many application cases [22, 29, 36, 42, 76, 100]. It is usually captured by sensor motes or sensors attached to mobile devices.
- *Profiled Input:* This group of context sources are proactively given information to enrich an entities context. Most commonly, this information is profiled by a user or a computing system:

  - *User:* The user is one of the most common and obvious context sources by proactively committing input to context-aware systems through a dedicated user interface. In most cases, this input consists of his personal preferences.
  - *Computing system:* a computing system may act as a context source by providing input to context-aware systems through well-defined interfaces.

- *Existing Context:* As stated earlier, context can be used to infer new context. Special inference components apply inferencing rules to conduct this process. Section 2.4 will discuss this approach in more detail.
- *Technical Infrastructure:* As stated before, the technological setting may provide valuable context information [42]. E.g., a cellular network may provide the position of a mobile host by identifying the current cell in which it is currently roaming. A mobile device's battery charge level is another example of this context source type.

Context Acquisition Type

This aspect focuses on the issue of how a device acquires contextual information. As depicted in figure 2.1, after it has been determined what context is to be acquired from the context sources, this aspect determines *how* context is acquired.

A device's context awareness may be differentiated to be *direct* or *indirect* [42]. In the case of indirect awareness, the entire sensing and processing of context occurs in the infrastructure while the mobile device obtains its context by means of communication. In contrast, a device has direct awareness of its context if it is able to obtain context autonomously by itself and independently of any infrastructure (e.g. by its sensors or user interface).

Contextual Information Type

A significant aspect about classifying contextual information is the aspect about its kind. This aspect is of importance after contextual information has been acquired, but not semantically determined yet. We distinguish between *internal* and *external* context information [81], which may both have a temporal dimension (static vs. dynamic).

External Information The external dimension of context depicts the physical environment of an entity [81]. It can be decomposed further as follows:

- *Location:* Besides the location of the entity itself, this aspect includes the proximity to other entities, too. Further, a location can be decomposed into three "shapes" (inspired by [61]):

  - *Point / Position:* depicting a concrete position represented by a set of distinctive coordinates.
  - *Zone:* denoting an area depicting the location.

  - *Distance:* denoting the distance to a reference point or a reference zone. A reference point yields a circular line as the depicted position (circle centered at a reference point). Defining a location, which is based on the distance from a reference zone, yields a line, which is equidistant to the reference zone at all times.

- *Environmental characteristics*, depicting all the physical and measurable characteristics of the entity's current surrounding, e.g. temperature, light level, noise-level, etc.

- *Non-environmental characteristics:* The physical surrounding is composed of more data than merrily environmental data. E.g., the presence of entities in the neighborhood, logical data associated with the entity location, etc. represent additional context information bound to the entity surrounding.

Internal Information The internal dimension of context depicts the contextual information coming from "within the entity" (i.e. not from its physical surrounding). It may also be considered as the information supplementing an entity's context together with the data about its physical surrounding (external context) [81]. We have identified the following types of internal context data:

- *Preferences:* Individual preferences set by an entity.

- *Rights and Privileges:* The permissions to access restricted resources [40].

- *Relationships:* As stated, entity contexts are highly interrelated. Hence, single entities may possess multiple relationships of different kinds to other entities.

- *Social Aspects:* Especially for persons (users) the question about social status and social relationships is an important part of context [49].

- *Other contextual Attributes:* Many other characteristics may be identified for an entity, all relevant for its context. However, due to the heterogeneous nature of context-awareness, further discussion is out of scope here.

All of the internal context information can be obtained in two ways. Either they are set by the entity itself, or they are derived from present context(s). An entity's current context holds an abundance of information. That information can be used to infer new (subsequent) context by defining inference rules, which dictate this inference process [36, 40]. Naturally, rules can apply to an arbitrary amount of contexts allowing the derivation of new contexts for multiple entities.

Temporal Characteristics Additionally to subdividing contextual information by its origin, we can also classify it according to its timely behavior [49]. *Static* context data remains invariant for its entire lifetime. It is usually defined by the user or other static input. *Dynamic* data is subject to continuous change, such as sensor readings and new context inferred from existing context. When regarding contextual information, its temporal characteristics have to be regarded as an additional dimension to their information type.

2.3 Context Modeling

Context modeling is the process of abstracting and representing contextual information for further processing. According to [22], this particularly includes the following aspects:

- Identification of the most appropriate contextual information that can model well enough the specific context in a certain domain.
- Identification and modeling of relations among pieces of contextual information.
- Identification of possible dynamic changes in contextual information and thus, modeling appropriate reactions to such changes.

Hence, modeling context is a technique focusing on how to find and relate contextual information that captures the observation of certain worlds of interest [22]. It decomposes into 2 subsequent phases: First, characteristics from real world are abstracted conceptually, as described in section 2.3.1. Afterward, this concept is mapped on a context model representing the context (section 2.3.2).

2.3.1 Conceptual Approach

Context modeling approaches can be classified into two, not necessarily disjoint, taxonomies [22]:

- *Context Theoretic Modeling*, based on modeling context as situations and changes in situation by actions.
- *Context Conceptual Modeling*, focusing on modeling context by mapping information to certain *concepts*.

Both approaches can be supplemented by the employment of *modals* to extend the semantics of contexts.

Context Theoretic Modeling

Context theoretic modeling is an approach, which aims at representing context primarily by fusing information describing *situations* of entities, that are dynamically changing by the occurrence of *actions* affecting those entities and thus their situations [22].

There are two alternatives in the context theoretical modeling approach. Context may either be described situation-centric or based on the entities' actions:

- *Context as situation composition*: Context is the composition of entities' situations captured from the entities' surrounding as well as derived from events/activities involving those entities [18, 22, 49]. An entity's situation may be represented as a snapshot of observable characteristics (i.e. values or contents), which be sensor readings, user input, etc. With such a snapshot forming static context at a single point in time, a composition of entities' snapshots, over time, forms dynamic context. Context can further be enhanced by assigning roles to entities and by defining relations between those [22].
- *Context as activity hierarchy*: Context can also be considered as a set of activities, which are evaluated based on their *performance* (meaning being performed) [81]. The central assumption of this approach is that *an entity performs an activity*, so that context is defined around the process of *performing* an entity (with that process being the relationship between entity and activity as shown in figure 2.2a). Hence, performing activities generates context by stating which activities an entity currently performs, which it may perform and which are not possible to perform. Identifying the current activity and inferring the according context from it can be done by analyzing the entity's surrounding (i.e. its situation[1]). This activity-centric context modeling approach further emphasizes an *activity hierarchy*. The hierarchy is defined by specialization and generalization of activities (e.g. *moving* specialized to *running* and *walking*, or generalized vice versa). Using such a hierarchy, the context of a specialized activity can be derived from a more general parent activity since contexts of generalized activities are usually known. Figure 2.2 visualizes the activity-centric modeling paradigm.

To avoid confusion we need to clarify in which relation we employ the term *situation*. Both modeling approaches describe the situational context of entities as we described contextual information argued earlier. The former approach rather focuses on observable characteristics while the latter approach infers possible actions at a particular point in time.

Context Conceptual Modeling

This type of context modeling describes context as concepts and the relations among such concepts as n-ary associations. Furthermore, such type of modeling categorizes context

[1]even though this step is related to building situational context, note that it merrily includes the acquisition of sensor data (e.g. temperature, etc.) to model context differently from the situational approach

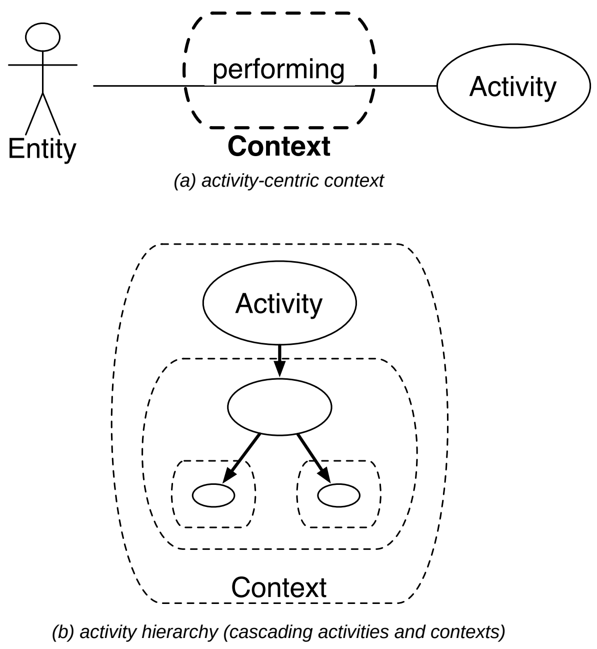

Entity
performing
Context
Activity
(a) activity-centric context
Activity
Context
(b) activity hierarchy (cascading activities and contexts)

Figure 2.2: Hierarchical activity-centric Context-modeling [81]

according to its prevalent characteristics [22]. Context conceptual modeling subdivides into 2 closely related alternatives:

- *Context as conceptual graph*: Context is mapped to *concepts* and relations representing associations between those concepts [49]. Such concepts may be represented by entities, so that multiple different entities may be put into relation to each other, thus modeling their contextual coherence as shown exemplary in figure 2.3. Entities can be enriched with properties, which are represented by attributes in the model. The associations between entities are directed and can be classified in 2 dimensions:
  - *static or dynamic*: depending on their timely behavior. Static associations represent invariant relationships among entities whereas dynamic associations are classified by the nature of their dynamics. E.g., Hendricksen et al. [49] differentiate between (1) associations derived from contextual data sensed by context sensors, (2) associations derived from other associations based on defined rules, and (3) profiled associations proactively defined by entities themselves.
  - *simple or composite*: depending on their complexity and amount of linked concepts. While simple associations put exactly two entity instances in relation to each other, composite associations may represent associations with many entity instances. E.g., figure 2.3 shows a visitor being interested in multiple exhibited objects by relating both the visitor and the exhibited object by a composite association. In addition, composite associations may have temporal constraints, such as the association between visitor and exhibition expresses a visiting period of maximum three hours.

This modeling approach is best visualized by a directed graph with the vertices representing concepts and edges denoting relations between concepts. Figure 2.3 shows a simple example of this technique yielding a directed graph. It is to be noted that such graph visualizes a modeling approach, which is to be clearly distinguished from the context models in the subsequent section 2.3.2.

- *Context as semantic graph*: Being similar to the conceptual graph approach, this approach is based on representing context as a set of propositions rather than a graphical representation of contextual information. However, the statements of propositional languages are visualized as a graph leaving it to be interpreted as a conceptual graph of semantics.

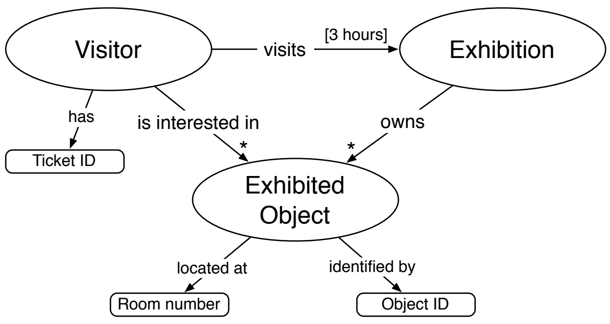


Figure 2.3: Conceptual Context Modeling Example

For both alternatives, contextual information may be either static or dynamic. Static information is obtained directly from appropriate sources. Dynamic context is captured by observing changing context over time.

Modals

The semantics of context models can be extended by *modals*, which are defined as specific properties of the world into which the context belongs [22]. They are used to describe changes of contextual information and can be sum up as follows:

- *Activity*: Context modeling is activity-oriented.
- *Belief*: Context modeling relies on belief committed by updates and revisions.
- *Probability*: Context models are probabilistic-based.
- *Time*: Context models dealing with temporal reasoning.
- *Fuzziness*: Context models are reasoning about uncertain information.
- *Category*: Context models are referring to more abstract or more specific contexts in terms of taxonomical interpretation.

2.3.2 Context Models

In order to create a context model, the conceptual model from section 2.3.1 has to be implemented appropriately by abstracting the conceptual model into structured information. A *context model* implements particular mechanisms to represent contexts and thus provides persistent storage and application of context. This section surveys commonly employed context models and discusses some associated aspects.

Summary and Evaluation of Context Models

An overview of representation models is given by [56] and [95]:

- *Key-value models*: Those are the most simple data structures associating context attributes with specific values of contextual information.
- *Markup scheme models*: These models consist of hierarchical data structures based on markup tags including attributes and comments. They are usually implemented as derivatives of SGML [12]. In some cases, markup scheme models are used to describe context as extensions of Composite Capabilities / Preferences Profile (CC/PP) [5] and User Agent Profile (UAProf) [15] to cover the high dynamics of contextual information.
- *Graphical models*: A quite intuitive approach to model context is to represent contextual entities and their relationships graphically. E.g., the conceptual modeling approach from the previous section 2.3.1 can be expressed best graphically, as seen on figure 2.3. Representative examples of graphical context models are listed below:
  - *Unified Modeling Language (UML)* [14] is a suitable instrument due to its strong graphical component. Its generic structure makes it appropriate for context modeling
  - *Object-Role Modeling (ORM)* [9] can be nicely utilized to represent context graphically by identifying facts, enriching those with types and roles, and putting them into relations and dependencies with other facts. A history component may be used to attach a time component to facts, as well as distinguish them to be either static or dynamic.
- *Object-oriented-models* consist of encapsulating contextual information into objects. The information can only be accessed through well-defined interfaces and is therefore hidden to from other objects. Due to the nature of object-oriented modeling this approach emphasizes reusability and controlled access to contextual information.
- *Logic-based models* represent a highly formal modeling approach. It is based on logics, which define conditions on which concluding expressions or facts may be derived from sets of other expressions or facts. Those conditions are described by rules in a formal system, so that the facts, expression and rules put all together define the context.

- *Ontology-based models* use ontologies, which are used to represent concepts and relations between concepts. They represent a uniform way for specifying the model's core concepts as well as sub-concepts and facts, thus enabling contextual knowledge sharing and reuse.

Depending on the conceptual approach when defining context models, one more realization alternatives from the list above are suitable realization options (e.g. ontologies for the conceptual graph model).

Ontologies

Current research indicates that ontologies are the most expressive context representation models [39, 95]. Ontologies provide a powerful paradigm for context modeling offering rich expressiveness and the supporting the dynamic aspects of context awareness. However, they require ontology engines managing the ontologies in use. Those engines generally have high requirements on resources, requiring the employing architecture to support those. This may have negative performance impacts on local context processing, where resource-constrained devices are employed [39].

As stated, ontologies represent *concepts* and *relationships* between concepts. The definition of concepts and their interrelations freely depends on the realization approach of the context aware application. Abstractions from the real world are usually mapped to concepts with relations interconnecting those concepts according to their real-world equivalents' relations. This may include entities such as item, person, location, environment, service, etc. [51, 37], which are mapped to concepts and represented by ontologies.

The ontologies representing individual entities must often be implemented in a certain frame to function properly. This frame is the same for all entities implying the need of ontologies being reusable. For this reason, ontology models often consist of two components [36, 37, 51]:

- *General ontology*: Those ontologies represent the general part of the model, which is used by all instances. It applies for all entities as it is. General ontologies are therefore used as a frame for ontologies representing entities.

- *Specific ontologies*: Those are employed entity-specific. Each entity may have unique characteristics that are represented by its own specific ontology only.

Concluding, ontologies representing an individual entity include a general part common in the ontology model, as well as a specific component for the entity's unique properties.

Beside their descriptive purposes, ontologies define sets of rules that are utilized by the inference engines using the context model to calculate the current context [36].

Ontologies are in use by numerous context-aware applications. The following listing illustrates an exemplary selection of some interesting picks:

- Christopoulou et al. [37] present an ontology to be used in their *E-Gadget Architecture*, which aims at making everyday objects context-aware and enhancing them to react dynamically to their changing environment. A lamp, that turns itself on when someone enters the room, because a doormat registered such event, is an illustrative example of this work.
- Huq et al. [51] present an approach to use ontologies to represent context-aware *meeting spaces*, whose users and installations collaborate context-aware.
- Khouja et al. [56] introduce an ontology model, which implements the *service-oriented-architecture* paradigm by introducing special concepts related to the correspondent SOA-entities (service requester, service registry and service provider).

Combination of Context Models

The practical implementation of context models may not be constrained to a single context model. The employment and mutual complementation of different context modeling approaches in a single system may highlight the individual strength of the alternatives chosen. E.g., object-oriented solutions may be combined and enriched with graphical components.

Biegel et al. [29] propose an interesting way of handling probabilistic aspects of context modeling. Their context-aware system employs a context model, which encapsulates a situational context into context hierarchies, implemented as an according data structure (markup hierarchies, graphically using trees). Those hierarchies define actions possible in such situations based on the specialization/generalization approach discussed in association with the activity hierarchy concept in section 2.3.1. To enrich the context with the probability of those actions to be taken, the activities in the hierarchy are represented by *Bayesian networks* [53], that are constructed from context sensors. Those networks allow a probabilistic conclusion on the actions represented in the context hierarchy.

2.3.3 Further Aspects of Context Modeling

In this section, we give some thoughts to aspect, which affect the context modeling process. In particular, we look at ways to semantically enrich context models with the notion of contextual relevancy.

Context Boundaries

An interesting aspect of context awareness is the utilization of *distance* of contextual relevance. In other words, we define how important contextual information surrounding an entity becomes in terms of distance. Although this aspect represents one of the very purposes of this work, we postpone the detailed discussion to the subsequent chapter 4. Here, we are about to briefly discuss the concept of bounding contextual data in the context of our survey. We do so by presenting two application cases depicting a data structure for bounding context: a *contextual boundary*[2].

Topological Proximity An approach presented by Roman et al. [85] conceptualizes context to be distributed on all participating nodes in a network. But a single node in the network is usually not interested in the complete context in the network, but rather only in relevant contextual information in its vicinity, which consist of the contextual data available at the proximate nodes. Thus, each node may define its context boundary, depending on its individual preferences. This boundary is therefore individually rooted at the respective node defining its relevant context.

Roman et al. [85] compute the boundary on the basis of the logical topology graph of the underlying network. The logical topology is derived from the physical topology by refining it in regard to application specifics. Defining the context boundary solely depends on the individual nodes preferences and may be implemented by any procedure that fits this purpose. With the context boundary set, the access to the bound context consists of accessing the contextual data at all nodes within the context boundary. The access is optimized by defining the optimal access route for each of those nodes. This is done by constructing a shortest-path-tree, including all of those nodes, and refining it by regarding cost aspects. Those cost aspects may directly influence context boundary, which may have to be adjusted in the process. The resultant data structure includes a minimum-cost-path for each node within the context boundary.

Since the physical network topology, the nodes' information base and individual preferences are usually constantly changing, the context boundary and the implied minimum-cost-cost-paths at each node have to be constantly updated.

Context spaces A different approach in bounding context information is introduced by Padovitz et al. [75, 77]. Similarly to the contextual map approach, context is split into contextual attributes and mapped to multi-dimensional space, which the authors called *context space*. By defining hyper-dimensional regions in the context space (i.e. subsets of the context space), the bounds for a set of possible situations are set. Such a region is defined as an accepted situation space including possible situations, which are clearly identified and accepted by a context-aware system. Hence, every context mapped to a point inside such a region depicts an accepted situation. Thus, the context boundary in this approach is depicted by the edges of a region in the context space.

Heterogeneity Aspects

The heterogeneity of mobile environment has a peripheral impact on context modeling. Especially the following aspects are to be regarded [39]:

- *Software heterogeneity*: Context models may have to address certain application requirements. Hence, the presence of heterogeneous software may have to be con-

[2]not to be confused with the contextual map's contextual boundaries

sidered. Additional problems arise when heterogeneous context models are to be employed simultaneously [92].

- *Architectural heterogeneity*: Context models may be bound to specific network domains, because only some context information may be relevant. Multiple heterogeneous networks may have to be considered.

Context quality

The quality of context reflects how accurately it describes the corresponding entity's real-world situation. In other words, it describes the degree of contextual error [49, 78, 76, 77]. Contextual errors have many possible causes. For one, readings from context sources, such as sensors afflicted with a certain level of error. Consequently, the resultant context built from this false information inherits the error. Another cause is the heterogeneity of context sources. Collecting data from many mutually different sources is complex process forcing to convert the heterogeneous input into a common representation, so that the collected data from all sensors may be used together. This transformation process always carries the risk of losing some of the information causing the common representation to be false. Building contexts from this information inherits the error, too.

There have been efforts to counteract this problem. We will return to this issue during the discussion about context-aware system design in the following section 2.4.

2.4 Context-aware Computing Systems

After having discussed the basics on how to define and model context, we put our focus on the design of systems utilizing context. We especially emphasize the requirements and architectural issues of context-aware systems.

2.4.1 Relevant Aspects and Requirements

Context-aware applications are required to consider certain aspects concerning context acquisition, modeling, access, handling and adaptation [22, 39]. The requirements of a context-aware application are characterized by the proper handling of the following aspects:

- *Context discovery*: The aim of context discovery is to locate and access contextual sources [22]. It may be utilized in conjunction with a serving context request (e.g., discovering the appropriate or approximate context pertinent to an entity). Context discovery covers the issues of service description, advertisement and discovery.

- *Context distribution*: The distribution policy specifies how access to distributed context information is supported. There are 2 basic approaches [39]:

  - *local context*, i.e. context data probed and used only at a device.

  - *distributed contexts* that can express situation of a set of entities that may be distributed over a network.

- *Context acquisition*: A mechanism to obtain context data from diverse context sources. This includes hardware sensors delivering information that conforms to a low-level data model [22, 42, 76], as well as software components acquiring contextual information from elsewhere [81].

- *Context aggregation*: A mechanism that provides context storing and integrity [22]. In case of a shared context model, the context aggregation forms a basis for merging correlated contextual information. The context composition is a specific kind of context aggregation, when the involved contexts are compatible with the same, or equivalent, context model.

- *Distributed composition of context*: Ubiquitous computer systems are derivatives of distributed systems, thus possibly lacking a central instance to provide the context model [95]. Composition and administration of a context model varies with notably high dynamics in terms of time, network topology and source.

- *Communication of context*: The communication policy specifies how context information is disseminated or gathered through the network by interested applications. The information can be distributed either *synchronously* or *asynchronously* [39].

- *Context consistency*: Context consistency enables the rationality of dynamically changing distributed context models [22]. Such mechanism, regarded as being an extended context aggregation mechanism, maintains the structure of the contextual model into higher levels of abstraction.

- *Handling of ambiguity and incompleteness*: The set of contextual information characterizing relevant entities in ubiquitous computing environments is usually incomplete and/or ambiguous, in particular if this information is gathered from sensor networks [95, 49, 77, 76, 78]. This should be covered by the context model.

- *Partial validation of context models*: It is desirable to be able to partially validate contextual knowledge on structure as well as on instance level against a context model, even if there is no single place or point in time where the contextual knowledge is available on one single node, since context is usually composed distributed [95].

- *Formality of context models*: It is always a challenge to describe contextual facts and interrelationships in a precise and traceable manner. It is highly desirable, that each participating party in an ubiquitous computing interaction shares the same interpretation of the data exchanged [95].

- *Context querying*: Context querying is the task of handling requests, that need to be executed over the context repository. Context retrieval tasks must be transparent to the requester and posed in a concerted form (e.g. by a querying language)

[22]. Context consumers should register interest in context in order to be notified asynchronously when it changes. Context querying can be implemented in 2 ways [39]:

  - *centralized* handled queries, in which a query is sent directly to the repository.
  - *distributed queries*, in which a query can be distributed among several repositories and the result is composed as a unique context information, that maintain the context.

- *Context adaptation*: The context-aware application should be capable of adapting its behavior according to contextual information [22]. Specifically, it automatically adapts the system configuration in response to a contextual change.

- *Context reasoning*: Context can be elaborated with reasoning mechanisms [22]. Context reasoning is a process for inferring new context, previously unidentified on the basis of the context currently known by the application. Basically speaking, context is reconfigured according to known context and newly acquired contextual data [88]. Reasoning tasks check context consistency and deduce high-level context according to contextual information. Specifically, it automatically adapts the system configuration in response to a contextual change.

- *Update of context*: The updating policy specifies how a context information is updated at a repository and how this update is disseminated to context consumers [39]. The most common policy is to send *frequent updates* in order to guarantee that context consumers access the up-to-date context information. However, this trivial approach is inefficient, since updates can either be committed too late (in case of fast changing context) or unnecessarily (in case of slowly changing context). *Proactive updates* committed upon a change of context remedy this problem but can result in a high update load in case of fast changing context. More sophisticated update semantics check for *certain conditions* that justify an update [61]. A slightly different approach is to enable context consumers to use *intelligent components* that infer the actual information based on a history, or evaluate the information's freshness through a comparison of time-stamps.

- *Context evaluation*: The context evaluation policy specifies how a context-aware application evaluates each attribute that composes a context information [39]. There are two approaches: *eager evaluation* and *lazy (on-demand) evaluation*.

- *Quality of context information*: Context information usually comes from heterogeneous context sources, such as sensors and software services. The quality and richness of that information can vary over time [22, 39]. As stated before, sensor readings can be afflicted with an uncertain error reducing the quality of the contextual data derived from such sensor readings. Transforming such low-level contextual information into higher levels of context is another source of error affecting the quality of context [49]. Furthermore, the lack of a universal context model and application-specific representation of the contextual data undermines the consistency of the sensed information. A mechanism for maintaining predefined sets of quality indicators and corresponding quality metrics is very important [22, 49]. Such indicators may be resolution, accuracy, repeatability, frequency and staleness of context. Context quality has a direct impact on context stability. Improper context data may cause an entities context to fluctuate even if its real-world context stays unchanged. Thus, maintaining contexts stability is an important requirement for context-aware systems [77].

- *Context integration*: Existing context models vary in the expressiveness they support, in semantics, and in the abstraction level of the conceptual entities [22]. Contextual information integration can be conducted whenever different context models are in accordance, not only with their semantics, but also with their similar domains of interest.
- *Applicability*: From the implementation perspective it is important, that a context model must be applicable within existing infrastructures of ubiquitous computing environments [95].
- *Context service placements*: The placement policy specifies where the context management is hosted. The simpler policy is the adoption of a *centralized* context service, but it imposes performance and scalability problems. *Distributed* context services can be implemented using four approaches [39]:
  - interconnecting context services distributed through a network
  - maintaining a context service in order to handle local context plus a remote service to disseminate context among other devices
  - implementing ad hoc services to enable context management in ad hoc networks
  - using a combination of the aforementioned solutions

2.4.2 Architectural Issues

This section takes a closer look on the architectural design of context-aware systems. We regard both static and dynamic aspects. Beginning with a list of involved components, we also reflect on the workflow of those systems.

Components

The composition of context-aware systems is characterized by a high degree of heterogeneity. However, all approaches agree in decoupling context capturing and context processing from application composition [36] by encapsulating the context management logic into middleware.

An analysis of the architectures of multiple context-aware systems presented in various publications [42, 52, 29, 36, 39, 100] identifies the following abstract components shared by the majority of those systems:

- *Context sensors*: Those sensors exist either as pieces of hardware sensing the physical environment, or as software component providing data from other context sources [52, 22, 49, 81, 42]. The primary task of both sensor types is the acquisition of raw data for further refinement into contextual information. In section 2.2.3, we have discussed and classified the possible context sources that context sensors acquire their data from. Concluding, we can depict the following types of sensors [42]:
  - *Visual and auditory*: Those used to capture aspects of the real world
  - *Location*: There are two basic options of how to determine the current location of a mobile device: (1) triangulation using the sensor infrastructure consisting of cells, or (2) using a satellite-based positioning system, such as GPS and GLONASS. A special approach is the application of visual sensors for location detection by recognizing visual waypoints.
  - *Other environmental sensors*: This set includes all other environmental sensors, not covered in the list so far. E.g., it may include motion detectors and temperature sensors.
  - *Software sensors*: Those "sensors" are software components that draw information from context sources other than the environment. This may include databases or network infrastructures.

  This list emphasizes the construction of context information from powerful single-purpose sensors. While this approach is quite practicable, it has several shortcomings concerning cost and the ability to capture dynamic aspects. The alternative approach proposes to replace such sensors by collections of multiple simple sensors, enabling the so called *multi-sensor context awareness* [42]. Even though simple sensors individually capture a much smaller portion of the environment than more complex sensors, the mass of simple sensors usually provides a more accurate awareness of the environment than their more powerful counterparts.

- *Context capturing interface*: Data acquired by sensors is usually uncertain and difficult to interpret by high-level components [95]. For this reason, sensor data needs to be refined for further processing by deriving a higher level of context from uncertain multi-modal sensor data - a process that may be called *sensor fusion* [29]. It is characterized by refining raw sensor data into data structures that are utilizable for higher application levels, thus providing a proper interface for the sensed environments [42]. Such data structures may reach from simple data types to sophisticated structures, such as Bayesian networks employed for probability calculations [53].

- *Context repository*: The context repository represents the persistent storage component in a context-aware system. It stores all context information currently known to the system. Thus, it implements the context model of choice (as discussed in section 2.3.2).

- *Context reasoning*: The context reasoning component is responsible for inferring new context based on the current contextual information in the context repository and new contextual data acquired through the context capturing interface. This process is put in practice by *inference engines* that work on the basis of rules [29, 52]. They fetch the contextual information from the repository and the contextual updates from the context capturing interfaces as input parameters, and they output an updated context model, that is committed to the context repository. Rules can be either of global or local scope. Global rules affect the entire context model. Local rules, however, have an effect on local entities only [36].

- *Context API*: The context API provides an interface for context-aware applications to actually utilize contextual information [39]. The employment of such an interface implements the commonly accepted paradigm of separating context management from application logic [36]. It actually represents the cut between those two by laying on top of the context management system making it transparent for context-aware applications.

- *Context application*: Applications accessing the context API can actually *use* the contextual information for their application-specific purposes. For this purpose, the context API provides well-defined context information, which is suitable for the accessing application. Concluding, the context API is usually application specific. Application-based employment of context includes the automatic execution of services based triggered by special contextual conditions, as well as the discovery and allocation of resources relevant to the current context [67]. The aspect of context application especially emphasizes the deployment of legacy applications [52].

- *Communication interface*: Since a context-aware system is distributed in nature, communication needs to be handled appropriately. This especially concerns the reasoning mechanisms, which may want to commit contextual updates into the network, and context capturing components, that may request contextual information from other nodes in the network.

- *Actuators*: Actuators are the counterparts of sensors. Unlike sensors, which *sense* their environment, the actuators' primary purpose is to *alter* the environment. Actuators are usually utilized as a result of a contextual update [29] requiring a proactive change of context.

Layered Architecture

The components discussed above form a flat representation of a generic context-aware system. In order to add levels of abstraction to it, we subdivide it into a hierarchy of layers with each layer representing information on a specific level of detail [36]. We proceed from low-level to high-level layers:

- *lexical level:* signals from sensors are abstracted into basic context events.

- *syntactical/representation level:* context events are translated to atomic context information, such as matching sensor data to real-world-properties.
- *reasoning level:* basic context information is refined and organized, context information is fused into a reasonable representation suitable for more sophisticated processing (context fusion in context hierarchies).
- *planning level:* context is evaluated, changes in context are detected, reactions to context changes are planned and scheduled.
- *interaction level:* reactions to context changes are executed in form of personal and collaborative interactions with the user and other hosts.

Figure 2.5 shows the coherence of the components and layers.

Workflow

So far, we have put our focus on the static aspects of context-aware systems only. To describe the dynamic behavior of such a system we can derive the workflow of capturing, storing and utilizing contextual information from the static architecture (components and layers) by extracting the components' tasks and putting those in sequence reasonably. Figure 2.4 visualizes the subsequently documented approach.

1. *Context sensing*: Detection and representation of contextual information [67]
   (a) *Acquisition of sensor data (low-level context)*: Raw data is captured by sensors, representing contextual information on the lowest level of abstraction (sensors on the lexical level) [36].
   (b) *Refinement of sensor data*: Raw data acquired by sensors is interpreted and represented in data structures to provide a higher level of context (context capturing interface on the syntactical level) [36].
2. *Context update and management*: The refined and well-presented contextual information is fetched and merged into the context repository. The overall context is updated in the process (context reasoning and context repository on the reasoning level) [36].
3. *Application of context*: The most current contextual information is fetched from the context repository through the context API [39] to be used by the context-aware application. This may occur by either pushing or pulling the information to the application level, dependent on the implementation of the application. The use of the current context includes the following aspects:
   - *Contextual adaptation*: Context-aware applications automatically adapt to the current context by updating/executing their according services and their internal states [67] (context application on the planning level [36]).

- *Contextual resource discovery*: The dynamic state of the context requires the dynamic location and association of resources relevant to the user's context.
- *Context augmentation*: The user's context is enriched by digital data, which is presented accordingly to the user.

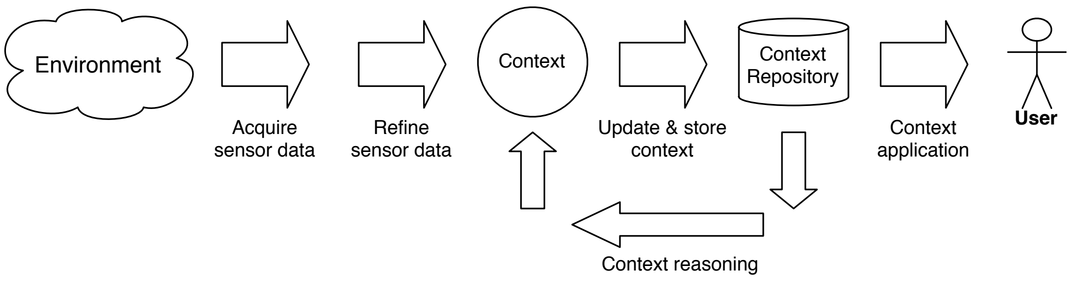


Figure 2.4: General Context Management Workflow

Summary and Implications

So far, we have identified common architectural traits of context-aware systems in regard to both structural (components) and behavioral (workflow) aspects. This enables us to conclude the architectural analysis by deriving a *general conceptual architecture for a context management system* providing contextual information for context-aware applications. Figure 2.5 visualizes this concept employing all of the relevant components on corresponding abstraction layers and implementing the context capturing process as discussed earlier in this section. It is to be noted that the visualized architecture depicts our particular view on context-aware system architectures, which we have elaborated during our survey on context-awareness.

The core of each system excelling awareness of context is the context model storing the current contextual information. As we have stated earlier, ontologies are the first choice made by most research groups due to their high degree of expressiveness. Since the employment of ontologies requires adequate hardware capabilities, one may be forced to fall back on less demanding context models if the available hardware is constrained.

The workflow of recognizing, updating and utilizing context is strongly inspired by the procedure described in the preceding subsection. *Sensors* (which may explicitly include software components as well) acquire raw environmental data, which is refined by the *context capturing interface.* This process is conducted by converting the raw data into contextual information. This means that the source data is abstracted into discrete data structures and enriched semantically, so that it becomes clear which data coming from which source contributes to which part of context. The resultant contextual information can be processed further by high-level components in the context management system. The contextual information is then committed to the *context repository*, which is responsible for the persistent storage of the context using an appropriate context model. The

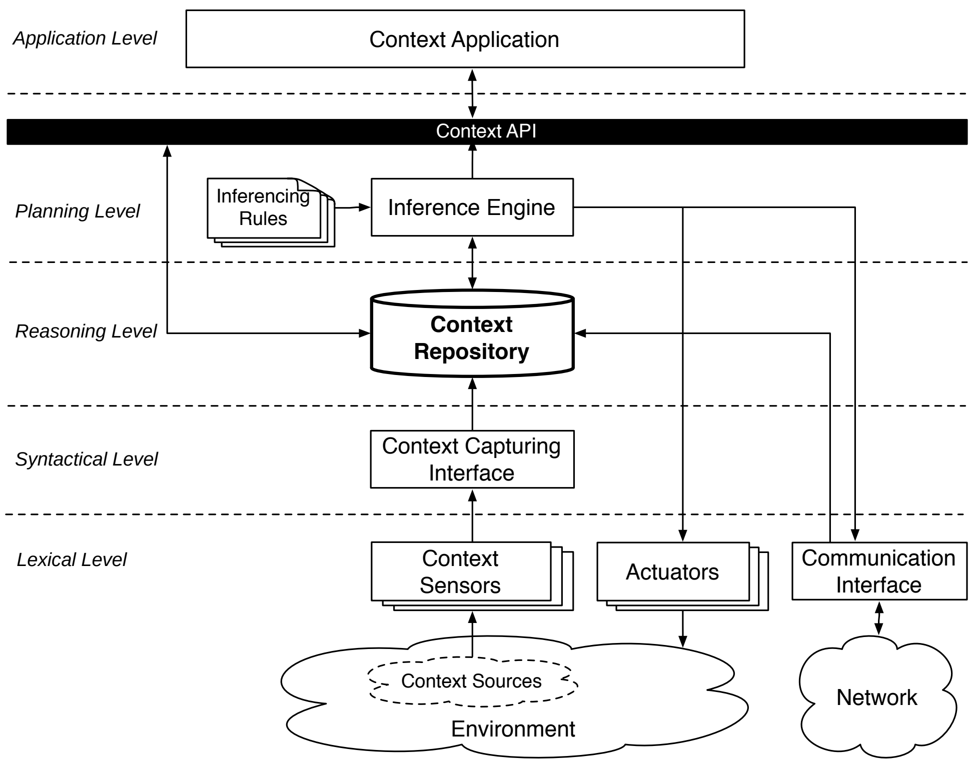


Figure 2.5: General Architecture for context-aware Systems

context repository controls the access to the contextual data and therefore functions as an interface to both the rest of the context management system and any context-aware applications utilizing context.

Context in the repository can be used to infer new context. This inference process is conducted by the *inference engine* according to *inference rules*, which define how new context is inferred. Both the rules and contextual information are loaded by the inference engine, which subsequently updates the context in the repository according to the inference rules. Since we are dealing with a distributed system, newly inferred context may be propagated to other nodes via the *communication interface*, which is attached to the communication hardware. The inference engine may also trigger any *actuators*, which are affected by the update of context. Contextual updates may also be received from other nodes in the network, so that the communication interface forwards them to the context repository.

The *context API* provides access to the context for user applications, and it represents the architectural cut between context management and context utilization as we have argued earlier [36]. The context API has direct access to the context repository, meaning it reads the current context and commits user updates into the context. The inference engine may also notify user applications through the context API, if the inference of new context requires it.

Regarding the architecture sketched in figure 2.5, we are dealing with an actual implication derived from various context-aware system architectures, thus going beyond a simple survey or enumeration on related research, respectively. For this reason, we clarify how it integrates into the contextual map's problem statement in section 1. For this purpose, we regard figure 1.2 from section 1.3 depicting the decomposed problem statement for contextual similarity handling. The reasoning level in the architectural concept (figure 2.5) corresponds to the model layer in the problem statement (figure 1.2). Our architectural concept includes the context repository, which stores reasoned and semantically accurate context. That is exactly what the contextual map model does. Its multi-dimensional context representation represents current and error-purged entity situations. The next upper level in our architectural concept, the planning level, corresponds to the output level in the problem statement. The planning level is dedicated to inferring new context, thus enriching present context without the acquisition of additional contextual information. The identification of similar context pairs and context clusters can be related to this procedure, since it also yields augmentation of the contextual map's current context data. The application levels in both representations represent another correspondence. Both are dedicated to the utilization of context data prepared by the respective lower levels.

2.4.3 Aspects of computational Distribution

Even though approaches featuring a centralized context management component are generally realizable, almost all of the current research emphasizes either distributed systems or hybrid approaches, where centralized component act as a subordinate system supplement. This fact implies that software providing context-awareness is deployed as middleware on the nodes in the distributed system. This section illuminates the aspect of dealing with geographically distributed and mobile entities.

Constrains

Since context-aware computing emphasizes the use of mobile devices, certain hardware restrictions must be considered [97]:

- *Power*: Mobile devices have a limited power supply, that has a direct impact on the performance of its components.
- *Communication*: Connections may become transient and occur non-deterministically.
- *Processing*: Mobile CPUs have low capabilities due to the low amount of available power and space.
- *Storage*: Storage components are non-mechanical and face the same problem of power and space availability. For this reason, mobile storage components are restrained, too.
- *User interface*: Due to the reduced size of mobile component, the user interface is forced to be very simple.

With those restriction applied, context-aware middleware further needs to enable mobile nodes to meet the following requirements [29]:

- *Sentience*: The node has to be able to sense its own environment by utilizing hardware or software sensors.
- *Autonomy*: Since we assume a distributed system, nodes are required to operate independently.
- *Pro-activeness*: Mobile nodes require the ability to act in anticipation of future goals.

Distributed context Spaces

The notion of *context spaces* is used to group contextual data under certain characteristics. In sections 2.1 and 2.3.3, we have already mentioned context spaces synonymously with similar contextual realms. Jacob et al. present an approach employing context spaces in a slightly different manner. They define context spaces on sets of distributed entities with similar contexts [52]. In their decentralized system, a context space is set of nodes, which commonly share information. That implies a bounded locality of context inside that context space. Thus, a context space allows a distinction between local and global context aware applications: the local ones only access the contextual information in their context space, global context-awareness denotes the utilization of the global context consisting of all subordinate contexts in the respective context spaces. Since context is being bounded, this approach has peripheral similarity to the context boundary aspect, introduced in section 2.3.3 [85].

Modularity

Since context-aware middleware needs to be deployed on a large number of devices, additional requirements regarding scalability and maintainability need to be considered. As stated before, the most significant architectural cut consists of separating the context management system from the context-aware application [36]. This enables the deployment of individual applications on the same context management system.

However, the context management system itself may have to be applied to different use cases - basically requiring different context management systems. Hence, unitizing the context management system seems reasonable [100]. A component used by a context management system may be used as a generic component on all devices. This usually includes the acquisition and provision of contextual information. Specific context-processing components can then be deployed on top of the generic ones. They usually implement the mechanisms how context is individually processed (update, storage, context API).

2.4.4 Heterogeneity Aspects

The impact of heterogeneity in pervasive environments has been stated several times. For context-aware systems deployed in those environments, this aspect especially complicates the context acquisition process due to heterogeneous context sources. Besides environmental aspect, application-specifity may cause heterogeneity having an impact on the technical design of context-aware middleware as well [39]:

- *Hardware heterogeneity*: Servers, workstations and mobile devices in a context-aware system have very different characteristics and capabilities, especially concerning platforms, network technologies, computational power. The challenge in middleware design is to make it deployable on all of those device types.
- *Software heterogeneity*: The software running on the devices may differ highly, too. This encompasses the presence of different operating systems and applications.
- *Architectural heterogeneity*: The communication between various nodes may not be uniformly defined neither. This heterogeneity aspect comprises of the existence of different network architectures and communication protocols.

The impact of heterogeneity has been discussed only in brief at this point. Section 2.7 presents an in-depth discussion about the heterogeneity issue.

2.4.5 Application Cases

In this section we conclude our context-aware computing survey by presenting a few applications showing awareness of context:

- *Contextual risk based access*: Diep et al. [40] employ an access scheme based on contextual risk calculation. It is based on 3 factors: availability, integrity and confidentiality. A context module acquires the context, a *risk assessment* component calculates the risk from the gathered contextual information by evaluating the outcome of possible actions, and finally, the *access control manager* decides about the grant of access based on both contextual information and the calculated risk value.
- *Pheromonial assessment of relevance of information*: Jacob et al. [52] introduce a system of context-aware and loosely coupled devices communicating within ad-hoc networks. *Personalized context relays (PCR)* represent middleware deployed on the devices, managing the context and enabling context-aware applications. The relevance of contextual information is inspired by biological *pheromones*: the more often an information is queried the more important it is and vice versa. Context aware applications are classified either local or global depending if they access context in their context space only, or if they require global context.
- *Location history* A cost effective way of enabling location histories in a context-aware environment is presented by Mantoro et al. [67]. The location, commonly considered as an important context attribute, of entities is detected, traced, stored in a database and made accessible by a querying language. Context agents are used to autonomously recognize user requests and fetch the required location information from the database. The freshness of this information depends on the most current

location update. The system is also able to predict locations based on the location history.

- *Utilization of the Object-Request-Broker (ORB) concept*: The context-aware approach developed by Yau et al. [100] implements a modular middleware for ad-hoc networks. Context-capturing core components (*R-ORB*) acquire contextual information and distribute it around the network implementing the ORB-paradigm. *Adaptive context containers (ADC)* represent individual context-processing components accessing the contextual information presented by the underlying R-ORB. Context-aware applications are executed on top of ADCs, utilizing the context provided by those.

- *Sentient objects*: Biegel et al. [29] introduce a framework, which emphasizes the conceptualization of *sentient objects.* It supports the development of context-aware software components, which are capable of autonomous sensory sensing, context abstraction and reasoning, and controlling of actuators. It uses Bayesian networks [53] to control probability aspects.

- *Location and context awareness*: A mobile computing middleware presented by Bellavista et al. [26] focuses on the *visibility* of user characteristics. Location is defined as the visibility of a user's position, whereas context denotes the visibility of resources associated with the user. The context management system emphasizes mobile agents to autonomously handle problems caused by the mobility aspect.

As presented, context-awareness is applicable in many utilization domains. On the one hand, the utilization of context is very suitable in distributed systems due to their dynamic behavior making context to change very quickly. This aspect makes mobile service development one the prime targets of our work. On the other hand, another interesting application domain is the development of internet services. Keeping this in mind, hybrid approaches implementing web-based mobile services are an important target group, too [90, 91].

2.5 Location-aware Computing

In the introductory section 1.2, we outlined the idea to borrow the mechanisms of geographical proximity detection for the identification of contextual similarities. Since proximity detection is settled in the location-awareness domain, this circumstance motivates a short survey about that research domain. Thus, this section introduces the area of location-aware computing in brief. For the technical aspects of the subsequent discussion, we assume mobile entity possessing a mobile device, which allows its localization.

2.5.1 Location and location-based Services

Location is an essential part of contextual information in the mobile computing domain and can generally be defined as *the property of physical position of users, devices and* *resources* [26]. The dynamic nature of this property has yielded a wide spectrum of sophisticated and highly dynamic services employing location-awareness [90], but summoning challenges in the fields of infrastructure design and location management as well [99].

Location-based services come with a set of requirements specific to their dedicated application domain [87, 90]:

- *Mobility:* denoting the presence of mobile objects implying the necessity to manage their locations appropriately.
- *Heterogeneity:* reflecting the issue that involved devices are highly heterogeneous regarding their individual capabilities.
- *Availability and responsiveness:* require services to be always available and minimize response times independently from the mobility aspect.
- *User interface*: designed according to the applying output restrictions, hence required to be simple, but efficient.
- *Distribution:* reflects the presence of distributed component comprising a single system, especially noting the option of absence of centralized components.
- *Transparency:* emphasizing the location-aware tier to encapsulate location management to upper layers, such as applications.

2.5.2 Location Detection

Determining the position of a mobile device may be partitioned into two basic approaches [90, 54]:

- *Zone-based:* In this approach, position determination is conducted by the infrastructure solely without including the mobile device into the process. Any location updates are initialized by location servers, which detect the presence in one of their attached infrastructural *zones*, implying that this approach is only feasible in client-server architectures. Consecutively, the server keeps autonomously track of the clients. Further, this approach can be decomposed as follows:
  - *Cellular:* Since most networks including mobile devices are cellular, a mobile device's position can be detected when it enters a cell and registers with the cell's base station. Since cells differ significantly in size, this determination method is quite imprecise leaving a large area of possible location instead of a unique point. There are sophisticated approaches to increase accuracy of this localization method, such as triangulation [90].
  - *Sensory:* Similar to the cellular detection mechanism, this approach works on the basis of tracking devices with *sensors*. Each sensor spans a detection area in which a mobile device can be detected by the sensor. However, cellular

detection implies that the entire roaming area is covered by cells whereas sensory detection may only cover spatially distributed and non-adjacent zones. Sensory detection is normally conducted by tracking the device without its apprehension, for example via RFID [87].

Despite the shortcomings of infrastructure-dependent location detection it is often the chosen course of action, since the requirements on mobile devices are considerably low. Given the fact, that the number of mobile devices roaming in a network may be very high, any distribution issues arising when involving mobile devices in location detection do not exist. Location detection is instantly possible with any device roaming the area covered by the infrastructure.

- *Map-based:* This approach focuses on locating a mobile device at an exact geographic point. Compared to zone-based detection mechanisms, this approach is far more accurate. However, location detection must be conducted by the mobile device itself using a satellite grid enabling location determination (e.g. GPS, GLONASS, Galileo). Subsequently, the mobile device commits location updates to a centralized server or to other mobile hosts [54].

  This approach allows implementation following both client-server and peer-to-peer architectures. However, it requires at least a small portion of application logic to be deployed to any involved mobile host decreasing the scalability of this approach in comparison to the zone-based mechanisms.

With location update mechanisms being an integral building block of location detection, the emphasis here lies on maximizing their efficiency by minimizing the amount of updates and maximizing the currency of location information. This paradigm is justified in the effort to minimize traffic in the network, in order to reduce costs and avoid flooding [67, 61, 57].

2.5.3 Location Models

Location models denote data structures and strategies to store location-related information in location-aware systems, including physical environments and locations themselves. Those models form the frame for representing location in databases and according infrastructures [67]. There are two basic approaches on implementing location models:

- *Cartesian:* Location models implementing this approach are based on Cartesian coordinates allowing geographic map-based models. A popular example of this model is an annotated road network, represented as a graph with attributes on its edges (streets) and vertices (intersections) [54]. The location of mobile nodes is projected upon this graph and location management is conducted on the basis of this model. Attributes in the graph are used to augment the models information, such as enriching edges (streets) with information about speed limit, width, etc.

- *Hierarchical:* This approach focuses on topologies organized in hierarchies. They are usually tree-based and reflect locational spaces in hierarchical relations to each other. For example, one may imagine a house, including rooms with each room containing tables and chairs [87]. Each of those entities (house, room, table, chair) may be regarded as a space anchored in a topological tree. As with Cartesian models, hierarchical models may be annotated to add auxiliary information, such as adding attributes to nodes in the tree hierarchy. However, unlike with Cartesian models, hierarchical models allow the insertion of auxiliary information inside the model's core structure. This means that non-location-related nodes may be inserted into the tree. Children of a node denoting a space may represent auxiliary information about that particular space. For example, the model proposed by Satoh [87] is based on the discussed tree hierarchy. The tree nodes represent spaces as well as auxiliary information such as auras (some sort of an object's virtual space) and services. A space-node's children being auras and services describe the auras and services being associated with the parent space.

Figure 2.6 visualizes those two models exemplary.

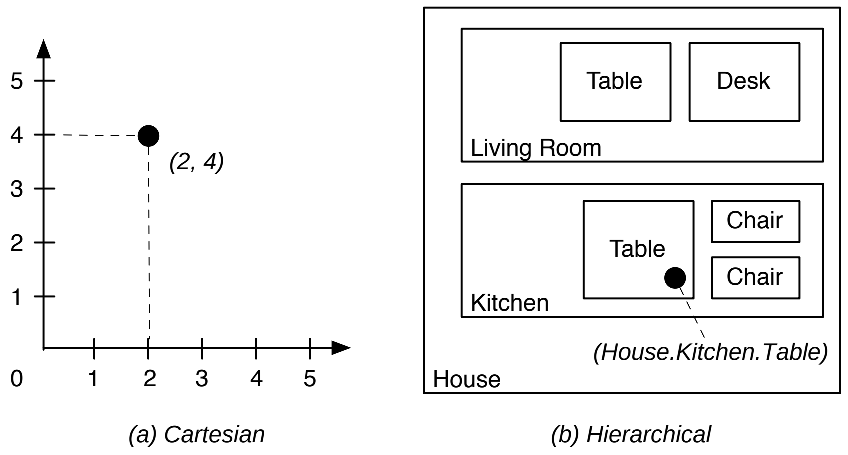


Figure 2.6: Location Model Examples

2.5.4 Location Management

With the location models introduced, this section focuses on how such models are utilized in managing location. The discussion emphasizes on how this information is processed in a location-aware system by regarding its architectural characteristics.

Architectural Layout

We begin our discussion by briefly outlining the two mutually excluding top-level design choices: client-server vs. peer-to peer.

Client-server The basic thought of location-aware client-server-based systems is centered around a centralized server taking care of all location management tasks [72]. Location updates are generated by monitoring movements of mobile hosts by means of the infrastructure or by receiving location updates from the clients, either autonomously or following the server's update request.

This basic 2-tier architecture, including the server and the mobile clients, can be extended to multiple tiers by including local servers into the architecture [99]. Local servers are responsible for local regions, where each local server manages a particular region, basically meaning that location management tasks are delegated from the central server to the local servers. Moreover, local servers can be grouped into multiple hierarchical levels depicting a differentiated granularity of regional partitioning. In this setting servers are structures in a tree-based hierarchy with the root being the central location server. A local server on a lower hierarchy level is responsible for a region, which is a subset of the responsible region of the server's corresponding parent server on the next higher level. Figure 2.7 shows the architectural layout of this system.

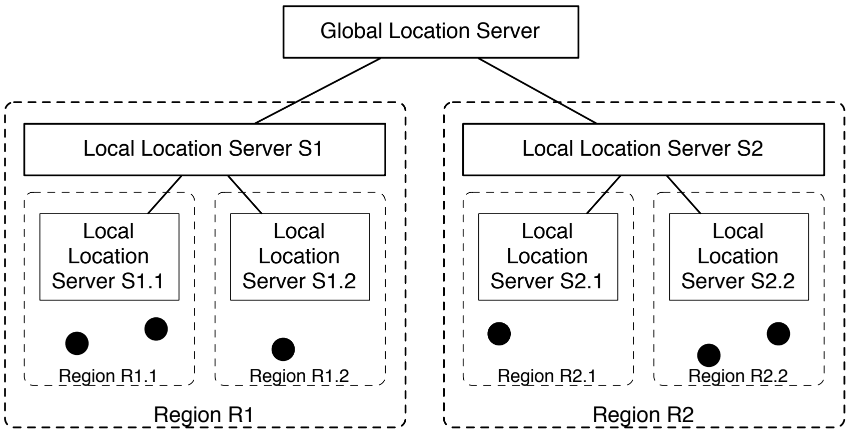


Figure 2.7: Multi-tier hierarchical Location Management System

The following fictional example is intended to illustrate this architecture. Consider a macroscopic location-based service being deployed in North America. With millions of mobile devices using this service, location management is extremely elaborate and cannot be processed by a basic 2-tier client-server system (one server, mobile clients). Hence, this architecture is extended into a hierarchy of local location servers with associated regions with every location server managing the mobile targets in its region only. Starting, there may be one server for Canada, one for the USA. Regarding the USA-server, one location server may be installed for each of the 50 states on the next lower level of this hierarchy with the USA being their parent server. In California, location servers for San Francisco and Los Angeles can be deployed on the next lower level, and so on.

Although this multi-tier approach is highly scalable, it raises communication issues on how to efficiently propagate location updates in the hierarchy [57]. Basically focusing on optimizing traffic caused by location updates, the basic principle addressing this issue consists of handling location updates from the bottom up in the hierarchy. Location updates are only propagated to the next higher level, if the update concerns this level at all. Consecutively, the resultant traffic stays as local as possible.

Peer-to-peer The absence of centralized components in mobile P2P-mashnets shifts location management tasks to the mobile peers themselves [87]. Naturally, this mainly includes handling the particular location model and processing location updates. The challenge in this approach is to keep the peers' current location data consistent albeit interaction between peers remains short and transient. Another issue is the presence of restrained capabilities existent on mobile peers [97], requiring location management to obey those constraints. Three basic approaches can be identified for location managements in P2P environments:

- *Each peer* manages its location model itself. Location updates are propagated during the brief interactions with other peers [87].
- *Temporary location servers* are peers that become a centralized node in the network if certain conditions are true [57]. They perform location management tasks until they are discharged of this duty.
- *Super peers* are mobile peers, that act as gateways for other peers [46]. Although this setting has close conceptual similarity with client-server approaches, it is to be emphasized, that super peers are full members of the P2P network.

Hierarchical partitioning of the covered area as introduced in the previous client-server discussion can be used to optimize both location management and traffic in P2P-networks as well. The regional partitioning itself is predefined and analogous to the client-server approach: regions on lower levels are subsets of parenting regions on higher levels [57]. In comparison to the client-server architecture, stationary local servers administrating those regions do not exist. Instead, peers become temporary location servers or location management is provided by super peers:

- *Temporary location servers*: Selection of temporary location servers must be uniformly enabled by all peers. Since broadcast mechanisms in P2P-networks are insufficient in terms of providing information about current location servers ubiquitously, selection and identification of temporary local servers can be conducted by algorithms using pre-defined hash-functions [57].
- *Super peers*: This approach comes with the requirement of at least one super peer located in a region. Taking into account that super peers are mobile, this requirement cannot be guaranteed. Hence, this approach must be complemented by mechanisms, such as the temporary location server approach, to work properly.

Workflow

The location-aware system (i.e. a server or a mobile device, depending on the architecture) initiates a location-based service either reactively or proactively:

- *reactively:* The system receives a service request coming from a mobile device.
- *proactively:* The system monitors its mobile hosts and detects the necessity to initiate a location-based service.

Although location-aware services are often very specific, the following general workflow can be devised as visualized in figure 2.8 by a UML activity diagram [54, 72, 61]:

1. *Location Detection:* The location of the mobile device (and it user) is detected as described in section 2.5.2. Besides the detection method, the initiation of a location detection event poses another issue. There are two fundamental mechanisms:
    - *Global Monitoring:* The mobile hosts' locations may be monitored continuously. This is only feasible in zone-based/cellular network-architectures, as described in section 2.5.2.
    - *Local Monitoring and Location Update:* Mobile hosts detect their location themselves and commit it as a location update to the entity executing the location-based service (server, another mobile host, etc.). Location updates are initiated, if certain prerequisites are given. The following enumeration highlights the most common approaches [61]:
        - *Polling:* The service-executing entity polls mobile hosts to report their positions.
        - *Periodic:* Mobile hosts commit their positions in regular intervals with the *update interval* dictating the time between two updates of each host.
        - *Distance-based:* A mobile host reports its position when the distance to the position of its last location updated exceeds a certain threshold, the *update distance.*
        - *Zone-based:* A position update is initialized by the mobile host upon entering or leaving a certain area, the *update zone.*
2. *Proximity/Separation Detection:* The change of location triggers a location-based service to be executed if certain prerequisites are given. Usually, those are defined by the presence and/or absence of the mobile host in certain areas. Detecting the mobile host entering such an area, *proximity* to this area is detected. The opposite case, when a mobile host is detected exiting such an area, triggers a *separation* event.
3. *Notification composition:* Upon detecting the necessity to execute a location-based service and to deliver information to the mobile device, the data to be delivered is composed according to the specification of both the client and the delivering entity (mobile device, server, etc.).

4. *Information Delivery:* The information is transmitted to the mobile host consuming the location-based service. Such data can be delivered on any logical channel supported by the mobile device, such as SMS, WAP-Push, MMS, etc.

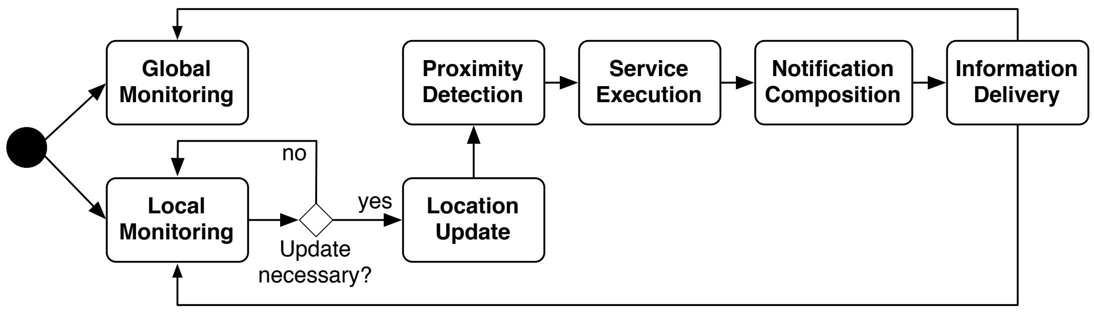


Figure 2.8: Workflow of a location-based Service Execution

Layered Architecture

Location-aware systems can generally be interpreted as middleware concepts, taking care of all location management tasks and offering high-level applications a location API. A layering reflecting this approach is depicted in figure 2.9, including the following levels [61] :

- *Positioning method:* Not being explicitly part of the middleware, this layer provides the technical position detection employing the proper technology, such as GPS, WLAN, etc.
- *Low-level position management:* The middleware's lower level is responsible for determining the current position and providing it to the higher levels in this chart. This particularly includes gathering the location data directly from the position technology by employing the update techniques discussed earlier (polling, periodic, distance-based, zone-based).
- *High-level position management:* Based on the most current location provided by the low-level position management layer, this middleware layer conducts more sophisticated analysis of the gathered locations. An excerpt is given below, such as clustering (detecting geographical clusters of mobile hosts) and proximity/separation-detection (see subsequent section 2.6). The results are provided by the *location API* on top of this layer.
- *Location-based application:* Location-aware applications and services may utilize all of the refined location-based data provided by the location API. The access is transparent hiding all of the tasks on the lower layers.

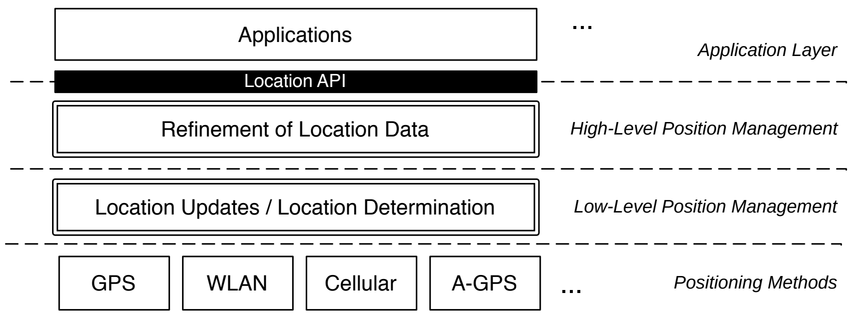

Applications
...
Application Layer
Location API
Refinement of Location Data
High-Level Position Management
Location Updates / Location Determination
Low-Level Position Management
GPS
WLAN
Cellular
A-GPS
...
Positioning Methods

Figure 2.9: Layered location-aware Middleware Concept

Location History and Location Prediction

Location history is the chronological sequence of a mobile host's locations. The main application cases of location history are to query a mobile host's location at a given time in the past or to predict its location at a given time in the future. While the former case simply requires a query from a database, the latter case requires more sophisticated calculations.

Predicting the location of mobile hosts enables location-based services to prepare their services in advance in order to provide them in the future when the target host is at its future location. Generally speaking, two approaches can be identified:

- *Route prediction:* This approach focuses on predicting the route of a moving mobile device. Its general workflow of predicting location works in three steps [54]:

  1. *Detect current location:* Location detection is processed as described in section 2.5.2.
  2. *Detect current trajectory:* A trajectory is interpreted as a mobile user's geographical path, i.e. his movement over time. In the Cartesian model, given in section 2.5.3, such a trajectory is a sequence of edges in the graph representing the map model.
     (a) A mobile host's trajectories are recorded and stored, becoming its *historic trajectories.*
     (b) To detect a mobile host's current trajectory, those are used to develop probabilistic models for future trajectory prediction. E.g., the probability of turning from a particular street to another at an intersection can be derived from the amount of such turns in the past.
     (c) The current trajectory of a mobile host is calculated from the current location, its presently covered way, and its current probabilistic model based on its historic trajectories.
  3. *Predict location:* Given a future point in time, the location is predicted by projecting the location on onto the trajectory at that given time.

- *Policy creation:* Another approach of predicting location is to create policies out of a mobile user's behavioral patterns [67]:

  1. The targeted mobile user's behavior is observed and patterns are derived out of it. Those patterns may include regular movements, regular stays at particular places, etc.
  2. Out of those *historic patterns*, dynamic policies can be created by combining them and adding probabilities of such patterns' occurrences. E.g., if a mobile user is located in room X during 3 Tuesdays in a period of 4 weeks, the probability of being at room X on Tuesday is 75%.
  3. Querying the user's position on Tuesday may result in "quite possibly in room X", given the policy, that the probability of 75% is corresponding to the output of "quite possible".

Both prediction approaches requires some kind of location history making it is an essential piece of information for location prediction.

2.5.5 Service Issues

This chapter outlines some aspects relevant for the development of location-based services. The following enumeration presents a rough overview:

- *Transparency:* Location-based services usually have transparent access to the location API provided by the location-aware middleware [87], meaning that the service itself does not have any knowledge about the location management at all. High-level location information is provided for readable access by the API, as are high-level triggers and listeners providing location events, such as proximity and separation events [61]. In summary, the location-based service is the highest level of any location-aware architecture. Thus, it is usually decoupled from the location management middleware (see section 2.4 for an analogous discussion concerning context-aware systems). This argumentation is illustrated in figure 2.9, where the location API represents this architectural cut.

- *Service initiation:* The initiation of a location-based service occurs on the application layer as drafted in figure 2.9. There are two possibilities how this may occur [61]:

  - *proactively:* The service or the user himself requests location-based information from the location API.
  - *reactively:* A trigger is activated by the location management middleware due to a location event, such as detection of proximity, separation, or the presence in an area of interest (e.g.).

- *Context-dependency:* Location-based services may rely on further contextual information besides context, implying the service to be executed in a scope of acquiring, refining and utilizing such contextual information [87].

2.5.6 Application Cases

We conclude this short survey on location awareness by presenting a selection of systems examples from this research domain.

- *Location prediction in a road network:* Karimi et al. have proposed and implemented a system to predict locations of mobile host in a Cartesian location model representing a road network [54]. It works on the basis of calculating the trajectory of a mobile device based on its historical movements and inferentially calculated movement probabilities (as described in section 2.5.4). Location prediction is used to prepare relevant information for the mobile user, where that information might be of his interest in the near future (e.g. traffic jams, etc.).
- *Hierarchical location management:* The efforts of the research group circled around Kieß et al. [57] are focused on the minimization of traffic in peer-to-peer networks by employing a hierarchical location model. The entire covered area is hierarchically partitioned into regions with cells being the smallest regions at the lowest level of the hierarchy. Each region on each hierarchy level has a responsible location-server (which is a mobile peer itself) roaming in a *responsible cell.* Responsible cells are dynamically dedicated cells including the dynamically selected location server managing a region on the lower hierarchy level. In addition, the model handles the dynamic assignment and propagation of those temporary location servers, since they are mobile peers and moving just as other peers are.
- *Location management in a P2P network:* The work presented by Satoh [87] proposes a location model for pure peer-to-peer networks. In comparison to the previous application case featuring peers functioning as temporary location servers, Satoh's approach relies on a principle omitting all kinds of centralized instances. Hierarchical location models are calculated and kept by each peer individually. Updates to the current model are derived from brief interactions with neighboring peers.

2.6 Proximity and Separation Detection

The detection of mobile entities approximating or departing each other depicts important application cases for location-based services [61]. Abstractly speaking, detecting proximity requires a subject, an object and a distance threshold. Proximity is detected if the subject is closer to the object than the distance threshold. While a subject is always represented by a mobile client, an object can be another mobile client, a physical object, a place or an area. Separation detection works analogously in the opposite direction: a subject separates from an object if having moved farther from it than the distance threshold. If separation and proximity are to be handled on the basis of different distances, we can distinguish between a proximity threshold and a separation threshold.

In the following, we drop the distinction between subject and object and generalize both to mobile entities. As with all of the services implying location, proximity and separation detection requires frequent location updates of the involved mobile entities. Based upon the location committed by the entities, proximity or separation can be detected. As depicted in section 2.5.4, there are four basic mechanisms on how to trigger a location update [61]: *polling* a mobile client to commit its location, *periodic updates* sent repeatedly at certain intervals, *distance-based* updates committed after having traveled a certain distance, and *zone-based updates* committed after entering a certain area.

A common endeavor on designing update semantics is the minimization of the amount of updates while keeping the location of the target as current as possible [20, 61, 57]. There are two efficient ways to achieve this objective in combination with detecting proximity or separation of mobile entities [61]. Both conduct the detection mechanism in 2-dimensional space following a mobile entity's location update. Both approaches work on the basis of defining geometrical regions in their area of coverage: circles and strips. Both approaches are introduced in detail in the subsequent sections 2.6.1 and 2.6.2. Independently from both approaches, the detection of proximity and separation is based on predefined proximity and separation thresholds. If distances between mobile hosts have been detected to have fallen below or have exceeded those thresholds, proximity or separation alerts are triggered, respectively.

2.6.1 Proximity Detection based on circular Zones

The first approach is based on circular areas centered around mobile hosts [61]. Every time a mobile entity commits a location update, the circle is redefined and centered around the entity's current position. The next location updates is committed by the entity upon leaving this circular area. This implies that the location-aware system remains unaware of the entity's exact location as long as the entity does not leave the circular area defined around it. However, depending on the radius of the circle, its current position is fairly approximate.

Given two mobile entities with their respectively defined circles around them, proximity and separation are detected upon calculating the smallest and largest possible distances. Both distances depend on the circles' radii, as it is visualized in figure 2.10a. For proximity detection, it is checked whether smallest possible distance is lower than the proximity threshold. Separation is detected analogously checking if the largest possible distance exceeds the separation threshold.

This zone-based approach eliminates the necessity of polling or periodic updates since updates are only committed once a mobile entity significantly changes its position. Dynamic radii and shifting circles further enhance the ratio of update amount to detection accuracy [61].

Extending those circle-based update semantics from two to an arbitrary number of entities yields no significant challenges. Every entity is concerned about its own radius of relevance when it comes to decide whether to commit an update. Concerning the location management, those semantics scale well. Given n entities, there are n circles to be regarded. The complexity is $O(n)$.

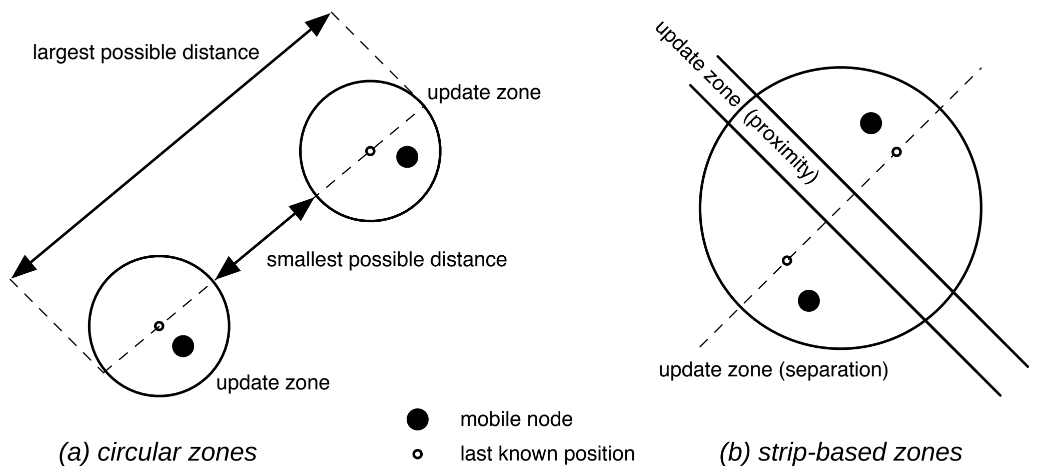


Figure 2.10: Proximity and Separation Detection

2.6.2 Strip-based Proximity Detection

The second approach focuses on defining an orthogonal strip between two mobile entities and spanning a circle around the center of the line between them, as seen in figure 2.10b. Both circle and strip are defined right after committing a location update. In contrast to the first approach, this mechanism requires each of the two entities to know the current position of the other entity. Without knowledge about the other's location, neither one would be able to define the strip.

Location updates are committed upon either entering the strip or leaving the circular zone. As stated, this method requires polling the potentially converging/separating partner entity's location. Proximity is detected upon a location update triggered by one of the entities entering the strip. After having polled the other entity's location it can be determined if it has come closer than the proximity threshold. Separation detection is conducted after a location update committed by a target leaving the circular zone. Separation occurred if the distance to the other entity (location has been polled as well) has exceeded the separation threshold. Figure 2.10b illustrates this approach.

Extending this approach from two to an arbitrary number of entities, we face the following problem. In this setting, an entity needs to define a strip in between itself and *every* other known entity [20]. Assuming that n mobile entities are registered in the system, every entity needs to define $n-1$ strips. This leads to two issues:

- The first issue concerns the number of strips known to each entity. We can observe an exponential increase of complexity when linearly increasing the entity count: $O\big(n(n-1)\big)$ with n entities calculating $n-1$ strips each. The solution is to partition the covered area. A straightforward approach is the subdivision into squared regions. In this setting, each square has usually eight neighbors (in directions N, NE, E, SE, S, SW, W, NW) and is labeled *live* if it is occupied with at least one entity. Further in this setting, each entity is only concerned about other entities in its own squares region and in the eight neighboring regions. Thus, each entity generates strips to entities located in those nine regions only. Besides the approach with squared regions, there are numerous other ideas of partitioning space (e.g. [71]), which can be employed to help managing mobile entities.

- The second problem arises out of the following observation. For each mobile entity, although the number of concerned proximate entities is reduced to those located in its own and neighboring region, the entity needs to monitor whether it enters one of its neighboring strips. Figure 2.11 shows this scenario. The neighboring strips create a bounding polygon, in which it is hard to track its own movements. While staying inside the polygon, no update is necessary. Leaving the polygon equals entering one of the strips and the necessity of committing an update. To reduce the elaborate monitoring process of checking whether leaving the bounding area, the polygon can be approximated by geometrical structures, which are much easier to monitor. E.g., Figure 2.11 shows a minimum bounding rectangle (MBR) approximating the bounding polygon. Despite generating more updates, an MBR is less elaborate to monitor.

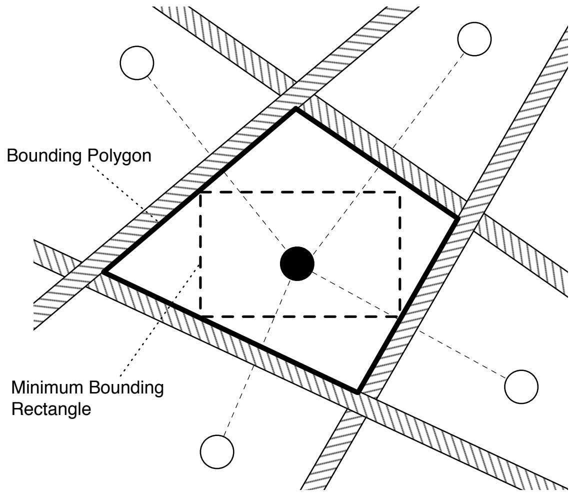


Figure 2.11: Strip-based Update Semantics with 4 Neighbors

It is to be recalled that the strip-based issues concern the proximity detection only. Separation detection is based on the circular-zoned update semantics (as discussed in section 2.6.1, which are considerably less complex. For this reason, we omit a further discussion at this point.

2.7 The Heterogeneity Issue

Context-awareness has a strong focus on mobile computing environments. In many cases, mobile devices represent context sources and hosts for context-aware applications. We will refer to mobile devices on multiple occasions when elaborating on the contextual map concept in the forthcoming chapters. However, mobile computing is characterized by a high level of heterogeneity, since there are only a few standards, which are commonly obeyed by the device manufacturers, software developers and network providers [90]. This aspect is especially reflected by heterogeneous characteristics of mobile devices, operating system, resources and network capabilities [39]. In this section, we provide a brief review about the impact of heterogeneity in mobile computing environments. It is to be regarded as a peripheral complement to the contextual map's core domains of context awareness and geographical proximity handling.

The goal of mobile computing suggests including devices spanning the entire hardware spectrum [69]. This argumentation includes the appliance of various use cases, including both the pervasive access to mobile services and ubiquitous communication between mobile hosts [90, 73]. Hence, those application cases can be reduced to the basic demand of *communication among heterogeneous devices in heterogeneous environment*. This statement can be refined as follows:

- *Communication among mobile devices:* This basic use case depicts the communication of at least two mobile hosts, both capable of roaming in correspondent networks and enabling the mobile devices' users to exploit this communication link.
- *Communication with back-end systems:* In this case, the user employs his mobile device to connect to a network's back-end system. This activity usually triggers a specific workflow on the back-end, e.g. a mobile service delivering a specific response to the requesting user.

Many research groups agree in handling heterogeneity by employing middleware solutions [39, 69, 35, 73]. The common idea behind those approaches is to position the middleware layer between the application layer at the top and the heterogeneous environment at the bottom as displayed in figure 2.12. Hence, transparent access is provided to the heterogeneous environment by masking the underlying heterogeneity.

2.7.1 Heterogeneity Abstraction

In mobile environments, the problem of heterogeneity concerns a wide range of architectural domains. A simple cut allows the abstraction of heterogeneity into three different views [39]:

- *Hardware heterogeneity:* Hardware heterogeneity reflects the presence of different devices with different capabilities, as well as different network technologies integrating those devices.

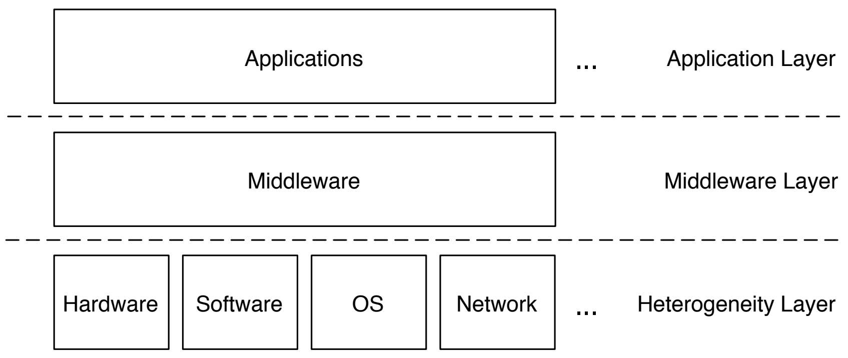


Figure 2.12: Heterogeneity-aware Middleware Design

- *Software heterogeneity:* Software heterogeneity is characterized by the presence of different applications and operating systems.
- *Architectural heterogeneity:* This heterogeneity aspect illustrates environments where network interconnections do not share any common architectural characteristics.

All of those heterogeneity aspects address the problems arising from the endeavor to achieve interoperability among different devices and systems. Interoperability may be decomposed into the following communication models:

- *Direct communication*: Mobile devices communicate directly with each other. Heterogeneity has a direct impact here, since communication among mobile devices must be based on commonly shared standards.
- *Brokered communication*: The communication link between devices is established by some sort of centralized instance. The challenge here is to make the server being able to address a heterogeneous mass of devices.
- *Unidirectional workflow activation*: A mobile user may trigger a specific workflow on a server by contacting it unidirectionally. In this case, the demand for support of heterogeneous callers applies again.
- *Service provision*: A mobile user employs his device to use services provided by a server by requesting the service and getting an appropriate response subsequently. Technically speaking, this application case describes an extension of the previously described universal workflow activation adding a response after completion of the workflow, so that the service is described by the delivery of responses following client requests. Again, the server must be able to deal with heterogeneous sets of devices.

Hardware Heterogeneity

Hardware heterogeneity simply reflects different *physical* implementations of the mobile devices' underlying technologies. Based on that, different capabilities of the particular devices can be derived. Those capabilities allow the heterogeneous set of devices to be structured accordingly. We categorize those capabilities under consideration of the following aspects of hardware heterogeneity, which are of particular interest in mobile environments:

- *Communication interfaces:* Those are the physical components, that connect the device with its surrounding, i.e. enabling access to the network [37, 24, 69]. There is an abundance of network technology available. Examples are given below:
  - *Wide Area Networks:* GSM, TETRA, 3G-networks, satellite phone networks, etc.
  - *Local Area Networks*: 802.11, etc.
  - *Piconets*: Bluetooth, Zigbee, etc.
- *UI capabilities:* User interfaces differ widely among mobile devices concerning both input (keypad, microphones, etc.) and output (displays, speakers, etc.) capabilities. Since this aspect concerns the interaction with the user, it aims at providing content to the user and gathering his commands, which can both be device-specific [91].
- *Performance:* Mobile devices are equipped with performance-constrained hardware due to limitations in the power supply and the available space. Although those restrictions apply to most of the mobile device spectrum, the level of restriction differs significantly. The reduction of performance especially concerns capabilities of computation, communication and storage [97].

Software Heterogeneity

Software heterogeneity concerns the set of *software payloads* on mobile devices. It can be decomposed into four categories:

- *Operating systems:* There are numerous operating systems running on mobile devices. They all handle their particular systems tasks differently creating a high degree of heterogeneity on the level of OS-internal architectures. Furthermore, operating system can be categorized as follows:
  - *Closed systems*: In many cases, the manufacturers of mobile devices provide a proprietary operating system on their products. Usually, those are closed, i.e. there are no APIs which might allow third-party software to run on those systems (e.g. Nokia OS, Series40).
  - *Open API:* Those operating systems provide common native APIs for third-party applications to run on them. Established representatives of such systems are SymbianOS [13] and Windows Mobile [8]. In recent years, more notable operating systems with open APIs have appeared in the mobile computing domain, such as iPhoneOS [4], Android [6] and WebOS [10].

- *Middleware APIs:* Some operating systems provide common non-operating-system APIs for third-party applications. Those APIs are designed to supplement the operating system. Hence they are shielded from the operating system's core functions making them less powerful than an operating system's native API (if existent). A prominent example for such a middleware API is J2ME [7].
- *Applications:* This abstraction criterion describes the mass of applications available for APIs provided by operating systems and middlware as stated above.
- *Application domains:* Applications can further be categorized by application domains (e.g. context awareness, mobile billing, etc.). Even though application domains differ among each other, software belonging to a particular domain often shares common principles.

Architectural Heterogeneity

Architectural heterogeneity addresses differences in any aspects focusing the architectural design of mobile computing systems, of which the following seem to matter most:

- *Network topology:* A network's static and dynamic settings can both differ greatly. Although the static architecture can only differ among complete networks, this aspect is relevant when mobile devices roam *between* such different networks. Dynamically changing network topologies even concern devices roaming *across* one network. Those issues particularly raise availability issues and imply accordingly working hand-over techniques [38].
- *Services*: Heterogeneity of services is reflected by the existence of a large spectrum of diverse services, which can be utilized by mobile devices. It especially concerns the services' protocols specifying access and result delivery [90].

2.7.2 Heterogeneity Handling

This section focuses on how to overcome the heterogeneity issues as decomposed in section 2.7.1 in order to maximize interoperability. As stated earlier, a key concept in approaching the problem is the design of middleware solutions, which are intended to make heterogeneity transparent. The following subsections discuss various concepts addressing this issue. First, techniques of abstracting heterogeneous data into proper representations are discussed. Subsequently, this information is employed to evaluate diverse operational techniques of overcoming the problems caused by heterogeneity. It is further to be noted, that all of these concepts presented in this section are *not* necessarily disjoint.

Representing Data from heterogeneous Sources

This section focuses on abstracting heterogeneous data into uniform representations in order to facilitate common *storage* of such data and *workflows* utilizing it.

Capabilities Heterogeneity defines itself by the presence of different capabilities in the same domains. In order to accomplish interoperability among heterogeneous devices, the first step is often to capture all of their capabilities and structure them into suitable representations. Those representations serve as a data basis for solutions adapting to the device heterogeneity.

There are plenty of representation techniques available, which are suitable for implementing such representations. Two examples are given below:

- *Device capability databases:* A DCDB stores the devices' capabilities and provides them to the system, which is communicating with the devices [91, 73]. It especially applies for understanding the devices' requests and providing according content to them. DCDBs are normally implemented as relational databases or using XML. A very neat example for the latter case is the open source project WURFL [79].
- *Ontologies:* A more sophisticated way of storing data is the employment of ontologies. Ontologies allow the representation of data as an interrelated set of concepts [95]. E.g., Christopoulou et al. [37] employ ontologies to describe the characteristics of heterogeneous devices. Just like the DCDBs, those ontologies solely focus on characteristics, which differ among the devices in question. However, since an ontology is usually intended to describe a device as a whole, the custom ontologies noted here may not suffice due to their sole focus on heterogeneous capabilities. Hence, to solve this issue they can be complemented by a general ontology, which describes device characteristics common to all target devices.

DCDBs and ontologies serve the same purpose, whereas ontologies may be regarded as a sophisticated refinement of DCDBs. Since both are storing information about devices, they will both be referred as *device databases* from now on.

Intermediary Representations The capability representation techniques have shown an approach how to reduce a heterogeneous spectrum of data to uniform representations, hence allowing uniform *storage* of heterogeneous data. Analogously, this principle can be applied to facilitate *workflows* of middlewares dealing with heterogeneous environments. In that case, the middleware uses uniform representations of data to allow communication between heterogeneous mobile devices among each other and any back-end system behind the middleware. Such representation represents a generalization of all supported communication semantics from the heterogeneous spectrum. It may be called *intermediary* since it is not feasible for neither any of the devices nor the middlware's back-end services. When communicating with either of those the data is dynamically adapted to the target's specification at runtime using DCDBs as described in the capabilities discussion. Concluding, intermediary representations introduce an interoperability layer in the middleware enabling heterogeneous communication through a generalizing approach, such as illustrated in figure 2.13. It is to be noted, that the middleware is not necessarily physically decoupled from the mobile device. Possible deployment option of middleware are discussed later on.

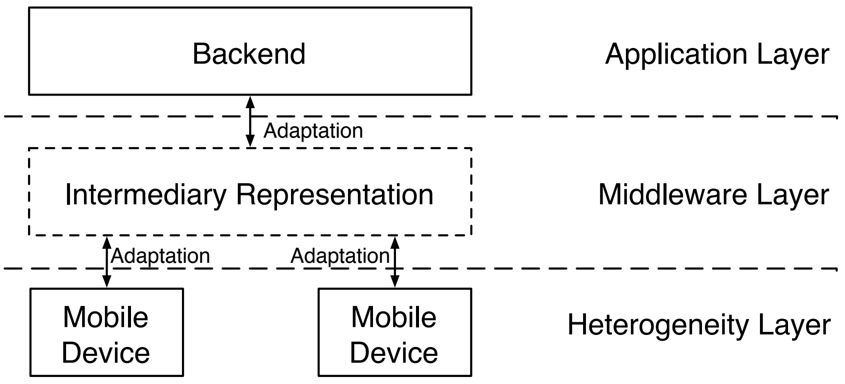


Figure 2.13: Intermediary Layer

In summary, an intermediary representation allows the middleware to utilize heterogeneity aspects uniformly at runtime. This concept can be particularly exploited for the following application domains:

- *Inter-device communication:* Intermediary representation can be used to represent all aspects relevant for the communication between devices, which differ in characteristics, protocols, etc. [73].
- *Content provision:* The utilization of intermediary representations can facilitate the provision of content to a heterogeneous spectrum of devices by first generating device-independent intermediary content, which is subsequently adapted to the receiving device [91]. This approach emphasizes the entire spectrum of defining content (structure, content, style, etc.).

Exploiting common Interfaces

One of the most basic approaches in enabling interoperability among mobile hosts is to identify their common interfaces and exploit them accordingly. Those interfaces allow the devices to be very different by hiding their individual heterogeneous characteristics behind their commonly shared interface specification and thus making themselves transparent to each other [24].

This thought is projectable on both heterogeneity domains of hardware and software, as depicted in section 2.7.1. Regarding heterogeneous hardware, this simply means that all devices communicate via the same technology and/or via the same protocols. E.g., devices from a large spectrum reaching from desktop computers to small handhelds may communicate wirelessly although being very different individually. The same principle applies for software heterogeneity, where the utilization of common APIs marks the correspondent approach. E.g., J2ME [7] or the Series60 API [11] provide uniform and widely spread interfaces for applications. The common idea behind the general approach and the applicability to the heterogeneity handling concerning both software and hardware is depicted in figure 2.14.

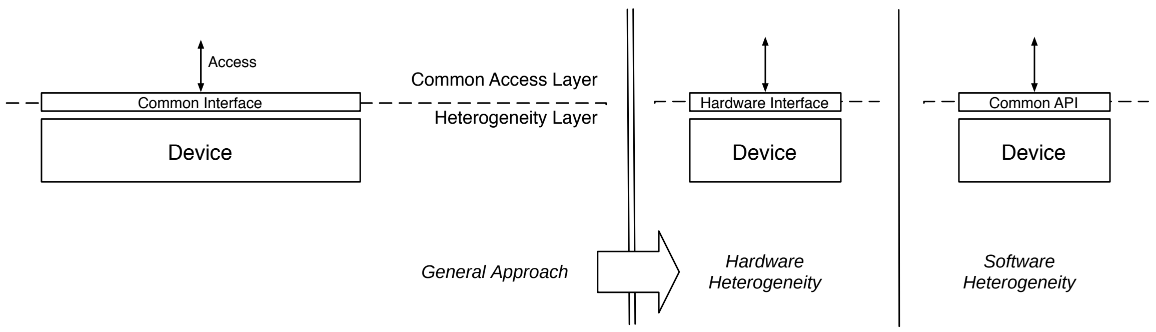


Figure 2.14: Common Interfaces

However, refinement of the common interface principle allows yet more sophisticated solutions. Common APIs on mobile devices provide a sort of *gateway* to the devices' heterogeneous subsystems to deploy more functionality. In this case, the common API is used to deploy and utilize device-specific implementations on any target device and to provide universal communication to the network. An illustrative example is the mode of operation of the Gaia Microserver, presented by Chan et al. [35]. It is intended to enable mobile devices to access a server platform for ubiquitous computing. Since the targeted devices are heterogeneous, the microserver would have been needed to be deployed as a device-specific package on each device. Instead, the J2ME middleware, which is shared among all target devices, is used to provide the desired device-independence, and thus the interoperability. A J2ME package containing *all* available device-specific implementations is deployed to the device and executed subsequently. The J2ME middleware selects and installs the appropriate specific implementation for the device. The resultant mode of operation consists of providing access to the ubiquitous computing platform in the back-end by accessing the device's native function through the device-specific implementation and by providing access to the back-end via the J2ME middleware. It is to be emphasized that this symbiotic solution between device-dependent implementations and the device-independent J2ME API is utilized by a single distribution.

Individual Adaptation

Direct communication among two heterogeneous partners requires that at least one of the communication partners adapts to the other's communication technique. This may include direct inter-device communication as well as communication between a device and a server. Extending this approach on a setting with devices implementing numerous different communication techniques requires uniform communication to satisfy the demand of general interoperability [73].

Apart from the necessity of using commonly shared hardware technology, which comes along with the direct communication approach, the concept of adapting communication to the current partner dynamically may handle heterogeneity issues in both hardware and software, as depicted in section 2.7.1. The following workflow enables this principle [91]:

1. As soon as the communication partner is identified, it is evident which communication standard is needed. Identification is conducted using a device detection mechanism exploiting a device database, which also contains the capabilities of the identified device.

2. Since the communication partner supports only one of many communication standards, an intermediary layer is inserted between the communicating entities for reasons explained previously. Incoming and outgoing communication is translated to the intermediary representation first before being committed to the communication partner or the adapting device (its back-end). The intermediary layer allows swift and scalable adaptation to any communication technique necessary.

The application cases of this principle, which is illustrated in figure 2.15, are various. The approach can be projected onto the client-server paradigm, so that a server adapts to a heterogeneous set of mobile devices, which implement heterogeneous communication techniques [91, 73]. An implementation based on a P2P architecture is imaginable, too, but raises question about facing the peers' low capabilities concerning computation and storage, with are both required for the adaptation process.

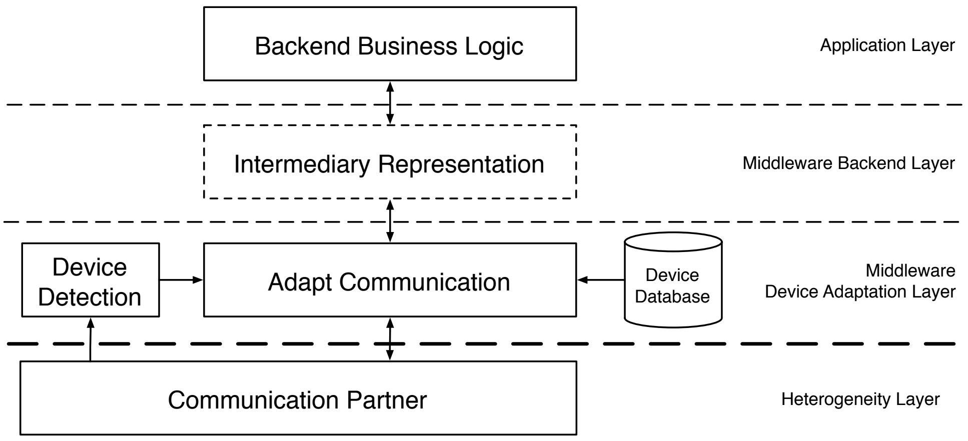


Figure 2.15: Individual Adaptation to Communication Partner

Bridges

The basic thought in the approach discussed in this section is the insertion of an entity between at least two heterogeneous systems or devices enabling the communication between those. This concept may be allegorically related to *bridges*, since they are intended to connect possibly totally different domains. Since the concept is very basic in nature, numerous instantiations are imaginable. The following discussion categorizes the general application cases:

- *Hardware bridges:* Those are dedicated stand-alone hardware devices negotiating the communication between heterogeneous communication partners. Those bridging devices provide interfaces for each of the supported heterogeneous domains. As an example, consider a laptop as a bridging device for heterogeneous sensors as depicted by Bartelt et al. [24] and Volgin et al. [96]. It is to be assumed, that those sensors may be connected to the laptop through its various interfaces (e.g. USB, PCMCIA, serial port, etc.). The laptop captures the sensory data, refines them into an appropriate format and commits them to the system's back-end wirelessly via 802.11. The conclusion of this example denotes heterogeneous data acquisition combined with a common communication mechanism, which masks the heterogeneous aspects of the data collectors.

- *Software bridges:* Such as hardware bridges provide interoperability between different hardware domains, software bridges enable the communication between different systems on the software level analogously. For this purpose, consider a *bridging framework*, as described by Nakazawa et al. [73]. Several levels are important on the abstraction layer dealing with software bridging:

  - *Transport-level bridging:* Transport-level bridging involves translation of protocols and data types inherent in the systems to be bridged. Since each system uses its own base protocols and data types for communication the bridging framework must be able to understand and translate each enabling devices from the bridged systems to communicate seamlessly with each other.
  - *Service-level bridging:* This level of bridging includes the discovery of new devices in bridged systems and offering appropriate services to them. Those services are not coercively uniform since each device may require different services and may offer services to others on its own.
  - *Device-level bridging:* Different device semantics are handled and translated by device-level bridging, including roles of devices and their compatibility across the different systems.

  Since all of the bridging levels discussed here are characterized by heterogeneous communication they are eligible for the adaptation mechanism discussed previously. They need information about how to adapt the communication to the correspondent device, requiring some sort of capability databases. Also, the translation of protocols, data types, services and device semantics require individual handling for each bridged system. In order not implement a translation technique for each pair of systems, an intermediary representation of information on each bridging level may greatly reduce the translation complexity.

Mobile Agents

The discussion so far has focused on the heterogeneity aspects concerning mobile devices' hardware and software (as depicted in section 2.7.1). A suitable way of addressing the remaining aspect of architectural heterogeneity is the employment of *mobile agents*. Those autonomous programs usually run on back-end systems (not on mobile devices) and may be used to individually address any arising heterogeneity issue. For example, consider a set of heterogeneous services, which are accessible by mobile devices. Mobile agents can be employed to handle service request for such devices by implementing the following workflow [23, 83]:

1. The mobile agent provides a uniform interface for mobile devices and accepts their service requests.
2. Subsequently, the agent queries the appropriate service and waits for its result delivery. This is the phase where the adaptation to the heterogeneous mass of service is applied.
3. Finally, the results are returned to the requesting device.

Since the adaptation to the heterogeneous service interfaces is handled by the mobile agent, mobile devices are provided transparent access to a set of heterogeneous services, thus successfully hiding the service heterogeneity from them.

2.7.3 Architectural Issues

After having decomposed the heterogeneity issue in mobile computing, and after having presented approaches to address those issues in the precedent sections 2.7.1 and 2.7.2, this section deals with aspects of realizing those approaches.

Communication Models and Middleware Application

The actual instantiation of the heterogeneity-addressing concepts are dependent on each individual application case. However, most application cases can be projected onto the four communication models depicted in section 2.7.1. Those can be further grouped into the generic communication paradigms of *client-server* and *peer-to-peer* as follows:

- *Client-server:* unidirectional workflow activation and service provision
- *Peer-to-Peer:* direct communication and brokered communication

The realization approaches of heterogeneity-handling middleware can roughly be categorized by those four communication principles.

Client-server Model Communication takes place between a mobile device denoting the client, and a server in the back-end. The heterogeneity-aware middleware may be deployed on the client [24], the server [91] or both [35] depending on the particular use cases. Those can be grouped by the communication models as follows:

- *Unidirectional workflow activation:* Since the communication flow is unidirectional in this case, the server simply needs to offer a set of common interfaces, which clients access to trigger workflows on the server. Individual adaptation to the clients' protocols is rudimentary and is limited to understanding the clients' invocations.
- *Service provision:* Basically being an extension of the previously stated communication model, the addition of a server response makes this case more complex [91]. Since the communication is bidirectional, device detection occurs so that the server is able to respond accordingly to the client. Subsequently, the requested service is performed by the server's application back-end and the results are committed to the middleware where they are stored in an intermediary format. Since the device is known, the middleware now adapts the service's results to the client's specification and returns them to the requesting device. Hence, individual adaptation must not only be used to understand the client's requests, but also to tailor device-specific responses, as illustrated earlier in figure 2.15.

  The service-provisioning infrastructure can be optimized by adhering the principles of service-oriented architecture (SOA). The SOA paradigm features a service-registry and corresponding lookup mechanisms for clients. It can easily be projected on a scenario encompassing heterogeneous devices and services with a correspondingly embedded middleware handling those heterogeneity aspects [24].

The communication models discussed here emphasize the deployment of the heterogeneity-aware middleware on the server only. However, special use cases may require deploying parts of the middleware onto the clients as well. Regarding this context it is to be noted that decoupling the middleware from the client potentially increases the range of the correspondent application since there is no need to deploy any middleware components on the devices [91].

Peer-to-Peer Model Communication occurs among multiple distributed and generally equally ranked peers. Two of our depicted communication models are applicable onto the P2P-architecture:

- *Direct communication:* Pure P2P networks are characterized by the absence of centralized components. Middleware enabling interoperability among heterogeneous peers is hence deployed on the peers themselves. Basically, this implies that communication partners always have to agree on common protocols prior to communicating with each other. With the exception of the underlying hardware requirements, the communication protocols are usually provided or complemented by the middleware. Hence individual adaptation and thus the use of intermediate representations are unnecessary since the middleware is the same on each client. This implies though, that middleware on mobile peers has a strong emphasis on the presence of common interfaces.

  This approach forces the middleware to adhere hardware restrictions, which are normally existent on mobile peers. Such are, e.g., capabilities concerning computational power, bandwidth and storage [97].

- *Brokered communication:* Brokers allow inter-device communication enabled by a centralized component, usually seen in hybrid P2P networks. The server-resident middleware negotiates the communication between heterogeneous partners [73] as it has been described in the software-bridging concept in section 2.7.2. Hence, device detection, common interfaces and individual adaptation are of significant importance.

Components

The discussion about approaches to face the heterogeneity issues allows the identification of several components relevant to each heterogeneity-aware middleware [91, 73]:

- *Device database*: A database storing all available information about devices, especially including device capabilities and semantics (as discussed in section 2.7.2).
- *Device detector*: A component dedicated to the identification of devices communicating with the middleware. It usually uses the device database for this purpose by matching the identification attribute with the corresponding entry in the database.
- *Intermediary storage*: The intermediate space is used for the device-independent representation of information. It has solely importance for the operation of the middleware itself and is intended to reduce complexity during the adaptation of information to specific devices (as discussed in section 2.7.2).
- *Translation adapters*: Those components translate device-specific information to the device-independent intermediary representation and vice versa. Abstractly speaking, there is an adapter for each device group sharing common capabilities (see section 2.7.2).
- *Common interfaces*: Middlewares offer a set of interfaces, which are commonly accessed by the heterogeneous communication partners. They are a simple approach to leverage transparent access on heterogeneous technology, as described in the common interface discussion in section 2.7.2.
- *Configuration modules*: The dynamic behavior of a middleware is dependent on their configuration. In order to keep it scalable and maintainable it is advisable to encapsulate its entire configuration in functional modules, which are dynamically loaded and interpreted at runtime. This especially applies for the following ones:
  - Templates for content and business logic
  - Intermediate syntax and semantics
  - Adaptation rules

Workflow

From the discussion of the heterogeneity handling strategies in section 2.7.1 and the derived generic components from the previous discussion, the following generic workflow can be conceptualized for a heterogeneity-aware system:

1. *Device detection*: The first step consists of identifying the heterogeneous communication partners, i.e. detecting the devices used. This is done by matching the devices' identification attribute with the corresponding entry in the device database.
2. *Interface selection*: With the device capabilities known, its interfaces are known as well. Hence, the next step consists of selecting the proper interface for the initiation of the logical communication link.
3. *Device-specific translation*: The device-specific output is translated to the system's device-independent intermediary representation and vice versa by using the corresponding translation adapters device-specifically. This translation occurs either unidirectionally or in both directions depending on the communication models depicted in section 2.7.1. E.g., for service provision only intermediary content is adapted to the target's specifications whereas direct and brokered communication both require translations both to and from intermediary representations.

2.7.4 Application Cases

This section outlines a selection of solutions implementing heterogeneity-aware middleware:

- *Deployment of device-dependent data:* At this point, the reader is reminded the common-interface-based deployment principle presented in section 2.7.2 with a device-independently utilizable gateway program enabling the operation of device-dependent implementations. The Gaia Microserver, introduced by Chan et al. [35] enables this principle by running such a program in any device's J2ME [7] sandbox. This program allows the installation of device-dependent code and enables uniform communication with the back-end system, a platform for ubiquitous computing.
- *Bridging device communication:* The bridging framework developed by Nakazawa et al. [73], which has been exemplary described in section 2.7.2 enables, is reminded to the reader at this point as well. Device capabilities and semantics are abstracted in a database, which is used to interpret incoming and adapt outgoing communication of any supported device. The framework uses an intermediary representation of all relevant data, which is adapted to the target's supported format on-demand. The bridging framework allows seamless interoperability among a wide spectrum of heterogeneous devices.
- *Dynamic content adaptation:* Rather than enabling interoperability, the idea behind [91] focuses on accurate content provision. It enables mobile devices to access

content via the internet, thus content displayable by the devices' web browsers (XHTML, WAP, cHTML, etc.). The requesting device is identified using a device database and the requested content is tailored to its specifications in two steps. First the content is generated in a device-independent intermediate format by taking the content from a static content definition and enriching it with dynamic data, which is only available at runtime. Subsequently, the second step consists of adapting the intermediate content to the requesting device using XSL-T transformation [16] and finally sending it to the requester.

- *Integration of heterogeneous mobile devices:* The system developed by Bartelt et al. [24] focuses on integrating heterogeneous devices into a SOA-based environment. It features a common communication bus based on 802.11 to which heterogeneous devices attach themselves using their individual service interfaces. Those interfaces bridge the common communication bus and the device-dependent service implementation on the device, thus achieving seamless interoperability among heterogeneous devices.

2.7.5 Summarizing Heterogeneity-Awareness

This survey has presented and evaluated techniques how to address heterogeneity aspects, which are present in mobile computing environments. From those approaches, a general architecture implementing those approaches can be derived as illustrated in figure 2.16. We have proceeded in the same manner as deriving the generalized context-aware architecture from section 2.4.2. The concept in figure 2.16 represents our view on heterogeneity-handling resulting from our survey.

The workflow to enable interoperability among heterogeneous devices and systems is basically inspired by the steps depicted earlier in section 2.7.3. First, the *device detector* identifies the partner device by matching the device's identification attribute with the corresponding entry in the *device database*. With this information available the middleware is able to negotiate communication between the *heterogeneous device* on the heterogeneity layer and the *parent application* on the application layer, as illustrated in figure 2.16. On the top, it provides a *transparent device interface* to applications, hiding all heterogeneity issues from them. On the bottom, it utilizes native hardware interfaces to connect to the communication partner. The middleware decomposes into two layers:

- *Device adaptation:* The core component on this level has been labeled the *adaptation engine*, which adapts all communication to the heterogeneous device. Each of its translation adapters is bound to a common interface present on the deployment system. It utilizes the device information from the device database to query information on how to actually adapt the communication to the partner device. For reasons of modularity, the translation rules are kept separately and interpreted at runtime. This approach allows the adaptation engine to handle heterogeneous devices through the common interfaces (bottom) and to provide a device-independent interface to the representation layer (top).

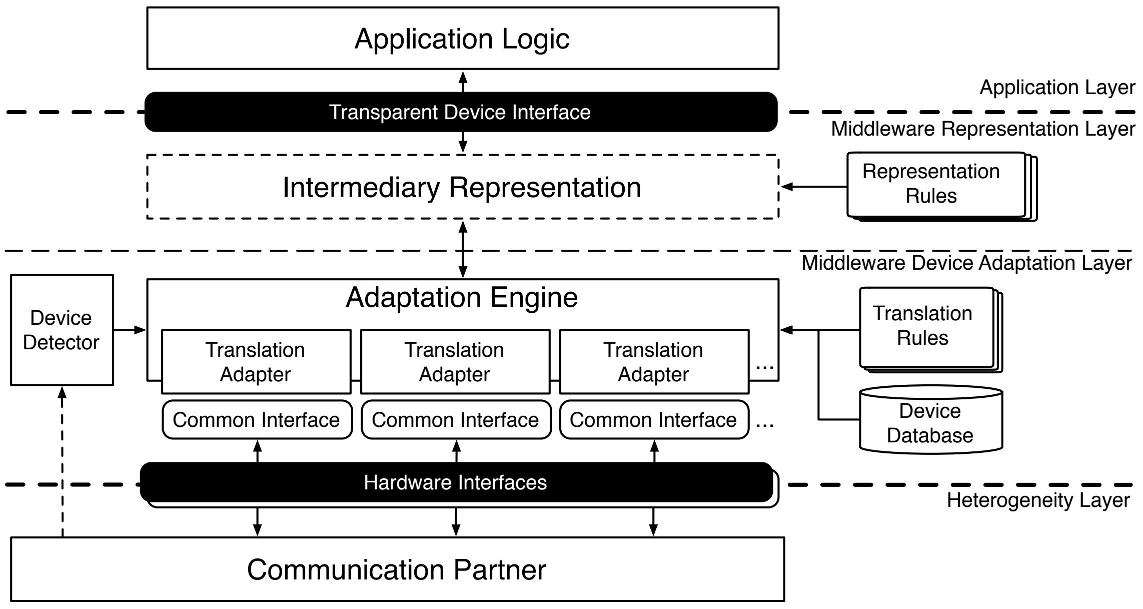


Figure 2.16: General Architecture for heterogeneity-aware Middleware

- *Data representation:* This layer's purpose is to make all information send through the middleware uniform and device-independent. It therefore uses an intermediate representation format.

The architecture presented here allows transparent communication with heterogeneous devices. The application logic depicted on top of the transparent device interface in figure 2.16 may be instantiated as applications in many domains, including the following exemplary selection:

- *Communication:* Heterogeneous devices communicate seamlessly with each other without caring for syntax, semantics or protocols.
- *Services:* Services may be offered to a wide spectrum of heterogeneous devices allowing service providers to extend the range of their service, meaning the increase of the amount of potential service users.
- *Interoperability:* Collaborative systems may include heterogeneous devices extending their range of potentially reachable devices.

2.8 Summary and Relevance

Concluding the chapter on related research, this section intends to bring all discussed research domains together and explain their relevance to the contextual map.

- *Context awareness:* The contextual map is a context model for representing and exploiting contextual information of various entity types. Hence, context awareness is the primary domain of our work. Understanding the functionality and workflows of context-aware systems enables us to integrate the contextual map into this domain. Sections 2.2, 2.3 and 2.4 have extensively surveyed this domain in regard to context definition, modeling and application. We have even depicted a placative architecture depicting the general working principles of context-aware systems in section 2.4.2 (see figure 2.5).

- *Geographical proximity detection:* This research branch of location-aware computing depicts important mechanisms that we base our contextual proximity detection on. The emphasis is put on maximizing the efficiency of those mechanisms, which apply for our use case as well. We aim at leveraging those mechanisms into multi-dimensional proximity detection applicable for our context. We have extensively surveyed the domain of location-aware computing while particularly emphasizing proximity detection in sections 2.5 and 2.6.

- *Heterogeneity:* Although not directly involved in the mechanisms for contextual similarity handling and therefore not crucial for the conceptualization of the contextual map, the heterogeneity issue has large influence on the actual realization of any context-aware system. Although the focus of this work is clearly put on theoretical concepts rather than implementation issues, we peripherally discuss this issue due to its impact on the context awareness domain. Section 2.7.1 has surveyed the domain yielding a placative architecture on generalized heterogeneity handling (see figure 2.16 in section 2.7.3).

With the background domains extensively discussed, the next chapters 3 and 4 provide an in-depth discussion on the contextual map's context model and application.

Chapter 3

The Contextual Map Model

In this section we introduce the contextual map model for storing contextual information. First of all, it needs to be clarified how the contextual map integrates into context-aware systems. For this purpose, we regard the placative context-aware architecture in figure 2.5 from section 2.4.2. The contextual map is a context model representing the situation of respective entities. In figure 2.5 this is exactly the task of the context repository. Concluding, the contextual map is a possible model for a context repository on the reasoning level of context-aware systems.

This section is structured as follows. After presenting the model's structural characteristics we focus on how contextual information is mapped into this model. We conclude this section with a survey of data models suitable for the contextual map. Further, to improve the reader's understanding, we are going to illustrate the working principle of the contextual map by employing an example, which exploits the context of a weather station.

3.1 Composition

The key idea of the contextual map model is to represent the context of entities in an n-dimensional realm. The context of a single entity corresponds to a single entry in the contextual map. We assume that the entity context is decomposed into n attributes. Thus, the entity's entry in the contextual map is composed of n coordinates and hence can be interpreted as a *position* in the map. Referring to our example, the context of our weather station denotes the current weather conditions at a specific location. Its context represents a single position in the contextual map, whereas the context of another weather station denotes another position in the map (since its context is different, i.e. it excels different location and weather conditions).

An entity's context, captured by various context sensors, is mapped into the n dimensions of the contextual map by employing a particular *mapping function*. Basically speaking, this function represents an abstract converter translating heterogeneous contextual attributes to vectors represented as points in the contextual map. We are going to sketch such a function in the subsequent section 3.2.

After mapping the information into the contextual map, the entity's current context is represented by n coordinates inside the map. For this reason and for the sake of simplicity, we call this n-dimensional point the *context* of its belonging entity. Thus, a context is made up of the values on the n dimensions of the contextual map. With that, each *dimension* represents a particular contextual attribute in the real-world context. We do not dilute the definition of context by doing so, since the n-dimensional point is a representation of an entity's context, which is semantically equivalent to any other context representation.

Analogously to the n-dimensional contextual map composition, one can imagine a 3-dimensional map of space, which is employed to represent the *location* of entities in space. We extend this 3-dimensional model by adding additional dimensions, in order to include further contextual information other than location. Finally, each dimension corresponds to an elementary contextual attribute of a complete piece of context. In the contextual map, a single dimension is the smallest unit of context representation. E.g., a weather station's temperature readings can be mapped to one dimension whereas its humidity measurement can be mapped to another. In this basic example, those two dimensions represent a 2-dimensional context map. A weather station's context is hence represented by the two current measurements of temperature and humidity in the map.

To further augment our model, we allow the contextual map to be partitioned into *contextual ranges*. A range is a subset of the map's dimensions, thus partitioning the contextual map into multiple disjoint dimension sets. Since a range groups a subset of all the map's dimensions, it actually represents a d-dimensional subspace of the contextual map (with $d < n$). Thus, ranges can be employed to group contextual information according its type. Since each dimension corresponds to a particular context attribute, a contextual range may group dimensions whose context attributes belong to a particular context type. For instance, our weather station's location is a piece of context, decomposing into three subordinate building blocks corresponding to the location's Cartesian coordinates. In the contextual map, three dimensions can be dedicated for coordinate representation, grouping them to the range R_{loc} dedicated to the representation of locations. Since the weather station's context includes more than merely its location, additional ranges must be included. E.g., an additional range R_{env} may include the station's environmental readings with a single dimension being dedicated to each sensor (e.g. temperature, humidity, etc.). Practically speaking, ranges represent "submaps" of the contextual map (corresponding to the map's according subspaces). Since they are grouping contextual attributes type-specifically, positions of contexts in the contextual map are especially expressive on range level. E.g., the representation of R_{env} shows the contextual position of the current weather. Those range-level context representations are of particular importance for the contextual map. We refer to those range-specific contextual positions as *range contexts*, depicting a set of d dimensions that represent d contextual attributes of a particular context type. Thus, range contexts represent complete contexts on a particular contextual range, which groups those d dimensions.

With this definition we have enabled contexts to be decomposed completely and disjointly into range contexts emphasizing and grouping contextual attributes on different ranges. To associate a range context with its according parent context, we introduce the *rcc*-operator (*range context* to *context*). Given a range context P_{range}, its parent context P_{full}, which is the inter-range complete context in the contextual map, can be derived by the *rcc*-operator:

$$P_{full} = rcc(P_{range}) \tag{3.1}$$

In addition to that, we also introduce the *crc*-operator (*context* to *range context*), which represents the inverted function to *rcc*. It maps a full context P_{full} to a range context P_{range} on a specified contextual range R:

$$P_{range} = crc(P_{full}, R) \tag{3.2}$$

Another addition to the model is the definition of *hierarchies* among ranges, which can be related to each other by means of specialization and generalization, resulting in tree-like hierarchies. For example, our weather station may issue a warning level denoting a single value for the current weather condition. This warning level may be assigned a range R_{alert}, which is specialized to the individual measurements of R_{env}. With this approach, we are able to inject causal dependencies into the contextual map, where range contexts on one range are dependent on range contexts of another one. Modeling such dependencies may be achieved by the definition of according rules that map values of a range's dimensions onto the dimensions of the dependent range. In summary, hierarchies serve as a structuring tool for dimensions and facilitate the process of inferring new context (see section 4.4).

The dimensions in the contextual map may be regarded as the axes of an n-dimensional Cartesian map model. Hence, all axes require well defined units of representation. The choice of an axis' quantification unit is virtually unrestricted, as long as it fits the contextual attribute, which it is supposed to represent (e.g. kilometers for location). However, within ranges, units have to be uniform spawning a d-dimensional Cartesian coordinate system for each range with d being the amount of the range's dimensions. E.g., the dimensions of the weather station's location range R_{loc} are defined in kilometers, whereas all of the environment range's dimensions R_{env} have to be defined in a uniform way representing the particular weather condition. For this example, we have chosen plain values between 0 and 100 to quantify R_{env} (see subsequent table 3.1). We will explain our motivation behind this choice upon discussing the context mapping procedure in the next section 3.2. Table 3.1 summarizes the ranges of the example introduced here.

Uniform quantification units on each range are necessary to enable efficient detection of contextual affinity (detailed discussion follows in chapter 4), i.e. contextual proximity. Since we are explicitly distinguishing between contextual *ranges* and the *entire* contextual map from this point onward, we need to clarify the depiction of dimensional count: We use d to refer to a range's number of dimension, whereas n denotes the number of dimensions of the entire context map.

The n-dimensional representation of an entity's context in the map is exploitable for further analysis. Changes of its context can alter the contextual correlation to other entities in regard to contextual boundaries. Since contextual boundaries usually define the degree of *similar context*, we can employ mechanisms known from 2- or 3-dimensional map

Range	# of dimensions	Axis quantification	Domain
R_{loc} (location)	3 (x, y, z)	Cartesian coordinates (kilometers)	unbounded: $(-\infty, \infty)$
R_{env} (environmental measurements)	4 (temperature, barometric pressure, humidity, wind speed)	plain values: 0 (min) to 100 (max)	bounded: [0, 100]
R_{alert} (alert level)	1 (alert condition)	Binary (danger, no danger)	bounded: $\{true, false\}$

Table 3.1: Example Ranges

models to detect proximity among contexts inside the contextual map, hence identifying contextual affinities by identifying *proximate context*, as explained later on in chapter 4.

3.2 Context Mapping

Based on the context model introduced in the previous section 3.1, it remains to be clarified how contextual information is mapped into the contextual map, i.e. how data acquired from context sources are mapped to the respective dimensions on the context map. At this point, figure 3.2 from the introductory section 1.2 is recalled. Back then, we have already sketched the mapping process. The figure shows that entity contexts are mapped into a multi-dimensional map model - the contextual map. This depiction of the mapping process can be extended to four steps as depicted in figure 3.1 (based on figure 1.1:

1. *Typification of context:* As stated in section 3.1, context needs to be typified in order to be processed, i.e. contextual information needs to be grouped according to type.

2. *Identification of attributes:* For each context type, distinct attributes are identified. All together, the context attributes depict the context structure.

3. *Quantification of attributes:* The identified attributes are each quantified by a scalar value according to the entity's situation. All together, the quantified context attributes depict the current entity situation, i.e. its context.

4. *Attribute mapping:* The quantified attributes are mapped into the contextual map's dimensions according to specific arithmetic rules.

The last step denotes the most significant step of all: How to map the attribute values into the contextual map? Obviously, a 1:1 mapping is rarely feasible since the attribute

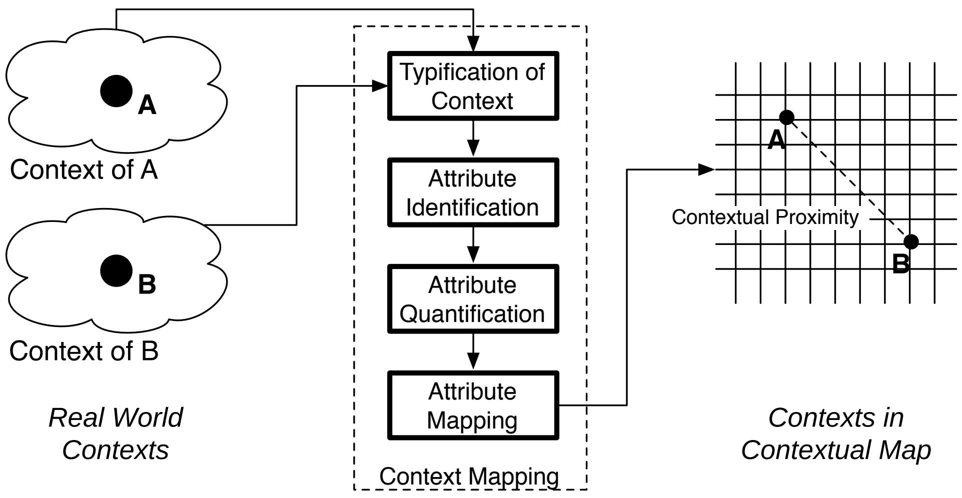


Figure 3.1: Context Mapping Steps

values are not uniform, but the dimensional quantification requires them to be uniform (see previous section 3.1 on quantifying dimensional units). The basic approach to tackle the problem is to employ a *mapping function*, which inputs quantified contextual data from according context sources and maps them to Cartesian coordinates of the diverse ranges in the contextual map. Formally, given an entity, its context is mapped to the contextual map as follows:

$$f_{mapping} : s \rightarrow \begin{pmatrix} v_1 \\ v_2 \\ ... \\ v_n \end{pmatrix} | s \in S, \vec{v} \in V_n \tag{3.3}$$

where S is the set of quantified context source data and V_n the correspondent n-dimensional realm in the contextual map. $\vec{v}$ represents a single context in the contextual map with $\vec{v}$ corresponding to the entity's real-world context. Due to the fragmentation of V_n into ranges (as introduced in section 3.1), the definition of $f_{mapping}$ is basically comprised of mapping individual contextual data to the correspondent ranges in V_n. It is to be noted, that S may cover a very heterogeneous spectrum of contextual sources (see section 2.7) implying heterogeneous definitions of $f_{mapping}$ as well. $f_{mapping}$ inputs the quantified context data from the context capturing interface (see figure 2.5) and transforms it to its correspondent Cartesian n-dimensional representation to be stored in the context map. We are about to depict a general workflow of that mapping process keeping in mind that heterogeneous application cases require to adapt the mapping process case-specifically.

At first, $f_{mapping}$ inputs well-defined context data which has been gathered from context sources throughout the system. Figure 3.2 illustrates this approach, where a context sensor is attached to each source delivering the raw readings to the context capturing interface, which corrects and quantifies those readings into well-defined context data. At

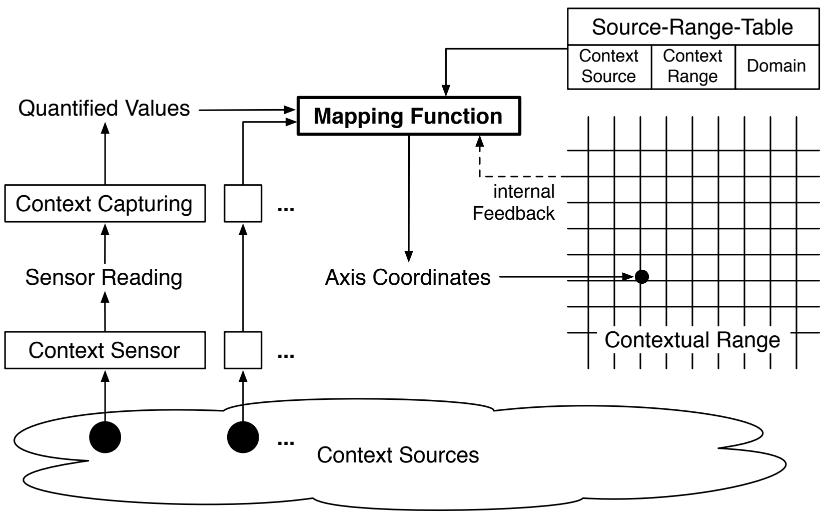


Figure 3.2: Context Mapping Procedure

this, $f_{mapping}$ distinguishes between the various ranges it needs to perform the mapping to, and thus, $f_{mapping}$ may consider multiple context sources to provide data for a single range as depicted in figure 3.2. For this reason, we need to determine the context sources' influential aspects on $f_{mapping}$. To determine which context sources are relevant for what range, we conduct a 2-level typification on context sources. First, we typify a context source according its origin:

- *Environment:* Contextual information is gathered from the environment, mainly by physical sensors attached to a mobile device associated with the entity, which the final context belongs to. E.g., a GPS sensor embedded in a mobile device reads the current position in the environment. Further, in our running weather station example, the environmental sensors gathering temperature, barometric pressure, etc. provide context data of this dedicated type.
- *Entity (e.g. user):* This context type denotes information coming directly from the entity. Given the circumstance that the entity is a physical user, this context source type depicts the input given by the user.
- *Databases:* Contextual information may also come from data stored in a system's back-end. Dynamic data, such as user subscriptions, addresses, etc. represent context data belonging to this context type.
- *Infrastructure:* The infrastructure of a context-aware system may provide valuable contextual information, too. E.g., in a GSM-network, the location of a mobile host known to the network (HLR-entry) may be provided by the infrastructure.
- *Devices:* Properties of mobile devices themselves comprise contextual information of these types. It included both hardware and software characteristics (e.g. operating system version, screen resolution, HTTP-user-agent tag, etc.) [91].

With the first level of typification defined, the second typification step distinguishes context sources according to their contextual characteristics. Context sources, which belong to the same contextual domain, are grouped to a contextual type. E.g., the environmental sensors in our weather station example can be grouped to sources providing data of one context type, namely the weather. Generally speaking, context types can be mapped bijectively to ranges, since ranges denote context consisting of similar contextual attributes, which context types actually represent. Hence, multiple context sources of the same context type may be mapped to the dimensions of the same contextual range (e.g. temperature reading may be mapped to the temperature dimension of R_{env}).

With the context sources typified, we introduce a data structure associating context sources to corresponding ranges. We call it the *Context-Range-Table (CRT)*, which possesses the following columns:

- *Context Sources* specify a list of context sources, from which a range gathers its information.
- *Context Range* depicts the range in question.
- *Range Domain:* defines the bounds of the range quantification. There are three options to this aspect:
  - *Unbounded:* The range is unbounded, hence spanning a domain ranging to infinity bidirectionally: $(-\infty, \infty)$. A point of origin ("0") is needed to fix the range quantification.
  - *Semi-bounded:* The range is bounded at its quantification origin only, spanning an unidirectionally bounded domain: $(0, \infty)$.
  - *Bounded:* The range is bounded bidirectionally specifying a predefined minimum and maximum: (min, max).

With the contextual data captured and quantified by the context capturing interface (as seen in figures 2.5 and 3.2), and with the context sources related to ranges, $f_{mapping}$ may be defined more specifically. Assume that the quantification yields vectors and scalars. With the CRT given, it is known which of those values are to be mapped to which range. Hence, it remains to be discussed how the mapping is processed generally. Scalars are mapped straightforwardly to the correspondent range dimension considering the relation between real world context and the target dimension inside the contextual range (e.g. temperature to temperature dimension of R_{env}). Mapping of vectors, however, leaves us two options:

- *Direct mapping:* The dimensions of the source vector are each mapped to one dimension in the contextual map.

$$f_{direct} : \begin{pmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{pmatrix} \rightarrow \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{pmatrix} | \vec{s} \in S, \vec{v} \in V_n \tag{3.4}$$

- *Arithmetic projection:* The vector's dimensions are mapped to less dimensions in the contextual map. More precisely: i dimensions of a vector are mapped to j dimension in the contextual map, with $i < j$. The most common variant of this approach isto map all of the vector's dimensions to a single one in the map.

$$f_{projective} : \begin{pmatrix} s_1 \\ s_2 \\ \dots \\ s_m \end{pmatrix} \rightarrow \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{pmatrix} | \vec{s} \in S, \vec{v} \in V_n, m > n \tag{3.5}$$

For reasons of completeness, it is to be noted, that both approaches are not mutually exclusive, hence allowing hybrid approaches as well.

From here on we proceed with a mapping technique complying with the general mechanism introduced above. We assume that the context of an entity is typified and hence given by a set of quantified scalars and vectors, each corresponding to its according type of context. Given our weather station example, we demonstrate this mapping procedure. We first define a representation for the station's context in the contextual map model. Subsequently, we define a mapping function, which defines how contextual data from context sensors is mapped into that map. We proceed as follows. First, we define three context types: location, environmental data, and the alert level. We assign each type a range with an according amount of dimensions in the contextual map. Consequently, we assign suitable quantification units to the dimensions of each range. Table 3.1 in the previous section 3.1 summarizes this setting. At this point, it is to be reminded that ranges roughly represent "submaps" with altogether comprising the entire contextual map.

With this setting, it is possible to individually define an example mapping function $f_{mapping}$, which transfers the weather station's context into the contextual map. First of all, we require $f_{mapping}$ to handle each context type and its corresponding range separately. $f_{mapping}$ takes the quantified contextual data captured and derived from the weather station's context and calculates its position in the contextual map adhering both range and axis definitions. Mapping location into the map's range R_{loc} is quite straight forward: The station's coordinates are translated to Cartesian coordinates in the map[1]. The coordinates for R_{env} are calculated by determining the relation of the current value to the pre-defined extremes. We have set the extremes as 0 and 100 denoting the minimum and maximum, respectively (see table 3.1). Let temperature to be read when adopting a value between -50°C and 50°C resulting in 100 adoptable units. With the axes of R_{env}'s dimensions ranging from 0 to 100, a currently measured temperature of 25°C corresponds to a coordinate of 75 for the temperature dimension. The mapping of the remaining environmental measurements works analogously. Mapping the station's currently issued alert level simply consists of assigning it the correspondent binary value in the map, for instance $false$ for normal weather conditions and $true$ for danger. Formally, we have assigned the following *domains* to our ranges:

- $R_{loc} = (-\infty, \infty)$

[1]note, that a coordinate origin (0, 0, 0) needs to be defined

- $R_{env} = [0, 100]$
- $R_{alert} = \{true, false\}$

Now, we can further formalize $f_{mapping}$. Given a context source i with its source ID (SID_i) providing a vector[2] of quantified values $\vec{s_i}$, $f_{mapping}$ maps this input to the according dimensions of the contextual map. The resultant map context gets assigned a value v_j on each affected dimension dim_j. The following definition is further generalized to an arbitrary amount of context sources:

$$f_{mapping} : (SID_i, \vec{s_i})^+ \rightarrow (dim_j, v_j)^+ \mid \vec{s} \in S, v \in V_n \tag{3.6}$$

For our example, this definition yields a custom setting of $f_{mapping}$. First, we identify the variables depicted in equation 3.6 before deriving a custom mapping function for our weather station use case:

- $SID_i = \big\{location\_sensor, temperature\_sensor, humitdity\_sensor, \ldots\big\}$ (for reasons of simplicity we restrain to regarding only three of five context sources as depicted in table 3.1)
- $\vec{s}_{loc}$, s_{temp} and s_{hum} represent quantified context data corresponding to their according SID
  - $\vec{s}_{loc}$ is given by two DMS-coordinates[3] s_{lat} and s_{long}, and the altitude s_{alt} in meters.
  - s_{temp} is given by a temperature value in °C.
  - s_{hum} is given by the ratio of partial pressure of water to vapor pressure of water, hence $s_{hum} < 1$.
- $dim_i(R)$ identifies the dimension i, which is in range R.
- x, y, z, $temp$ and hum represent contextual coordinates in the context map.
  - x, y and z represent Cartesian location coordinates on R_{loc}, measured in kilometers. We set the point of origin at the geographical crossing of the equator and the zero meridian (which is a few hundred kilometers off the coast of Ghana in the South Atlantic). Since s_{lat} and s_{long} are given in DMS-coordinates, we transfer them to the Cartesian coordinates x and y expressing the Cartesian pendant to latitudes and longitudes, respectively. This is achieved by transforming degrees ($s(deg)$), minutes ($s(min)$) and seconds ($s(sec)$) to kilometers in decimal notation. One latitude degree corresponds to about 111.3 kilometers of distance equidistantly whereas one longitude degree corresponds to the same kilometer value only on the equator, where it is maximal. The distance

[2]this explicitly includes scalars, which have the form of 1-dimensional vectors

[3]Degrees, Minutes, Seconds - corresponding to latitude and longitude

corresponding to longitudes decreases with increasing proximity to the poles. For this reason, that distance is dependent on the cosine of the local latitude. With x and y defined the distance from the coordinate origin corresponding to latitude and longitude, respectively, z remains to be defined. Since s_{alt} is given in meters, it is simply converted to kilometers. See the subsequent equation 3.7 for a formal definition of the mapping depicted above.

– *temp* and *hum* are unit-less Cartesian coordinates on R_{env} representing *temp*erature and *hum*idity, respectively. Since s_{temp} is expected to be between $-50°C$ and $50°C$, it is mapped accordingly to *temp* in R_{env}'s domain $[0, 100]$. Also, s_{hum} is mapped accordingly to *hum* in that domain, keeping in mind that $s_{hum} < 1$ always holds.

With those prerequisites, $f_{mapping}$ may be defined as follows:

$$f_{mapping}: \begin{cases} \Big(location\_sensor, \begin{pmatrix} s_{lat} \\ s_{long} \\ s_{alt} \end{pmatrix}\Big) & \to \begin{array}{l} \Big\{ \Big(dim_x(R_{loc}), x\Big), \Big(dim_y(R_{loc}), y\Big), \Big(dim_z(R_{loc}), z\Big) \Bigm| \\ x = 111.3 * \Big(s_{lat}(deg) \pm \frac{s_{lat}(min)}{60} \pm \frac{s_{lat}(sec)}{3600}\Big) \\ y = 111.3 * cos(s_{lat})\Big(s_{long}(deg) \pm \frac{s_{long}(min)}{60} \pm \frac{s_{long}(sec)}{3600}\Big) \\ z = s_{alt}/1000 \Big\} \end{array} \\ \Big(temperature\_sensor, s_{temp}\Big) & \to \Big(dim_{temp}(R_{env}), temp\Big) \Bigm| temp = (s_{temp} + 50°C)\frac{1}{°C} \\ \Big(humidity\_sensor, s_{hum}\Big) & \to \Big(dim_{hum}(R_{env}), hum\Big) \Bigm| hum = s_{hum} * 100 \\ \ldots \end{cases} \tag{3.7}$$

with corresponding depictions for Cartesian location coordinates, temperature and humidity. For reasons of simplicity, the definition in equation 3.7 represents an excerpt of $f_{mapping}$ lacking the complete mapping rules for the environmental sensor sources.

The mapping function presented here covers a special use case, and must not be seen as universally valid. The heterogeneous spectrum of context-aware application cases requires $f_{mapping}$ to be adapted individually depending on the required application cases.

Since the context of entities is dynamic, the mapping must be employed iteratively to ensure the most current context to be stored in the contextual map. Such context actuality is achieved by frequent contextual updates. The according update semantics are discussed later on in section 4.2.

3.3 Data Model

In this section, we briefly present the data structures suitable for the contextual map. In doing so we revert to proven indexing techniques for n-dimensional data sets. The following survey explicitly emphasizes the "curse of dimensionality", which refers to the exponential degradation of performance with a linear increase of dimensions when querying or updating multidimensional indexing structures.

3.3.1 Indexing Structures in multi-dimensional Space

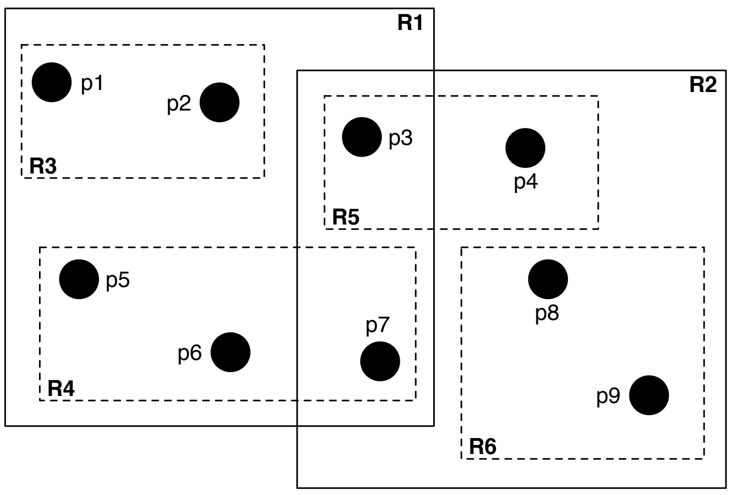
The task we are facing is to store an arbitrary amount of contexts into a suitable data structure. Since every context is represented as an n-dimensional vector in the map, we have surveyed several index-trees, which are capable of storing and - even more importantly - efficiently querying those vectors. Our focus is put on R-trees, kd-trees and its derivatives, as surveyed in [101].

R-trees and its Derivates

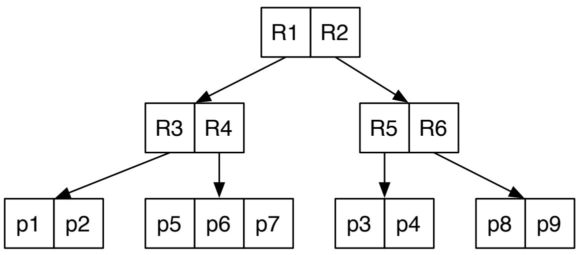
This tree-based data structure represents a multidimensional representation of B-trees [66]. Thus, they are height-balanced and ordered providing sophisticated and efficient algorithms for updating and querying. Objects in n-dimensional space are represented in an R-tree's leaf nodes. A leaf node's entry consists of a definition of a hyper-rectangular minimum bounding rectangle (MBR) encompassing the object. A non-leaf node in an R-tree consists of multiple entries of which each denotes an MBR definition and a child-pointer to a child node. Thus, each child node has a correspondent entry in its parent node. Further, all entries in a child node denote spatial definitions (MBRs) *inside* the parent entry's MBR. Practically speaking, a non-leaf node's entry bounds the space of all its corresponding child branch. Hence, the space defined by a non-leaf-node's MBRs encompasses all space defined by its children nodes. Figure 3.3 shows an example R-tree definition. It represents *points* instead of spatial objects since we regard the contextual map's contexts, which are denoted by points in $\mathbb{R}^n$. However, since points are spatial objects with size 0 in $\mathbb{R}^n$, there is no mentionable difference between R-tree representation of spatial objects and points.

There have been numerous extensions and variants of R-trees. The following list briefly surveys the most common ones:

- *R+ trees and R* trees* focus on minimizing the overlap of bounding boxes (as in figure 3.3 R1 and R2 overlap with points p3 and p7) by employing more sophisticated updating algorithms but preserving the R-tree structure as described [25, 93].
- *X-trees* represent a variant introducing an additional node type - the super node - to overcome the phenomenon of exponentially degrading performance with linear increase of dimensions ("dimensionality curse") [28].
- The *SS-tree* (*S*imilarity *S*earch tree) has a similar structure to that of an R-tree, but instead of bounding boxes (MBRs) it uses bounding spheres [98]. Since the bounding spheres are defined by a radius, they are insensitive against changes of dimensionality. MBRs, on the other hand, are bound by two n-dimensional points. An increase of n implies an exponential increase in coordinates necessary to define an MBR.
- The *SR-tree* (Sphere/Rectangle-tree) [55] represents an improvement of the SS-tree. Bounding spheres as used by the SS-tree occupy a much larger region of space than rectangles do, which leads to performance drawbacks. The SR-tree combines



(a) Representation in 2-space



(b) Representation R-tree

Figure 3.3: R-tree Representation of Points in $\mathbb{R}^2$

spherical and rectangular bounding structures allowing space to be partitioned in considerably smaller regions than the SS-tree.

kd-trees

A slightly different approach in indexing points and objects is given by the k-dimensional-tree, also abbreviated as *kd-tree* [101]. k-dimensional space is partitioned completely disjointly. Each level of the tree is designated to disjointly split a dimension of $\mathbb{R}^k$. In contrast to the R-tree, all of the kd-tree's nodes represent points in $\mathbb{R}^k$. Each point is used to align a splitting plane on the correspondent tree-level and dimension, respectively. Further, kd-trees are binary and ordered. Hence, given a node N in a kd-tree denoting a point P and splitting dimension d, N's left neighbor denotes a point with a coordinate on dimension d lighter than that of P. N's right neighbor denotes a point with a d-coordinate greater than that of P. Figure 3.4 shows an example kd-tree in $\mathbb{R}^2$.

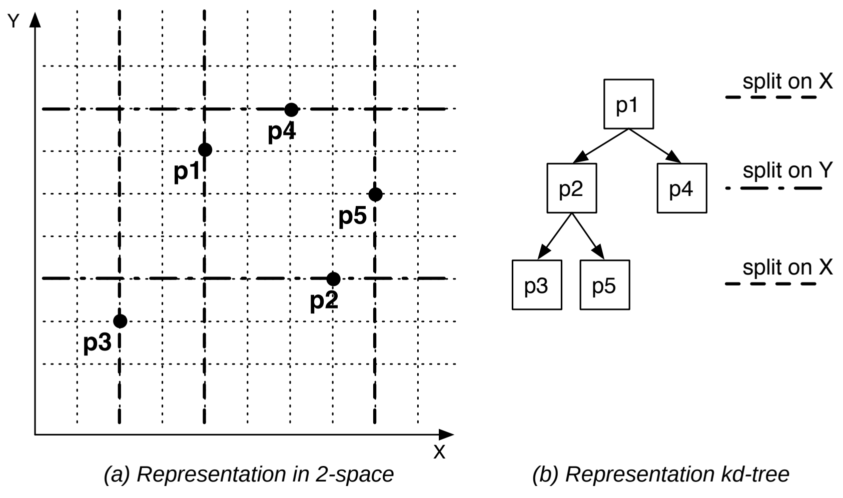


Figure 3.4: kd-tree Representation of Points in $\mathbb{R}^2$

Hybrid-trees

The authors of [34] present a rather unconventional approach. They partition existing indexing structures into two disjoint techniques: pure data partitioning (DP) index structures (e.g. R-tree, SS-tree, SR-tree) or pure space partitioning (SP) ones (e.g. kd-tree). They identify relevant aspects to tackle the high dimensional indexing problem from both techniques. Finally, the positive aspects of both index structure types are combined yielding the hybrid tree. The basic approach is to take a kd-tree as a basis and extent it by adding a corresponding bounding region to each its nodes. So the hybrid tree is actually a SP-based kd-tree is extended by the DP-based region bounding principle. By defining a mapping from the kd-tree based representation to an array of bounding regions, DP-based search, insertion and deletion algorithms are applied directly to the hybrid tree.

According to the authors it scales well even at very high dimensionality and significantly outperforms the other index structures discussed here. Presented in 1999, the hybrid tree is the most recent data structure in our survey.

3.3.2 Optimization Principles

Multi-dimensional indexing yields a problematic phenomenon when dimensionality increases. A linear increase of the number of dimensions yields an exponential increase in processing cost concerning the execution of queries and updates upon an index. Considerable research has been conducted on breaking this ”curse of dimensionality”. The two basic principles are listed below [101]:

- *Dimensionality reduction:* Neglecting dimensions representing less important data allows the construction of indexes on reduced space. Although querying performance is enhanced by this step, the result sets of queries may contain false positives, i.e. points which match criteria in the reduced space, but *not* in the source space with its full number of dimensions. The criteria of picking important dimensions and neglecting others are manifold. E.g., an illustrative criterion is the focus on dimensions modeling important attributes.

- *Filtering and refining:* This 2-step querying approach focuses on quickly determining a subset from which is known that it contains all results (filtering) and subsequently examining this subset for points complying with the query (refining). The filtering step is focused on speed rather than accuracy, so even that all points of the final result are included, the existence of false positives cannot be excluded. The refining step remedies this situation. Since being applied on a small subset only, it scales well. Lots of different criteria for both filtering and refining exist here, too. An illustrative example approach is the partitioning of space. First, all spatial parts concerned by a query are identified. Second, all points inside those concerned parts are checked on compliance with the query.

3.3.3 Contextual Map Indexing

With a brief survey on multi-dimensional indexing given, it remains to evaluate it against the applicability on the contextual map model. We are facing an n-dimensional data space, where n may be considerably high. Since each dimension represents a little contextual attribute of the real world, which is mapped into the contextual map, many contextual attributes imply a large quantity of dimensions and automatically the issues coming with the "dimensionality curse".

The first straight-forward approach to tackle this issue is to break up the representation of contexts range-specifically. As we have argued earlier in section 3.1, ranges represent context types. Context types, which are not related in the real world cause the correspondent ranges not to be related in the contextual map, neither. Thus, separating ranges and building an index for each range results in the following benefit. Logically, d-dimensional contextual ranges have less dimensions than the entire n-dimensional contextual map. In addition, $d \ll n$ usually holds. With only a few d dimensions indexed for each range (d varies range-dependently), those indexes are efficient. Due to the exponential cost growth implied by the increase of dimensionality, many less-dimensional indexes are more efficient than a single n-dimensional index. Regarding the weather station example from the previous sections, the location and weather ranges are indexed separately and expected to work more efficiently than an index covering both.

Concerning the choice of an adequate indexing structure from the survey in section 3.3.1, we have made the following evaluation in regard to the contextual map model:

- *R+ tree and R* tree:* Despite the efforts on minimizing overlap, those R-tree improvements do not scale well when it comes to very high dimensionality. However, since indexes are built range-specifically, a considerably lower number of dimensions is indexed individually. Thus, R-trees may offer a good trade-off between efficiency and simplicity.

- *X-tree & Hybrid tree:* Those trees are designed for very high dimensionality. For ranges with many dimensions that have to be covered by an index, X-trees may offer a reasonable choice.
- *SS-tree & SR-tree:* The spherical nature of space bounding discloses advantages in contextual proximity detection. Since contextual proximity is distance based, it manifests itself spherically in the contextual map (see subsequent chapter 4). Given a context in the map, affine contexts are situated in its spherical proximity. Hence, the SS-tree, and thus its outperforming cousin, the SR-tree, may be a reasonable choice for indexing contexts.
- *kd-tree:* Despite the lack of spatial overlap implied by the structure of kd-trees, we have not identified any advantages concerning the contextual map. kd-trees are behaving especially well in nearest-neighbor searches, which have only peripheral relevance for the contextual map (see subsequent section 4.4.2).

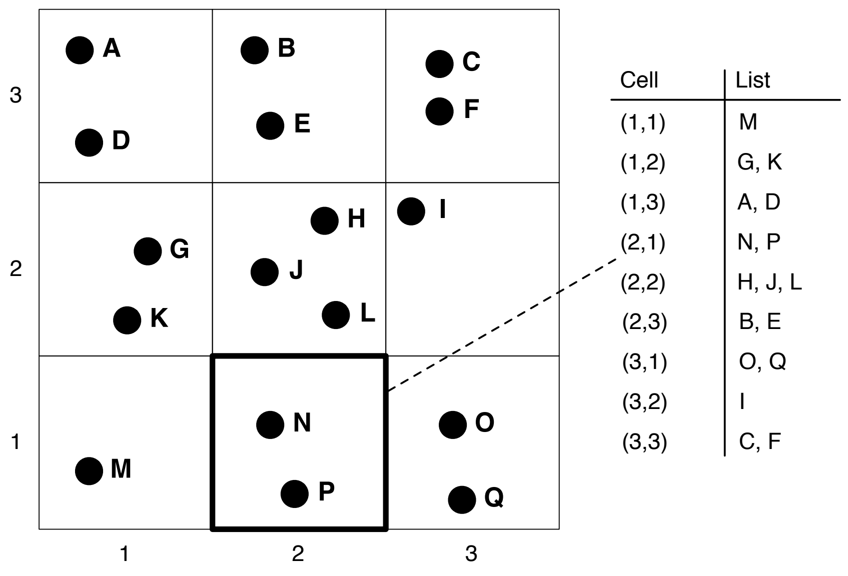
Table 3.2 summarizes the suitability of index structures for the contextual map.

Index structure	Practical characteristics	Relevance to Contextual map
R*tree, R+ tree	simple & efficient	for simple context ranges
X-tree	scales well with high dimensionality	high dimensional ranges
SS-tree / SR-tree	spherical data model	context proximity monitoring
Hybrid tree	advanced, most recent index structure in the survey (1999), scales well with high dimensionality	high dimensional ranges
kd-tree	disjoint space partitioning	none

Table 3.2: Index Structures

3.3.4 Space-partitioning Grids

An alternative to tree-based indexing is to partition a contextual range's data space into a grid of equally sized subspaces, so called *cells*. Each of those cells represents a d-dimensional hypercube. Hence, each range context is positioned uniquely in one cell. Following this argumentation, we can derive lists of included range contexts for each cell as illustrated in figure 3.5. However, since no other structuring criterion applies on those lists, the set of those included range contexts can be indexed using one of the indexes presented in section 3.3.1.



Cell	List
(1,1)	M
(1,2)	G, K
(1,3)	A, D
(2,1)	N, P
(2,2)	H, J, L
(2,3)	B, E
(3,1)	O, Q
(3,2)	I
(3,3)	C, F

Figure 3.5: Contextual Range Grid

Querying a range context in the spatial grid basically means determining its current parent call. Updating adds the necessity to determine its cells before and after the update, respectively, and adjust those cells' list structure (obviously skipped if range context stays within its cell). The additional indexing of those lists can obviously speed up the access on the spatial grid.

However, the scalability issues concerning the dimension count also apply for the grid-based space-partitioning. With an increasing number of dimensions, it becomes increasingly cumbersome to navigate in the grid. The choice whether to employ tree-based indexes or spatial grids for multi-dimensional storage depends on the use case. Research indicates that both data structures scale differently and specifically. To answer what works best for the contextual map, we will return to this issue in chapter 4 when discussing the applicability of the contextual map.

We can interpret the denoted spatial grid as a basic index for range contexts, such as the sophisticated trees form the according range context indexes for the tree-based techniques discussed in section 3.3.1. Concluding, both the spatial grid and tree-based structures are suitable structures to index contextual ranges, thus allowing fast access to the affected range contexts.

Chapter 4

Application of the Contextual Map

So far, we have solely depicted the contextual map model in the previous chapter 3. In this section we discuss its applicability. We assume a setting where autonomous entities issue updates committing their individual *local* contextual information to a central contextual map. The sum of all contexts, which the contextual map holds, represents the *global* context. Figure 4.1 visualizes this argumentation.

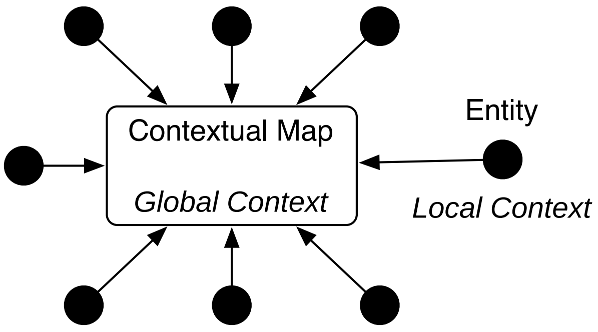


Figure 4.1: Global and local Context

According to changes in the global context contextual affinities between entities are monitored and exploited. This section focuses on the basic working principles of the contextual map handling contextual updates and identifying contextual proximity:

1. First, we discuss how to define *affinity among contextual information* and derive the definition of context boundaries, which form the basis for enabling triggers based on contextual proximity/affinity. With context boundaries defined, the principles of measuring the proximity of contexts are introduced, hence allowing to determine the affinity between those contexts (section 4.1).

2. Since the determination of contextual affinity requires current contextual information, we subsequently reflect on efficient *update semantics* for entities providing contextual updates. Those update semantics aim at maximizing the up-to-dateness of the global context despite of a minimized amount of updates (section 4.2).

3. After having determined *when* to commit an update, we discuss *what* to include into the update before committing it. Section 4.3 discusses the procedure of *update composition* by selecting appropriate data to be included in the update.

4. We proceed by presenting strategies to only apply *contextual affinity checks* on contexts, which have actually been affected by an update (section 4.4). For this purpose, we reflect on identifying contexts in the surrounding of an updated contexts, i.e. contexts that are close to the updated one, hence narrowing the set of contexts to those actually affected by an update.

5. After having introduced the basic contextual proximity handling mechanisms for a single contextual update, we extend those mechanisms to be applied on multiple updates, thus enabling *monitoring contextual affinities* over time (section 4.5).

6. We close this chapter by explaining *contextual boundaries management* in a monitored dynamically changing contextual map (section 4.6) and summarizing the overall *workflow* of identifying contextual proximity among entities (section 4.7).

Figure 4.2 visualizes the relevant aspects from the enumeration above in relation to a typical contextual proximity detection cycle. It illustrates the core functionality of the contextual map by sketching how contextual proximity is detected. In the remainder of this chapter, we will refer to exactly this sketch in figure 4.2 when discussing the individual aspects from the enumeration above.

4.1 Contextual Affinities

At this point, it is to be emphasized that meanings of contextual similarity, contextual affinity and contextual proximity are very similar, if not equal.

- *Contextual similarity* expresses entity contexts to be similar to each other.
- *Contextual affinity* expresses the same notion with emphasis on contexts being affine to each other.
- *Contextual proximity* emphasizes the similarity notion is regard to the contextual map in which contexts are *proximate* to each other in regard to the Euclidean distance between each other.

In this section a metric for monitoring contextual proximity is introduced. We present the concept of contextual boundaries, which represent the base for further applications of the contextual map.

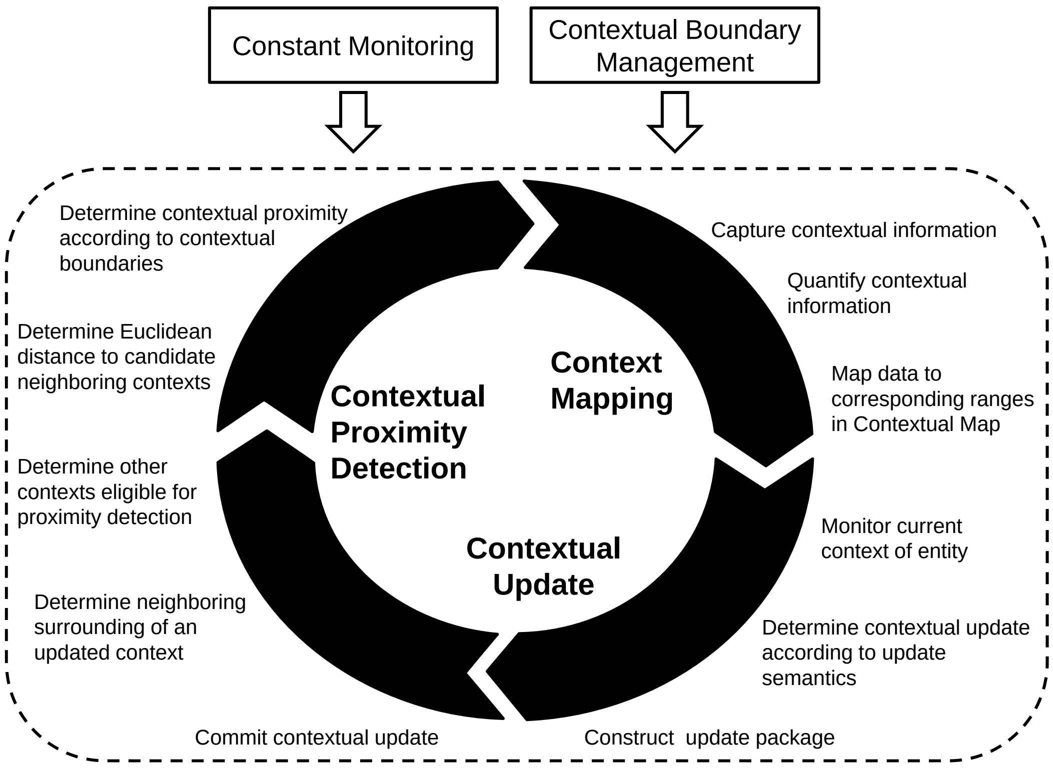

Constant Monitoring
Contextual Boundary Management
Determine contextual proximity according to contextual boundaries
Determine Euclidean distance to candidate neighboring contexts
Determine other contexts eligible for proximity detection
Determine neighboring surrounding of an updated context
Commit contextual update
Construct update package
Determine contextual update according to update semantics
Monitor current context of entity
Map data to corresponding ranges in Contextual Map
Quantify contextual information
Capture contextual information
Contextual Proximity Detection
Context Mapping
Contextual Update

Figure 4.2: Contextual Proximity Detection Workflow

4.1.1 Context Boundaries

Contextual boundaries represent the degree of similarity between contextual information. More precisely, contextual boundaries may be defined between different entities to determine the affinity of their individual contexts. This affinity expresses how interesting or - contrarily - uninteresting another entity's context is. Basically, this may be interpreted as contextual proximity, which has already been introduced in the previous chapters. Concluding, context getting *closer* also raises its relevancy for the concerned entity [85].

For example, a car driver may be interested in proper weather conditions on his route, hence taking action if a proximate weather station reports strong weather. We define the contextual boundaries between the entities *car driver* and *weather station* on two ranges: R_{loc} and R_{env} (compare table 3.1). Both ranges are part of the context of each entity. A weather station's physical position is represented in R_{loc}, its currently measured environmental readings are iteratively mapped to R_{env}. The car driver's context is represented by mapping his current location to R_{loc} and his personal conception about not-agreeable weather to R_{env}. We now aim at defining a context boundary, which delimits the driver's ability to drive from the situation of getting into extreme weather that keeps him from driving. Basically, we want to bound the situation of driving in bad weather by a boundary, so that the driver gets alerted when *crossing* this boundary. Obviously, we have to define the boundary on both ranges, R_{loc} and R_{env}, since both are relevant for modeling the described use case. Concerning R_{loc}, the boundary expresses a weather station's distance to the driver, so that it actually becomes of his interest. On R_{env}, the boundary focuses on how close the current weather at a station may come to the weather conditions regarded as critical by the driver. As it can be observed, the boundary on both R_{loc} and R_{env} is represented by a *proximity threshold* defining proximity between two contexts and setting the boundary on the specific range. The contextual boundary between the entities *driver* and *weather station* gets crossed if the driver gets close to the weather station, which reports conditions close the driver's critical criteria. Figure 4.3 illustrates the definition and crossing of this example boundary in the described scenario.

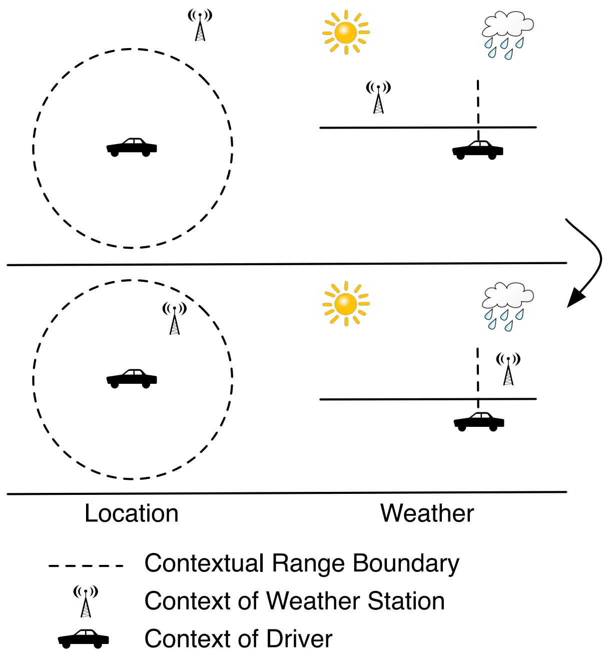


Figure 4.3: Crossing Context Boundary Example

In summary, a context boundary defines a specific *degree of affinity* between different contexts. The boundary is composed of proximity thresholds defined individually on relevant ranges. Each threshold defines range-specific degree of affinity concerning the range-specific context. Concluding, a crossing of such a boundary is dependent on those thresholds. Given this example, we can formally define contextual boundaries as follows:

- Contextual information is typified into ranges enabling contextual proximity. Each range may be composed of several dimensions (e.g. location decomposes into its three Cartesian planes x, y and z).
- A context boundary B may affect several ranges $R_1, ..., R_m$ with $m \leq o$ and r denoting the total number of ranges in the contextual map. This reduces the number of affected ranges to those, which are contextually relevant to B.

- A proximity threshold t_i is defined on each contextual range R_i, which is relevant to B. Each t_i is a single value dependent on all of the R_i's dimensions, hence, defining the degree of alikeness of R_i's contexts. Basically speaking, the threshold represents the *distance*, which delimits the range's contextual information to be "proximate" or not.
- The set of thresholds t_i of all ranges $R_1, ..., R_i, ..., R_m$ relevant to B defines the contextual boundary:

$$B = (t_1, ..., t_m) \tag{4.1}$$

It is to be noted that the ranges relevant to a context boundary have been enumerated, i.e. R_i is part of such enumeration and does not represent a rage ID. This expresses the fact that a context boundary affects a subset of all ranges, i.e. $m \leq r$.

4.1.2 Detecting Affinity among Contexts

With the principle of contextual boundaries defined, this section focuses on how contextual boundaries can be exploited using the contextual map model. As we have argued earlier, range contexts that are close to each other on range level are contextually affine in regard to the context attributes represented by that range. A context boundary may erect concrete delimiters for that context affinity by the definition of its range-specific thresholds. Hence crossing context boundaries corresponds to different entities' contexts getting either more or less affine (or rather contextually "closer/farther" to/from each other), thus crossing the affinity degree specified by the boundary.

With context boundaries defined in a contextual map, the next step is to determine crossings of those boundaries indicating changes of contextual affinities. Each of a context boundary's thresholds is defined on a single range. For this reason the check for two contexts having crossed the boundary by converging or separating from each other is performed range-specifically. It is to be reminded that each range represents a d-dimensional "submap" of the contextual map. Thus, we proceed by calculating the range-level *distances* between the two contexts, hence determining the Euclidean distances between their range contexts on each range affected by the boundary. We proceed range by range, and check, whether the Euclidean distance between the two correspondent range contexts has fallen below or exceeded the threshold of the current range.

Given d dimensions on a range R, and two contexts, P and Q, the Euclidean distance between the range contexts of P and Q on R is denoted by:

$$d_{\overline{PQ}}(R) = \sqrt{\sum_{i=1}^{d}(p_i - q_i)^2} \tag{4.2}$$

with p_i and q_i being the coordinates of dimension i of the range contexts P and Q, respectively: $P = (p_1, ..., p_i, ..., p_d)$ and $Q = (q_1, ..., q_i, ..., q_d)$.

With the distance $d_{\overline{PQ}}(R)$ calculated, we employ proximity and separation detection to determine, if the contexts of the two entities are "closing" or "separating" on range level. This task may be conducted by applying the proximity detection mechanisms discussed earlier in section 2.6 [61]. Consequently, the thresholds of a contextual boundary serve as proximity/separation thresholds to enable proximity or separation alerts. Such an alert is triggered when all of the boundary's thresholds have been breached, i.e. when the boundary has been crossed on each relevant range.

Formally: Let there be a context boundary $B = (t_1, ..., t_m)$ affecting m distinct ranges in the contextual map. Let t_i be the threshold defined for range i with $0 < i \leq m$. Further, let

$$A = (a_1, ..., a_i, ..., a_m) \tag{4.3}$$

be the set of Boolean variables where each a_i expresses proximity between P and Q on range R_i. In this case, proximity - meaning that P and Q are *within* the boundary's extent on R_i - is given when P and Q are closer to each other than t_i on R_i:

$$a_i = \begin{cases} true & \text{if } d_{\overline{PQ}}(R_i) \leq t_i \\ false & \text{else} \end{cases} \tag{4.4}$$

On the other hand, contextual differentness - meaning the opposite of contextual proximity so that P and Q are *outside* the boundary's extent on at least one R_i - is given when P and Q are farther from each other than t_i on at least one R_i (the $false$ case in equation 4.4).

Concluding, P and Q are contextually proximate upon the entire set A being $true$:

$$a_1 \wedge ... \wedge a_i \wedge ... \wedge a_m = true \tag{4.5}$$

Contextual differentness is given upon at least one value in A being $false$:

$$a_1 \wedge ... \wedge a_i \wedge ... \wedge a_m = false \tag{4.6}$$

At this point, we can formally discuss the definition of proximity and separation alerts in the contextual map. The detection of converging and separating context corresponds to the geographical detection of proximity and separation in 3-space (section 2.6). For this purpose, it helps to regard the "direction" of exceeding a context boundary's thresholds. If $d_{\overline{PQ}}(R_i)$ changes exceeding t_i (from its lower to its upper bound), a separating crossing of B is indicated (separation) on R_i. The opposite case, when $d_{\overline{PQ}}(R_i)$ falls below t_i (from its upper to its lower bound), indicates a converging crossing of B on R_i (proximity). This observation allows us to identify contextual convergence and separation on individual contextual ranges.

In order to detect boundary crossings at all, we need to observe all ranges that the boundary B affects. For this purpose, knowledge about $d_{\overline{PQ}}(R_i)\ \forall i$ at two different times is required. For this purpose, set A from equation 4.3 is replaced by A_{old} and A_{new}. Both of the new sets represent the according Boolean variables (i.e. the state of contextual proximity between P and Q) at those two times, where A_{old} precedes A_{new}. If both sets match the criteria of being alternately false and true, proximity or separation alerts are triggered:

- A proximity alert is triggered upon $A_{old} = false$ (equation 4.6) and $A_{new} = true$ (equation 4.5), i.e. P and Q have become contextually proximate in regard to B. It is to be noted that it suffices that a single $a_i \in A_{old}$ becomes true in A_{new}. In fact, $d_{\overline{PQ}}(R_i)$ may all have dropped below t_i on each range R_i in the past. But for the actual event of contextual convergence (that triggers the proximity alert), we regard the time frame between the point in time that A_{old} and A_{new} have been valid. It is only required that A_{old} becomes true in regard to A_{new}.

- A separation alert is triggered upon the opposite condition: $A_{old} = true$ (equation 4.5) and $A_{new} = false$ (equation 4.6), i.e. P and Q have contextually separated in regard to B. It is to be noted that a single $a_i \in A_{old}$ becoming false in A_{new} is enough to trigger the separation alert, since the requirement for contextual proximity regarding boundary B is not given anymore.

The triggering of such proximity or separation alerts corresponds to the contexts P and Q getting either more or less affine, respectively, hence having crossed the boundary B.

As stated, the detection of a context boundary crossing requires knowledge about distances between P and Q at two different times (before and after the crossing). As with the particular mechanism introduced in section 2.6, such determinations are made upon receiving an update from an affected entity. In contrast to section 2.6, we are dealing with contextual updates rather than with location updates. The corresponding update semantics, which have been extended to work in d-dimensional space, are discussed in the subsequent section 4.2.

Returning to our weather station example, we assume that the contexts of both the car driver and surrounding weather stations are iteratively captured and mapped to the contextual map. Hence, each weather station and the car driver get a constantly updated contextual map position associated with their respective context. The car driver has defined a contextual boundary defining dangerous weather, at which he is not willing to drive (i.e. weather coming close to his critical weather definition). Hence this boundary is decomposed into two ranges R_{loc} and R_{env} as depicted earlier in section 4.1.1 and figure 4.3: on R_{loc} expressing the driver's tolerated distance to a weather station reporting bad weather, and on R_{env} the tolerance interval to weather conditions, at which the driver refuses driving. Let the boundary be further defined as 10 kilometers on R_{loc} and 15 units on R_{env} (as depicted in table 3.1 in section 3). The distances between the contexts of driver D and weather station W on each range are computed upon receiving a contextual update using the Euclidean distance formula (equation 4.2). In this example, such an update may yield changing weather conditions and/or changing driver's location. As soon as the driver gets closer than 10 kilometers to a weather station, which reports weather less than 15 units distant to the driver's critical weather definition, the defined context boundary is crossed and a proximity alert is triggered. Hence, the driver may take action changing his route or stop driving. This corresponds the following Euclidean distances between range contexts of D and W:

$$d_{\overline{DW}}(R_{loc}) = 10 \tag{4.7}$$

$$d_{\overline{DW}}(R_{env}) = 15 \tag{4.8}$$

When mapped onto thresholds of a context boundary B, that boundary is defined as follows:

$$B = (t_{loc} = 10, t_{env} = 15) \tag{4.9}$$

4.2 Update Semantics

An obvious precondition for detecting proximity/separation of contexts in the contextual map is the knowledge about the contexts' most current positions in the map. This context actuality is achieved by contextual updates, i.e. updates containing an entity's most current context information. The challenge is to define efficient update semantics, i.e. when and how updates are to be committed, so that a minimal number of updates deliver the most current context possible. E.g., frequent updates issued in short intervals deliver very actual context information, but are highly inefficient, since most of them are redundant due to missing context changes.

In section 2.6, we have outlined efficient semantics for issuing location updates to detect geographical proximity among mobile hosts [61]. Those mechanisms allow the acquisition of the most current context with a minimal number of location updates. With little effort, we can transfer those principles on the contextual map. Both the geographical setting in section 2.6 and the contextual map are settled in multi-dimensional Euclidean space (with the geographical setting being 2-dimensional expressing width and depth). The proximity and separation detection mechanisms from section 2.6 work zone-based based on circles and strips. Recalling the statements from that section, those update zones present a spatial controlling mechanism for entity updates in 2-dimensional space. This means that an entity entering or leaving an update zone is eligible for issuing an update containing its most current coordinates (see figure 2.10).

Before extending those mechanisms on the contextual map model, it is to be reminded that entities have an n-dimensional representation of their respective context in the context map. It is to be recalled that this n-dimensional context decomposes into multiple d-dimensional range contexts. As stated earlier, d is range-specific depicting the number of dimensions of a particular range whereas n denotes the contextual map's total number of dimensions.

The extension of the update zone definitions in 2-dimensional space requires to define correspondent update zones in d-dimensional context ranges. The Euclidean distances defining contextual proximity are defined range-specifically. Consequently, update zones need to be defined range-specifically, too. Hence, the extension of the circular and strip-based update zones in 2-space to an arbitrary amount of dimensions concludes the definition of d-dimensional hyperspheres and hyperplanes, respectively. The following sections present an in-depth discussion on both of these extensions.

4.2.1 Hyperspheres (extended 2-dimensional Circles Method)

In the 2-dimensional realm ($\mathbb{R}^2$), contextual updates are sent upon leaving the circular zone spawned around the according entity's position during the last update, which thus has contained its last known position (see figure 2.10). Considering one additional dimension, thus the real-world-space ($\mathbb{R}^3$), the update zone gets sphere-shaped. Extending this principle on the d-dimensional realm ($\mathbb{R}^d$), we define an accordant hypersphere. Regarding the contextual map model, we center this hypersphere around the entity's last committed position of its context in the contextual map.

A d-dimensional hypersphere is called a $(d-1)$-sphere[1] S_{d-1}. The set points of such a sphere, $(x_1, ..., x_d)$, is defined by the following formula:

$$(x_1, ..., x_d) \in S_{d-1} \Leftrightarrow r = \sqrt{\sum_{i=1}^{d}(x_i - c_i)^2} \tag{4.10}$$

with $C = (c_1, ..., c_d)$ being the hypersphere's center point and r its radius.

In our case, we need to define a hypersphere bounding an update zone in $\mathbb{R}^d$. For this purpose, we interpret equation 4.10 as follows: the center point C is the position of the range context committed last, and r stands for the radius of the update zone. This definition allows us to determine, if the current range context P manifests itself at a position outside the hypersphere, i.e. if the Euclidean distance (equation 4.2) between P and C is greater than r.

Formally: Given the current range context $P = (p_1, ..., p_d)$, the last committed range context $C = (c_1, ..., c_d)$ and the update zone bounded by a hypersphere centered around C with radius r, if

$$\sqrt{\sum_{i=1}^{d}(p_i - c_i)^2} > r \tag{4.11}$$

holds, a contextual update is triggered, since P has left the hyperspherical update zone.

It is to noted that the bound of update zones, denoted by the radii of hyperspheres, are to be set dynamically based on different criteria as depicted by Küpper and Treu [61] for efficient proximity and separation detection in $\mathbb{R}^2$. Küpper and Treu aim at minimizing the amount of updates by dynamically altering an update zone's center and

[1]an n-sphere of radius r is defined as the set of points in $(n+1)$-dimensional Euclidean space which are at distance r from a central point

radius. With the Euclidean distance definition (equation 4.2) given, those principles are easily expandable on $\mathbb{R}^d$.

4.2.2 Hyperplanes (extended 2-dimensional Strips Method)

For proximity detection in $\mathbb{R}^2$, strips are spawned orthogonally between two mobile nodes eligible for proximity detection (again, see figure 2.10). Updates are sent by a mobile node upon entering such a strip. Thus, the update zone is bounded by two lines. In $\mathbb{R}^3$, such an update zone is bounded by two parallel planes. Thus, the d-dimensional analogy requires the definition of two $(d-1)$-dimensional hyperplanes, bounding the update-zone in $\mathbb{R}^d$. Since this approach is still settled in Euclidean space, we can formally define those two hyperplanes as follows. Let there be two contexts, P and Q, with according coordinates in the contextual map. The definition of hyperplanes in $\mathbb{R}^d$ works analogously as the definition of 2-dimensional planes in $\mathbb{R}^3$. We employ a base point B on the plane and a unit normal vector (aka surface vector) $\vec{n_0}$, which is orthogonal to the plane and has the vector length $|\vec{n_0}| = 1$. The *Hessian normal form* for the definition of (hyper)planes has the following form, then:

$$\vec{b} \cdot \vec{n_0} = d_b, \tag{4.12}$$

with $\vec{b}$ being the position vector of the base point B and d_b its distance to the coordinate system's point of origin.

Hence, to define the two hyperplanes for strip-based proximity detection, we first define one unit normal vector $\vec{n_0}$ for both planes, since the hyperplanes need to be parallel. Second, we define two base points R and S, which span up the two hyperplanes.

Formally: Given a unit normal vector $\vec{n_0}$ and two position vectors $\vec{r}$ and $\vec{s}$ (corresponding to the base points R and$S$, respectively), two hyperplanes are defined as follows:

$$\vec{r} \cdot \vec{n_0} = d_r, \vec{s} \cdot \vec{n_0} = d_s \tag{4.13}$$

with d_r and d_s denoting the respective distances of R and S to the coordinate system's origin.

With this definition given, $\vec{n_0}$, R and S need to be specified more precisely. Since both hyperplanes are parallel and bound the update zone exactly between P and Q, we put our focus on the line connecting both contexts, i.e. $\overline{PQ}$. To derive $\vec{n_0}$ we first define a regular normal vector $\vec{n}$. $\vec{n}$ expresses the direction of $\overline{PQ}$ and is therefore defined as follows:

$$\vec{n} = \begin{pmatrix} p_i - q_i \\ ... \\ p_d - q_d \end{pmatrix} \tag{4.14}$$

The definition of $\vec{n}$ implies that both hyperplanes are orthogonal to $\overline{PQ}$. Now, to obtain the *unit* normal vector $\vec{n_0}$, we need to normalize $\vec{n}$, i.e. shortening its length to 1. This is achieved by dividing $\vec{n}$ by its length $|\vec{n}|$. The definition of $|\vec{n}|$ is:

$$|\vec{n}| = \sqrt{\sum_{i=1}^{d} n_i^2} \tag{4.15}$$

Hence, $\vec{n_0}$ is defined as follows:

$$\vec{n_0} = \frac{\vec{n}}{|\vec{n}|} = \begin{pmatrix} (p_i - q_i)/|\vec{n}| \\ ... \\ (p_d - q_d)/|\vec{n}| \end{pmatrix} \tag{4.16}$$

With $\vec{n_0}$ "aligning" the hyperplanes accordingly (parallel), they still need to be "positioned" correctly, namely exactly between P and Q. For this reason, we regard the middle point C of $\overline{PQ}$, which is defined naturally:

$$C = (\frac{p_i + q_i}{2}, ..., \frac{p_d + q_d}{2}) \tag{4.17}$$

For reasons of simplicity, we position the base points of both hyperplanes, R and S, onto $\overline{PQ}$[2]. At this setting, R and S denote the intersections of $\overline{PQ}$ with the two hyperplanes. Given this assumption, both base points have to be equidistant from the center point C, in order to position the update strip exactly in between P and Q.

Formally: Given an update strip of width w (the Euclidean distance between both hyperplanes) and a middle point $C = (c_1, ..., c_d)$ between two contexts (and also between both hyperplanes), the hyperplanes' base points R and S and their corresponding location vectors $\vec{r} = (r_1, ..., r_d)$ and $\vec{s} = (s_1, ..., s_d)$ are restricted as follows:

$$\sqrt{\sum_{i=1}^{d}(c_i - r_i)^2} = \frac{w}{2} = \sqrt{\sum_{i=1}^{d}(c_i - s_i)^2} \tag{4.18}$$

and

$$\vec{r} \cdot \vec{n_0} - d_r = 0 = \vec{s} \cdot \vec{n_0} - d_s \tag{4.19}$$

with equation 4.18 defining the equidistance of both R and S to C (see Euclidian distance from equation 4.2) and equation 4.19 requiring them to be included in the according hyperplane (derived from equation 4.13 defining both hyperplanes in Hessian normal form), respectively. Complementary speaking, equations 4.19 and 4.13 infer that R and S are positioned on $\overline{PQ}$, since those positions are the only ones on the hyperplanes with a distance of $\frac{w}{2}$ from C.

With the bounding hyperplanes defined, it remains to elaborate the update condition, i.e. detecting contexts within the strip-area. Assuming that a context P has changed within the contextual map, the update condition can be determined by calculating its distances to the two bounding hyperplanes. If both distances are lighter than w, P is located within the update strip, triggering a contextual update.

[2]placing base point anywhere on the plane would denote a valid hyperplane definition

Formally, given an update strip of width w bounded by two hyperplanes defined by location vectors $\vec{r}$ and $\vec{s}$ and a unit normal vector $\vec{n_0}$, a context $P = (p_1, ..., p_n)$ is positioned inside the strip if

$$|\vec{p} \cdot \vec{n_0} - d_r| < w > |\vec{p} \cdot \vec{n_0} - d_s| \tag{4.20}$$

holds, with w being the Euclidean distance between both planes and $\vec{p}$ denoting the location vector corresponding to P. Assuming $\overline{RS}$ is orthogonal to both hyperplanes, w corresponds to the distance between the base points:

$$w = \sqrt{\sum_{i=1}^{d} (r_i - s_i)^2} \tag{4.21}$$

Figure 4.4 illustrates the strip-based update technique for proximity detection in $\mathbb{R}^2$. Since the strip is always bounded by $(d-1)$-dimensional hyperplanes, the bounds are simple lines in this case (1-dimensional hyperplanes). However, the illustrated principle can easily be extended on an arbitrary dimensional realm.

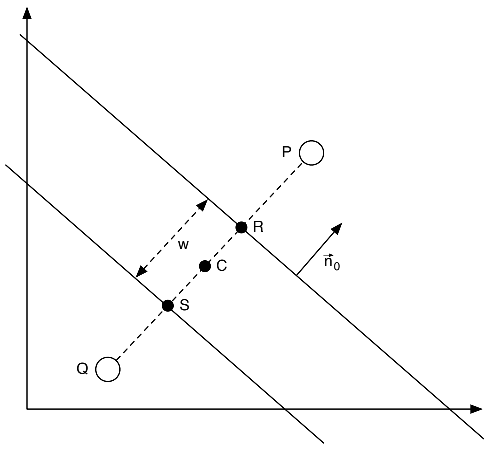


Figure 4.4: Update Strip Definition

Finally, we can now trigger a contextual update, if one of both contexts enters the strip-area bounded be the two defined hyperplanes. However, this method is only feasible to conduct proximity detection between the two contexts.

For separation detection the strip-method cannot be applied, as argued in section 2.6. In order to conduct separation detection, we define an update zone bounded by a hypersphere, centered at C (see equation 4.10). The contextual update is then triggered upon a context "leaving" the hypersphere, as defined by equation 4.11.

As with hyperspherical update zones in section 4.2.1, it is to noted that the bounds of the update strips, denoted by the strips' widths, are to be set dynamically based on criteria depicted by Küpper and Treu [61] for efficient proximity and separation detection in $\mathbb{R}^2$. Since the strip is repositioned and altered in width, the criteria can be easily extended to $\mathbb{R}^d$ using the definitions above.

4.2.3 Application of Update Semantics

To demonstrate the applicability of the update semantics presented here, we return to the example with the driver and the weather station. We have two contexts, the driver D and the station S. We further regard the two relevant ranges in the contextual map: location R_{loc} and weather conditions R_{env}. However, the dynamic changes, which take place here, are restricted to the weather station's current weather and the driver's location. Concluding, we have to pay attention to D's position in R_{loc} and S's coordinates in R_{env}, only. To proceed with this example we employ update zones bounded by hyperspheres. Therefore, hyperspheres are defined around D and S on their respective ranges R_{loc} and R_{env}. For D, it seems reasonable to select a radius of 10 on R_{loc}, denoting an update necessity from the driver every 10 kilometers (neglecting the third dimension z as denoted in table 3.1). For the update-sphere of S, we define a radius of 5 on R_{env}. This corresponds to an update triggered when the weather conditions at the weather station change by 5 units corresponding to a change of $\frac{1}{20}$ of the scale (see range's axis definition in table 3.1).

4.3 Update Composition

With the update semantics discussed in the previous section, an entity is able to determine a suitable moment *when* to commit an update. Now it needs determine *what* context information to include in that update. Merging all contextual information available into an update package would yield a large amount of unnecessary information. The recipient of the update may already know a large portion of the entity's context from the entity's preceding updates. Hence, the entity ought to select appropriate context information for its update.

In an n-dimensional context map, each entity's context consists of n values denoting the context position in the map. Naturally, with changing context the position changes as well. When committing an update, the entity propagates its changed context among the network. However, not all of the entity's n contextual coordinates may have changed in between updates. In such a case, it is only feasible to include actually changed context data in an update. Subsequently, we sketch a simple algorithm for controlled update composition based on changed context.

First, we introduce an additional data structure to be employed in parallel with an entity's contextual map: an n-dimensional Boolean vector, which denotes the entity's context validity in the network. The *validity vector* expresses which portions of the entity's context have been committed at a certain point in time - and thus are known in the network - and which parts of the context have not been propagated yet - thus unknown to the network. Consequently, the vector consists of n Boolean entries of which each i-th entry corresponds to the i-th entry in the contextual map denoting whether the map entry has been propagated since its last change or not. The validity vector is significantly smaller than the contextual map itself since it consists of 1-bit entries. Its main purpose is to prepare the context update, which is transmitted in form of another data structure, the *update vector*. We return to the validity vector first and use its definition to derive the update vector. Formally, the validity vector can be defined as follows. Given an entity's context $P = (p_1, ..., p_i, ..., p_n)$ in the contextual map, the validity vector V is defined as follows:

$$V = \begin{pmatrix} v_1 \\ ... \\ v_i \\ ... \\ v_n \end{pmatrix} \Bigg| \; v_i = \begin{cases} true & \text{if current } p_i \text{ committed by an update} \\ false & \text{else} \end{cases} \tag{4.22}$$

V is initialized together with P. All of its entries are set to false, since no update of P have been committed yet. While the entity, keeps its own context stored and updated locally, the update composition process is conducted as follows:

1. According to the entity's update semantics an update is triggered. Thus, the update composition is initiated.

2. All elements p_i of P, for which $v_i = false$ are extracted and merged into the update vector U.

   $$U = \begin{pmatrix} u_1 \\ ... \\ u_i \\ ... \\ u_n \end{pmatrix} \Bigg| \; u_i = \begin{cases} p_i & \text{if } v_i = false \text{ committed by an update} \\ \oslash & \text{else} \end{cases} \tag{4.23}$$

   Hence, an example update vector in an 8-dimensional setting corresponding to the weather station example may look as follows:

   $$U = (\oslash, \oslash, \oslash, 75, 68, 55, \oslash, true) \tag{4.24}$$

   assuming that p_1 through p_3 correspond to the weather station's coordinates, p_4 through p_7 denoting its environmental measurements and p_8 expressing its current alert level (comparing U to table 3.1, it decomposes into the ranges R_{loc}, R_{env}, R_{alert}). Here, the location context has remained unchanged, which seems logical, since we have been regarding a stationary weather station. But three of the environmental values have changed so that alert level must have been set to *true* indicating strong weather.

3. The update is committed to the system by transferring U to the target (e.g. a central server hosting the global contextual map).

4. The entire validity vector is set to *true*, since the current context of P is known by the network:

   $$v_i = true \; \forall i \leq n \; \big| \; v_i \in V \tag{4.25}$$

5. Every time when a contextual value p_i changes due to the entity's changing real-world-context, the correspondent value v_i in the validity vector is invalidated: $v_i = false$. This step is iterated until the next update is triggered according to the update semantics. In that case, the workflow sketched here starts over.

4.4 Efficient Update Handling

In the previous sections 4.2 and 4.3, we have discussed the update preparation procedure. We have presented the update semantics and update composition techniques for *update-sending entities* committing their most current context information. This section focuses on how those updates are handled by *update-receiving entities*. In order to determine the adequate time for an update, every entity monitors its own context only. For this reason, the focus of this section is put on how contextual updates are merged into the global context and how this information is used to monitor individual entities' contexts in regard to context boundaries.

For reasons of simplicity, we assume that one contextual update originates at one entity, which intends to update its own context in the contextual map. Concluding, the entity's context position in the map has potentially changed since the entity's last update. After having adjusted the context's coordinates it has to be determined whether any other contexts have converged to or separated from the updated context. Those converging and separating contexts indicate changes of contextual affinity among each other. Hence, contextual boundaries may have been crossed. However, since only map regions near the updated context are concerned for context boundary checks, the contexts in this regions - the context's *vicinity* - have to be determined. Thus, the context's vicinity represents the spatial neighborhood of a particular context and it represents the spatial region that is to be searched for boundary-crossing contexts. In summary, the update process can be divided into three steps, executed consecutively:

1. Update the contextual map according to the contextual update. This especially concerns the proper update on the map's data model (section 4.4.1).

2. Determine all contexts in the vicinity of the updated context. The resultant contexts have possibly crossed a context boundary to the updated context by converging to or separating from it (section 4.4.2).

3. Perform proximity and separation detection on the most updated context and contexts in its vicinity. Crossing of context boundaries is detected by this procedure (section 4.4.3).

The subsequent discussion in this section focuses on each of those three steps.

4.4.1 Updating the Contextual Map

Committing the contextual information contained by an entity's contextual update to the contextual map marks the first step in the receiver-sided update process. Before the actual update procedure on the data model is sketched, we take a look at some aspects concerning this process:

- As we have stated in section 3.1, the definition of *hierarchies* enables a suitable way of grouping and structuring ranges. Ranges can especially be utilized for typifying contextual information. Thus, they are suitable for grouping contextual information of the same type. E.g., location and weather conditions are context information types in our running example, each dedicated a range in the contextual map.
- Concerning the update process, only the *contextual ranges affected* by the update are updated. Hierarchies support this task since they provide rapid access to the ranges, which have to be updated. Furthermore, dependencies inside hierarchies can be exploited. If a parent range is dependent on its children and the contextual update affects its children, the parent range needs to be updated additionally. Although this seems to counteract the efficiency efforts, we will see the benefit of this approach shortly.
- Another aspect in minimizing effort during the update process is the exploitation of *redundancies among ranges hierarchies*. Given a parent range, which is dependent on its child ranges. If the child ranges are composed of information that specialize the parent and considered redundant according to the current proximity/separation-detection task, they can be completely neglected. Proximity/separation detection is only performed at the parent, even though the contextual update has explicitly affected the child ranges. This is possible, since dependencies between parents and children are subject for updating as stated before.

The actual step of updating the contextual map is closely related with the underlying data model. A brief survey of employable indexing structures has been provided in section 3.3 we have identified two techniques: tree-based indexes (section 3.3.1) and equidistant spatial grids (section 3.3.4). Each indexing technique comes with corresponding index-manipulation algorithms. Those are commonly focused on efficiently updating, removing and inserting nodes in their associated index-structure. As stated, efficiency is a high priority due to its exponential degradation coming with the increase of dimensionality.

An atomic update committed by an entity contains its own most current context. More precisely, it may contain the context changes of which the update's recipient is unaware of, as discussed previously in section 4.3. Basically speaking, given an n-dimensional context map, a single entity's update consist of at most n values of which each corresponds to a piece of context information, as denoted in section 3.1. In summary, those values comprise the entity's current and changed context.

A single context usually corresponds to multiple entries in the range-specific indexes, where there is one entry in each of contextual ranges' indexes depicting the full context's respective range context. In a tree-based structure, a range context is a single node in an index tree. In a spatial grid, it represents an element in a cell's list. We distinguish the following update operations performed on an index:

- *Regular Update:* This update operation updates a single entity's context representation and the corresponding range contexts in the contextual map's respective indexes.
  - *Index trees:* The corresponding node holding the range context data is identified in the tree (e.g. by a depth-first search) and its values are altered given the update values. Depending on the index tree in use, the tree must be adapted to fit its criteria. E.g., an update in an R-tree implies the tree to be reordered and balanced.
  - *Spatial grid:* The corresponding new parent cell is identified and the range context is added to the cell's inclusion list. In parallel, the range context is removed from the list of its previously known parent cell (before the update).
- *Insertion:* This update applies if a completely new context is inserted into the contextual map. This may occur if a new entity registers in the context map. According range contexts are created and merged into the according indexes.
  - *Index trees:* The index tree needs to be expanded by a new node associated with the new range context. This implies inserting that new node at the appropriate place in the tree and bringing the tree into its required shape. E.g., when inserting a new node into an R-tree, it must be inserted at the correct position in the tree, and the tree may have to be rebalanced afterwards.
  - *Spatial grid:* First, the according cell is identified in the grid, then the range context is added to the cell's inclusion list.
- *Removal:* In case that an entity exits the coverage of the contextual map, its corresponding context needs to be removed from the range-specific index structures.
  - *Index tree:* The node corresponding to the according range context must be removed from the index tree. This requires identifying that node inside the tree, deleting it from the tree and bringing the tree into a state fitting its criteria. E.g., removing a node from an R-tree only requires to re-balance it after deleting the affected node.
  - *Spatial grid:* The removal of a range context works analogously to the insertion case. The determination of its parent cell is followed by its removal from the cell's inclusion list.

The discussion about update operations is both theoretical and brief. A thorough discussion is provided in [101] and thus out of scope at this point.

4.4.2 Context Vicinity Definition

In the previous section 4.4.1 we have identified the problem of defining the region affected by a contextual update in regard to context boundaries. More precisely, given a freshly updated context C, its coordinates in the contextual map may have changed. This change implies that its surrounding contexts in the contextual map may be eligible for proximity or separation detection. In theory, all contexts encompassed in the contextual map may have potentially come close to C crossing a defined context boundary. However, the contexts "near" C are the most likely ones to have crossed a context boundary by either separating from or converging to C. Those contexts, which are all positioned in the *vicinity* of C, are of special interest for us. Proximity and separation detection, and thus context boundary checks, are performed on those contexts only, because only those are eligible for having crossed a context boundary.

At this point, we have to examine the context model on range level once again. As we have argued in section 4.1.2, proximity and separation detection is performed on ranges, since the contextual boundaries dictating the triggering threshold are defined range-specifically. For this reason we are concerned about C's surrounding regions on each range. We call those regions *range vicinities* denoting the d-dimensional Euclidean space surrounding C's range contexts.

Formally, given a range context C on range R, the range vicinity $V_{range}(C, R)$ groups all range contexts near C:

$$V_{range}(C,R) = \Big\{P = (p_1, ..., p_d) \;\Big|\; P \text{ is a range context "near" } C \text{ on } R\Big\} \tag{4.26}$$

This rather informal definition lacks a statement on exactly how "near" to C a range context P may be positioned. The answer crucially depends on the range vicinity's spatial definition, which is about to be discussed shortly. It is to be noted that we use the term "range vicinity" when referring both to a range context C's the neighboring *space* and the *set of other range contexts* in that space. Further, given m ranges in the contextual map, we call the set of full contexts[3], which corresponds to the union of all m range vicinities, the *contextual vicinity* of C:

$$V_{context}(C) = \left\{ \bigcup_{0<i\leq m} rcc\Big(V_{range}(C, R_i)\Big) \right\} \tag{4.27}$$

Hence, $V_{context}(C)$ groups all full contexts together out of all range contexts, which are listed in C's range vicinities (for the *rcc*-operator, see section 3.1). Last but not least, we define the superset of m different range vicinities denoting the *contextual range vicinity list* of C:

$$V_{list}(C) = \Big\{\big(V_{range}(C, R_i)\big)^+\Big\} = \Big\{V_{range}(C, R_1),\ \dots\ , V_{range}(C, R_m)\Big\} \tag{4.28}$$

In the rest of this section, we put special emphasis on range vicinities. To efficiently capture a context's range vicinity, we have identified the following aspects, which we consider relevant for defining range vicinities and identifying their encompassed contexts:

[3]as stated in section 3.1, full contexts are range-independent defined all n dimensions of the contextual map

- *Spatial shape of the range vicinity in $\mathbb{R}^d$:* In the first place, it has to be determined what the range vicinity is supposed to "look like", i.e. what shape it is supposed to assume. This consideration allows us to express how the range vicinity is actually bounded. In the forthcoming discussion, the most common shaping techniques, i.e. spherical and cubical shapes, are taken into focus yielding the definition of hyperspherical and hypercubical range vicinities.

- *Space partitioning of $\mathbb{R}^d$:* With the spatial shape of a range vicinity chosen, the portion of d-space belonging to the range vicinity needs to be determined. Obviously, this is needed to identify the proximate contexts lying inside that portion of space. The naive approach to check *all* contexts in the contextual map on membership of the range vicinity space is enormously elaborate and hence nonsensical. To counteract the naive approach, the d-space is first partitioned according to specific techniques. This leaves the entire space divided into numerous parts of space. The next step is to determine those spatial parts, which are actually part of the range vicinity (i.e. which overlap with the range vicinity space). Identifying contexts belonging to range vicinities is left to those spatial parts only. So in summary, space partitioning greatly enhances the efficiency of identifying range vicinity contexts.

- *Data model of the contextual map:* Since all range contexts are indexed in the contextual map's data model (see section 3.3), the data model itself plays a significant role in defining range vicinities.

Regarding these aspect, we have identified two basic approaches on efficiently defining range vicinities. Those two approaches are basically differentiated by the *spatial shape criterion*, which provides us with the most differentiated strategies for each approach. In the following discussion, focus is put on hyperspherical and hypercubical range vicinities. In both cases, we consider a single range vicinity $V_{range}(C, R)$, thus the vicinity of a freshly updated range context C on range R.

Bounding hypersphere

The natural approach of defining $V_{range}(C, R)$ is to define a hypersphere around it with all other range contexts within that hypersphere belonging to $V_{range}(C, R)$. By centering this hypersphere at C, the hypersphere then represents $V_{range}(C, R)$ with its surface bounding $V_{range}(C, R)$ equidistantly. The extent of $V_{range}(C, R)$ - and thus the hypersphere's radius - is directly dependent on the contextual boundaries defined on the range. Since a contextual boundary's individual range characteristic is defined by a threshold delimiter (see section 4.1.1) for converging and separating contexts, a contextual boundary can be interpreted as a hyperspherical boundary on range level. It can be easily observed that this range level context boundary definition (boundary threshold) is based on the same setting as the hyperspherical range vicinity $V_{range}(C, R)$. For this reason, a hyperspherical range vicinity's extent must equal the maximum threshold t_{max} out of all context boundaries defined R. Given k contextual boundaries $B_1, ...B_k$, we define t_{max} as follows:

$$t_{max} = max(t_R) \;\Big|\; t_R \in B_i, 0 < i \leq k \tag{4.29}$$

This constraint enables us to capture all range contexts, which potentially cross a contextual boundary defined on the range. Thus, $V_{range}(C, R)$ is defined as follows:

$$V_{range}(C, R) = \Big\{(P = (p_1, ..., p_d) \;\Big|\; d_{\overline{PC}}(R) \leq t_{max}\Big\} \tag{4.30}$$

Figure 4.5 visualizes the setting discussed above in $\mathbb{R}^2$ (bounding hypersphere with radius t_{max}).

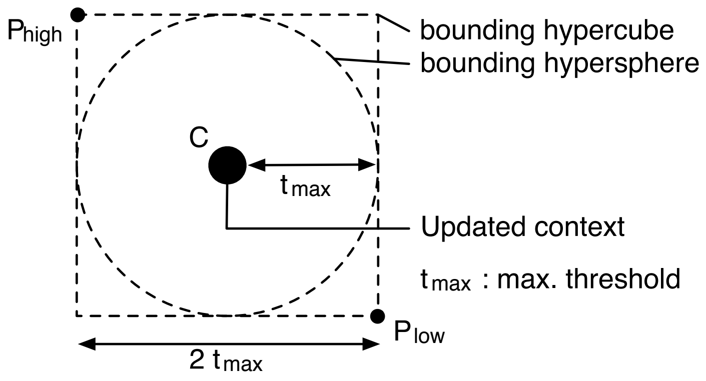


Figure 4.5: Bounding Hypersphere and Hypercube in $\mathbb{R}^2$

It is to be noted, that there still may be contexts beyond $V_{range}(C, R)$, which may have separated from C crossing a contextual boundary. Those contexts lie outside $V_{range}(C, R)$ now, so one may assume that they are not be included in the boundary check. Shortly, in section 4.4.3 we will explain why a radius of t_{max} suffices despite this observation.

Following this argumentation, a hypersphere with a radius of t_{max} offers a suitable approach for defining range vicinities. To efficiently identify all range contexts within $V_{range}(C, R)$, i.e. all $P \in V_{range}(C, R)$, we have identified two approaches:

- the hyperspherical range query
- the incremental nearest-neighbor search

Hyperspherical range queries This type of query focuses on the identification of points and spatial objects inside of hyperspherical ranges[4] in $\mathbb{R}^d$. The general approach on conducting those queries is to exploit space-partitioning for identifying spatial parts overlapping with the queried hypersphere and then determining all points lying inside that queried range:

1. The d-space is partitioned according to the specific querying technique.

[4]here, we are referring to spatial ranges, *not* ranges as defined by the contextual map

2. The hyperspherical query determines a query point forming the center of the query range (hypersphere).
3. All parts of space (according to space-partitioning) overlapping the query hypersphere are identified.
4. All points contained by the resultant parts of space are extracted.
5. Using the Euclidean distance check (equation 4.2), the distances between all of those resultant points and the query point (center of hyperspherical range) are determined. The points yielding a distance below the range hypersphere's radius are contained in the query range, hence representing the result set.

The scheme presented here represents a brief strategy. The next step is to apply it on a contextual range R in the contextual map. In order to determine $V_{range}(C, R)$, we perform a hyperspherical range query on R using C as query point and t_{max} as radius. The resultant points represent the range contexts of $V_{range}(C, R)$. There are numerous querying techniques implementing the scheme above. Subsequently, two interesting representatives are presented.

We begin with a basic hyperspherical range query scheme using R-trees. Although R-trees are outperformed by many other index trees, the strategy presented in the following is simple and thus illustrative. As we have discussed in section 3.3, an R-tree's inner nodes are comprised of entries, which denote hyper-rectangular spaces that are further divided by the nodes' children. This means that all spaces denoted by an R-tree non-leaf node's child nodes are encompassed by the space denoted by their parent node. Since an R-tree reflects its space partitioning approach in its own structure, we can exploit its space partitioning to identify spaces overlapping with an hyperspherical query range. Employing a depth-first search on the tree, we can prune all branches, whose parent nodes denote non-overlapping space. While conducting this searching scheme, we continuously narrow the search space. Eventually, the search arrives at leaf nodes, which represents points in regions overlapping the query range (parent nodes have not been pruned due to overlapping query range). Those points are potential candidates for the query range so that their distance to the query point, and hence membership in the query range, are determined. For example, consider this technique being applied on an R-tree that partitions space as shown in figure 4.6 (the R-tree is similar to the one in figure 3.3). The query is centered at $p8$. It identifies the MBRs $R2$, $R5$ and $R6$ being affected by the query while the remaining $R1$ and thus also $R3$ and $R4$ are pruned (compared to figure 3.3(b), the entire left branch of the R-tree is pruned). The points in $R5$ and $R6$ are checked whether they are included by the query. The results yields $p4$ and $p9$ ($p3$ and $p7$ are pruned). Note that we have applied the filtering and refining principle as introduced in section 3.3.2.

The second and quite neat approach of partitioning space in $\mathbb{R}^d$ together with a hyperspherical querying technique corresponding to the scheme above is featured by Lee et al. [62]. Their SPY-TEC approach stands for *spherical pyramid technique* partitioning bounded d-dimensional space into an amount of $2d$ spherical pyramids. Figure 4.7 shows

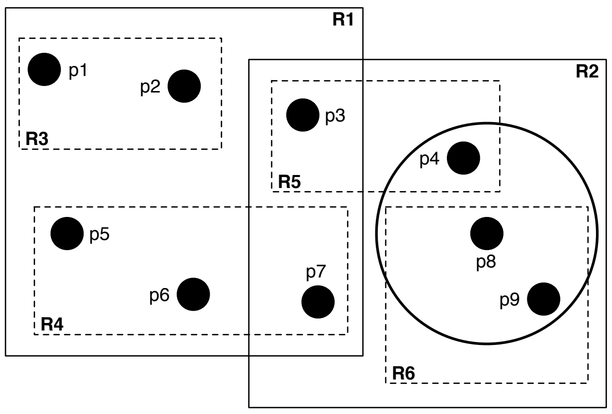


Figure 4.6: Hyperspheric Range Query on R-tree ($\mathbb{R}^2$)

this concept applied on bounded 2-dimensional space. A spherical pyramid (SP) is inspired by a regular hyperpyramid with the difference of possessing a spherical base. Just as spheres and cubes, regular pyramids can be extended to hyperpyramids representing their multi-dimensional counterparts. A hyperpyramid in $\mathbb{R}^2$ represents a triangle with two edges and a base line. In $\mathbb{R}^3$, a hyperpyramid represents a real-world pyramid with four edges and a base square. Consequently, a hyperpyramid in $\mathbb{R}^d$ consists of a top point, d edges and an $d-1$-dimensional base hypercube. Altering hyperpyramids to SPY-TEC's d-dimensional SPs, the hypercubical base is replaces by a hyperspherical one. Figure 4.7 shows four SPs in 2-space. As this concept dictates, d-space is always partitioned into an amount of $2d$ SPs. Further, the space needs to be bounded and encompassed by a hypersphere whose center represents the top of each SP and whose surface denotes the spherical bases of the SPs (as seen in figure 4.7). With the space partitioned into SPs, each SP is further dividable into slices (each slice is a "layer" of the SP as shown in figure 4.7). The SPY-TEC query also implements the filtering and refining technique. Given a query point and radius, all slices of different SPs overlapping the query hyperspherical range are determined in the first step. The points in all those slices represent the subset eligible for the second refining step to determine which of the point are actually inside the query range (false positive elimination using Euclidean distance check, see equation 4.2). According to the authors of SPY-TEC [62], their method performs significantly better than algorithms performed on indexing structures, which have been discussed before.

It is to be noted that the calculated distances between points in the query range and the query point itself are usually determined in the final phase of a hyperspherical querying algorithm. Those distances correspond to the contextual range distances between an updated context (query point) and contexts in its vicinity (points in query range). Since exactly those distances are needed for the subsequent context boundary check, it is practicable to remember them and store them along with a context's range vicinity (section 4.4.3).

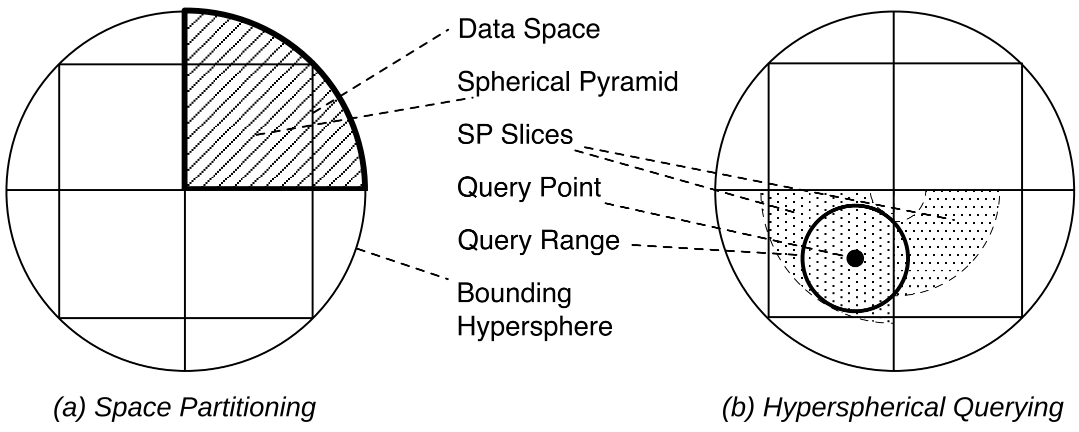


Figure 4.7: SPY-TEC Space Partitioning and Querying in $\mathbb{R}^2$

Incremental nearest-neighbor search A slightly different approach from the bounding strategies discussed so far is the application of *nearest-neighbor* algorithms. A k-nearest-neighbor (k-NN) algorithm in Euclidean space delivers the k nearest neighbors of a query point q. Since those algorithms have reached a mature state they work fast and efficiently [50, 71, 86]. We exploit this characteristic for our purpose by iteratively applying a k-NN algorithm at the query point C. It is to be reminded that C denotes the updated context and hence the center of the range vicinity $V_{range}(C, R)$. We iteratively execute a k-NN-algorithm incrementing k in each iteration.

We begin with $k = 1$. Subsequently, we continue with an iterative execution loop of the k-NN algorithm. In each k^{th} iteration the algorithm yields a set N_k of the k nearest neighbors of C. Since we have defined a hyperspherical vicinity bound with radius t_{max}, it is checked if the distance between the k^{th} neighbor[5] in N_k exceeds t_{max}. If so, $V_{range}(C, R)$ has been determined comprising all range contexts in N_k except the k^{th} one. Hence, the iteration loop is terminated and N_k is rid of C's k^{th} neighbor $P = (p_1, ..., p_d)$:

$$V_{range}(C, R) = N_k \setminus \Big\{(p_1, ..., p_d)\Big\} \quad \Bigg| \quad \sqrt{\sum_{j=1}^{d}(p_j - c_j)^2} \geq t_{max} \tag{4.31}$$

Depending on the actual choice of a k-NN strategy, the algorithm itself most likely works iteratively until delivering k neighboring range contexts. It may be practicable to perform the boundary checks within the algorithms iterations, incrementing k as needed and thus forcing the algorithm to terminate when all range contexts in $V_{range}(C, R)$ have been determined and checked against defined context boundaries.

[5]the k-th neighbor is the farthest neighbor from the query point (in this case C)

Bounding hypercube

Although a hyperspherical range vicinity represents the natural bound of a context's surrounding due to its exact representation of its extent t_{max}, its determination is somewhat elaborate. For this reason, we may dilute this vicinity definition by bounding $V_{range}(C, R)$ by a hypercube of the same extent. Since the data model indexes contexts efficiently (see section 3.3) the range contexts inside a hypercubical $V_{range}(C, R)$ can be determined quickly. See figure 4.5 for a visualization of this approach in $\mathbb{R}^2$.

The basic tool for determining a hypercubical $V_{range}(C, R)$ is the *multi-dimensional range query*. Given a hyper-rectangular query range[6] in multi-dimensional space denoted by two bounding points P_{low} and P_{high}, this type of query is performed upon the underlying indexing structure yielding all points in the query range. We have discussed several indexing structures for points in multi-dimensional space in section 3.3. The majority of the subsequent analysis focuses on index trees (from the discussion in section 3.3.1. After that, we take a look on how to apply bounding hypercubes on the spatial grid index (discussion in section 3.3.4).

Index trees perform efficiently on accessing multi-dimensional data. Hence, multi-dimensional range queries perform efficiently, too. In order to use the multi-dimensional range query for hypercubical range vicinities, we employ the d-dimensional range query with d being the amount of dimensions of the affected contextual range[7] R. We define an equidistant unidimensional query range on each of R's d dimensions. The size of the query range on each dimension implies the size of the hypercube bounding $V_{range}(C, R)$. It can be derived from the discussion about the size of the bounding hypersphere. There, we have identified the maximum threshold t_{max} of all contextual boundaries' thresholds defined on R. Since the bounding hypercube is supposed to be of the same extend as the bounding hypersphere, the size of each dimensional query range must equal $2t_{max}$ corresponding to the according hypersphere's diameter. Figure 4.5 visualizes this setting comparing a hyperspherical and hypercubical range vicinity in $\mathbb{R}^2$. In addition and as shown by the figure, the range context C, which defines $V_{range}(C, R)$, is situated exactly in the center of the hypercube.

Formally, given a d-dimensional contextual range R and a correspondent query range in $\mathbb{R}^d$ bounded by $P_{low} = (PL_1, ..., PL_d)$ and $P_{high} = (PH_1, ..., PH_d)$ with $PH_i > PL_i$, and a query range size of $2t_{max}$, a hypercubical query range of that extent is given if the following equation holds:

$$PH_i - PL_i = 2t_{max} \; \forall \; i \leq d \tag{4.32}$$

$V_{range}(C, R)$, bounded by such a hypercube, is then defined around $C = (c_1, ..., c_d)$ given $P_{low} = (PL_1, ..., PL_d)$ and $P_{high} = (PH_1, ..., PH_d)$ with C being located in the center of the bounding hypercube:

[6] we refer to *ranges* in multi-dimensional space, *not* ranges in connection with range vicinities and range contexts

[7] here, we refer to *contextual ranges* as defined by the contextual map

$$PL_i = c_i - \frac{PH_i - PL_i}{2} = c_i - t_{max} \ \forall \ i \leq d, \tag{4.33}$$

$$PH_i = c_i + \frac{PH_i - PL_i}{2} = c_i + t_{max} \ \forall \ i \leq d \tag{4.34}$$

Hence, $V_{range}(C, R)$ can be defined encompassing all range contexts within the range of P_{low} and P_{high}:

$$V_{range}(C, R) = \Big\{(P = (p_1, ..., p_d) \ \big| \ PL_i \leq p_i \leq PH_i\Big\} \tag{4.35}$$

The basic prerequisite for the d-dimensional range query is the space partitioning strategy given by the index structure in use. For example, given the contextual map indexed by an R-tree, a depth-first search can be applied upon it. This search works similarly as querying hyperspherical ranges on an R-tree (see discussion above about hyperspherical range queries). As we have argued earlier, all branches with parent nodes denoting spaces not overlapping with the query range are pruned. As a result, all leaf nodes which are reached by the search are either inside the query range or close to the query range. The latter ones are pruned so that upon the search terminating the set of all points within the query range has been found. The efficiency gained by this approach is given by two aspects:

- The search prunes branches in the R-tree, which contain points of spaces outside the query range.
- The R-tree is a balanced B-tree, optimizing search times of algorithms applied on it.

Performing hypercubical range queries are also well suited to be applied on spatial grids (section 3.3.4). However, it works best, if $V_{range}(C, R)$ is approximated even further. In the discussion above, we have argued that $V_{range}(C, R)$ should have an exact extend of $2t_{max}$. Regarding the spatial grid, we may simply choose a query range encompassing exactly all those cells overlapping with that space. Concluding, even that this query range is a little bit larger than the one depicted above, the fact that query range is a set of complete cells (the union of those cells yields the query range in return) allows us to simply determine the query's result by summing up all range contexts from the included cells' inclusion lists. Given a cell size of u with C located in its cell D bounded by $D_{low} = (DL_1, ..., DL_d)$ and $D_{high} = (DH_1, ..., DH_d)$, $P_{low} = (PL_1, ..., PL_d)$ and $P_{high} = (PH_1, ..., PH_d)$ with $PH_i > PL_i$ are defined as follows:

$$PL_i = DL_i - v_i w \ \forall \ i \leq d \ \big| \ v_i \in \mathbb{N}_0 \tag{4.36}$$

$$PH_i = DH_i + w_i u \ \forall \ i \leq d \ \big| \ w_i \in \mathbb{N}_0 \tag{4.37}$$

with v_i and w_i being the iterators counting how many of the surrounding cells as to be included, so that the following equations holds:

$$c_i - PL_i \geq t_{max} \wedge c_i - PL_i - u < t_{max} \; \forall \; i \leq d \; \big| \; v_i \in \mathbb{N}_0 \tag{4.38}$$

$$PH_i - c_i \geq t_{max} \wedge PH_i - c_i - u < t_{max} \; \forall \; i \leq d \; \big| \; v_i \in \mathbb{N}_0 \tag{4.39}$$

$$(PH_i - PL_i) \bmod u = 0 \; \forall \; i \leq d \tag{4.40}$$

wit equations 4.38 and 4.39 defining the range's extent and figure 4.40 expressing that the range encompasses complete cells. Figure 4.8 shows an exemplary $V_{range}(C, R)$ in a spatial grid.

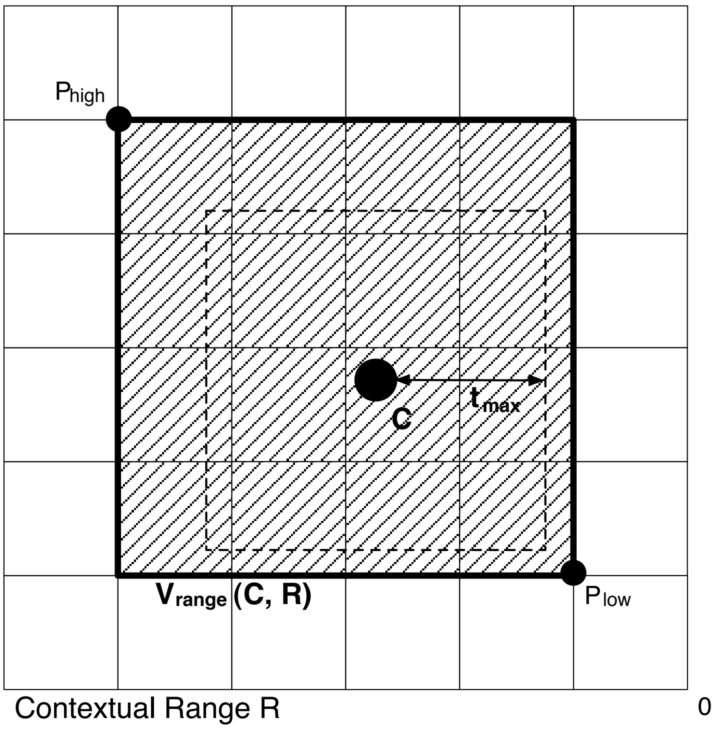


Figure 4.8: Hypercubical Range Vicinity in a spatial Grid

To the time being, lots of research has been conducted on this topic. Besides the strategy sketched above, there are numerous variants and improvements on d-dimensional range querying. These approaches are comprised of both indexing structures [30, 41, 66, 101, 80] and multidimensional querying strategies [33, 58, 80, 71]. Concerning indexing trees, special emphasis is put on variants of the R-tree [25, 93], which is a wide-spread structure for multidimensional indexing. All of the approaches focus on its performance optimization to overcome the significant shortcomings of storing multi-dimensional data ("curse of dimensionality", section 3.3).

From the discussion so far, we can derive a general workflow of processing a hypercubical query on the contextual map (i.e. a contextual range) yielding all contexts inside the queried context's range vicinity (i.e. the updated context):

1. The d-space is partitioned according to the specific querying technique, i.e corresponding to the indexing structure in use.
2. The hypercubical query determines a query point forming the center of the hypercube and denoting the range context C, which the queried range vicinity belongs to.
3. All parts of space (according to space-partitioning) overlapping the query hypercube are identified.
4. All points/contexts contained by the resultant parts of space are extracted.
5. Each of the resultant contexts is checked on membership in the query hypercube. The complying contexts represent the range vicinity $V_{range}(C, R)$.

4.4.3 Boundary Check

After determining contexts in the vicinity $V_{context}(C)$ of a context C in the contextual map, the next step is to determine whether any of those proximate contexts in $V_{context}(C)$ have crossed a contextual boundary to C. Recalling the discussion in section 4.1, contextual boundaries define the degree of affinity between two contexts. In section 4.4.2, we have argued that $V_{context}(C)$ consists of multiple range vicinities $V_{range}(C, R_i)$ (with i enumerating the contextual ranges). Since contextual ranges group contextual attributes of the same type by mapping those onto the range's dimensions, the contextual boundary check is performed range-specifically on each $V_{range}(C, R_i) \in V_{list}(C)$, as we have explained on previous occasions.

The general picture of the contextual boundary check is to first analyze all ranges on which the examined boundary applies to, i.e. on which it has defined any thresholds. With all ranges examined knowing which pairs of contexts have exceeded threshold on which range (i.e. range-specific boundary crossing), we can determine crossings of the examined boundary. Such a crossing is detected for a pair of contexts, if all the boundary's thresholds have been exceeded between the two contexts. In the following discussion we first focus on the range-specific aspects before concluding with the overall setting in the contextual map.

Given a specific contextual range R and an updated context C, we proceed as follows. First, we determine if any range contexts in $V_{range}(C, R)$ have crossed a contextual boundary on the examined range R. Recalling the definition of contextual boundaries, a single boundary defines one threshold on each of the ranges, which concern the boundary. Thus, regarding a single range, a contextual boundary is defined by a proximity threshold t on that range, hence defining a delimiting distance between C and any other range context. Further, multiple boundaries may result in multiple threshold definitions on a single range (again, one threshold specific to R per boundary only). Let us assume that there are three thresholds of three boundaries defined on R, as depicted in figure 4.9. Figure 4.9 shows the range vicinity $V_{range}(C, R)$ of the updated context C with the three boundary thresholds t_1, t_2 and t_3 with the latter also denoting the maximum threshold

t_{max} defined on R. $V_{range}(C, R)$ encompasses eight other range contexts $P_1, ..., P_8$, which are logically contextually proximate to C (to a degree t_{max}) on R.

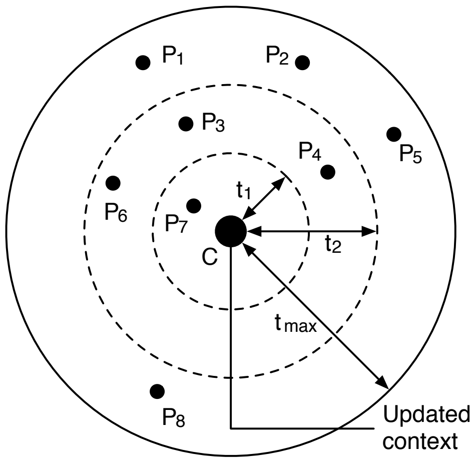


Figure 4.9: Boundary Thresholds in Range Vicinity ($\mathbb{R}^2$)

A crossing of a specific boundary on the examined range R is denoted by a range context getting closer than t to C or farther than t from C, with t being the boundary's threshold defined on R. Actually, this approach corresponds closely to the principles of proximity and separation detection [61] with t denoting both the separation and proximity distance (see section 2.6). From here forth, we use the term *range crossings* when referring to boundaries which have been "crossed" range-specifically as explained here (i.e. threshold values that have been exceeded positively or negatively in terms of separation and convergence, respectively).

In order to determine a range crossing on range R we need to know the distances between neighboring range contexts and C *before* and *after* an update on C (compare discussion about proximity sets as defined in equation 4.3 denoting convergence and separation in section 4.1). For this reason, we calculate all distances $d_{\overline{PC}}(R)$ between every range context P in $V_{range}(C, R)$ and C *before* the incoming update is merged into the context map. We use the Euclidean distance formula from equation 4.2 to determine the distance between C and any other multi-dimensional point denoting a range context in $V_{range}(C, R)$ (see section 4.1.2). We store the distances in a list D_{pre}(C) denoting the distances of range contexts to C before the update. Hence, each list element is a tuple denoting a range context P and its distance to C:

$$D_{pre/post}(C) = \Big\{ \big(P, d_{\overline{PC}}(R)\big)^* \Big\} \quad \Big| \quad P \in V_{range}(C, R) \tag{4.41}$$

After committing the update, the distances are all recalculated and stored analogously in a new list $D_{post}(C)$ listing the distances between range contexts and C *after* the update. With both $D_{pre}(C)$ and $D_{post}(C)$ we can almost perform the range crossing check. For this purpose another list is introduced: the *threshold range list.* It lists which range contexts are closer than a threshold's value to C. Thus, each list entry is a tuple depicting a threshold t defined on the range and a list of range contexts, which are all closer than t to C:

$$T_{pre/post}(C) = \left\{ \left(t, \left(P \mid d_{\overline{PC}}(R) < t \right)^* \right)^* \right\} \;\Big|\; P \in V_{range}(C, R) \tag{4.42}$$

Regarding the example setting in figure 4.9 the threshold range list comprises three entries, one for each threshold, i.e. t_1, t_2 and t_3 (t_{max}). The range contexts, which are associated with each entry are listed in table 4.1.

Threshold	Range contexts
t_1	P_7
t_2	P_3, P_4, P_6, P_7
$t_3 = t_{max}$	P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8

Table 4.1: Threshold Range List of Figure 4.9

As with the distance lists, we need two threshold range lists, $T_{pre}(C)$ and $T_{post}(C)$ stating the situation before and after the update, respectively. Both $T_{pre}(C)$ and $T_{post}(C)$ are defined analogously. With those two lists, range crossings can now be determined. The number of entries in $T_{pre}(C)$ and $T_{post}(C)$ is equal. Both denote the same number of thresholds. The identification of range crossings is conducted by comparing the entries from $T_{pre}(C)$ and $T_{post}(C)$, which are associated with the same threshold. Let us consider a single threshold t and its two corresponding list entries from $T_{pre}(C)$ and $T_{post}(C)$, respectively:

$$(t, Q_{pre}), (t, Q_{post}) \;\Big|\; Q_{pre/post} = \left\{ \left(P \mid d_{\overline{PC}}(R) < t \right)^* \right\} \tag{4.43}$$

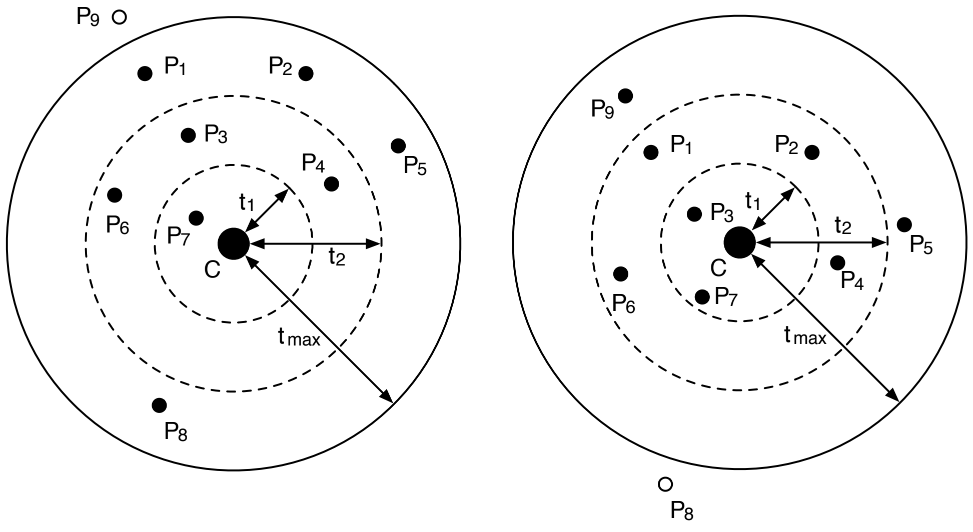
with Q_{pre} and Q_{post} denoting the set of range contexts closer than t to C before and after the update, respectively. Because of the update occurring in between the generations of Q_{pre} and Q_{post}, it can be stated that Q_{pre} is transformed into Q_{post} by the update commitment. This transformation implies that range contexts may either disappear from the set or new range contexts may show up in it. Those observations have the following meaning regarding range crossings:

- *Contexts disappearing:* A range context P disappearing from Q_{pre} and hence missing in Q_{post} indicates that P has moved farther away from C and that the distance between the two has just exceeded t during the update. This observation implies a range crossing for t having occurred between C and P, since $d_{\overline{PC}}(R) < t$ holds before and $d_{\overline{PC}}(R) \geq t$ after the update (see section 4.1). Since P has moved farther away from C, C and P have become *less affine* in terms of contextual proximity.

- *Contexts emerging:* A range context emerging in Q_{post} but having been missing in Q_{pre} indicates that P has moved closer to C and that the distance between the two has just fallen below t during the update. This observation also implies a range crossing for t having occurred between C and P, since $d_{\overline{PC}}(R) \geq t$ holds before and $d_{\overline{PC}}(R) < t$ after the update (again, see section 4.1). Since P has come closer to C, C and P have become *more affine*.

- *No changes:* If $Q_{pre} = Q_{post}$ holds no range crossing for t has occurred. The distances between the included range contexts and C have not changed as significantly as to affect t.

The procedure of comparing Q_{pre} and Q_{post} is conducted on every threshold defined in the range vicinity $V_{range}(C, R)$. To improve the readers understanding for the procedure described above, we show a practical example. Returning to the setting in figure 4.9, there are three thresholds to be examined. The threshold range list for $V_{range}(C, R)$ displayed there has been listed in table 4.1. Assuming that an update is committed on context C, this list can be regarded as $T_{pre}(C)$ denoting the range vicinity's initial state. Let us assume further that the update causes C to "move up" on the particular range in $\mathbb{R}^2$ as depicted in figure 4.10[8]. As a consequence $T_{post}(C)$ can be derived from the resultant range vicinity - as shown on the right side of figure 4.10. $T_{post}(C)$ is listed in table 4.2.



(a) Range vicinity with T_{pre} *(b) Range vicinity with T_{post}*

Figure 4.10: Range Crossings in $\mathbb{R}^2$

With $T_{pre}(C)$ and $T_{post}(C)$ in tables 4.1 and 4.2, both can be checked for changes inflicted by the update. The results indicated in table 4.3 show the following range crossings in regard to C:

- P_3 has become more affine causing a range crossing on t_1
- P_1 and P_2 have become more affine causing 2 range crossings on t_2

[8]note, that we have extended figure 4.9 by range context P_9 for illustration purposes

Threshold	Range contexts
t_1	P_3, P_7
t_2	P_1, P_2, P_3, P_4, P_6, P_7
$t_3 = t_{max}$	P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_9

Table 4.2: Threshold Range List after Update

- P_9 has become more affine causing a range crossing on t_3
- P_8 has become less affine causing a range crossing on t_3

Threshold	Q_{pre} (table 4.1)	Q_{post} (table 4.2)	removed	inserted
t_1	P_7	P_3, P_7	$\emptyset$	P_3
t_2	P_3, P_4, P_6, P_7	P_1, P_2, P_3, P_4, P_6, P_7	$\emptyset$	P_1, P_2
$t_3 = t_{max}$	P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8	P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_9	P_8	P_9

Table 4.3: Changes in Threshold Range List (figure 4.10)

At this point we have identified which thresholds have been exceeded and if the corresponding range crossings denote range context pairs getting more or less affine. This range-specific boundary check for context C is performed on every range in the contextual map. Basically, we have identified the range-local changes in contextual proximity caused by a contextual update. After this procedure is finished, we can derive the crossing of the complete contextual boundaries from the previously determined range crossings. For any context pair including C and another context from $V_{context}(C)$, all thresholds, which exhibit range crossings, are known. Hence, we can determine which boundaries have been crossed by checking all range crossings:

1. Given a range crossing between range contexts C (updated context C) and P on range R, we identify the affected boundary B. The range crossing always denotes an exceeding of a threshold t, which belongs to a particular boundary, hence B. Formally: $t \in B$ holds.
2. We fetch all threshold range lists of C's range contexts on all *other* ranges than R. Next, we extract the entry corresponding to t from each of the threshold range lists, i.e. (t, Q_{post}). We now have to regard two cases:
   - If the range crossing denotes that t has been exceeded from its upper to its lower bound, a range-specific *convergence* on R occurred. Thus, we check if P is included in all Q_{post}.

  - If so, C is contextually proximate to P on all ranges defined by B (equation 4.5 holds) and B has been crossed (because this has not been the case before the update yet)
  - If not, we have to check if there were converging range crossings on all those ranges on which P was missing in Q_{post}. If so, C is contextually proximate to P on all ranges defined by B and B has been crossed. If not, B has not been crossed.

- If the range crossing denotes that t has been exceeded from its lower to its upper bound, a range-specific *separation* on R occurred. Thus, we perform the same check as with a converging range-crossing, i.e. we check if P is included in all Q_{post}. If so, a contextual separation between C and P in regard to B has occurred since C and P are not sufficiently contextually proximate to satisfy the contextual similarity degree defined by B (equation 4.6 holds). If not, B has not been crossed.

3. all range crossings on other ranges affecting both the boundary B and contexts C and P, are pruned, since it has been determined if C and P have crossed B or not. We proceed by applying this enumerated checking procedure on the remaining unchecked range crossings.

A crossed context boundary is finally eligible to trigger any affinity alerts defined on P or C. Closing, it is to be remarked that the boundary check procedure can be compared to proximity and separation detection in $\mathbb{R}^n$. Here, we exploit this approach for context-aware computing for monitoring affinities among contexts.

4.5 Monitoring the Contextual Map

In the past sections 4.3 and 4.4 we have discussed how updates are composed and handled. Both concerns are limited to a single point in time. In this section, we add a timely aspect, thus presenting principles how to continuously observe multiple updates over a period of time.

Monitoring the contextual map basically means monitoring its contextual ranges. Observing a range over time implies to regard all range contexts present in that range. So far, we have always regarded a single updated range context. Thus, this section also extends the principles from the previous sections to an arbitrary number of updated range contexts.

We proceed as follows: First, we identify the setting in which range monitoring occurs (section 4.5.1). After that, we discuss how we employ the contexts' range vicinities to continuously monitor affinities between entities, i.e. contextual proximity among their belonging contexts (section 4.5.2).

4.5.1 Contextual Range Monitoring

In section 4.4 we have described how a single contextual update is processed to determine contextual proximity in conjunction with contextual boundaries, i.e. we have shown how a single update inflicts boundary crossings and how those are detected. In a contextual range, however, we usually face an arbitrary number of range contexts that change dynamically at random times following contextual updates of their corresponding entities. Hence, we extend the principles from section 4.4 to an arbitrary number of range contexts that change their position in the contextual range dynamically over time. Hence, let $C(t)$ denote the range context C at time t on range R with d dimensions:

$$C(t) = \Big(c_1(t), ..., c_d(t)\Big) \tag{4.44}$$

In order to perform contextual affinity detection on multiple entities, ideally, their most current context should be known. This would require a continuous stream of updates from each entity committing its actual context information. This is obviously impractical. Since contextual updates are only committed from time to time according to the update semantics in use (see section 4.2) the most current context is always an approximation of an entity's real-time context. Thus, a contextual range represents an approximation of the overall situation regarding the context type which the range represents (e.g. weather context from section 4.1).

Following this argumentation, the quality of the overall current state of a contextual range depends on the quality of the individual contexts, which can be broken down to two basic influencing factors

- Context actuality / freshness of data
- Context fluctuation

We reflect on both aspects in the subsequent discussion before closing this section with sketching the monitoring mechanisms applied onto a contextual range. It is to be noted that since we regard contextual ranges here, we *always* refer to range contexts (even if not explicitly writing so), *not* to range-global contexts in the contextual map.

Context actuality

A range context's actuality is a decisive criterion for the overall freshness of the contextual range. It is primarily dependent on the update semantics in use. Zone based update semantics as presented earlier in section 4.2 influence the context freshness by the proper definition of their update zones. Smaller update zones cause more frequent updates, which lead to more current context in turn. Distance-based update semantics behave equally. On the other hand, when using periodic interval-based update semantics context freshness is dependent on "context velocity". The context velocity can be regarded as a measure how fast a context changes and how fast it "moves" through the contextual range. Fast moving contexts change significantly over time, hence reducing the correctness of the last committed update accordingly.

An option to enhance actuality of contexts is to *predict* their location in the context range according to their location history [76]. A range context's location history consists of points in the contextual range that it updated in the past. For "moving" contexts, those points yield a trajectory, which allows the prediction of future locations. In terms of context-awareness, we gain limited ability to predict an entity's future context (see figure 4.11). Of course, due to the inability to foresee the future, we explicitly declare this ability as *limited*.

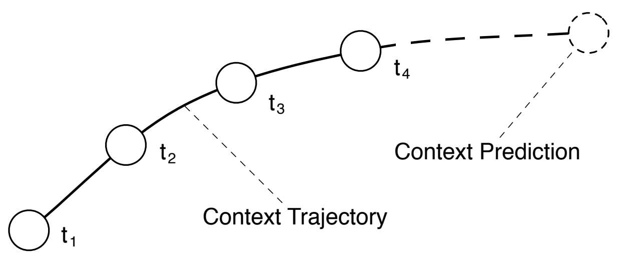


Figure 4.11: Context Prediction based on Trajectory between Times t_1 and t_4

A simple example of predicting a range context's C location is not to regard its trajectory, but merrily its last known position[9] $C(t_0)$, velocity v_0 and direction[10] $\vec{d_0}$ at a time t_0. Missing any subsequent update, we can predict its location at a time past t_0, say t_1, by simply applying the laws of mechanical physics:

$$C(t_1) = C(t_0) + \vec{d_0} v_0 (t_1 - t_0) \tag{4.45}$$

Prediction of range contexts can provide a more actual context location for entities, which have not recently committed contextual updates and whose contexts may have become obsolete in the real world. For this reason, predicting an entity's range context according to its trajectory for the *current* time may contribute to a more accurate context range.

Context fluctuation

Another aspect impacting the quality of context is contextual fluctuation [77]. Apparently unchanged context of an entity may change slightly over time due to imprecise sensor readings. This causes the corresponding context representation on the according contextual range to fluctuate, i.e. to "move around". Figure 4.12 illustrates a fluctuating context.

Padovitz et al. [77] have approached this phenomenon by determining an ideal state inside a fluctuation area. The ideal state is represented as a point in space (i.e. in the contextual range) denoting an *ideal context*, which can be regarded as the actual position of the concerned fluctuating range context. By neglecting the range context's *real* position

[9]remember that $C(t_0) = \big(c_1(t_0), ...c_d(t_0)\big)$ is a vector depicting the location of context C at time t_0

[10]normalized vector with length $|\vec{d_0}| = 1$

(determined by updates from the belonging entity) and using the ideal context instead eliminates the impact of fluctuation. However, in order to determine the ideal context, the fluctuating updates from the belonging entity require that ideal context to be calculated. Multiple updates yield the context's positions at different times allowing to calculate the *mean*, which represents the ideal context, as approximated in figure 4.12.

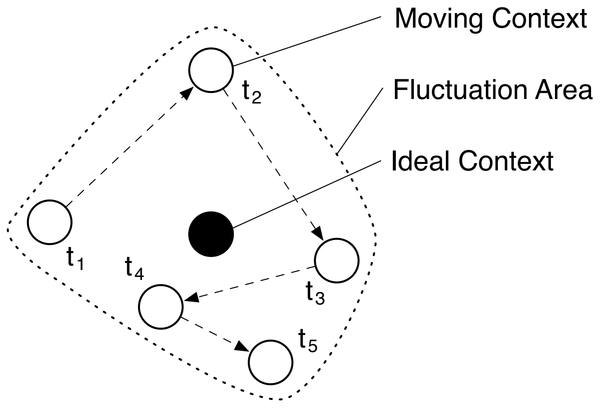


Figure 4.12: Context Fluctuating in Timeframe between t_1 and t_5

At this, the fluctuation degree deg_{fluct} represents the quality metric for the fluctuation aspect. Three basic approaches can be employed to determine it:

- The ideal state can also be employed to measure the fluctuation degree by iteratively determining the Euclidean distance $d_{\overline{CI}}(R)$ (see equation 4.2) between the real range context C and the ideal context point I. Assuming that n updates yield n distances $d_1, ..., d_n$ between C and I at n points in time $t_1, ..., t_n$, the fluctuation degree deg_{fluct} is the average of all distances:

$$deg_{fluct} = \frac{\sum_{i=1}^{n} d_i}{n} \;\Big|\; d_i = d_{\overline{C(t_i)I}}(R) \tag{4.46}$$

- Alternatively, the fluctuation degree can be determined by summing up the context's total distance travelled during the timeframe $[t_1, t_n]$ and determining its "average speed". Hence, deg_{fluct} is defined as follows:

$$deg_{fluct} = \frac{\sum_{i=2}^{n} d_{\overline{C(t_i)C(t_{i-1})}}(R)}{\Delta t} \;\Big|\; \Delta t = t_n - t_1 \tag{4.47}$$

- A third option of measuring context fluctuation is to dynamically determine the size of its fluctuation area (see figure 4.12) over particular periods of time. Since a context may fluctuate erratically, it may yield a non-symmetric polygonal area, which may be difficult to compute. A simple solution to remedy this issue is to determine the minimal bounding rectangle $M = (m_1, ..., m_d)$ of the fluctuation area with d depicting the number of dimensions of the concerned context range:

$$M = \Big(m_i \mid m_i = min\big(c_i(t)\big) \forall t \in \{t_1, ..., t_n\}, \forall i \leq d\Big) \tag{4.48}$$

with n depicting the number of points in time where a contextual update revealed the context's actual state and t_i denoting such time point. Following, deg_{fluct} equals the area of M:

$$deg_{fluct} = \prod_{i=1}^{n} m_i \tag{4.49}$$

Besides the fluctuation of stationary range contexts, fluctuations of moving contexts can be regarded, too. In that case, an ideal trajectory is iteratively calculated instead of an ideal context point. However, the detailed realization of this mechanism is out of the scope of this work.

Monitoring Techniques

After making clear the impact of context actuality and fluctuation, the major remaining question is: how to efficiently monitor numerous contexts in a contextual range over time and detect proximity among them? The solution to this issue is the application of *continuous range monitoring* mechanisms [33, 70, 80]. It is important to mention that *range* is used in a different context here, namely monitoring ranges of Euclidean space. This is not converging with the contextual map's ranges in any way. Range monitoring is applied on multi-dimensional space yielding which points are in which query ranges over time. Basically, a query range is a subspace of the entirely monitored space, which is associated with a result set of moving points currently inside that subspace.

We have to be careful not to confuse the meanings of the word "range". On the one hand, we talk about query ranges, which define areas of interest inside multi-dimensional space, on the other hand there are contextual ranges, which depict the contextual situation of a particular context type in the contextual map.

Applying multi-dimensional range-monitoring on the contextual map, each range vicinity (surrounding a range context on a contextual range) equals a query range (a part of space with an associated result set of points). Hence, we deal with equally sized[11] query ranges with an extent of the maximum threshold t_{max} defined on the contextual range (see section 4.4). The points inside the query range / range vicinity are the range contexts close to the context owning the range vicinity (located in its middle) and hence represent the result set of the range query. Concluding, when monitoring a contextual range, monitoring moving points and moving range queries is equivalent to monitoring changing range contexts and changing range vicinities. Table 4.4 shows how general range monitoring is applied on monitoring contextual ranges. In the following section 4.5.2, we discuss this approach in more detail.

[11]equally sized query ranges on the particular contextual range in question

Query range monitoring	Contextual range monitoring
multi-dimensional space	Euclidean space with Cartesian coordinate system
moving point	range context representing (partial) context of an entity
query range	range vicinity of a range context

Table 4.4: Query Range Monitoring and contextual Range Monitoring

Another important aspect of continuously observing a contextual range is the dynamic forming of context clusters. We have identified this aspect early in section 1.3 for determining a set of currently similar contexts in the contextual map. However, at this point we postpone the clustering topic and focus on monitoring non-grouped contexts. Chapter 5.2 provides an in-depth discussion about clustering contexts.

4.5.2 Monitoring multiple Range Vicinities

In section 4.5.1 we have argued that range monitoring mechanisms as a basis for monitoring range vicinities of contexts in contextual ranges. Although we have identified index structures as efficient tools for quickly querying high-dimensional data (see discussions in sections 3.3 and 4.4.2), monitoring a particular part of space over time by iteratively performing range queries on the index has significant drawbacks. Although indices enable queries to quickly locate high-dimensional data sets within a particular range of space, they are still elaborate [80]. For this reason, research has been working on finding alternatives for the monitoring use case. Prabhakar et al. have indexed the queries instead of the points employing updates on point locations on their *Query-Index* [80]. Other approaches focus on iterative mechanisms [33, 70] where each query is provided with its initial result set upon initialization and where respective result sets are iteratively updated upon point updates. In other words, there is only one huge effort to perform during the initialization when it is necessary to determine which points are in which query range. Afterwards, updating individual result sets concerned by point updates poses minimal effort. Partitioning space, which is mostly grid-like for simplicity reasons, facilitate this approach of dynamically managing query spaces, point locations and location updates [33, 70, 71]. This is strongly related to the spatial grid model introduced in section 3.3.4.

Following this short survey about range monitoring approaches, we discuss the applicability of selected principles on our contextual range setting. First, we present the theoretical approach based on our discussion in previous sections of this work. Subsequently, we deduce practical improvements based on the surveyed approaches. We begin our discussion with the approaches introduced in section 4.4, where we regard the update of a single range context C. For C, the space of interest consists of the range vicinity[12] $V_{range}(C, R)$ with the extent t_{max} (see section 4.4.2). First, we extend the scope of view from a single range context to multiple range contexts by regarding the threshold range list $T(C)$ of the updated range context C. Any change in $T(C)$ does not only concern

[12]for reasons of simplicity we regard C as the complete context as well

C, but all other changed range contexts as well. E.g., in figure 4.10 and the belonging table 4.3 we can observe the change of $T(C)$ caused by an update. The range contexts P_1, P_2, P_3, P_8 and P_9 have changed in $T(C)$ by appearing in a different set (t, Q) with t denoting a threshold and Q the set of range contexts closer to C than the threshold t. This means that all of those range contexts have possibly crossed a contextual boundary by causing a range crossing (see section 4.4.3). The update in $T(C)$ requires updates in the threshold range lists of the respective changed range contexts, too. The following cases apply:

- *Range context added:* This case occurs when a new range context P is inserted in the set Q of an entry (t, Q) of the threshold range list $T(C)$. P has come closer to C than the threshold t. That is why we need to commit this change to the threshold range list of P as well. Hence, we need to insert C into the set Q of the entry (t, Q) in $T(P)$.

- *Range context removed:* This case works analogously to the insertion case. When a range context P is removed from the set Q of an entry (t, Q) of the threshold range list $T(C)$, P has moved farther from C than the threshold t. Thus, we commit this change to the threshold range list of P by removing C from the set Q of the entry (t, Q) in $T(P)$.

In summary, a change to a threshold range list of a range context expressed by the insertion or removal of a neighboring range context also concerns the threshold range list of the inserted or removed range context.

At this point, we are able to satisfy an arbitrary number of contexts that are affected by the update of a single context. All of the affected contexts' range vicinities are up to date with their respective threshold range lists denoting the current state of their range vicinities. In summary, the following iterative process enables continuous monitoring of a contextual range:

1. Upon receiving a contextual update for a context, determine its new range vicinity $V_{range}(C, R)$ of its respective range context C by fetching all neighboring range contexts within $V_{range}(C, R)$.

2. Update C's threshold range list $T(C)$ by comparing the stored version $T_{pre}(C)$ with the current distances to the determined neighboring range contexts in $V_{range}(C, R)$. By doing so, derive an updated version of the threshold range list: $T_{post}(C)$.

3. All range contexts in $T(C)$, which have been inserted or removed from one of the tuple entries of $T(C)$ require an equivalent update in their own respective threshold range list (as discussed above). Further, those contexts have possibly crossed a boundary defined on the contextual range R. Process this event adequately.

In regard to query range monitoring, satisfying affected range vicinities equals satisfying the corresponding query ranges. By storing a threshold range list for each range context and updating it iteratively we dynamically manage the current result set of our query ranges (inspired by the approaches in [33, 70]).

The remaining issue poses the selection of contexts within the updated context's vicinity. Naturally, this depends on the choice of the underlying data structure that stores contexts, i.e. the coordinates inside a contextual range. We have two options:

- *Index on context points:* Naturally, a proven approach is to index all range context coordinates in an index structure as depicted in section 3.3. Each time when a range context is updated, the index is updated instantly by the according UPDATE-operation of the index structure in use. For the subsequent determination of the range vicinity, we perform a range query on the index (with the new range context position in the center of the query range) yielding all points, i.e. range contexts, within the queried range of space (see section 4.4.2). The index represents the master structure for a contextual range. We do not memorize context point positions anywhere else but in the index. As discussed earlier, this approach comes with pros and cons. Tree-like indices enable fast access to high-dimensional data. However, despite that, the curse of dimensionality has a significant impact on that.

- *Index on space partitions:* The authors of [80] claim that point indices perform poorly in query range monitoring. Hence, we can fall back on the alternative using the spatial grid from section 3.3.4 to index range contexts. We partition the entire contextual range's space into a grid of equally sized parts of space. In this setting, each of those cells has hypercubical shape. Each cell possesses a list of range contexts currently included in its particular part of space [20, 70].

  Upon receiving a contextual update, the affected range context may have either moved inside its cell or moved to another cell. In the former case, the range context's new position is updated in its cell's list. In the latter case, the range context is removed from its former cell's list, its new cell is determined and its current coordinates are inserted in its new cell's list. Cells can be indexed, so that they can be determined quickly.

  Capturing a range context's range vicinity $V_{range}(C, R)$ means selecting a set of cells as depicted in figure 4.13. Since rectangular regions are easy to query, we select the hypercubical set of cells that encompasses $V_{range}(C, R)$ with its extent of t_{max}. This approximation tolerates the inclusion of false positives, which are situated outside of $V_{range}(C, R)$, but inside the hypercubical range query. However, those are pruned by the following distance check, which is used to infer the new threshold range list $T(C)$ and the detection of possible contextual boundary crossings.

4.6 Contextual Boundary Management

As discussed, contextual boundaries serve as an important metric for determining contextual proximity, i.e. affinity between contexts. For this reason, context boundaries are tied in deeply into the contextual map, especially by the threshold range lists of which each is

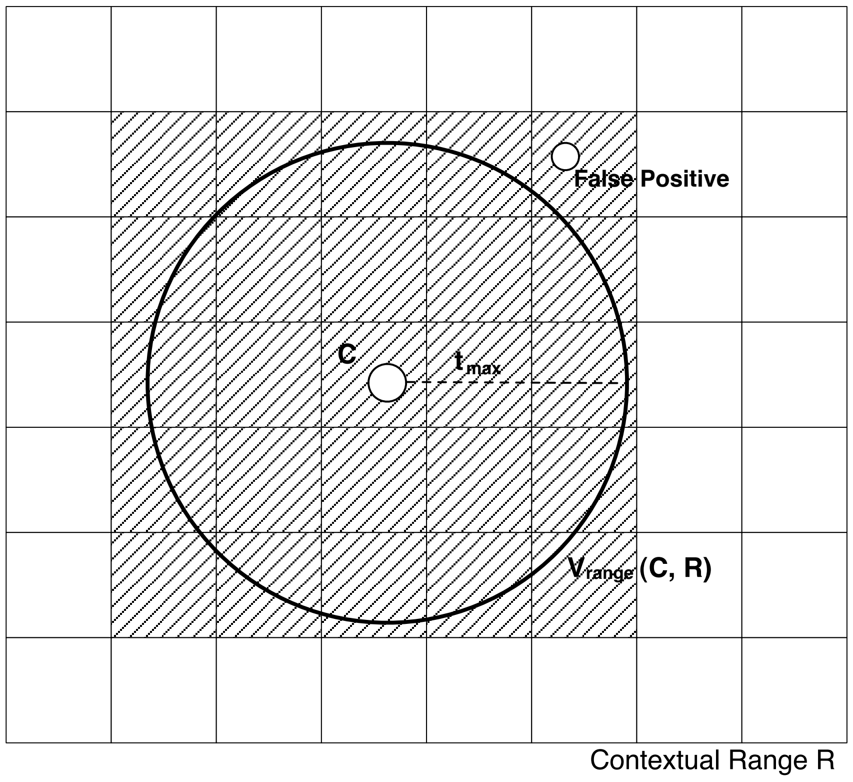


Figure 4.13: Range Vicinity $V_{range}(C, R)$ in grid-based Contextual Range R

stored with its belonging range contexts. Thus, inserting new boundaries and removing existing ones is a cumbersome task, which gets more elaborate the more contexts and ranges exist. In the following discussion, we regard the two operations of inserting and deleting a contextual boundary:

- *Insert:* Inserting a new boundary equals updating all concerned threshold range lists. A new contextual boundary has an arbitrary number of thresholds, which concern a set of disjoint contextual ranges. Each threshold concerns one of those ranges. To add the boundary to the contextual map, we need to update each of the affected ranges by registering the according threshold. Assuming a threshold t_i from a new boundary B. t_i is defined on R_i, which means that all range contexts in R_i need to be updated. In particular, this means updating their threshold range lists[13]. We distinguish three cases on how to update a single threshold range list $T(C)$ with C representing the range context whose threshold range list is being updated:
  - t_i *is minimal:* This means that there is no smaller threshold than t_i defined on R_i. For this case, the *existing* minimal threshold t_{min} is of concern. Concluding, the corresponding tuple in each threshold range list is (t_{min}, Q_{min}) (see equation

[13]updating does *not* concern the range contexts' coordinates here

4.43). We now need to determine a new entry (t_i, Q_i) for the new threshold t_i. Since Q_{min} contains all range contexts for the new set Q_i, all range contexts in Q_{min} are checked against t_i. This means that the distance between each range context P from Q_{min} and the C is determined. If the distance is below t_i, P is inserted into Q_i.

  - *t_i is neither minimal nor maximal:* This case describes the situation that t_i is in the mid-range of all thresholds defined on R_i. In this case, we are concerned about the next larger and next smaller threshold, i.e. the two neighboring threshold in terms of their values compared to t_i. Let the next larger threshold be $t_>$ and the next smaller one $t_<$ with their corresponding entries $(t_>, Q_>)$ and $(t_<, Q_<)$. The difference between $Q_>$ and $Q_<$, i.e. $Q_> \cap Q_<$ includes all candidates for Q_i. Checking all candidates' distance to C yields the result set for Q_i by including all candidates with the that distance being smaller than t_i.
  - *t_i is maximal:* This case occurs, if there is no larger threshold defined on R_i, i.e. $t_i > t_{max}$. In the former two discussed cases, we have taken advantage of existing entries in $T(C)$ since the range vicinity $V_{range}(C, R)$ has stayed unchanged. This approach is insufficient in this case, since the $V_{range}(C, R)$ needs to be extended and there may be new range contexts to be included. All entries in $T(C)$ remain, but the new entry (t_i, Q_i) needs to be determined by a hyperspherical range query centered at C with extent t_i. It yields the result set Q_i, allowing to complete $T(C)$ with t_i becoming the new maximum t_{max}.

- *Delete:* The deletion of a context boundary is comparatively easy. Deleting (t_i, Q_i) from $T(C)$ suffices, since all other tuples in $T(C)$ remain unaffected.

As stated, the *Insert* and *Delete* Operations depicted above describe processes that need to be conducted on every range context (we have denoted it as C in the discussion above) of every context range affected by the new boundary.

4.7 Overall Workflow

With the basic working principle of the contextual map model discussed, we are now about to summarize the general workflow of detecting affinities between contexts.

1. *Context capturing*: The first step consists of abstracting the environmental context into the context model. The contextual map has to be set up according to this context, i.e. ranges and dimensions are to be defined (section 3.1). Subsequently, the initial context is mapped into the contextual map. This includes the identification of entities possessing their own definable contexts, typifying that context, identifying and quantifying distinct contextual attributes (by the context capturing interface as depicted in figure 2.5), and finally mapping them into the contextual map (section 3.2). As a result, every entity possesses a *position* in the map, corresponding to its initial context.

2. *Setting of contextual boundary:* In the next step, the contextual boundaries have to be defined, so that contextual affinity detection becomes possible. This includes the identification of ranges affecting the according boundary and defining the delimiter thresholds on those ranges (section 4.1.1). Contextual boundaries may change upon a use-case-specific demand (section 4.6).

3. *Definition of update semantics:* The entities' terms for committing updates about their most current contexts must be defined. If adopting zone-based semantics, this includes the selection of the appropriate bounding model for the update zones in $\mathbb{R}^n$, in particular hypersphere and hyperplane models (section 4.2). The size of the update zones (hypersphere = diameter, hyperplanes = distance between each other) is application-specific and subject to dynamic changes.

4. *Monitoring current local context:* It is assumed, that a mobile host is attached to each entity. It keeps track of the entity's current context. If zone-based update semantics are applied, a contextual map representation of its entity is maintained locally. The mobile host captures the most current context from its context sensors and mapped into its local contextual map (section 3.2) as described in step 1. Thus, the entity's changing context corresponds to changing coordinates of its contextual position in the map. Since local context monitoring is used for zone-based update semantics, it is irrelevant if other than zone-based update semantics are in use.

5. *Determine contextual update:* If zone-based update semantics are in use, the constant local monitoring of the entity's context enables the determination to commit a contextual update. This is exactly the case, when the entity's context leaves the update zone in its local contextual map (section 4.2). If other than zone-based update semantics are employed, the time for a contextual update is determined accordingly. In either case, if considered necessary, a contextual update is committed to the system by the entity's mobile host.

6. *Determine relevant vicinity:* The global contextual map model receives the entity's update (see figure 4.1) and prepares to examine its impact on the global context, i.e. if the entity's context got more or less affine to other contextually proximate contexts. First the affected context boundaries are identified, then the relevant vicinity of the entity's context in the global contextual map is determined (section 4.4). This vicinity, which is split up on the contextual map's ranges, defines the relevant *space* of proximity examinations.

7. *Contextual affinity detection:* The distances from the entity's context to all contexts in the entity's relevant vicinity are determined using the Euclidean distance formula (equation 4.2 in section 4.1.2). This process is conducted on each range affecting a relevant context boundary.

8. *Trigger contextual proximity and separation alerts:* The calculated distances are checked with the affected contextual boundaries. If a change of distance equals a crossing of a boundary (section 4.1.1), a proximity or separation alert is triggered.

9. *Application-specific action:* The proximity or separation alert is processed application-specifically. This may include committing the recently identified context changes to various entities, i.e. committing an inverse contextual update.

10. *Global context monitoring:* The changes in the global context, which are continuously caused by incoming contextual updates sent from entities, are monitored and processed accordingly (section 4.5). Upon receiving an update, the workflow from step 4 to step 9 is iterated. Context boundaries managed upon demand (section 4.6).

Figure 4.14 illustrates the workflow described above as a UML activity diagram.

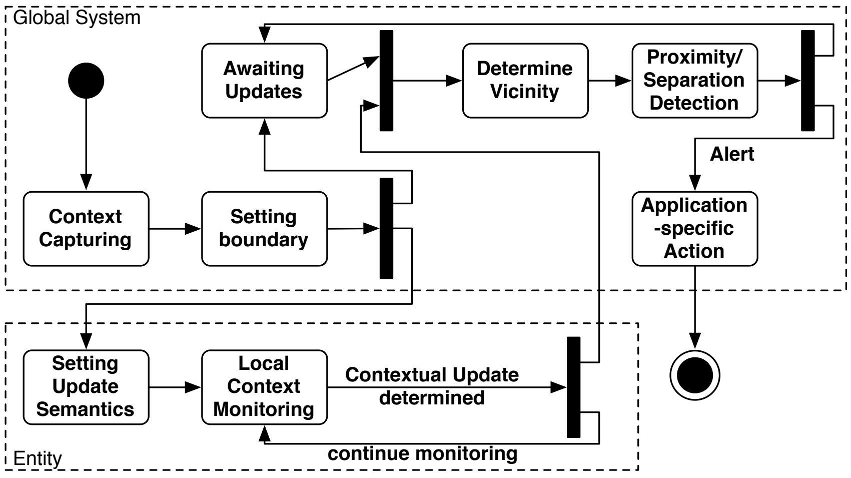


Figure 4.14: Workflow

Chapter 5

Exploiting contextual Similarity

So far, we have extensively explored the aspect of similarity between *two* contexts. In this chapter, we explore the possibilities to utilize arbitrary sets of similar contexts. In section 1.3 we have identified two main application cases of the contextual map (see figure 1.2):

- the application of the *contextual similarity query*
- and the determination of *context clusters*

In section 5.1, we explain the working principle of the contextual similarity query. We proceed with section 5.2 focusing on the determination of groups of similar contexts, the context clusters. We close with an analysis of how to interpret clusters of similar contexts as situation bounds in real-world-contexts in section 5.3. We present the notion of the *super situation* (as introduced in section 1.3) derived from context clusters and depicting a particular contextual realm, i.e. a state space for certain contexts (i.e. situations).

5.1 Contextual Similarity Queries

Given a query context C, the similarity query delivers a result set containing all contexts that are similar to C. In this chapter, we focus on the actual application of the contextual similarity query. We distinguish two variants, which differ by their definition of the similarity degree, i.e. the metric for defining the query's bound:

- similarity degree set by *contextual boundaries*, i.e. a contextual boundary is used as query parameter and the bound of a similarity query is inferred from that boundary (discussed in section 5.1.1)
- similarity degree set by a *custom parameters*, i.e. the bound of the similarity query is set directly (discussed in section 5.1.2)

5.1.1 Boundary-defined Similarity Query

Following our discussion so far, the contextual map's metrics for bounding similarity are grouped into contextual boundaries. Recalling, a boundary B defines thresholds on those contextual ranges that are of semantic concern for B. Hence, assuming $B = (t_1, ..., t_m)$ defining m thresholds on m contextual ranges with t_i denoting a threshold defined on range R_i (see section 4.1.1).

We start with regarding the query context C, which depicts the query point for the contextual similarity query. Since we regard multiple contextual ranges, we need to carefully distinguish between *contexts* and *range contexts*, with range contexts existing in a particular contextual range only, whereas contexts comprise of multiple range contexts on all (applicable) contextual ranges. In section 3.1, we have introduced this basic data model. To perform a similarity query on C, we need to regard all range contexts of C on the according ranges R_i that are affected by B. As stated, B defines a threshold t_i on each of those ranges. Since B is the query's only parameter by defining the required similarity degree, we need to identify all range contexts in the radius t_i on each of the contextual ranges R_i affected by B. Hence, we divide the similarity query in m sub-queries, with each sub-query executed on a range R_i (i.e., each range that a threshold t_i is defined on with $B = (t_1, ..., t_i, ..., t_m)$). With this approach we identify similar range contexts of C on each affected range. So, we proceed as follows on each affected range by executing the according sub-query: we regard the range context of C on the particular range R_i as the query point. Using the *crc*-Operator, the corresponding range context is $C_{range} = crc(C, R_i)$. Now, we are interested in all range contexts that are similar to C on that range, i.e. the range contexts that are contextually proximate to C_{range}. The similarity degree is defined by the corresponding threshold t_i in B. Hence, since t_i is defined on R_i, it is included in C_{range}'s threshold range list[1] $T(C_{range})$. The corresponding entry in $T(C_{range})$ is the tuple (t_i, Q_i) with Q_i denoting all range contexts closer than t_i to C_{range}. For the range R_i, Q_i almost depicts the result set for the sub-query responsible for R_i. Since the contextual similarity query is performed on inter-range level, it aims at contexts, not only range contexts. Thus, the corresponding contexts of all range contexts in Q_i yields the result set of the sub-query for R_i. We can express that with the *rcc*-Operator. Thus, the result set for the sub-query on range R_i is the intersection of all m result sets yielded by the sub-queries:

$$rcc(Q_i) \tag{5.1}$$

We conduct such a sub-query on each of the m ranges depicted by B. As a result, we get m result sets $rcc(Q_i)$ with $i \leq m$. Taken together, those result sets contain all contexts similar to context C in any regard to boundary B. From this set, we need to extract those contexts that are similar to context C entirely in regard to boundary B, meaning that those contexts are similar to C in regard to *all* thresholds defined in B. Hence, the result set of the contextual similarity query is:

[1] t_i is actually included in every range context's threshold range list

$$CSQ(C,B) = \bigcap_{i=1}^{m} \Big(rcc(Q_i)\Big) \;\Bigg|\; B = (t_1, ..., t_i, ..., t_m), (t_i, Q_i) \in T\big(crc(C, R_i)\big), i \leq m \tag{5.2}$$

In some cases it may be practical to determine a set of similar contexts in regard to multiple existing context boundaries. In that case we have to regard all thresholds (and ranges) defined in those boundaries and perform a sub-query on each corresponding range. In this approach, it may occur that multiple thresholds are defined on a single range. Ranges are only exclusive in a boundary definition, not across boundaries. In that case, we employ the maximal threshold of all thresholds defined on a range by the boundaries employed by the similarity query. This way, every boundary is satisfied by the query. Formally:

$$CSQ\big(C, (B_1, ..., B_m)\big) = \bigcap \Big(rcc(Q_i)\Big) \;\Bigg|\; \begin{array}{l} \bigcup_{j=1}^{m} B_j = \big\{t_i\big\}^+, j \leq m, \forall i : \\ \big(max(t_i), Q_i\big) \in T\big(crc(C, R_i)\big) \end{array} \tag{5.3}$$

5.1.2 Custom-parameter-defined Similarity Query

In the previous section, we have employed contextual boundaries as a bound for querying similar contexts. Contextual boundaries are suitable for this purpose, because they are deeply tied into individual contexts' data structures (such as the threshold range lists). However, in some cases it may be desirable to execute a contextual similarity query, which is not described by any existing boundary. So far, the approach to solve the problem would be to create a new boundary, which suits the desired similarity query. But since inserting a new boundary into the contextual map is highly elaborate, it may not outweigh the benefit of a custom similarity query that does not rely on any contextual boundaries.

To avoid employing a context boundary, we first need to specify our custom parameters for the similarity query. We define the similarity degree e_i for each range R_i that needs to be considered. Grouped together, the array of those similarity degrees is actually no different than a context boundary with its thresholds. However, in this case, the similarity degrees are completely unknown to the contextual map. None of them appears in any threshold range list. For this reason, we call them similarity degrees in the *parameter array* and explicitly not thresholds in a contextual boundary. Thus, the parameter array groups the similarity degrees and maps each similarity degree to its belonging range.

With the parameter array of similarity thresholds given, we execute a sub-query on each considered contextual range (as already explained in section 5.1.1). Since the range contexts' threshold range lists are useless in this case, we need to extract the similar range contexts from different data structures. Depending on the indexing structure employed on range contexts, we conduct a range query either on the context range's spatial-point-index or on the context range's spatial grid. In section 3.3 we have introduced both of those options. Given a similarity degree e_i (for range R_i) from the parameter array $E = (e_1, ..., e_i, ..., e_m)$, we perform one of the following according operations on the belonging contextual range:

- *Index tree:* In this case, all range contexts are stored in a high-performance indexing tree as discussed in section 3.3.1 (an index tree is always responsible for one contextual range). We perform a hyperspherical range query with extent e_i (radius) on the index, so that all range contexts in the query range, the space of radius e_i, are determined.

  A hypercubical range query of extent $2e_i$ (edge length) may be less elaborate and hence executed alternatively. However, this approach requires a post-processing step for the elimination of false positives, which may be outside the hyperspherical query range. We do this by a Euclidean distance check on the range contexts in the result set and the query context.

- *Spatial grid:* In case a contextual range's range contexts are stored in a spatial grid, we select the set of cells that encompass the hyperspherical region of extent e_i, similarly as depicted in figure 4.13. Since this is a hypercubical range query, we need to prune the false positives in a second post-processing step here, too.

Assuming that each sub-query yields a result set S_i for its belonging range R_i, the result for the whole similarity query is determined analogously to equation 5.2, namely by the intersection of the result sets S_i with $i \leq m$:

$$CSQ(C, E) = \bigcap_{i=1}^{m} \Big(rcc(S_i)\Big) \Bigg| \, E = (e_1, ..., e_i, ..., e_m) \tag{5.4}$$

5.2 Clustering Contexts in the Contextual Map

Contextual clustering depicts the second application case utilizing groups of similar contexts as depicted in section 1.3. *Context clusters* can be defined as sets of contexts sharing contextual affinity, hence providing a grouping criterion for contexts in the contextual map. There are two basic parameters for detecting clusters:

- The *maximum distance*, which may exist between any two contexts in the cluster. We label it the *inner cluster distance* throughout this chapter.
- A selection of *contextual ranges*, which a cluster includes. The contexts' coordinates relevant for clustering are interpreted on range level, as with the other mechanisms introduced so far. However, a cluster definition may span over multiple ranges. This implies, that the clustering criterion (maximum distance) is applied on each range, since the Euclidean distance metric is applicable on range level only (i.e. on a single range at a time).

Returning to the example with the cautious driver in section 4.1.1, assume that there are multiple drivers all concerned about the agreeable weather conditions along their routes. They all individually define an individual (and fictional) weather condition, which they regard as critical. Since all of those weather conditions are positioned in the contextual map, similar entries can be detected as clusters on the environmental measurement range. With such a cluster detected, the contextual map can give recommendations about defining a critical weather condition for new drivers, for example.

In this section, we describe the mechanisms of contextual clustering applied on the contextual map. After surveying basic clustering algorithms in subsection 5.2.1, we explain how those can be applied on the contextual map in subsection 5.2.2. The focus is put on how to cluster contexts, which span across multiple contextual ranges. We close with a discussion about two important aspects that concern the detection of context clusters in the contextual map: the aspects of dynamic changes and high dimensionality discussed in subsections 5.2.3 and 5.2.4, respectively.

5.2.1 Clustering Algorithms

This section presents a brief review of three clustering algorithms that may be employed for the contextual map. Clustering algorithms are applicable for any Euclidean space with an arbitrary number of dimensions. In this section, we introduce three algorithms.

k-means Clustering

K-means clustering aims at identifying k clusters in a set of n multidimensional points [65]. The algorithm starts with randomly selecting k points, which serve as initial centroids of the clusters. The clusters are determined by an iterative process by first adding all remaining points to the cluster with the nearest centroid and recalculating each clusters new centroid by calculating its *mean*. The iterations are conducted until convergence is reached, i.e. until all means remain unchanged. In detail, the following steps are conducted:

1. Given n multi-dimensional points, k initial points are selected for the k resultant clusters. Each of the k points represents the initial *mean* of its belonging cluster.
2. Iteratively repeat the clustering procedure by assigning each point to its nearest cluster:
   (a) For each of the n points, determine the closest mean and assign the point to the cluster that this particular mean belongs to.
   (b) With all n points assigned to one of the k clusters, the mean of each cluster is recalculated.

   This procedure is iterated until no changes occur anymore. Upon recalculating a cluster's mean, it is possible that the mean moves (step 2b in iteration i). A moved mean may imply that new points are added to the cluster from other clusters or that points included in the cluster are assigned to other clusters with more proximate means (step 2a in iteration $i + 1$). A different point constellation in a cluster automatically implies a different mean (step 2a in iteration $i + 1$). Eventually, all clusters' means converge to a final position yielding the final clusters.

Hierarchical Clustering

Hierarchical clustering focuses on representing all given points in a hierarchical tree of clusters. The root represents a cluster encompassing all given points. The leaf nodes represent individual points. All nodes in between root and leaves represent clusters on different hierarchical levels in the tree. The hierarchical level of a cluster node depicts its vertical position in the tree (as measured from the root to the leaves). The metric for determining hierarchical levels of cluster nodes is the maximal distance d_h between points belonging to a cluster node (maximal *inner cluster distance* on a particular level in the *h*ierarchy)[2]. Thus, a cluster node depicts a group of points where no point in the cluster is farther away from another point in the same cluster than d_h, whereas d_h defines the hierarchical level inside the tree. Figure 5.1 depicts a hierarchical clustering example.

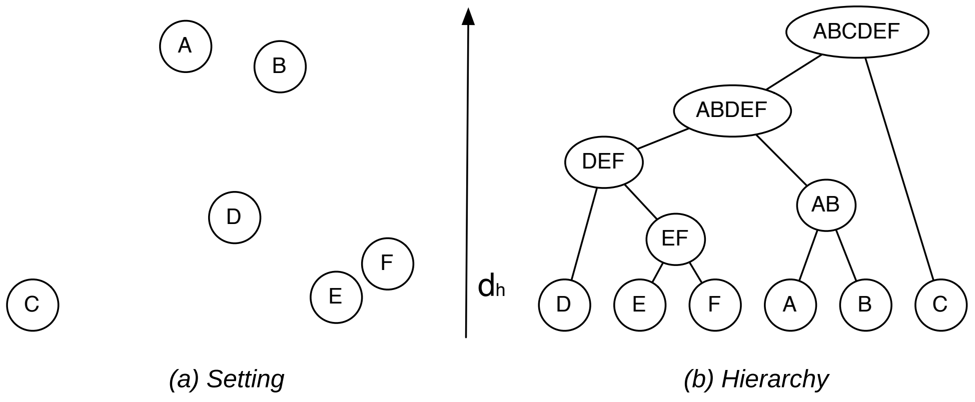


Figure 5.1: Hierarchical Clustering

The hierarchical cluster tree can be constructed by two approaches:

- *Agglomerative (bottom-up):* In this approach the hierarchy is constructed from the leaves to the root. We start regarding individual points and group them to clusters subsequently. As stated, the hierarchy level of a cluster node is dependent on the maximum distance d_h between the points belonging to the cluster denoted by the node. The bottom-up approach requires to start with the lowest hierarchy level (the leaves) with $d_h = 0$. Hence, only nodes depicting individual points populate the lowest level. They can be interpreted as one-element clusters. Subsequently, we move up the hierarchy, i.e. increasing d_h. With d_h increasing, we particularly observe the neighborhood of identified clusters. This inevitably leads to finding other clusters within the current maximal distance of clusters, i.e. the current value of d_h. In such a case, when another cluster is identified within the current distance d_h, both cluster nodes are merged into a shared parent node. Thus, a parent node always represents a cluster consisting of its two merged child cluster nodes. The process

[2]here, d_h is a dynamic depicting the maximal inner distance on a particular hierarchy level, thus changing when regarding a different level

terminates when all clusters have been merged to the root node encompassing all existing points.

- *Decisive (top-down):* The opposite approach constructs the hierarchy from the root to the leaves. d_h is initialized as the maximum distance between two neighbors (e.g., distance between C and D in figure 5.1a). Starting with the hierarchy's root node encompassing all points, d_h is continuously decreased and cluster nodes are consecutively broken up according to the current value of d_h. Once the inner distance between points in a cluster node exceeds d_h, the cluster node is broken up into several child nodes (usually two[3]) depicting clusters with an inner distance not exceeding d_h. This process terminates upon having determined all points as leaf clusters at the bottom of the hierarchy tree.

Minimum-spanning-tree Clustering

This clustering approach identifies clusters among a minimal tree spanned along all existing points in the observed space [102, 43]. In the first step, a minimum spanning tree (MST) along all points is created. Thus, this tree $T = (V, E)$ consist of vertices V that denote the points in the observed space and edges E spanned along them, whereas the sum of all edge weights is minimal. There exist multiple algorithms for constructing an MST [32, 60, 74, 82] of which the ones of Kruskal [60] and Prim [82] are the two most common ones that serve as bases for most optimizations. An MST is a simple graph with no spatial characteristics with only vertices and edge lengths known. The points' coordinates are unconsidered. In Euclidean space, the MST's vertices are points in space. Hence, the MST manifests itself as a multi-dimensional structure with its nodes and connections (i.e. vertices and edges) aligned at their according multi-dimensional positions, hence having a spatial shape. It is called the Euclidean minimum spanning tree (EMST). Thus, the EMST is an extension of MST, which represents the EMST as a traditional graph $T = (V, E)$, a plain tree with weighted edges that has no spatial characteristics. This means that the actual coordinates of the EMST's points represented by the vertices in V are not preserved in the original MST. Contrarily, the edges' lengths in the EMST are preserved as edge weights in E, however. Figure 5.2 illustrates the coherence between the MST and its spatial counterpart, the EMST, based on the same observed space as introduced in figure 5.1.

For the clustering algorithm, both MST and EMST represent sufficient structures, although the MST depicts the less complex structure of both. Given an MST spanned between all existing points in the data set and a parameter d denoting the maximal tolerated distance among points within a cluster, all edges with a length exceeding d are erased. Thus, T collapses into multiple subtrees, whereas each subtree depicts a cluster. Figure 5.3 illustrates this clustering procedure based on the setting from figure 5.2.

More sophisticated variants of the algorithm employ a different threshold parameter to prune edges. E.g., the authors of [43] employ the deviation from the average edge

[3]if there is exactly one pair of points whose distance is d_h; the number of children equals the number of sub-clusters divided at pairs of points with exact distance d_h

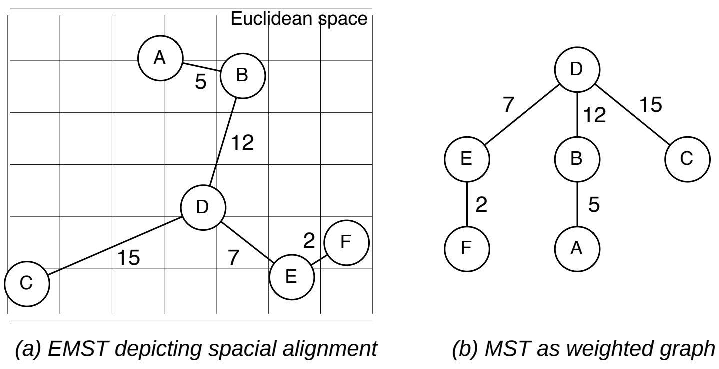

Euclidean space
A
B
C
D
E
F
5
12
15
7
2
(a) EMST depicting spacial alignment
(b) MST as weighted graph

Figure 5.2: Minimum Spanning Tree Representations

length[4] in the entire MST as the threshold parameter. If an edge length deviates from the average edge length more than the parameter, it is pruned.

For our forthcoming discussion, we focus on the EMST representing a spatial minimum-spanning-tree among an arbitrary number of multi-dimensional points in Euclidean space (as depicted in figure 5.2a).

5.2.2 Contexts and Range Contexts

After having surveyed common clustering algorithms, this section focuses on applying them on the contextual map model. Similarly as with context monitoring (section 4.5), context cluster detection is conducted in a bottom-up approach.

1. First, clustering algorithms are employed on range contexts on each of their respective contextual ranges.

2. After this, clustered range contexts are inferred to their respective range-global complete contexts, so that range-global clusters can be determined.

Clustering Range Contexts

A contextual range is an Euclidean space with its n-dimensional vectors representing range contexts depicting a dedicated context type in the real world (e.g. environment on R_{env} as presented in chapter 3). We start the context cluster determination on contextual ranges by clustering range contexts according to a traditional clustering algorithm, e.g. as presented in section 5.2.1. Generally, we are looking for an arbitrary number of clusters in the context range, leaving us the hierarchical and MST-based clustering approaches. For more specific use cases, when a certain number of clusters is to be determined, k-means clustering and its derivatives can be employed. Whatever case applies, the clustering algorithm yields a set of clusters, of which each contains range contexts that are similar to each other, i.e. contextually proximate. In the subsequent discussion we focus on finding

[4]here, the average edge length is the average distance between a point and its closest neighbor

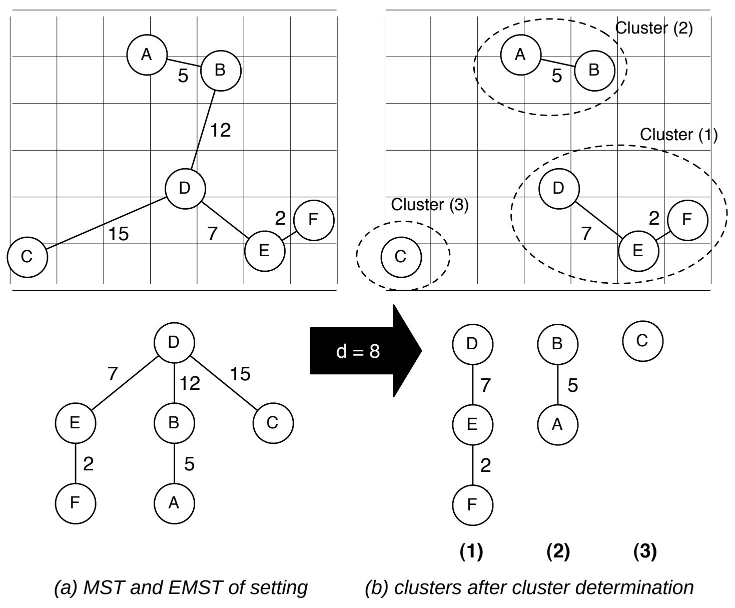

A
B
C
D
E
F
5
12
15
7
2
d = 8
Cluster (1)
Cluster (2)
Cluster (3)
(1)
(2)
(3)
(a) MST and EMST of setting
(b) clusters after cluster determination

Figure 5.3: Minimum Spanning Tree Clustering with $d = 8$

an arbitrary number of clusters given a threshold t depicting the maximum distance among range contexts in a cluster, thus briefly discussing hierarchical and MST-based clustering.

Hierarchical clustering As stated, hierarchical clustering decomposes into its agglomerative and decisive variants. Both construct a hierarchical tree of clusters with a cluster's inner distance defining the hierarchical level.

- *Agglomerative:* The bottom-up variant starts with individual range contexts and iteratively groups them to clusters. In the beginning, each range context represents a cluster for itself depicting a leaf in the hierarchy. The key activity in building up the hierarchy is an iterative nearest surrounding search conducted on individual range contexts. The surrounding of each range context belonging to a cluster is searched for the closest range context that is part of another cluster. Upon finding such a range context, both clusters are merged in their parent node, which is created in this step. This parent node represents the cluster merged out of the two child-clusters, thus consecutively constructing the hierarchy from the bottom up. As stated, this search is conducted iteratively while the radius of the scanned surrounding is continuously increased according to the current hierarchy level, i.e. the maximum inner cluster distance d_h, respectively. Eventually, a cluster encompassing all range contexts will emerge as the hierarchy's root node.
- *Decisive:* The top-down approach starts with the root cluster encompassing all range contexts. The root cluster is iteratively split up into sub-clusters with decreasing inner distance d_h. The procedure is split into two phases:

1. First, all distances between range contexts have to be determined. With m range contexts this equals $\frac{m^2-m}{2}$ distances to be calculated[5]. The resultant distances are ordered yielding an ordered list of range context pairs. The top of the list denotes the range context pair with the longest distance between each other in the entire contextual range.
2. We initialize the parameter d_h with the distance at the top of the list. The list's range context pairs are consecutively deleted from the top of the list proceeding down the list. d_h is continuously decreased and a range context pair is deleted once the distance between them becomes larger than the current value of the decreasing d_h. To determine if the deleted pair's range contexts denote a cluster split, we need to find out if both range contexts belong to the same parent cluster and if they belong to different clusters with an inner distance not exceeding the current d_h. If both of those conditions hold, there is a cluster split. Assume, that a pair of range contexts (A, B) has been deleted and both A and B belong to the same known cluster (the cluster node has already been assigned to the hierarchical tree). Further, A now possibly belongs to cluster X_A and B possibly to a different cluster X_B, where the inner distance in both clusters does not exceed d_h. To check this assumption, we create a symmetric relation R from the remaining list and extend it to its reflexive transitive closure R^*. If $(A, B) \in R^*$ holds, there is no cluster split. Otherwise A and B are in different clusters with their inner distance not exceeding the current d_h. The two clusters are determined with the help of R^*. A range context C is assigned to one of the two cluster splits X_A or X_B, respectively.

$$C \in X_A \Leftrightarrow (A, C) \in R^* \tag{5.5}$$

$$C \in X_B \Leftrightarrow (B, C) \in R^* \tag{5.6}$$

   With all affected range contexts (those from the parent cluster) assigned to either X_A or X_B, both clusters become child nodes of the regarded parent cluster node in the hierarchical tree.

With the hierarchy constructed, we need to determine the range context clusters for our given threshold d, which depicts the maximum inner distance we allow in a cluster. Starting at the hierarchy root, we search the tree by using a depth-first-inspired search. We descend the tree recursively. Every time upon finding a cluster node whose hierarchy level d_h satisfies the condition $d_h \leq d$, we remember this node and stop the recursive descent. All of those remembered node depict our range context clusters, because they are the largest clusters, which just about satisfy our condition $d_h \leq d$.

[5]every selection of two out of m range contexts: $\frac{m!}{2!(m-2)!} = \frac{m(m-1)}{2} = \frac{m^2-m}{2}$

MST-based clustering This algorithm splits up into two parts: First, constructing the Euclidean minimum spanning tree (EMST) among all known range contexts, and second, using this tree-structure to determine clusters in the contextual range. It is to be reminded that the EMST differs from the plain MST only by explicitly including coordinates of vertices.

1. *Constructing the EMST:* In order to find the EMST of a set of range contexts, we need to define a graph $G = (V, E)$ to perform the MST-algorithm on. Obviously, the range contexts represent the vertices V. The naive approach to define the edges E is to define an edge between all pairs of range contexts, hence interconnecting all range contexts, and perform an MST-algorithm on G. Since this approach handles a potentially enormous number of edges (namely $\frac{m^2-m}{2}$, as explained earlier), we focus on two ways on how to reduce the edge count. In the first approach, we define G with a reduced number of edges with G containing the EMST of all range contexts. The *Delaunay triangulation* represents such fast-computable graph, from which the EMST can be extracted by using a standard MST-algorithm (e.g. Prim or Kruskal as mentioned in section 5.2.1). The second approach builds up the EMST directly from scratch using a nearest surrounding search.

   - *Delaunay triangulation:* Since both Prim's [82] and Kruskal's [60] algorithms require a graph as input, the key idea is to insert a pre-processing step by efficiently reducing the number of edges in the graph containing all range contexts (its vertices). A Delaunay graph $DT(P)$ is a triangulation of a point set P, so that no more than three points are cocircular [103]. $DT(P)$ can be calculated in $n \log n$ time [45]. With P denoting all range contexts of a contextual range R, $DT(P)$ can be efficiently determined so that the EMST spanning the entire range context set P can be determined by the algorithm of Prim [82] or Kruskal [60], respectively.
   - *Incremental nearest surrounding search:* Prim's [82] and Kruskal's [60] algorithms (and many others) require a graph from which they determine its MST. Our second approach is inspired by Prim's algorithm [82], but the MST is built from scratch (without a graph) and continuously extended by choosing the shortest (i.e. least-weighted) outgoing edge, which does not yet belong to the MST. Following this principle, the EMST is built up by iteratively extending it until all range contexts are covered. We start by randomly choosing a single range context, which represents the EMST root (or more precisely, the first vertex, since an MST does not need a dedicated root node). After that, we iteratively extend the tree by finding the range context, which is nearest to the EMST and adding it to the EMST by inserting a minimal edge. The algorithm terminates upon all range contexts added to the EMST.

2. *Perform clustering:* Using the EMST, we cluster the contextual range according to the clustering parameter d, which may represent the maximum inner cluster distance or the maximum tolerated distance deviation between two range contexts

in absolute or relative regard to the standard edge-length deviation in the EMST. To do so, we iteratively remove edges that do not comply with d, i.e. which are too long. The resultant forest, which is created by continuously splitting up the EMST, represents the resultant clusters with each tree representing a single cluster.

Clusters may become large and irregular in shape. This implies that two range contexts on opposite sides of a cluster's rim may not be close, i.e. contextually proximate, at all (figure 5.4 shows an example of an irregular cluster with two non-proximate range contexts). For this reason, it is practical to subdivide a large cluster into multiple smaller clusters in which range contexts are contextually proximate to each other. Hence, in addition to the maximal tolerated distance among neighboring range contexts within a cluster (inner distance), the maximum tolerated distance among all range contexts must be introduced as an additional parameter.

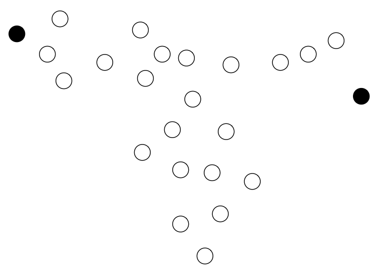


Figure 5.4: Example of an irregular Context Cluster

Assuming, we are given those two parameters: the maximal inner distance d and the maximal proximity p, we first identify clusters in the contextual range using d. Clustering is conducted as discussed above by employing the hierarchical or MST-based clustering technique. After that, each cluster is analyzed if it contains range contexts whose similarity to each other exceeds p. This is done by iteratively performing the following two consecutive steps on an individual cluster C:

1. *Determine non-proximate range contexts:* First of all, a pair of range contexts with a similarity exceeding p has to be determined within the cluster. Formally speaking, regarding a contextual range R, a pair of range contexts (P, Q) must be determined that satisfies the following condition:

$$d_{\overline{PQ}}(R) > p \mid P, Q \in C \tag{5.7}$$

with $d_{\overline{PQ}}(R)$ denoting the Euclidean distance between P and Q on range R as defined in equation 4.2 In theory, all distances among range context pairs would need to be known. If clustering was performed on the basis of the decisive hierarchical approach all of those distances are already known since this clustering approach requires them in advance (see the preceding discussion). Practically, we only need to know distances between contexts at the rim of a cluster, since they are sufficiently meaningful to determine non-proximate range context pairs. If having employed MST-based clustering, the leaves of the cluster's MST (i.e. the MST of C, which is a subtree of the MST spanned over the entire range) depict the range contexts at the cluster rim, hence pairs of those may hold the condition 5.7.

2. *Split cluster:* Having found a non-proximate arbitrary range context pair (P, Q) that satisfies condition 5.7, C is split into three new disjoint clusters. Beginning at P and Q, a nearest surrounding search with radius p is conducted around each of those range contexts. Both searches yield all range contexts in the ambit of P and Q, respectively (vicinity of radius p). We denote those result sets S_P and S_Q, respectively. The range contexts in both result sets denote two of the new clusters that C is split up to. We denote the new clusters C_P and C_Q, respectively:

$$\left.\begin{array}{l} C_P = S_P \cap C \\ C_Q = S_Q \cap C \end{array}\right| \forall P, Q \in (S_P \cup S_Q) : d_{\overline{PQ}}(R) \leq p \tag{5.8}$$

All remaining range contexts in C that are neither part of C_P nor C_Q are part of the remaining cluster split:

$$C_{rest} = C \setminus (S_P \cup S_Q) \tag{5.9}$$

C_{rest} represents the cluster that still may contain non-proximate range contexts in regard to p whereas C_P and C_Q do not. Thus, the process is iteratively repeated on C_{rest} (and each subsequent resultant C_{rest})[6] until no two range contexts exist in C_{rest} whose distance between each other does not exceed p.

After finishing, the initial cluster C is split up into multiple clusters, so that none of the split sub-clusters contains non-proximate range contexts that satisfy condition 5.7.

Consider the exemplary cluster C from figure 5.4. The range contexts on the cluster's rim are not proximate to each other if facing opposite sides of the cluster - thus not contextually similar to each other. Therefore, we subdivide C into multiple smaller clusters so that each range context is similar to its belonging cluster's range contexts in a degree not exceeding p. In the first step, we choose two range contexts A_1 and B_1 at opposite sides of the cluster rim and determine two appropriately sized clusters that include A_1 and B_1. We denote those clusters C_{A_1} and C_{B_1}, respectively, as depicted in figure 5.5. Both are sub-clusters of C. The remaining range contexts in C, which are not contained in neither sub-cluster, are grouped to the "rest-cluster" C_{rest_1}.

During the first step, we have determined two new clusters where no two range contexts are less similar than p (in terms of similarity degree) to each other while being members of the same cluster (logically, not C_{rest_1}, since this is the "rest-cluster"). We repeat this step iteratively on the rest-cluster until the rest-cluster contains no two range contexts that are less similar than p. In our example, one more step suffices. C_{rest_1} is split into C_{A_2} and C_{B_2}, but since C_{rest_1} is small enough, the algorithm terminates as depicted in figure 5.6.

[6]with C_{rest} becoming C in the next iteration

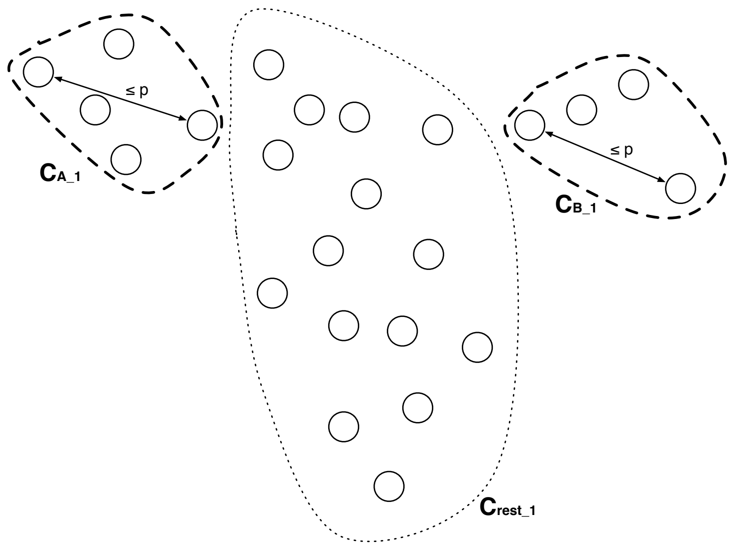


Figure 5.5: Splitting an irregular Context Clusters - Step 1

Clustering range-global complete Contexts

Clustering contexts that consist of multiple range contexts on multiple contextual ranges is inspired by the technique of mapping range contexts to complete contexts in a contextual similarity query - as discussed in section 5.1.2. The initial setting consists of many clusters of range contexts on many ranges. In order to derive range-global context clusters, we need to identify those range-global contexts, which are clustered in contextual proximity on range-level (hence, clustered in the given range context clusters).

First of all, we need an ordered representation of clusters, sorted by contextual range. For this purpose, we introduce the following definition:

$$cluster(R) = \{C_1(R), ..., C_i(R), ..., C_n(R)\} \tag{5.10}$$

defines all n clusters $C_i(R)$ of range contexts on contextual range R with $i \leq n$. Next, we need to specify a *relevance array* (similar to the parameter array introduced in section 5.1.2) depicting the contextual ranges that are relevant for the current range-global clustering process. The array $E = (R_1, ..., R_m)$ specifies m contextual ranges whose clusters will be considered. With the relevance array given, range-global context clusters are derived from range-specific context clusters by grouping contexts to clusters according to the cluster memberships of their range contexts. Given two contexts P and Q together with a relevance array E, they are part of a cluster if the following condition holds:

$$\exists j : crc(P, R_i) \in C_j(R_i) \wedge crc(Q, R_i) \in C_j(R_i)\ \forall R_i \in E, i \leq m \tag{5.11}$$

depicting that the range contexts[7] of P and Q are part of the same range context cluster on each relevant contextual range defined by E. Determining the context cluster equals

[7]crc-operator derives range contexts from range-global contexts

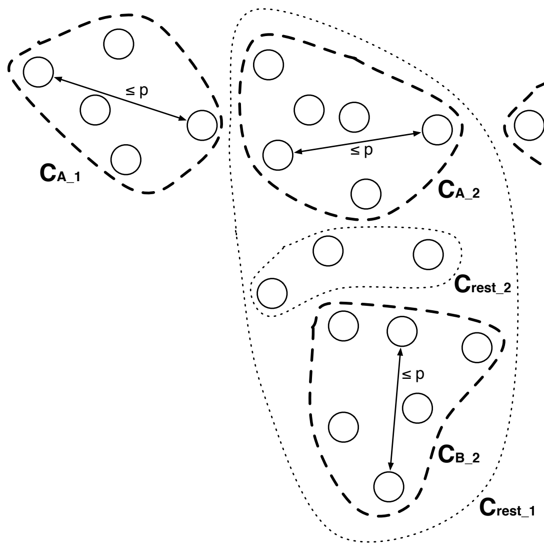


Figure 5.6: Splitting an irregular Context Clusters - Step 2 (final)

cross-combining all range context clusters on different ranges and derive the corresponding range-global clusters. We conduct this operation with the help of the Cartesian product of all $cluster(R_i)$ sets:

$$\prod_{i=1}^{m} cluster(R_i) \ni \big(C(R_1), ..., C(R_i), ..., C(R_m)\big) \;\Big|\; C(R_i) \in cluster(R_i) \wedge R_i \in E \;\forall i \leq m \tag{5.12}$$

This Cartesian product contains all possible permutations of different ranges of which each entry is an ordered tuple. As an example, consider three ranges with range context clusters as depicted in figure 5.7 and the Cartesian product of those range context clusters as derived in table 5.1. Each tuple element $C(R)$ represents a cluster of range contexts on range R.

Contextual Range	Clusters in Range: $cluster(R)$
R_1	$\Big\{C_1(R_1)\Big\} = \{A, B, C, D, E\}$
R_2	$\Big\{C_1(R_2), C_2(R_2)\Big\} = \Big\{\{A, D\}, \{B, C, E\}\Big\}$
R_3	$\Big\{C_1(R_3), C_2(R_3)\Big\} = \Big\{\{A, C, D\}, \{B, E\}\Big\}$
$\prod_{i=1}^{3} cluster(R_i) =$	$\left\{ \begin{matrix} \Big(C_1(R_1), C_1(R_2), C_1(R_3)\Big), \Big(C_1(R_1), C_1(R_2), C_2(R_3)\Big), \\ \Big(C_1(R_1), C_2(R_2), C_1(R_3)\Big), \Big(C_1(R_1), C_2(R_2), C_2(R_3)\Big) \end{matrix} \right\}$

Table 5.1: Example for Cartesian Product of Range Context Clusters

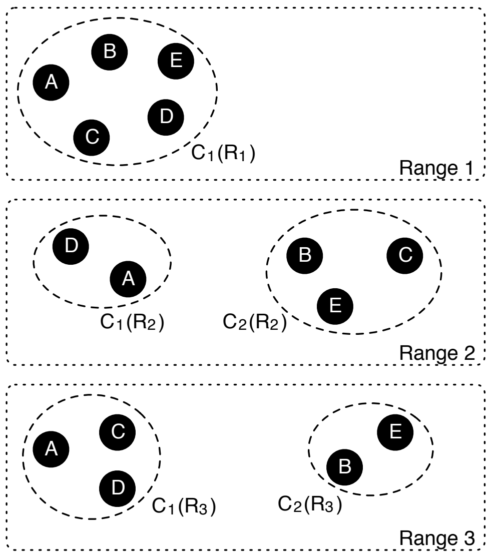


Figure 5.7: Example Context Clusters

We assume that the parameter array $E = (R_1, R_2, R_3)$ affects all three ranges. With the Cartesian product given, each tuple depicts a possible range-global context cluster. If a context appears in *each* of a tuple's elements, it is part of each range-local cluster depicted by the tuple. This argumentation implies that exactly those contexts that are part of each of the tuple-denoted range cluster are clustered on each relevant range. Those contexts compose the range-global context cluster.

However, we have neglected a neat separation between range contexts and complete contexts during this argumentation. Since each tuple in the Cartesian product represents a range-local cluster, we proceed as follows. First, we derive the range-global contexts[8] from all range contexts in the Cartesian product. Each tuple now contains elements, of which each depicts complete contexts clustered on a particular range. Second, regarding a single tuple, we intersect all of the tuple's elements (which still depict range-local clusters), so that we get a single range-global context cluster:

$$C_{context} = \bigcap_{i=1}^{m} rcc\big(C(R_i)\big) \;\Big|\; \big(C(R_1), ..., C(R_i), ..., C(R_m)\big) \in \prod_{i=1}^{m} cluster(R_i) \tag{5.13}$$

Hence, the complete set of all range-global clusters can be inferred by applying this procedure on all tuples of the Cartesian product:

$$(C_{context})^* \left| \begin{array}{l} C_{context} = \bigcap_{i=1}^{m} rcc\big(C(R_i)\big)\ \forall\big(C(R_1), ..., C(R_i), ..., C(R_m)\big) \in \prod_{i=1}^{m} cluster(R_i) \\ \wedge\ C(R_i) \in cluster(R_i) \\ \wedge\ R_i \in E\ \forall i \leq m \end{array} \right. \tag{5.14}$$

[8]rcc-operator derives range-global contexts from range contexts

Applying this procedure on the example in figure 5.7 with table 5.1, we denote the intersections taken in table 5.2. This table shows the intersections of the elements in each of the Cartesian product's tuples, thus the context clusters.

Tuple	Context clusters
$\big(C_1(R_1), C_1(R_2), C_1(R_3)\big)$	$\{A, D\}$
$\big(C_1(R_1), C_1(R_2), C_2(R_3)\big)$	$\emptyset$
$\big(C_1(R_1), C_2(R_2), C_1(R_3)\big)$	$\{C\}$
$\big(C_1(R_1), C_2(R_2), C_2(R_3)\big)$	{B, E}

Table 5.2: Example for Cartesian product of Range Context Clusters

It is to be noted that many of the intersections in equations 5.13 and 5.14 may yield an empty set, which means that there is no cluster for the according Cartesian product tuple. In our example in table 5.2, it occurs once. We also have a 1-element cluster $\{C\}$ in the result. Depending on the application case, this may not be contextually relevant. For this reason, a context-aware application may only be interested in context clusters with a minimum size or larger.

5.2.3 Dynamic Clustering

In section 4.5 we have outlined the mechanisms to dynamically monitor the context map with its contextual ranges. We have described how to efficiently handle a potentially high number of simultaneous contextual updates. At this point, we continue the discussion by introducing cluster monitoring in a dynamically changing contextual map.

As argued, changes in the contextual map are caused by contextual updates affecting contextual ranges. Contextual updates affect contexts and range contexts, which both form clusters in the contextual map. Hence, dynamic changes in range contexts imply dynamically changing context clusters. In this section, two disjoint approaches are presented: Iterative clustering and individual cluster updates.

Iterative Clustering

In section 5.2.2, we have sketched possible clustering algorithms applied on contextual ranges. We have employed tree-based data structures to determine range context clusters, namely minimum spanning trees and hierarchies. In theory, dynamical management of those data structures enables dynamic cluster monitoring in turn. The data structures are *iteratively* adapted according to incoming context updates. This approach yields the following considerations in regard to hierarchical- and MST-based clustering:

- *Hierarchical:* Maintaining a cluster hierarchy dynamically means to rebuild its branches locally. Recalling the hierarchical clustering algorithm from section 5.2.1, the hierarchy is a binary tree with each node having its hierarchy level stored. Given a contextual update, it has to be determined from which node to rebuild the tree. There is exactly one path between the leaf-node denoting an updated range context C and the root node denoting the entire range context set. A contextual update on C invalidates nodes on that path beginning at the leaf denoting C. Nodes on that path are invalidated, if all clustered range contexts depicted by an affected node depict an inner cluster distance exceeding the node's hierarchy level (a contextual update on C may cause C not to belong to clusters anymore, which are on the path between the$C$-leaf and the hierarchy root). Since clusters are grouped together in the hierarchy with increasing hierarchy level, we need to find the first node on the described path for which the clustering condition still holds, i.e. which is not invalidated, assuming we begin searching from the C-leaf. The hierarchy tree needs to be rebuilt beginning at this node (i.e. all of its child nodes need to be re-determined). Analogously, the search for this node can be conducted from the opposite direction of the path beginning at the hierarchy's root.

- *MST-based:* Dynamically maintaining an EMST of a set of range contexts decomposes into dynamical determination of Delaunay triangulations and dynamic MST-algorithms. As stated in section 5.2.2, a Delaunay triangulation greatly reduces the graph among range contexts to determine the EMST from. Hence, dynamically updating the Delaunay triangulation - as presented by the authors of [44] - may significantly facilitate dynamically maintaining the according EMST. This approach can be further enhanced by dynamic MST-algorithms, which have been studied extensively [19].

It is to be noted that maintaining the support tree - both a hierarchy and an MST - still requires the conduction of the clustering procedure both range-specifically and range-globally, as depicted in the previous section 5.2.2.

Individual Cluster Update

Dynamically maintaining the according clustering algorithm's support structure (hierarchy or MST) is an elaborate task, which gets more demanding the more contexts are maintained in the contextual map. For single contextual updates, it is more practical to pursue a different approach. Assuming that an initial set of contexts has been clustered using an algorithm of choice initializing the monitoring process. Changes in clusters are determined following individual contextual updates. Following this approach, a single contextual update is regarded and checked if it has changed any cluster configuration. The process can be decomposed as follows:

1. *Initialization:* The context clusters are initialized using a clustering algorithm of choice.

2. *Monitoring:* Dynamically maintaining context clusters occurs contextual-range-specifically. Upon receiving a contextual update, all affected ranges are checked for range-specific context cluster changes before deriving range-global context cluster changes. The following process is conducted continuously:

(a) A received contextual update contains information about the changed context C of an entity. The data may span across multiple contextual ranges.

(b) On each affected contextual range R: the new coordinates of the belonging range context, say $P = crc(C, R)$, are determined. It is checked, if P still belongs to its cluster $C_P(R)$ by determining the distance $d_{\overline{PQ}}(R)$ to its nearest neighbor Q and checking if $d_{\overline{PQ}}(R)$ against the maximum tolerated inner cluster distance d. The following cases are possible[9]:

- $d_{\overline{PQ}}(R) \leq d \wedge Q \in C_P(R)$: P still belongs to its cluster, to which it was belonging before the update:

$$P \in C_P(R) \tag{5.15}$$

- $d_{\overline{PQ}}(R) \leq d \wedge Q \notin C_P(R) \wedge Q \in C_Q(R)$: P does not belongs to the cluster, to which it was belonging before the update, anymore. It belongs to the cluster, to which its nearest neighbor belongs:

$$P \notin C_P(R) \wedge P \in C_Q(R) \tag{5.16}$$

- $d_{\overline{PQ}}(R) > d$: P does not belong to any existing cluster since it is too far away from any neighboring range context: $P \notin C_i(R)\ \forall i : C_i(R) \in cluster(R)$. It represents a new one-element cluster C_{new} that is added to the range's cluster set $cluser(R)$:

$$P \in C_{new}(R) \wedge C_{new}(R) \in cluster(R) \tag{5.17}$$

(c) The final step consists of partially performing range-global clustering (as discussed in the previous section 5.2.2) for all Cartesian product tuples (as depicted in definition 5.13) that include a range with a change of clusters (i.e. the cases denoted by definitions 5.16 and 5.17).

Evaluation

At this point, we have identified two possible ways of updating cluster configurations in the contextual map:

- Updating the support structure and perform the appropriate clustering algorithm on it (iterative clustering)
- Individual update of known cluster configurations (individual cluster update)

Concluding the discussion of both mechanisms, iterative clustering is much more expensive than individual cluster updating. Since iterative clustering has global scope, i.e. it is applied on the entire contextual range, all range contexts are clustered first range-specifically and the range-globally as described previously in section 5.2.2. On the other

[9]it is to be noted that $P \in C_P(R)$ has been true *before* the contextual update

hand, individual cluster updates are performed locally by pinpointing the "location" of a cluster change and committing it to the affected clusters. Range-specific clustering does not occur and the range-global clustering steps can additionally be reduced to the affected contextual ranges.

Thus, the individual cluster update approach clearly wins the cost-per-update evaluation. However, it is only capable of processing *one* single update at a time. Given the circumstances of a high contextual updates emergence, processing those updates and updating the cluster setting in the contextual map in real-time may become problematic. In that case, storing contextual updates and process them jointly in a single update step at certain time intervals may prove reasonable. Hence, applying iterative clustering in such a scenario remains the better alternative, since that updating procedure is capable of handling an arbitrary amount of contextual updates. Table 5.3 summarizes the applicability of both approaches.

Update Variant	Locality	Drawbacks	Benefits
Iterative Clustering	global	expensive: in each iteration, both range-specific and range-global clustering is conducted over entire contextual range	capable of processing multiple contextual updates in a single updating step, e.g. updates at particular intervals
Individual Cluster Update	local	each iteration handles only one single contextual update	low cost per update, efficient range-global update mechanism

Table 5.3: Comparison of Context Cluster Update Mechanisms

5.2.4 High-dimensional Data Clustering

The range-specific clustering step may be complicated by a high dimensionality count. Earlier in section 3.3, we have stated that high dimensional data causes their accessing operations to scale poorly - a circumstance that is also called the "curse of dimensionality". Clustering high dimensional data is no exception to this aspect. This means that context clustering in the contextual map gets inevitably impeded by the dimensionality curse if the number of dimensions is high. For context-aware applications that provide rich context (meaning many contextual attributes are given) the contextual map must be declared with a sufficient number of dimensions. In that case, scalability problems must be expected.

However, the issue has been studied and approaches to remedy the problem have been devised. All approaches agree in preprocessing the full-dimensional search space prior to cluster detection. Two approaches are presented in the following discussion:

- cluster-search in dimensional subspaces
- 2-pass clustering using cheap metric pre-selection

Cluster-search in dimensional Subspaces

Kriegel et al. [59] provide an overview about a family of clustering techniques, which focus on finding clusters in dimensional subspaces, i.e. spaces defined by a subset of dimensions out of the full number of dimensions (*not* to be confused with hyperspatial subspace of the full-dimensional space). This basically means that the search space is reduced to hyperplanes in the full-dimensional space. Since the complexity increases exponentially with a linear growth of dimensions, the individual examination of more feasible subspaces counteracts the "curse of dimensionality". The final clustering result for the complete search space is consolidated from the individual clustering results of the subspaces. Hence, subspace clustering decomposes into three steps:

1. *Find all relevant subspaces:* This step aims at determining *which* dimensional subspaces are to be selected for clustering.
2. *Find clusters in subspaces:* Clustering is performed on each subspace. Since each of the subspaces has a significantly reduced search-space in comparison with the full-dimensional search space, clustering is computationally far less demanding. In addition, the number of subspaces scales linearly - not exponentially as dimensions do.
3. *Cluster determination:* The final clustering result for the complete search space is consolidated from the individual clustering results of the subspaces.

The determination of suitable subspaces is basically influenced by the orientation of the according subspace's hyperplane. Such subspaces are categorized into either axis-parallel or arbitrarily oriented hyperplanes. In the following, we present a brief overview on subspace clustering. For an in-depth discussion, we refer to the survey in [59] and the research referenced there.

Clustering in axis-parallel subspaces Subspaces of this kind are oriented axially parallel to axes denoting dimensions of the full-dimensional search space. Hence, the number of axis-parallel subspaces is finite, albeit still large. Categorizing clustering techniques based on axis-parallel subspaces needs to be conducted aspect-oriented. The first aspect is the *general* approach to the problem, namely three in total:

- *Projected clustering:* All possible subspaces are determined. Each point is assigned to exactly one subspace cluster.
- *Soft projected clustering:* A "softer" variant of the former approach does not require "hard" assignment of points to clusters. Further, the number of clusters is given in advance (as with k-means clustering, compare section 5.2.1).
- *Subspace clustering*[10]*:* As the most straightforward approach, it requires the determination of all possible subspaces and checking them for clusters.

[10]this approach is confusingly named identically with this general clustering approach, since cluster determination in dimensional subspaces is widely referred to *subspace clustering* as well

The second aspect in categorizing axis-parallel clustering techniques is a more *specific* fine-coarsed algorithmic view. It denotes whether clusters are determined from top-down or bottom-up:

- *Top-down:* Subspaces are determined from full-dimensional space and clusters are derived from local neighborhoods, meaning that local searches for neighbors are the driving approach.
- *Bottom-up:* First, all 1-dimensional subspaces (smallest kind of all) are determined. We now make use of the following observation: If subspace S contains a cluster, then any subspace $T \subseteq S$ must also contain a cluster. The reverse implication, if a subspace T does not contain a cluster, then any superspace $S \supseteq T$ also cannot contain a cluster. Thus, this observation can be used for pruning superspaces, that is, excluding specific subspaces from consideration, until convergence is reached. Thus, actual clustering is performed on only a fraction of possible subspaces.

Clustering in arbitrarily aligned subspaces Since the orientation of those hyperplanes is arbitrary, there is an infinite number of possible subspaces, so identifying and searching all of them in infeasible. For this reason, clustering on those subspaces is also called *oriented clustering*[11]. Further, it has been observed that attributes that points accommodated on a common hyperplane appear to follow linear dependencies among the attributes participating in the description of the hyperplane. Since linear dependencies result in the observation of strong linear correlations among these attributes, we call this type of clustering also *correlation clustering.* The general approach of determining arbitrarily oriented subspaces that are searched for clusters is to employ Principal Component Analysis [59]. That, however, is out of the scope of this work.

Pattern-based clustering Another interesting approach in dimensional subspace clustering is to represent all points in the search space as a point-attribute matrix A, where rows usually depict the points in the data set and columns the attribute values of the according points. Cluster detection is based on finding sub-matrices denoting *patterns* in A. Thus, a sub-matrix J represents an identified subspace cluster. An in-depth discussion about pattern-based clustering algorithm requires an extensive excursion into that domain and thus out of scope of this work. For further reading, Kriegel's survey on subspace clustering [59] is emphasized.

Clustering using cheap Metric Pre-selection

McCallum et al. [68] propose a clustering technique, which reminds the *filtering and refining* principle from section 3.3.2. It is a 2-phase clustering technique, which greatly decreases the complexity of clustering a high-dimensional data set:

[11]The name comes from the arbitrary *orientation* of the subspaces

1. In the first phase, a cheap approximation metric is defined to partition the data set into multiple subsets, of which each includes elements similar to each other in regard to the selected metric. Thus, this metric approximates clusters by quickly partitioning the data set into overlapping subsets, called *canopies*. This is done by applying the metric on all elements, and group those elements that are similar to each other in regard to the metric. As a result, a canopy includes elements that are candidates for a cluster. Since all of those cluster candidates are approximated, canopies are *not disjoint*. The loss in accuracy accounts for an according gain in performance. Thus, each canopy may include false positives, i.e. cluster candidates that actually do not belong to the cluster represented by the canopy.

   The important aspect about this pre-processing step is the cost of the metric, which can be a cheap distance metric or classification metric. This enables quick (hence inaccurate) pre-selection of data entries for according clusters.

2. The second phase conducts the actual clustering on individual canopies. This time, the clustering metrics need to be accurate making the second phase more expensive. However, clustering is performed on individual canopies, not on the entire data set. This greatly reduces the effort by minimizing the number of cluster candidates. False positives are pruned by the clustering process yielding a set of clusters after processing each canopy. Each canopy yields one cluster at most.

This approach can unquestionably be applied to contextual range clustering. The challenge here is to define an appropriate approximation metric for phase 1. Possible choices include the Manhatten distance (another metric in Euclidean space, although cheaper than Euclidean distance) or a context classification metric. Due to the impact of heterogeneity in contextual attributes (which are eventually mapped to contextual range dimensions), this remains an application-specific issue. Phase 2 is conducted using the Euclidean distance metric.

5.3 Context Clusters and Contextual Realms

In the previous section we have explained how context clusters are determined. However, the semantic meaning of context clusters remains to be clarified. This section discusses the relevance of context clusters in regard to context-awareness. As we have outlined in the introductory section 1.3, clusters of contexts define a situational realm in multi-dimensional space depicting a bounded set of possible contexts. So this time, we are not considering individual contexts that are collated with a possessing entity, but a particular space that an entity context can exist in. Thus, in the contextual map this space represents regions spanned by numeric ranges of the map's dimensions. We have dubbed such a region of space a *contextual realm*. It includes a particular set of possible situations, i.e. contexts. The coherence of contexts, situations and context clusters are subject of this section.

5.3.1 Contexts, Situations and Contextual Realms

Throughout this work, we have defined a context as a single *point* in multi-dimensional space. Since a context can basically manifest itself anywhere in the contextual map, it is reasonable to introduce multi-dimensional *regions* as an additional spatial structuring criterion to improve context classification. Such a region in the contextual map may be considered as a *contextual realm* bounding possible context[12]. That way, contextual realms define multi-dimensional areas in the contextual map, which depict specific contextual characteristics. E.g., considering the weather station example from chapter 3, we can define a region in the hyperspace of the environmental contextual range R_{env} depicting bad weather by defining appropriate ranges of values on all four dimensions belonging to R_{env} (see table 3.1). We can call this example region as the "contextual realm depicting bad weather". An entity with its range context on R_{env} situated inside this realm expresses bad weather as part of its overall context. From here on, we refer to *contextual realms* as regions of spaces inside the contextual map.

As we have stated in sections 2.2 and 2.3, a single context represents the real-world situation of an entity in an abstract way [21, 78]. Thus, extending this mapping on contextual realms, each realm depicts a "super-situation" encompassing all possible real-world situations corresponding to it (much like an abstract set of acceptable situations complying with specific contextual parameters depicted by the contextual realm, e.g. "bad weather"). Such contextual realms can be defined in two ways:

- *Cluster-defined:* On the one hand, contextual realms can be defined by contextual range clusters. Clusters of range contexts occupy a particular region of space. This space can be defined a contextual realm. Naturally, such a realm is *derived* from the overall context situation of the observed environment, since the contexts included in those clusters define the realm by the union of their respective situations.
- *Arbitrarily defined:* On the other hand, contextual realms can be defined freely. Defining a range of values for each individual dimension of the contextual map yields a single contextual realm. In contrast to cluster-defined realms, those realms are not dependent on the current overall context situation of the environment, but rather on *arbitrarily defined* parameters following particular contextual characteristics (e.g. the "bad weather" example).

It is to be noted that contexts inside a contextual realm are usually similar to each other. Thus, contextual affinity can be exploited by using contextual realms accordingly. As we have stated earlier, contextual affinity is examined on multiple contextual ranges. For this reason, contextual realms can also exist on multiple ranges. In that case, we face a set of spatial regions, i.e. "sub-realms", with each region defining a situational space on a particular contextual range. Assuming that a contextual realm encompasses m sub-realms and each sub-realm defines a space A_i on its range R_i. The contextual realm C is then defined as the set of those sub-realms:

[12]see also the theory of context spaces introduced by Padovitz et al. [75] outlined in section 2.1

$$C = \Big\{A_i \; \forall \; i \leq m\Big\}^+ \tag{5.18}$$

Section 5.3.3 presents an in-depth discussion about an example realm encompassing multiple ranges. In the remainder of this section, we refer to the term *contextual realm* for both realm types: the singular contextual realm composed of a single space on one contextual range, as well as the inter-range contextual realm, which decomposes into multiple singular realms on individual contextual ranges (one per range). We do so, because the latter case is a generalization of the former. A contextual realm defined as one single space on one particular contextual range is a contextual realm with $m = 1$ in regard to definition 5.18.

5.3.2 Cluster-defined Contextual Realms

As stated in the previous section 5.3.1, contextual realms can be derived from context clusters. In this section, we give a detailed discussion about this type of contextual realm including their relevance to high-level context-awareness.

Definition and Properties

As we have argued earlier in section 5.2.2, context clusters consist of contexts, which are part of the same range context clusters on the contextual range level. Practically speaking, the global contexts' range contexts span the contextual realm on each range. However, the actual shape of the contextual realm (on range level) is not necessarily trivial. Figure 5.8 shows possible forms of a contextual realm spanned by a context cluster on a particular range. The actual shape is highly application-specific. Also, the illustrated variants differ in computational complexity. While 5.8a and 5.8b can be determined fairly easily by determining the minimums and maximums of the range contexts' dimensional values, 5.8c consists of many connected hyper-cuboids. 5.8d is probably the least feasible option of deriving the contextual realm from the cluster.

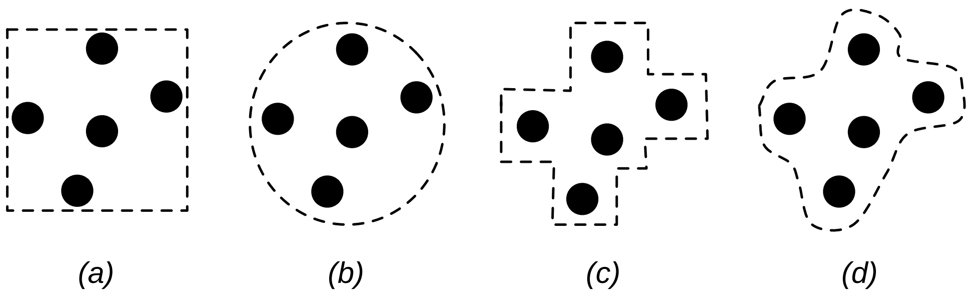


Figure 5.8: Cluster-defined contextual Realm Shapes

Following this argumentation, two basic attributes can be associated with a cluster-derived contextual realm:

- *Space:* We have briefly discussed the possibilities on how to define the *shape* of a cluster-defined realm. Complementary speaking, the *size* or *volume* of the realm is another attribute. Naturally, the determination of the realm's size is dependent on its shape:
  - *Hyper-rectangular (figure 5.8a):* Determining the minimum and maximum on each relevant dimension of clustered contexts yields the bound of the hyper-rectangle bounding the context realm. Its size can easily be determined.
  - *Hyper-elliptic (figure 5.8b):* The minimum hyper-elliptic space encompassing the cluster defines the contextual realm.
  - *Hyper-polygonal (figure 5.8c):* In the given example, the context realm decomposes into multiple hyper-rectangular subspaces. Hence, the overall size is determined by summing up the subspaces' individual sizes. Different hyper-polygonal shapes, which may have application-specific origins, can be handled analogously.
  - *Irregular (figure 5.8c):* For reasons of completeness, this variant is listed here, too. However, determining its size is highly application-specific and thus, out of scope of this work.
- *Context density:* The realm's context density is simply derived from the number of contexts in the cluster and the size of the contextual realm. The size can be determined by geometrically calculating the volume of the realm's shape (see enumeration above).

Until now, we have been talking about clusters and contextual realms on contextual range level, i.e. contextual realms derived from range context clusters. After the preceding discussion, the reasons should be obvious: the determination of those realms is dependent on a uniform axis unit, which is only guaranteed on intra-range level. Regarding a range-global context cluster, we can observe that each of its range context clusters (i.e. a cluster that groups range contexts on a particular range as part of a range-global cluster) spans a single contextual realm on the particular range (the space that is occupied by the particular range context cluster). Hence, all of those range-specific realms can be grouped together and considered as part of an inter-range super-realm. Basically, we regard each of the range-specific realms as a sub-realm of the inter-range super-realm. For the global context cluster, this is the contextual realm, which is composed of its individual sub-realms with each sub-realm depicting the space spanned by a range cluster on its corresponding context range. Figure 5.9 shows an imaginary inter-range contextual realm, which consists of three sub-realms on three correspondent contextual ranges, each of them derived of a cluster on range level. In section 5.3.1 we have already discussed such inter-range realms and defined them in equation 5.18.

Contextual Relevance

After having defined and discussed the physical specifics of cluster-derived contextual realms, it remains to be discussed what they represent in terms of context-aware appli-

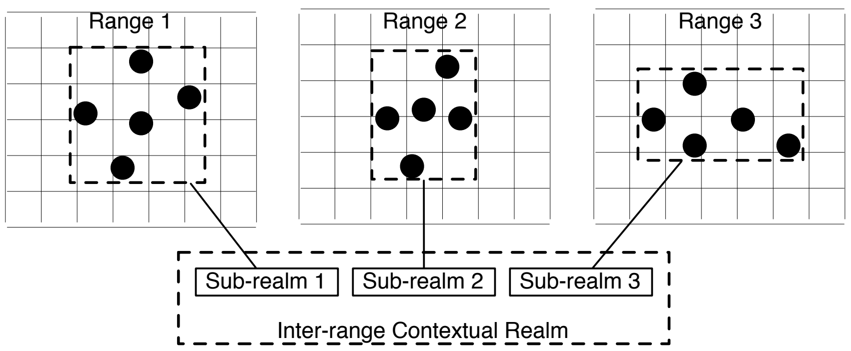


Figure 5.9: Range-global cluster-derived contextual Realm

cability. A cluster-derived contextual realm represents a set of particular contexts, which are clustered, i.e. similar to each other in terms of clustering. For context-aware applications, such a contextual realm is of significant interest. It states that there are entities with similar contexts and that their contexts exist in exactly the space defined by that contextual realm. The space is bounded and since it is cluster-derived, it is densely populated by entity contexts. The context-aware application is given the ability to specifically focus on adapting its services for entity situations in this specific context realm. The context-density is an additional metric for the interest of a context-aware application for the context realm in question. The higher it is the more contexts are present in smaller space, enabling the context-aware application to conduct service adaption for a smaller realm but serving a higher number of entities.

Following this argumentation, it is reasonable to define a "focus point" for each contextual realm, which serves as a representative context for context-aware applications. Such a *representative context* does not need to be a real context associated with an entity. It can be a fictional context consisting of contextual attributes, which describe the context cluster (i.e. the context realm) best. Consequently, for a singular context realm on range-level the *cluster's mean* represents a suitable point for a range-local representative context (comparable to the ideal context in section 4.5.1, e.g. figure 4.12). For globally cluster-derived realms with multiple sub-realms, the range-global representative context is composed of the range cluster means of the ranges belonging to the context realm. Finally, context-sensitive application can use that representative context to adapt to the entire associated context realm. Since all contexts in the realm are similar to each other, adapting to the representative context (cluster mean) only instead adapting to all individual contexts in the cluster is feasible.

With the representative context defined, we can enrich the contextual realm by introducing two additional metrics expressing its convergence to the representative context on range-level. For now, we stick to the singular context realms on range-level:

- *Individual context convergence:* The convergence (or deviation) of an individual range context in the context realm to the representative context can be defined by determining the Euclidean distance between the two on range-level. Consequently, this metric states how well the "cluster context" or "realm context" fits an individual range context, or how well context-aware applications adapt to that individual range context when adapting to the representative context. Since context clusters can be derived to contextual realms, membership in a cluster is to be regarded equivalently with the inclusion in the corresponding context realm, hence making the convergence metric equivalent for cluster membership, too.

- *Contextual realm convergence:* The convergence of the entire context realm to the representative context can be described by the average deviation of range contexts inside the realm to the representative context on range-level. It can be viewed as a quality metric of the realm, since less deviation means that context-sensitive application serve the realm's range contexts better, i.e. they tailor their services more fittingly to the entities with their contexts inside that realm. In other words, this quality statement can be made for a set of clustered range contexts, since context clusters can be derived to contextual realms on range-level.

Extending those convergence metrics from the enumeration above on inter-range realms is not feasible, since the Euclidean distance metric is only applicable on contextual range level. For this reason, convergence can only be stated in regard to a specific sub-realm. For contexts in such inter-range realms, their convergence can only be stated for a particular sub-realm on a contextual range. The same principle applies for the contextual realms themselves. Realm convergence can only be expressed individually for each sub-realm.

5.3.3 Arbitrarily defined Contextual Realms

In some cases it is reasonable to define regions arbitrarily in the contextual map. The presence of contexts inside such contextual realms is irrelevant for their definition. Since we do not use context clusters (which are strictly structured according to contextual ranges) to define those realms, we can handle the context range separation less strictly. Defining an arbitrary contextual realm requires the selection of m contextual ranges, with an arbitrary number of coherent subspaces (i.e. subspaces compose a sub-realms on range level) defined on each range. The spatial coherency constraint forces the realm to be connected "in one piece" on each of the m affected ranges. In regard to contextual proximity it is reasonable not to spatially break up a contextual realm. In contrast to cluster-derived realms discussed previously in section 5.3.2, arbitrary defined realms depict inter-range contextual realms in the contextual map.

To formally define an arbitrary contextual realm, we introduce the set of subspaces belonging to a inter-range realm C on contextual range R. We call this set *range_realm* encompassing r subspaces A_i on range R:

$$range\_realm(R, C) = \bigcup_{i=1}^{r} A_i \tag{5.19}$$

This leads us to the definition of the complete inter-range contextual realm C, which is an ordered set of subspaces of all m ranges:

$$C = \{range\_realm(R_i, C) \ \forall \ i \leq m\} \tag{5.20}$$

The arbitrary definition of a contextual realm takes the opposite way of defining a context realm by the presence of a context cluster, as described in section 5.3.2. The latter approach derives a realm from an *existent* contextual setting whereas the arbitrary approach depicts a *possible* contextual setting. This argumentation implies to foresee possible situations and describe them with an according contextual realm. In section 5.3.1 we have sketched an exemplary contextual realm depicting bad weather conditions. Consider the contextual map example, which has been introduced in chapter 3 and specified in table 3.1: a location range R_{loc} and environmental readings range R_{env}. To demonstrate an arbitrary context realm we define it for the contextual setting "bad weather in Bavaria" (which is a state in Southern Germany). For R_{env} we define the contextual space for "bad weather conditions", as depicted in table 5.4 (for reasons of simplicity we use the mapping defined in the associated mapping function from definition 3.7). This yields a (hyper-)rectangular sub-realm E on R_{env} with a size of 15×20 (as defined in table 5.4). For R_{loc}, we define the contextual space that covers Bavaria. We do that by defining seven rectangular sub-realms L_1 through L_7 for R_{loc} as depicted in figure 5.10[13]. The single space defined on R_{env} and the seven spaces defined on R_{loc} define the context realm:

$$\begin{aligned} C_{weather} &= \Big\{range\_realm(R_{loc}, C_{weather}), range\_realm(R_{env}, C_{weather})\Big\} \\ &= \Big\{L_1 \cup L_2 \cup L_3 \cup L_4 \cup L_5 \cup L_6 \cup L_7, E\Big\} \end{aligned} \tag{5.21}$$

Dimension	Minimum value	Maximum value
dim_{temp}	45 ($-5°C$)	60 ($-10°C$)
dim_{hum}	80 (80%)	100 (100%)

Table 5.4: Example Realm on R_{env}

With arbitrarily defined context realms context-aware application can tailor their services to situations that are known and identified beforehand. Entities with contexts in such a realm can be served uniformly. In regard to the given example, entities in Bavaria requesting weather warnings can be delivered a service, which is employed for the entire context realm.

Generally speaking, space and context density (as discussed in section 5.3.2) are metrics that apply for arbitrary contextual realms, too. Further, the convergence metrics - both for individual contexts and the entire realm - can be applied. The representative context can be composed of each context range's centroid of its coherent subspaces. However, the quality statement associated with the convergence metrics, which we have mainly employed to evaluate automatic context-adaptation for cluster-derived realms does not have

[13]showing a map of Germany with Bavaria, the area to be covered by the sub-realm

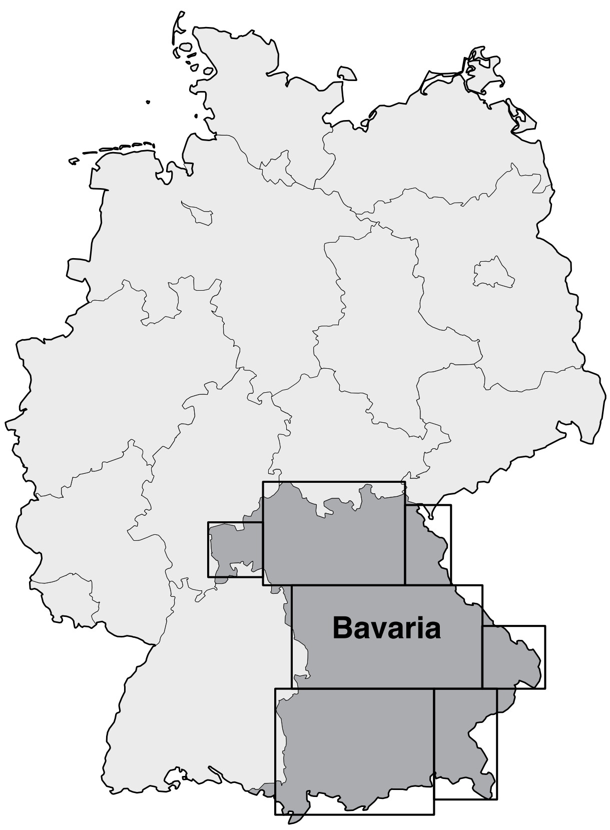


Figure 5.10: Example Realm on R_{loc}

such a strong impact on arbitrary realms. Cluster-derived realms depict existing situational realms of existing entities. Context-aware applications need to dynamically adapt to those as good as possible. Arbitrarily defined realms depict predefined situations, not caring about existing entities, thus lacking the necessity to serve them adequately. For this reason, the denoted quality metrics are of much larger importance for cluster-derived realms, although they are applicable to arbitrarily defines realms, too.

5.3.4 Operations on Contextual Realms

In this section, we sketch operations that may be applied on contextual realms.

- *Context inclusion:* Given a context and a realm, this basic operation determines if the context is included in the realm. Since context realms are split into sub-realms on multiple contextual ranges, the inclusion operation is broken up into several sub-operations of which every one checks the inclusion of the corresponding range context in a sub-realm on the particular context range.
- *Realm overlap:* The amount of overlap between multiple contextual realms yields an absolute amount of contextual convergence between those realms on contextual

range-level, i.e. it is stated how similar the realms are to each other. However, this does *not* include any statement on equivalency of the considered context realms. Together with the knowledge about realm sizes, this operation can naturally be used to determine whether a smaller context realm is completely included in a larger one. The overlap volume of m multiple context realms C_1 through C_m on range R is denoted by $V_{overlap}\big(\{C_1, ..., C_i, ..., C_m\}, R\big)$.

It is to be noted that C_i may also denote an inter-range contextual realm. In that case the correspondent sub-realm on R is used for the overlap operation. Again, the result yields a range-specific value, since only a particular range can be regarded due to non-uniform axis definitions on inter-range level.

- *Realm equivalence:* The degree of equivalence is based on the proximity of representative contexts on range-level. However, this only indicates that the according realms are anchored at the same contextual setting. In addition, the size of the realms matters accordingly. For this reason, we introduce two different equivalency metrics: the *punctual equivalency degree* with no regard to realm size, and the *general equivalency degree* depicting overall realm equivalency with regard to realm size.

  - the *punctual equivalency degree* for contextual realms where just the distance between representative contexts matters. Given a set of contextual realms, the average of the distances between all pairs of representative contexts on range-level denotes this metric.
  - A more expressive equivalency metric can be inferred by considering the size of the regarded context realms. We regard the case on a single contextual range first. On range R, the ratio between the overlap volume and total volume of context realms yields the *general equivalence degree*:

$$deg_{equiv}\big(\{C_1, ..., C_m\}, R\big) = \frac{V_{overlap}\big(\{C_1, ..., C_m\}, R\big)}{\sum_{i=1}^{m} V(C_i, R)} \tag{5.22}$$

where there are m contextual realms involved and $V_i(C_i, R)$ denotes the spatial volume of the contextual realm C_i on range R. This equation denotes that the range-specific general equivalence degree ranges from 0 (not equivalent) to 1 (totally equivalent). Extending this setting on the case covering multi-range realms, we define the inter-range equivalence degree as the average of the range-specific equivalence degrees:

$$deg_{equiv}(\{C_1, ..., C_m\}) = \frac{1}{n}\sum_{i=1}^{n} deg_{equiv}\Big(\{C_1, ..., C_m\}, R_i\Big) \tag{5.23}$$

yielding the equivalence degree of the context realm set on all n ranges. As its range-specific parts, it ranges from 0 to 1, too.

5.3.5 Contextual Realm Monitoring

Monitoring contextual realms basically equals observing the realm characteristics over time. We have provided a detailed discussion about those characteristics in the preceding sections. Closing, they are summed up as follows:

- realm convergence on range level
- context density
- multiple-realm operation results
  - realm overlap
  - realm equivalence

Monitoring contextual realms basically leaves the two approaches that have been discussed in section 5.2.3. Either each of the metrics above is updated after each single contextual update, or they are updated accumulatively in certain intervals. The advantages and disadvantages remain the same as presented. In the remainder of this section, we illuminate the monitoring mechanisms for cluster-derived and arbitrarily defined contextual realms in regard to the aspects of a realm's size, position and context population.

Cluster-derived Realms The metrics above are of particular interest for those realms, since the realms are constantly changing their position, size and context population. Dynamic monitoring of clusters (as discussed in section 5.2.3) allows an efficient update of the according metrics. In the case of the individual update approach, each metric is adjusted after a contextual update, e.g. together with the individual cluster update. Alternatively, the interval-based accumulative update procedure requires recalculating all metrics according to the current state of the entire cluster.

Arbitrarily defined Realms Since those realms are predefined, size and position of those realms do not change. Hence, the multi-realm operation results remain static, since they take exactly those two parameters into account. However, since the context population is dynamic, context density and realm convergence represent dynamic metrics. Updating the realm metrics after each contextual update results in the individual adjustment of each metric - as with metrics in cluster-derived contextual realms. Accumulative updates require performing range queries on the encompassed realm space in regular intervals. The range queries deliver the contexts currently positioned inside the realm for the determination of the realm's convergence degree and context density.

Chapter 6

System-specific Aspects

Throughout the preceding chapters we have presented the concept of the contextual map model in an in-depth discussion. This chapter presents first thoughts on realization aspects by discussing the deployment of the contextual map in a distributed system (with potentially mobile hosts) in a heterogeneous environment. The subsequent sections 6.1 and 6.2 handle the aspects of distribution and heterogeneity, respectively.

6.1 Distribution and Mobility

Since the contextual map model handles entities with dynamic contexts, it is clearly applicable to ubiquitous computing environments. Systems settled in this domain naturally exhibit a high degree of distribution and mobility. This section discusses the applicability of the contextual map in such distributed environments. We consider entities to be equipped with mobile hosting devices that are used to commit and receive contextual information. This enables a context-aware system that employs the contextual map to be partially deployed on those numerously distributed devices. Thus, we regard an environment with many entities that are mobile in terms of their geographical location.

6.1.1 Context Model Distribution

In a distributed system, the contextual map itself can be deployed differently on the various distributed components. For this purpose, we have defined three specifications of the context model, which basically differ in the scope of an entity's contextual neighborhood in the map.

- *Local Map:* In section 4.7, the contextual map has been employed to monitor a single entity's context, in order to determine the need to trigger a contextual update. According to the zone-based update semantics (see section 4.2), such an update would be triggered if the entity context has changed significantly enough to justify an update. Thus, the local map only serves as a data structure to implement those zone-based update semantics by the context-owning entity. Any information about entity contexts other than its own is not present.

- *Vicinity Map:* This specification is centered around an entity's context and encompasses all of the other entities' contexts, which are currently known to the entity in its physical or logical neighborhood. This set of contexts literally defines the entity's known contextual vicinity within its physical or logical surrounding. It is to be noted that we refer to the context's physical and/or logical surrounding, thus encompassing the contexts, from which updates have been received. It is *not* to be mistaken for context vicinities as discussed in section 4.4.2.
- *Global Map:* The global contextual map is not centered around single entities, but encompasses all contexts of entities known by the entire context-aware system. Thus, a global map denotes a system's global context that encompasses the individual contexts of all known entities.

Figure 6.1 illustrates the three specifications with the local map missing any other contexts in contrast to the other map specifications. Comparing those three alternatives, they all possess fundamentally different expressiveness. The local map depicts a single entity context only, the vicinity map possesses context information about neighboring entities, and the global map is omniscient with knowledge about all existent entity contexts.

One can observe a direct dependency between expressiveness of the map model and its required capabilities concerning calculation and storage on its hosting device. The more expressive a context model is the more capabilities it requires on its hosting device. The spectrum of possible hosting devices is vast making capability requirements an important aspect. Table 6.1 provides an overview about the dependency between expressiveness and capability requirements.

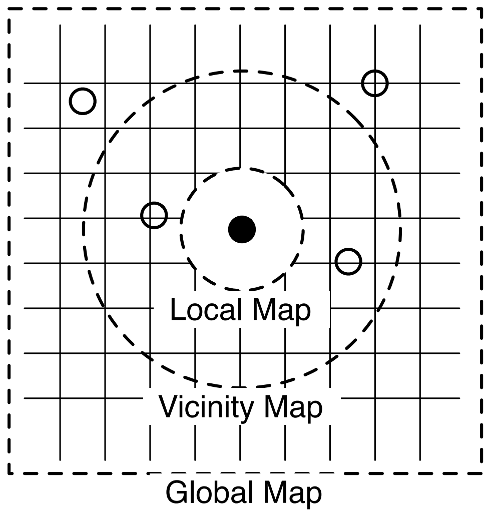


Figure 6.1: Distribution of the Contextual Map

It is crucial to keep in mind that our discussion about distribution of entities focuses on techniques related to communication, updating and deployment implied by distributed

Map Model	Expressiveness / Contextual Compass	Capability Requirements
Global Map	high, encompasses current context of all entities known to the context-aware system	high (store and analyze entire global context)
Vicinity Map	moderate, only contexts of neighboring entities is visible and current	moderate (confined to neighboring entity contexts)
Local Map	low, none but own context available to an entity	low (monitor own context for update determination)

Table 6.1: Contextual Map Variants

systems. In our discussion, the location data implied by distribution has nothing to do with entity contexts. The emphasis of this entire section is the focus on *transportability* and *deployment* of context information in a distributed system. The importance of location affecting entity contexts is completely irrelevant here. Any implication on entity contexts caused by entity location is a matter of processing entity context and has already been discussed previously in chapter 4.

6.1.2 Employable Architectures

This section discusses the applicability of the contextual map model on diverse distributed architectures. In particular, we focus on the client-server paradigm, peer-to-peer networks, hybrid peer-to-peer systems and wireless sensor networks (figure 6.2). We have identified the following aspects being of importance:

- *Map distribution:* With the global map providing the richest amount of information (including all known entity contexts), it is naturally not generally applicable to distributed systems. According to the restrictions implied by distribution, the less capable map specifications may have to be employed. E.g., in a resource-constrained system, a vicinity map or even a local map may have to be used.
- *Update Semantics:* The contextual map complies with the deployment in highly dynamic environments, requiring contextual updates to provide the overall system with current contextual information. The semantics for the consequently required contextual updates have already been discussed in section 4.2. They include polling, periodic and zone-based mechanisms. We are about to enrich this discussion with the applicability of update semantics to distributed system architectures.
- *Capabilities:* The presence of mobile hosts implies the existence of significant hardware restrictions [97]. Since the contextual map model requires a certain level of performance it may not be able to be deployed on any available host.

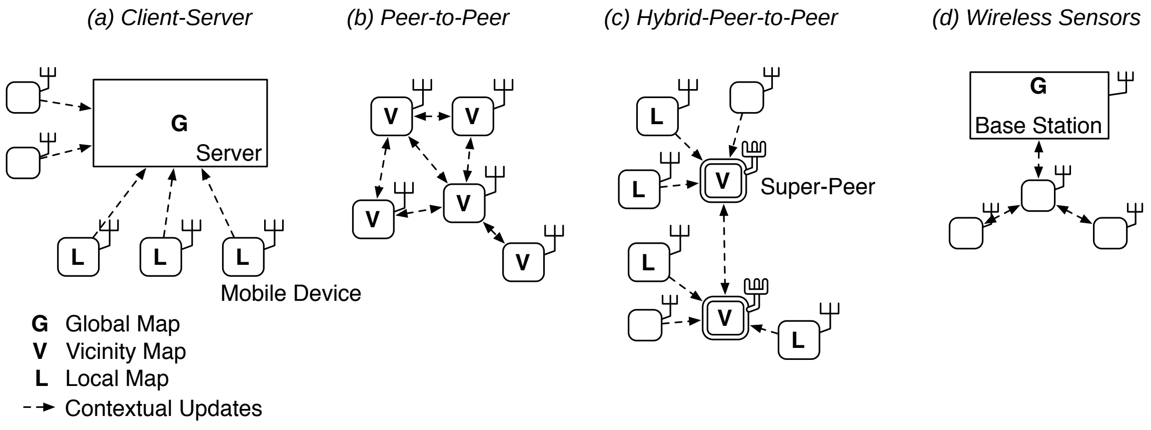


Figure 6.2: Distributed Architectures

As stated before, we assume that entities are attached to a device committing their contexts to the repository implementing the contextual map model. The entities are explicitly thought to be mobile implying the usage of mobile devices with restricted capabilities.

Client-Server Paradigm

This scenario encompasses an omnipotent server, where all of the contextual information from mobile clients converges. It corresponds to the first architecture depicted in figure 6.2 (far left).

Map distribution Since the context management occurs on the server, it is suitable to employ a *global map* there. It provides a complete snapshot of the entire context situation of the covered area. Clients may be equipped with a *local map* to enable efficient zone-based updates (updates are triggered after sufficient change of context). However, if required by application cases, any deployment on the clients can be avoided as well in order to render the clients as lightweight as possible (see map configuration examples in figure 6.2). This may be necessary when client devices have extreme hardware restrictions unable to even support a local map.

Update Semantics The issue to determine when a contextual update is appropriate has been discussed earlier. Given a local map on the client device, an update zone in the contextual map can be defined as the metric for defining context being sufficiently different to justify an update (see section 4.2). Alternatively, given a lightweight client missing a local map, one must fall back on conventional methods, particularly polling updates and periodic updates (see section 2.6).

Capabilities The omnipotent server allows any deployment thought necessary by corresponding use cases. Restrictions only apply to mobile clients, which may not be able to fully support the deployment of a local contextual map. However, those less potent client devices may still be integrated into this architecture neglecting the zone-based update semantics in $\mathbb{R}^n$.

Peer-to-Peer Networks (P2P)

Pure P2P-networks lack any central components for coordinating interoperability and context acquisition. Hence, in our discussion we focus on mobile peers roaming in wireless networks. This setting is visualized by the second architecture in figure 6.2 (second from the left).

Map distribution The lack of a central instance in a P2P-based system makes it impossible to employ a global context representation, forcing the utilization of map specifications other than the global map. Further, the lack of a global context representation requires the peers to have information about other contexts. Hence, peers build up a contextual map centered around their own context by exchanging information about other contexts with neighboring peers (see figure 6.2). This process reflects the local maintenance of a map about contexts inside a peer's known physical/logical vicinity. Accordingly, the *vicinity context map* is the map model of choice. As a consequence, the global context is represented by the sum of all peers' vicinity maps[1]. None of the peers is able to construct a global and current context representation in its map model. As a consequence, no implications on global context can be conducted in the entire network. Assuming that peers exchange context information not only about themselves but about their neighborhood as well, i.e. committing all known context from their entire vicinity in updates, each peer receives context information from other non-neighboring peers over time. This may occur, since context information from remote peers can "dig through" to a peer by being passed in multiple updates. Eventually, a peer may receive complete context information covering an area, which is much larger than its neighborhood. However, it takes time to receive remote context information due to the numerous update iterations. For this reason, remote context may be outdated once it arrives. It seems reasonable to let a peer decide when to prune outdated context, so that it is neither included in its vicinity map nor committed further to neighboring peers.

Update Semantics The strategies of efficiently determining suitable times for issuing contextual updates are only applicable, if the update's recipient is always reachable. In P2P-networks, this is not the case. For this reason, the update semantics have to be restricted to exchanging updates between peers during their periods of interaction between each other. Since those connections between peers are usually transient, updates are committed upon establishing *contact* with another peer. Those "initial updates" may not only include the own context, but also a set of all relevant other contexts. This is reasonable, since the update's recipient may be unable to determine the other entity's

[1]since vicinity maps are not disjoint, the global context is represented by the most current context available in any vicinity map

contexts included in the update. Finally, the update strategies encompassing polling, periodic updates and the zone-based updates in $\mathbb{R}^n$ complement the update semantics for peers during their interaction periods.

Capabilities In this setting, we face a set of mobile peers with restrained capabilities, which comprise the entire system. However, the mobile devices need to provide sufficient computational power and storage. Both are necessary for updating and hosting a vicinity map. Devices, which do not excel the minimal requirements, cannot participate in this system. Alternatively, they can be employed in a hybrid-P2P system (see forthcoming discussion).

Hybrid Peer-to-peer Networks (H-P2P)

H-P2P-networks [46] share the same layout as their pure P2P-counterparts. The distinguishable difference is the existence of *super-peers* in H-P2P architectures as shown in the third architecture in figure 6.2 (second from the right). Super peers are dedicated mobile nodes, which function as regional coordinating components within the network. Although slightly exhibiting client-server techniques, super-peers are only accessible by peers in reach.

Map distribution The existence of super-peers allows applying a light-weight client-server approach to the H-P2P architecture. Super-peers are deployed with a *vicinity map*, as their regular counterparts in the pure P2P-architecture. With the super-peers equipped with a vicinity map, it suffices to use *local maps* on the remaining peers (see figure 6.2). The super-peers actually act as they were servers for all peers in reach. As with pure P2P-networks, a global context representation is not available to any peer or super peer. An approximation of larger areas beyond neighboring peers is only available to super peers under the restrictions of context actuality as discussed previously in regard to pure P2P-networks.

Update Semantics In H-P2P-networks, contextual updates are propagated analogously to P2P-networks, except that regular peers do not receive contextual updates. With a local map, they are unable to process other contexts but their own.

Capabilities Basically, the same restrictions impacting the design of P2P-networks are applicable to H-P2P as well. However, since local maps are less resource-consuming, capability requirements on regular peers are less demanding. As we have stated earlier already, it is even possible to employ peers without any contextual map deployment, hence minimizing the capability demands on those. However, the latter minimization efforts come with the cost of neglecting the zone-based update semantics in $\mathbb{R}^n$. Periodic updates and polling are the remaining strategies of choice then.

Wireless Sensor Networks (WSN)

In mobile WSNs, sensor nodes operate with minimal hardware capabilities. They are usually small in size, deployed in the wild and supposed to gather data over long periods of time. Hence, minimizing energy consumption is crucial. As a consequence, hardware capabilities are minimal. Sensor nodes are usually capable of processing simple tasks, storing several bytes of information and transmit them over-the-air to a server component, namely the base station. Usually, the base station needs to be in reach of a sensor node for transmission, since the node's range is limited due to its hardware restrictions. WSNs are a specialization of the more general client-server-architecture discussed earlier. In contrast to the more general instance, WSNs focus on heavily restrained clients. The fourth architecture in figure 6.2 (far right) denotes the WSN architecture.

Map distribution With the base station attached to a capable machine, it can maintain a *global map* for the entire system. The sensor nodes, however, do not have the capability to host any context model (see figure 6.2).

Update Semantics and Capabilities Due to minimal hardware and transmission range of the sensor nodes, updates can only be issued upon contact with the base station. Given the case that sensor nodes are able to communicate among each other, updates can also be exchanged upon contact between nodes. This approach implies that nodes are capable of storing contextual information other than their own, which they forward to other nodes upon contact. Hence, those foreign updates are not processed, but stored and readied for transmission to the base station. Because of the limited storage capabilities, the buffer for foreign updates may be full before reaching the base station. In this case, it is prudent to erase the oldest updates, since they most likely contain the most obsolete context information. However, dropping aged context information may result in some nodes' context data not reaching the base station at all. As a consequence, the global map at the server may not have an encompassing representation of the global context. In addition, the nodes' context data may take a considerable time to get to the base station and thus into the global map. The actuality of context is further degraded by this circumstance. For this reason, the global map's capabilities in WSNs have to be diluted from the observation in table 6.1. Although the base station's global map aims at constructing current and all-encompassing context, it cannot be guaranteed due to the discussed architectural restrictions.

Evaluation

In the foregoing discussion we have regarded four different architectures in respect to three aspects: distribution of the contextual map, update semantics and capabilities. We can sketch a summarizing statement as depicted in table 6.2.

In addition, this discussion allows us to infer architectural metrics that inherently depend on each other:

Architecture	Map Distribution (number)	Update Semantics	Capabilities
Client-Server	Global (1), Local (arbitrary)	update zones, periodic, polling	potent
P2P	Vicinity (arbitrary),	on contact	limited
H-P2P	Vicinity (arbitrary), Local (arbitrary)	on contact	limited
WSN	Global (1), none (arbitrary)	on contact	minimal

Table 6.2: Architectures of a distributed Contextual Map

- *Context actuality:* This aspect denotes how current the system's knowledge about the overall contextual situation is. It is directly dependent on the update semantics and the underlying infrastructure. It may be quantified by determining on how *current* and *complete* contextual information is at a specified entity or any context-management component of the context-aware system. E.g., the server in a client-server architecture has very current information bundled at one place whereas a P2P node has only a few, incomplete and probably outdated context information.

- *Deployment cost:* The overall costs for deploying an architecture are depicted by that aspect. Larger and more complex infrastructures logically require higher costs of implementation and setup.

- *Capabilities:* The capabilities of the architecture in terms between minimal and potent. Although this metric cannot be quantified as easily as the previous two, it is still significant on the conceptual - non-numerical - level.

Those three metrics depend on each other as visualized in figure 6.3. As expected, the basic statement of this observation is that context actuality is expensive. The lower the capabilities are, the lower the context actuality is. The same relation generally applies to the deployment costs. A potent (and thus expensive) infrastructure, which generally comes with relatively high deployment costs, enables higher context actuality. Low deployment costs normally restrict the ability to provide current context.

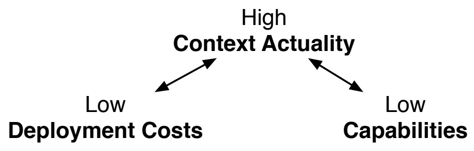


Figure 6.3: Dependencies among Distribution Metrics

6.1.3 Distributed Updates of Context

The distribution of the contextual map model has a significant impact on keeping the system-wide context current efficiently. Contextual updates are constructed and propagated by mobile hosts contributing to the actuality of the system-wide context. In the following discussion the composition of updates in a distributed system is given a closer look. In section 4.3 we have introduced the data structures employed for exchanging updates between *two* entities - the updating sender and the update receiver. Here, it is necessary to extend those principles to an arbitrary number of entities. Recalling section 4.3, validity vectors control the data to be selected for an update whereas update vectors denote the actual data included in an update (see equations 4.22 and 4.23). This section discusses the application of this mechanism on the distributed architectures evaluated in the previous section 6.1.2.

Client-server

Since every entity communicates with the server only, the management is reduced to one validity vector per entity. Basically speaking, this architecture reflects the basic communication principle introduced in section 4.3 since the entities' updating efforts are isolated and hence not interfering with each other. The entities commit updates while the server manages the system-wide context. Recalling the update composing technique from section 4.3, both update and validity vectors are managed only by the clients.

Peer-to-peer and Hybrid-P2P

In P2P networks, entities communicate with numerous other entities. In contrast to the previously evaluated client-server paradigm, in this architecture entities are responsible for system-wide context management in addition to just conduct contextual updating. Since an entity may be in contact with multiple communication partners, it needs to manage multiple validity vectors. One validity vector for each communication partner allows to distinguish which data to include into an update for this particular partner once an update is required. In addition to managing own updates, each entity may need to manage *foreign updates* as stated in section 6.1.2. It is to be reminded that such updates are passed throughout the system to maximize the system-wide actuality, i.e. providing each entity with the most current system-wide context, which is available through its neighbors. For this scenario, the mechanism from section 4.3 is extended by the following aspects:

- A validity vector $V_{P,Q}$ stored at an entity E (see equation 4.22) denotes which context data of entity P has been committed to Q by entity E. More precisely, P denotes an entity present *somewhere* in the network that E is aware of, whereas Q is a *communication partner* of E. This definition assumes that E and Q have possessed a physical link between each other for an interaction period in the past (which may well last to the present). This means that E stores a permutation of validity vectors for each encountered communication partner and each (remote) entity, which E has received an originating foreign update from. E.g., figure 6.4 illustrates six entities with transient connections among each other. Each entity hosts validity vectors for its own contextual changes to be committed to its neighbors. E.g. A hosts $V_{A,C}$, $V_{A,D}$ and $V_{A,E}$ for committing updates of its own context. When receiving a foreign update, there is a vector added for each communication partner (except the one from which receiving the update) denoting the context of the entity where the update information *originated*. E.g. the update vector U_B including updates about B's context has been passed via C to A, so that A prepares to distribute this information to its neighbors by adding $V_{B,D}$ and $V_{B,E}$ to its stored set of validity vectors. A vector $V_{B,C}$ is intentionally neglected for an obvious reason: C has committed B's foreign update, thus knowing B's context as well as A does. There are no updates about B to be sent from A to C at this time[2].

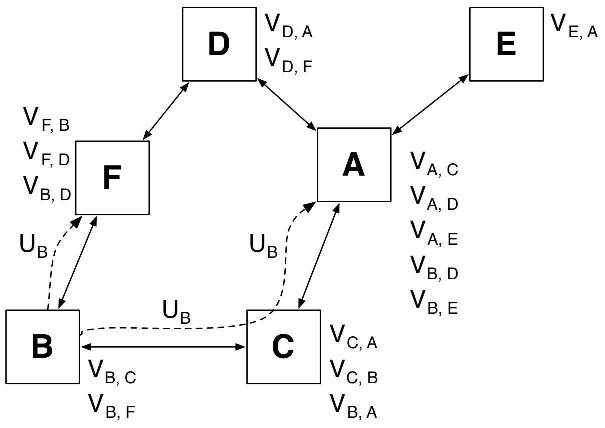


Figure 6.4: Distributed Updates in P2P Networks

- An update vector U_P (see equation 4.23) includes context data about an entity P exchanged between any two entities in the network. It is basically constructed as described in section 4.3. As stated and shown in figure 6.4, P can be any entity in the network. Hence, U_P can be distributed as a foreign update. Upon receiving an update vector U_P, the receiving entity merges the updated context data about P into its locally stored vicinity map (see section 6.1.2). Since updates from P may arrive from different directions, i.e. U_P may have been distributed via different paths, context data concerning the same contextual attribute (one element of U_P) may arrive in different updates. Since we are facing a highly heterogeneous topology in P2P-networks, the time of arrival does not allow any conclusion about the actuality of arriving updates. For this reason, it is prudent to attach a timestamp to each data element in U_P denoting its time of creation. Hence, when P initially creates and distributes U_P, it attaches the current time to each element in U_P. Hence, an update vector U_P depicts update-worthy context data about the context of an entity P as follows:

[2]although this may change when U_B is received from another *direction*, e.g. via F and D

$$U_P = \begin{pmatrix} u_1 \\ \dots \\ u_i \\ \dots \\ u_n \end{pmatrix} \Bigg| \, u_i = \begin{cases} (p_i, time_i) & \text{if } v_i = false \\ \oslash & \text{else} \end{cases} \tag{6.1}$$

with $P = (p_1, ..., p_i, ..., p_n)$ denoting the context entity P and $time_i$ depicting the creation time of p_i, i.e. the time when p_i was committed into an update by P, and $V = (v_1, ..., v_i, ..., v_n)$ being the affected update vector of P. After U_P is first created it is passed through the network as follows. When an entity receives U_P, it compares all u_i with the context information about P in its locally stored vicinity map. If $time_i$ is older than the correspondent data stored in the vicinity map, u_i is pruned. Otherwise, context P in the vicinity map is updated by p_i. This reasoning implies that $time_i$ is associated with contextual information in the vicinity map, too. For further distributing the context information of P a foreign update is used. For this purpose U_P is newly constructed according to the remote context P in the vicinity map before committed farther into the network. The validity vector concerning P and the communication partner to which U_P is sent influences the construction of U_P as denoted in equation 6.1. E.g., if sent to a neighboring entity Q, $V_{P,Q}$ is taken into account. Binding U_P to the vicinity map of the issuing entity ensures the most current context information to be distributed, since the vicinity maps holds the most current context information available to the issuing entity. Figure 6.5 illustrates the principles discussed here.

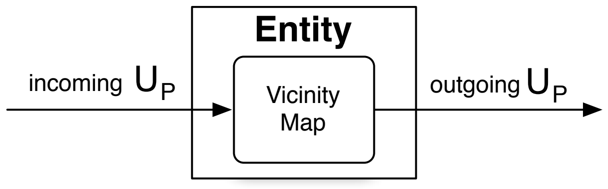


Figure 6.5: Update Vector passing in P2P Architectures.

- Both validity and update vectors have local scopes. A validity vector is only feasible at its hosting entity. E.g., in figure 6.4, both entities A and F each store a validity vector $V_{A,D}$. Although having the same description, they are not equal. Formally, to ensure a system-wide unique ID, a validity vector would need to include its hosting entity in its ID. For reasons of simplicity, this aspect has been neglected in this discussion so far. A suitable ID for a globally feasible validity vector may be $V_{A,D}(A)$ and $V_{A,D}(F)$ for the previous example. Update vectors are only feasible for the sending and receiving entity, since every update vector is reconstructed by the sender and not passed through in the state in which it has been received. Globally feasible update vectors may be augmented by their sending and receiving entities, e.g. $U_B(B, F)$ for an update vector being transmitted between entities B and F, as shown in figure 6.5.

Hybrid-P2P-networks employ a hybrid approach concerning the composition and distribution of updates. The distribution of updates among super peers functions analogously to the P2P model. Concerning the communication between super peers and regular peers, updates are only committed by regular peers. The communication between those basically works analogously to the client-server model reflecting the basic principles, which have been introduced in section 4.3.

Wireless Sensor Networks

The update composition techniques in mobile wireless sensor networks are inspired by the techniques presented for P2P models. However, due to a missing map model on the nodes, foreign updates are *forwarded* to neighboring nodes (not reconstructed as in P2P networks). Concluding, an update vector U_P stays unchanged for the duration of its lifetime ranging from its creation at P till its commitment to the base station. In this case the $time_i$ entry can be substituted by a creation time for the entire vector, say $time_P$. An update then consists of an update package including U_P and $time_P$ with U_P being constructed in its basic form as shown in equation 4.23. Foreign updates are stored in a buffer, which is emptied upon committing all included updates to the base station. Limited storage capabilities usually bound the buffer's capacity. If an update is received whose $time_P$ is older than a correspondent update in the buffer, U_P is pruned.

Due to significant resource constrains, some aspects may also need to be regarded:

- The use of a validity vectors may be neglected if it is impeded by calculation and storage constraints applying to the sensor nodes. This measure results in update overhead by including redundant information in update vectors, but it decreases computation and storage requirements since validity vectors are neither calculated nor stored.
- Since update packages are handled according to the "store and forward" principle, a node's buffer for foreign updates may run full. In this case, there are two basic options to pursue: First, the oldest update is removed from the buffer and substituted by the new one (regardless from which entity the oldest update originates). Second, updates may be prioritized. In this case, the oldest update with the lowest priority is removed from the buffer. This raises the chances of highly prioritized updates to reach the base station. Concluding, such an approach requires a threesome update package:

$$(U_P, time_P, priority_P) \tag{6.2}$$

6.1.4 Deployment

Concerning the architectures discussed in section 6.1.2, the issue of deployment primarily concerns individual mobile devices, which employ one of the contextual map options, i.e. either the vicinity map or its local counterpart. To run such a component on the mobile device, the most suitable way is to deploy it as middleware, i.e. between the device's operating system and the user interface. There are two options to do this:

- make use of an *existing* middleware platform that is running on the device (e.g. J2ME [7])
- deploy the appropriate map model as middleware itself by using the operating system's API

In both cases, the contextual map then acts as a middleware between context-aware applications and the device's enabling operating system. Considerations about hardware are unnecessary since the operating system enables transparent access to the device's hardware. In order to function appropriately, the employment of such a middleware needs to concur with the following aspects:

- The *hardware capabilities* of the carrier device need to comply with the middleware's hardware requirements. This aspect applies to the case when the contextual map is employed as stand-alone middleware. If embedded into existing middleware, the device usually possesses sufficient capabilities of supporting the hosted middleware.
- The *access to context sources*, which are controlled by the operating system, needs to be enabled and maximized in terms of availability. This includes on-device sensors as well as context sources outside the device (most likely connected wirelessly). The more fine-grained access to context sources is available the more accurate the context monitoring is.

Concerning server components as listed in the client-server architecture and WSNs, we do elaborate further on the deployment issue due to the large degree of freedom and lack of restrictions on server machines.

6.2 The Impact of Heterogeneity

Throughout this work, we have argued that heterogeneity has a tremendous impact on context-aware computing and thus, on the contextual map. In the introduction (chapter 1) we have already emphasized the problem with heterogeneous context sources producing error-prone low-level context information that is unusable for further utilization (unless refined). We have outlined this issue in section 2.2.2 and proposed an approach in section 2.4.2 denoting the *context capturing interface* refining heterogeneous context data and thus masking heterogeneity from the context management system (see figure 2.5). We have also provided a peripheral discussion about heterogeneous context models and context-aware middleware systems in sections 2.3.3 and 2.4.4, respectively. In section 2.7, we have provided an in depth discussion on the problem of handling cooperation among heterogeneous technologies (see figure 2.16). In section 3.2, we have further argued that the design of context-aware systems is highly application-specific. E.g., we have discovered that mapping context into our contextual map does not follow any generalizable rules, but rules dictated by the application case.

This chapter illuminates the impact of heterogeneity on the contextual map. We describe several applicable heterogeneity aspects that apply on our context model. First of all, we discuss the environment from which the contextual map (and any other application) gathers its context data from. Subsequently, we illuminate the heterogeneity of context, mainly focusing on the character of context answering the question: what makes up context and how is context actually composed? The last section is dedicated to application heterogeneity and how it influences a context-aware middleware system.

6.2.1 Heterogeneous Environment

In this section, we summarize the heterogeneity issues originating from the environment containing the sources of context. If applied to the real-world-environment, context-aware systems can collect an enormous amount of data from it, refining it to context, and make it employable for high-level context-aware applications. We have depicted this workflow in figure 2.5. The vast diversity of context sources makes gathering context information an elaborate task, since each context source may require a different access approach. As denoted earlier, we access context sources with context sensors. Such a context sensor can be represented by anything capable of collecting the context information from the context source. E.g., a temperature sensor (context sensor) may extract the temperature (context data) from the air (context source). Or a system daemon (context sensor) may query a distant database (context source) for a user's social network subscriptions (context information). Concluding, context sources, information and sensors are dependent on each other, but this chain can be represented by virtually anything.

Besides handling heterogeneous context sources and sensors, context-aware systems need to be able to communicate with a wide spectrum of diverse communication partners. As the context sources, communication partners are scattered throughout the environment that a context-aware system is deployed in. In section 2.7 we have discussed the issue of heterogeneous communication, and we have derived a suggestion how to handle it (section 2.7.3, visualized in figure 2.16). Communication partners have heterogeneous attributes (system specification, middleware, etc.) and employ heterogeneous means of communication (i.e., communication medium, protocols, etc.). However, enabling communication among them is essential, in order to make context-awareness ubiquitously available in a heterogeneous environment. Ubiquity requires information to be meshed together, so that it is pervasively available.

In order to deal with the two issues discussed above - namely heterogeneous context sources and communication partners - we unify both problems into a combined architectural sketch. As shown in figure 6.6, the heterogeneous environment is accessed by a two-tier architecture handling heterogeneous context acquisition and communication.

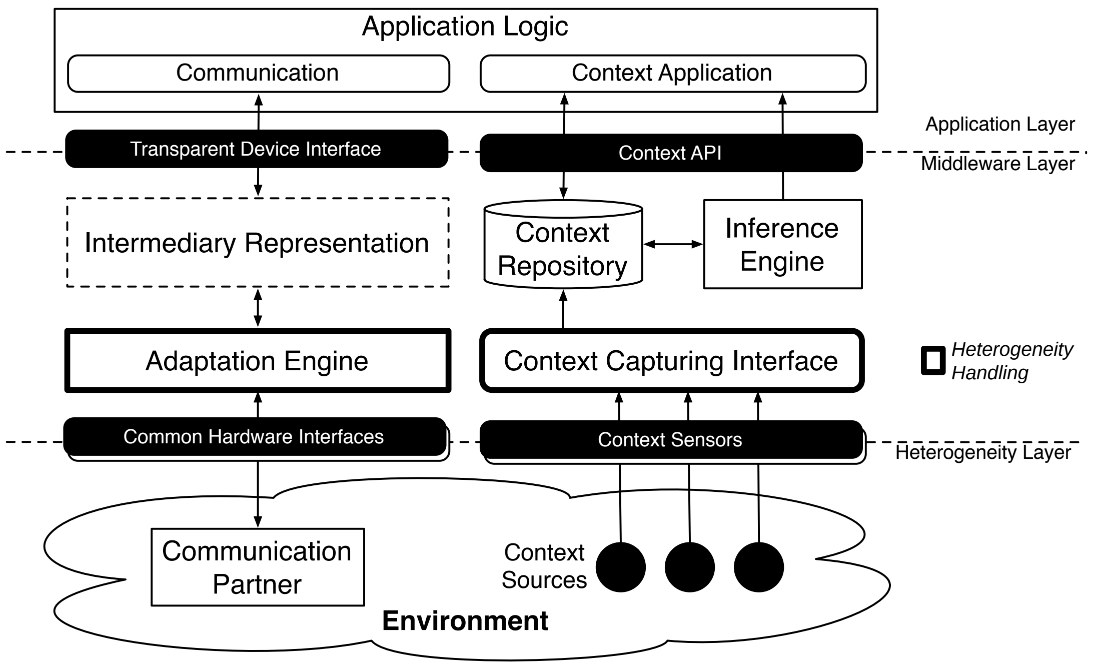


Figure 6.6: Environmental Heterogeneity Handling.

The context acquisition tier (right side) is inspired by our depiction in figure 2.5 and handles heterogeneous context sources. Diverse context sensors acquire raw context information from context sources that are subsequently refined by the context capturing interface. The data is highly error-prone and heterogeneous, since it comes from potentially very different and unreliable sensors. The context capturing interface removes possible errors and converts the data into a uniform representation, so that it can be used for context management on the higher levels (i.e. storage in the context repository and derivation of new context by the inference engine). There are plenty of possibilities to correct faulty context information. E.g., we can correct fluctuation of gathered contextual information as depicted by [77] (discussed in section 4.5.1) or we can check gathered contextual data against heuristics to identify obviously false information. The error-free data is then transformed into a common representation and stored in the context repository. For the contextual map, this generalization step corresponds to mapping contextual data into the contextual map as dictated by the mapping functions. As depicted in section 3.2, we consider the function's input to be error-free (hence preprocessed) and thus given as vectors (including 1-dimensional vectors, i.e. scalars). We also need to know where to map the data, i.e. how we map the input values to the dimensions of the contextual map's ranges. Hence, the mapping function serves as the bridge between the heterogeneous context sources and the homogeneous data in the contextual map. Regarding figure 6.6, the mapping function is situated in the context capturing interface mapping the data into the context repository harboring the contextual map.

The communication tier in figure 6.6 (left side) is inspired by our architecture from figure 2.16. It does not implement any core mechanisms required for context-aware computing, but it enables ubiquity for context-awareness and thus, represents an important peripheral building block for a context-aware system. As we have discussed in section 2.7.2, common interfaces represent the key communicating with heterogeneous partners. Those enable the actual sending and reception of data. Received data from heterogeneous communication partners may not be understood by the recipient at first. This requires the recipient to adapt to the data. The adaptation engine in the architecture transforms heterogeneous data into a common intermediary representation, hence bridging the heterogeneity implied by the diversity of communication partners. For the contextual map, the relevance here consists in the ability to commit and receive contextual updates (extensively discussed in chapter 4) to or from any entity with arbitrary characteristics.

The architecture in figure 6.6 is layered according to specific purposes:

- *Heterogeneity layer:* This is where all the diversity of the environment is situated. The context sources and communication endpoints are placed on this layer.
- *Middleware layer:* This layer handles the heterogeneity issues employing the two-tier suggestion as discussed above. On its lower side, it provides interfaces for access to the heterogeneous environment, in order to gather and commit data. Those interfaces are represented by common interfaces and context sensors, respectively. Since data going through those interfaces is non-uniform, dedicated adaptation components are attached right to them. They are represented by the adaptation engine and the context capturing interface. They handle the heterogeneous data and represent the actual heterogeneity bridge to the system. The transparent device interface (TDI) and context API on the layer's top side provide transparent access to communication and context data for context-aware applications.
- *Application layer:* Applications can transparently access contextual data and communicate throughout an entire distributed context-aware system by accessing the TDI and context API, respectively.

6.2.2 Heterogeneity of Context Information

In the previous section we have discussed about *where* to get contextual data from. However, an important prerequisite is to define *what* is actually to be considered as context. The amount of contextual data, which resembles entity contexts and which is available from the environment, is vast. However, it is to be clarified which data contributes to a specific context, i.e. the contextual information associated with a particular entity. E.g., the scenario in chapters 3 and 4 depicts weather stations, car drivers, weather conditions and locations. So we are considering two entity types and two context types. However, there is much more context data surrounding both weather stations and car drivers. E.g., the additional context of a car driver can be the type of car he is driving, his current speed, level of fatigue, and much more. With a weather station, we can argue analogously. Additionally, car drivers and weather stations are not the only existing entities in the real-world-environment. They do not even represent a closed system in terms of being connected to other entities. In fact, they have relationships to many other entities that have not been considered by the given scenario. E.g., cars, streets, station operators, etc. are additional entities roaming close to the regarded ones. However, all the additional available data is of no importance for the exemplary application case depicted above. We confine entities and their context only to the data that is actually important to us: car drivers, weather stations, locations and a selected set of weather conditions.

Using this example, it becomes clear how the abundance of context information is utilized for the contextual map. Given a particular application scenario, the heterogeneous and vastly available context data is *circumvented* to fit the application case. In particular, this means selecting the appropriate entity types and context sources.

- *Entity types:* The application case dictates which entities are relevant for contextual monitoring and notification. Determining the relevant entity types usually yields the set of concerning entities.

- *Context sources:* In order to acquire contextual information about entities and determined their contexts, the appropriate context sources need to be determined. Usually, a particular context type consists of contextual data provided by a specific set of context sources.

The example above is intentionally kept simple. All we need is two entity types (car driver, weather station) and two context groups (weather conditions, location). This selective context bounding procedure is driven by the underlying application case. The *general* (and unusable) context given by the environment is specified by application case. The resulting *specific* context is exploitable by the underlying application. Context-aware application cases may differ greatly (see next section 6.2.3) making the regarded contexts application-specific, hence heterogeneous. However, within a single application-specific utilization of the contextual map, heterogeneity does not occur.

Closing, we regard the aspects relevant for determining contexts for the contextual map. Table 6.3 shows how we map aspects from the heterogeneous environment to the contextual map, i.e. how context in the contextual map is composed (and thus confined from all available context data). Context sources provide data, thus they evaluate contextual attributes, and the data is eventually mapped to the diverse dimensions of the contextual map. Definable context types correspond to contextual ranges, since ranges group attributes. Entities that actually possess context are represented as multi-dimensional point in the contextual map and on the Cartesian contextual ranges, respectively. The definition of specific contextual situations corresponds to the definition of arbitrary contextual realms in the contextual map (see section 5.3.3).

6.2.3 Application Heterogeneity

In the previous section we have already mentioned that the diversity of different application cases influences the setting of the contextual map. For this reason, the setting of the contextual map is completely application-dependent and thus, heterogeneous. Since

Environment (heterogeneous)	Contextual Map (homogeneous)
contextual attributes	map dimensions
context sources	values on dimensions
context types	contextual ranges
entities	contexts and range contexts (points in map)
specific situations	contextual realms

Table 6.3: Aspects of Context and the Contextual Map

context-aware applications can implement virtually any imaginable use cases, the heterogeneity spectrum on adjusting the contextual map to application specifics is accordingly wide. When adapting the contextual map to a particular context-aware application, we consider four aspects: contextual attributes, contextual ranges, the mapping function and the context API.

- *Contextual attributes:* The context-aware application, which is to be served, usually expects a very specific subset of the overall available context information. Hence, all relevant context sources that contribute to this particular context have to be identified and mapped to the contextual attributes in question. Those attributes should be quantifiable with discrete vector values.
- *Contextual ranges:* Given the application-specificity and the according contextual attributes, the resulting context can be typified. For each resulting context type, a contextual range may be created in the contextual map. Usually, the contextual attributes depict the dimensions spanning the contextual ranges.
- *Mapping function:* With the quantified context attributes given, they need to be inserted into the contextual map, i.e. they need to be mapped on the corresponding dimensions of the contextual ranges. The mapping function is used to conduct this task. There is no general directive how this can be done. The application case dictates how contextual data is to be handled, i.e. hoe quantified contextual attributes are mapped to the contextual ranges. Thus, the mapping function is completely application-dependent.
- *Context API:* With the contextual ranges filled with contexts, it is to be determined which data is needed by the context-aware application. Thus, the context API is to be equipped with according interfaces that allow the application to query contextual data and to be notified by the context modes (opposite direction), respectively.

See figure 6.6 for a visualization of those concepts. Summarizing the discussion above, the entire context processing chain is required to bridge application-induced heterogeneity.

Chapter 7

Prototypic System Design

With the theoretical background about the contextual map extensively discussed in the foregoing chapters, we now focus on validating the context model. For this purpose, we aim at utilizing the contextual map in a real-world scenario. This chapter presents the design of a prototypic system for that utilization. The utilization procedure itself is documented in the next chapter 8. It is to be noted that the scope of the prototype is very basic. We aim at deploying a proof-of-concept prototype that is capable of determining contextual proximity and separation for two different entities. Other utilization cases are excluded.

In this chapter, we present the prototypic system as follows: After depicting the scope and requirements applied on the prototype in sections 7.1, we present the static and dynamic characteristics of the system design in sections 7.2 and 7.3. Strictly speaking, we present the system's static architecture and the most important workflows applied on it.

7.1 Requirement Specification and Scope

The prototype implements the basic concepts that have been presented so far in this work. However, realizing the contextual map model leaves us a wide range of level of implementation detail, which depends on the amount of stated requirements. In this section, we define the scope of the prototype and enumerate the requirements exacted on it.

7.1.1 Requirements

We aim at demonstrating the contextual map's core ability of detecting contextually proximate entities and to provide this functionality to higher level applications. The subsequent listing summarizes the core functionalities of the contextual map, which are implemented by the prototype.

- *Contextual range definition:* Contextual ranges allow an entity's context to be partitioned type-specifically. It can be represented as quantified scalar attributes. Each of those attributes represents a dimension on a contextual range. As a consequence, both range contexts (range dependent representation) and full contexts (inter-range representation) are supported. The contextual ranges form the basis of the contextual map model.

- *Context mapping:* Utilizing the input from context sources, this data is mapped into the contextual map. A mapping component implements the mapping function projecting the contextual input data onto the contextual range's dimensions.
- *Contextual affinity definition:* Contextual boundaries are applied on multiple contextual ranges to define contextual affinity. A threshold is defined on each affected range depicting the degree of similarity of two entities' contexts on the respective range. The contextual boundary defines the contextual similarity degree on inter-range level.
- *Contextual affinity detection:* Contextual affinity management is implemented by using the Euclidean distance metric applied on contextual boundary thresholds. Entity contexts are monitored and identified as contextually proximate, if the Euclidean distances of their belonging range contexts on all relevant contextual ranges are below the thresholds of the respective contextual boundary (see section 4.4.3). The detection mechanism is applicable to two entities.
- *Contextual vicinity utilization:* Only the neighborhoods of range contexts are taken into account when determining Euclidean distances to adjacent range contexts on a contextual range (see section 4.4.2 for vicinity definitions).
- *Update semantics:* The contextual map is only updated if the contextual change is adequately significant implementing the dynamic circles strategy (see section 2.6.1).
- *Context API:* The contextual affinity management system is transparently placed behind a well-formed interface providing controlled access for context-aware applications. It provides the following functionality:
  - create custom application-specific contextual maps with individual ranges and boundaries.
  - register entities for contextual affinity detection.
  - notify the context-aware client system of contextual proximity and separation between two registered entities.

We mainly focus on functional requirements to deliver a proof-of-concept prototype. Non-functional aspects, such as scalability and performance have only influenced our implementation in regard to demonstrating functional capabilities.

7.1.2 Scope of Prototype

The purpose of the prototype is to deliver a proof-of-concept implementation by implementing the basic functionality of the contextual map and make it applicable to a simple real-world scenario.

- *Contextual affinity management:* The prototype allows the definition of the basic data structures for contextual affinity management, i.e. contextual ranges and boundaries. It is capable of detecting contextual proximity between two registered entities according to a defined boundary. This represents the basic functionality of the contextual map. We do not manage any entities but their contexts. We represent those contexts as multidimensional points and do not consider any other information about the belonging entities but their entity type. We do not implement any higher application cases, such as contextual similarity queries or contextual clustering, neither. However, we do provide the basis for that, i.e. the contextual proximity detection mechanism for two different entities.
- *Scenario and application:* The prototype is applied on a real-world-inspired scenario implementing a particular application case. A simulation emulates real-world entities committing their contexts to the contextual map. According to the application-specifically defined boundaries, contextual proximity is detected and reported by the prototypic model implementing the contextual map.
- *Proof-of-concept:* The proper functionality of the context model and the scenario simulation utilizing that model demonstrates the contextual map's applicability. It is documented in the next chapter 8.
- *Architecture:* The prototype implements the client-server architecture as depicted in section 6.1.2. We do not regard any other architecture that have been discussed in section 6.1. We also exclude any heterogeneity issues that have been discussed in section 6.2.

7.2 System Architecture

This section sketches the static architecture of the contextual map prototype. We introduce the main system components and depict their alignment in the system model subsequently.

7.2.1 Component Specification

The model is composed of the subsequently enumerated modules.

- *Core model module:* Contextual ranges and their assigned dimensions are managed by this module providing the basic structure of the contextual map.

  - *Contextual ranges:* There are three types of contextual ranges differing in their bounding as depicted in section 3.1: bounded, semi-bounded and unbounded ranges. Ranges represent a particular domain of contextual data, which is quantified uniformly in the contextual map.
  - *Range dimensions:* A dimension is a range's axis and represents a dedicated contextual attribute. It is therefore statically assigned to its parents range and quantified according to its given quantification unit. Thus, all of a range's dimensions are quantified using the same unit dictated by the range.

- *Context module:* This component delivers context management to the system. Entity contexts and their respective range contexts are managed here.
  - *Contexts:* Contexts are associated with their belonging range contexts (one per range) and possess an *entity type* denoting the type of the entity that they belong to (e.g. "car", "person", "place", etc.).
  - *Range contexts:* Each range context possesses multiple contextual values that represent the quantification on the belonging contextual range's dimensions, thus depicting the multi-dimensional "position" in that range.

- *Contextual boundary module:* Defining contextual proximity degrees is handled by this component. It manages contextual boundaries with each boundary possessing thresholds for the contextual ranges of relevance (see section 4.1.1). Boundaries can be applied for all existing entities or to individual entities only. It can also be constrained to certain entity types. Defining entity types applies to two cases: on the one hand, entities that are regarded for a contextual proximity check can be narrowed by entity types. On the other hand, entities that are reported to be contextually proximate can be narrowed by entity types, too.

  In either case, a boundary applies to entities' range contexts on the relevant ranges defined by the boundary.

- *Contextual proximity detection:* This module handles contextual affinity detection and management using the contextual map's core module (context ranges), the context module (entity contexts) and the boundary module (affinity definitions). It monitors contextual affinities among range contexts by determining contextual changes following a context update committed by an entity. If boundaries have been crossed, this event is propagated to the context-aware system using the contextual map.

- *Context API:* The context API serves as the interface between the contextual map and any context-aware system using the contextual map. It provides the following functionality.
  - *Initialization:* The API provides dedicated access for creating all necessary data in the contextual map. This especially affects contextual ranges, context mapping rules and contextual boundaries.

  - *Contextual updates:* Updates on contextual data sent by entities are committed via the context API to the contextual map. The context API module implements the *mapping rules* responsible for mapping entity update data on the dimensions of the contextual map. The mapping rules serve as the main access point for entities to commit their context data. If an entity commits its data to one or multiple mapping rules, each of the used mapping rules transforms the data accordingly, so it can be mapped on the proper dimensions (and thus contextual ranges) in the contextual map.
  - *Boundary crossing notifications:* Notifications about boundary crossings are sent through the context API in the opposite direction notifying the context-aware system above. Such a notification includes the pair of entities that have either contextually closed in to each other or separated from each other, together with the contextual boundary under which this affinity change has occurred.

- *Indexing:* This module stores the contextual data used by the contextual map. I.e., this includes applying an efficient data structure to permanently store and query contexts efficiently (see section 3.3.1).

Figure 7.1 shows how those components are aligned to each other.

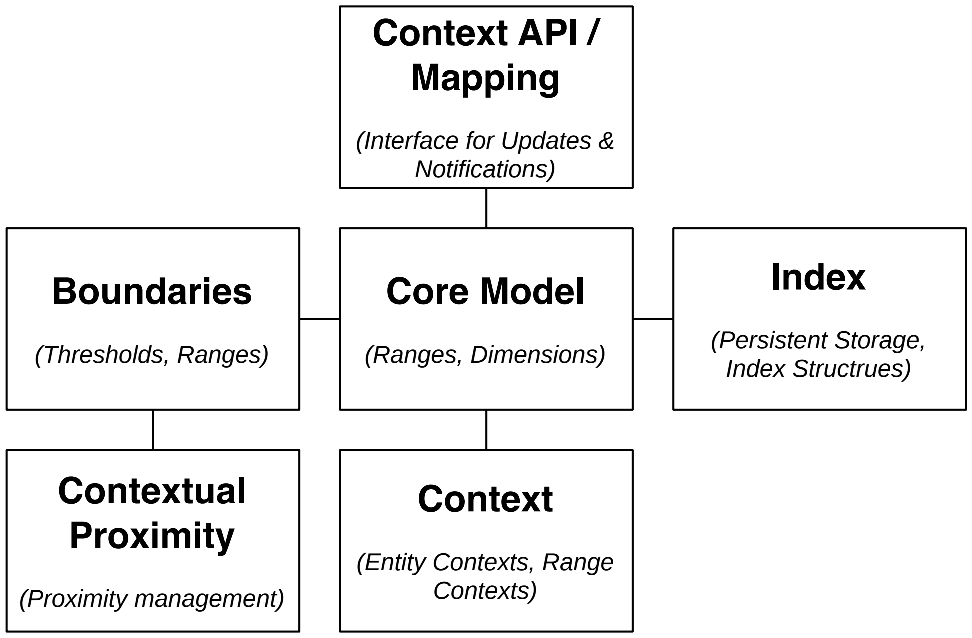


Figure 7.1: Components of the Contextual Map Prototype

7.2.2 Model Specification

With the components introduced, we can finally provide a complete sketch of the architectural model. Figure 7.2 shows the static model as a UML class diagram [14]. Note that figure 7.2 is a specialization of figure 7.1 where each package in figure 7.2 corresponds to a component in figure 7.1. We briefly explain the functionality of each package in regard to the component specification of the previous section 7.2.1:

- *Core model:* implements the basic map composition required to capture and handle entity contexts
- *Context:* enables representation of entity contexts
- *Index:* enables persistent and efficient storage of entity contexts
- *Boundary:* allows definition of contextual similarity degrees
- *Proximity:* enables contextual proximity detection
- *Mapping:* provides an interface for utilizing the contextual map

Core Model Package (contextualmap.core.model)

This package implements the contextual map's core structure. The `ContextualMap` class aggregates all relevant contextual ranges that are represented by the `ContextualRange` class. In addition, it is associated with the `IndexController` class (Index Package, see below), which controls persistent context storage, and the `BoundaryListener` class (Boundary package, see below), which is responsible for identifying crossings of contextual boundaries.

The `ContextualRange` superclass is abstract and specialized by three subclasses depicting the different contextual ranges in the contextual map: bounded, semi-bounded and unbounded. The `ContextualDimension` class aggregates here. It depicts that a contextual range is composed of multiple contextual dimensions.

Context Package (contextualmap.core.model.context)

This package manages entity contexts in the contextual map. The `Context` class represents a complete entity context and is thus composed of range contexts, which are represented by the `RangeContext` class. Each range context consists of multiple scalar values (one per dimension on that particular contextual range) represented by the `ContextualValue` class. The type of an entity is given by the `EntityType` class.

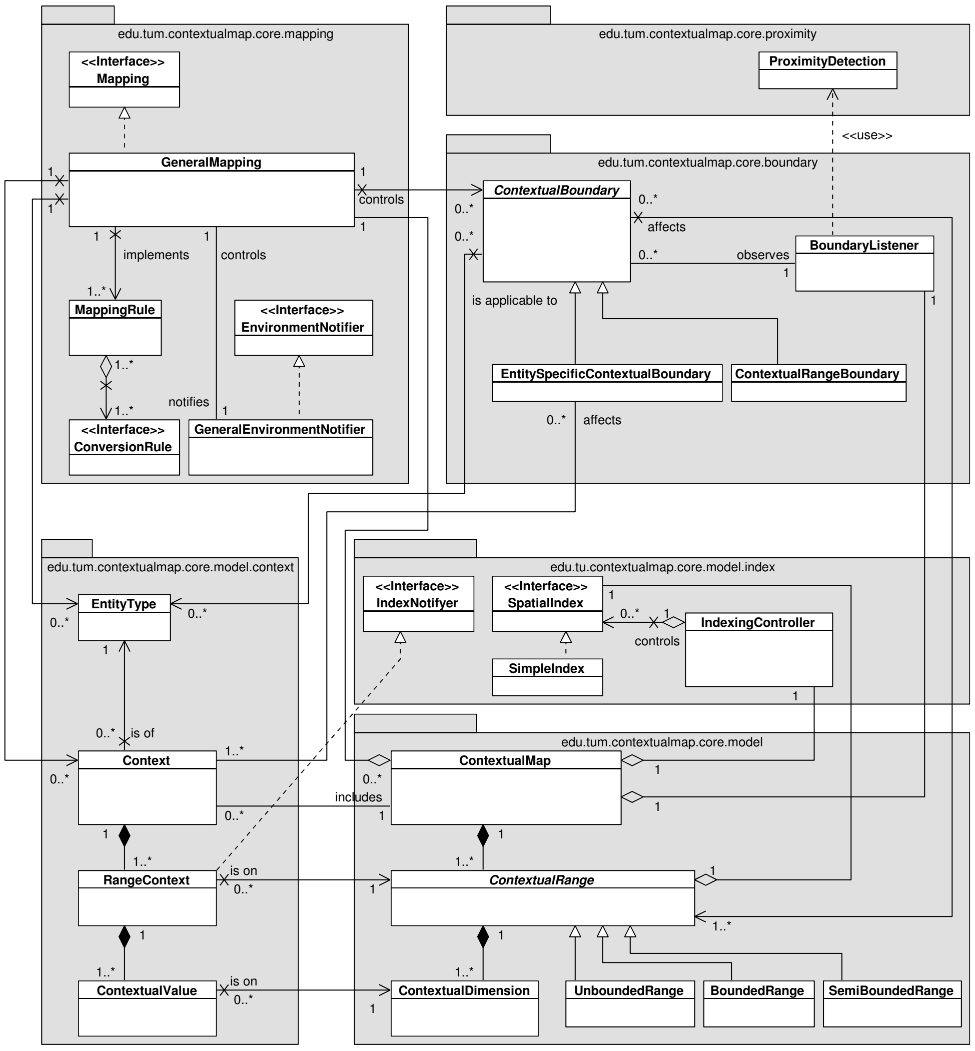

edu.tum.contextualmap.core.mapping
<<Interface>> Mapping
GeneralMapping
implements
controls
MappingRule
<<Interface>> EnvironmentNotifier
notifies
<<Interface>> ConversionRule
GeneralEnvironmentNotifier
edu.tum.contextualmap.core.proximity
ProximityDetection
<<use>>
edu.tum.contextualmap.core.boundary
ContextualBoundary
affects
observes
BoundaryListener
is applicable to
EntitySpecificContextualBoundary
ContextualRangeBoundary
affects
edu.tum.contextualmap.core.model.context
EntityType
is of
Context
RangeContext
is on
ContextualValue
is on
edu.tu.contextualmap.core.model.index
<<Interface>> IndexNotifyer
<<Interface>> SpatialIndex
IndexingController
controls
SimpleIndex
edu.tum.contextualmap.core.model
ContextualMap
includes
ContextualRange
ContextualDimension
UnboundedRange
BoundedRange
SemiBoundedRange

Figure 7.2: Static Prototype Architecture

Index Package (contextualmap.core.model.index)

This package manages the data structure that provides persistent storage for contexts. The `IndexController` class provides access to all index structure used in the contextual map model. The index structures themselves must be implemented by a class implementing the `SpatialIndex` interface that dictates proper access to a spatial indexing structure. We use a very basic index structure realized by the `SimpleIndex` class. The `IndexNotifier` interface is implemented by all classes that trigger an update on contexts and thus, their respective representation in the index structure.

Contextual Boundary Package (contextualmap.core.boundary)

Contextual boundary management is provided by this package. The abstract `ContextualBoundary` class represents a boundary. It is a superclass to two specializations:

- The `ContextualRangeBoundary` class represents the boundary that is applicable to all range contexts of affected ranges. It is thus associated to the `ContextualRange` class. It is the boundary type that we have been talking about throughout this work.
- The `RangeContextBoundary` class represents a boundary that is only applicable to certain range contexts, not contextual ranges. It represents a range-independent alternative if not all range contexts on a range are supposed to be affected by the boundary.

It is to be noted that `ContextualBoundary` possesses an association to `EntityType`, allowing to constrain its validity on particular entity types, as discussed earlier in section 7.2.1.

This package also enables detection of boundary crossings and thus contextual proximity detection. An observer patterns causes the `BoundaryListener` class to observe the `ContextualRange` class (Core package, see above). Contextual ranges notify the listener when any changes to their included range contexts have occurred, so that boundary crossings can be identified. The `RangeContext` class (Context package, see above) is embedded in another observer pattern with the `ContextualRange` class in which the latter observes the former. Since `RangeContext` implements the `IndexNotifier` interface, it is the class that is detecting contextual changes first. In that case, the contextual values are updated and the observing `ContextualRange` class is notified of the changes implementing the first observer pattern. `ContextualRange` notifies the `BoundaryListener` class implementing the other observer pattern. Figure 7.3 visualizes those two observer patterns in our architecture.

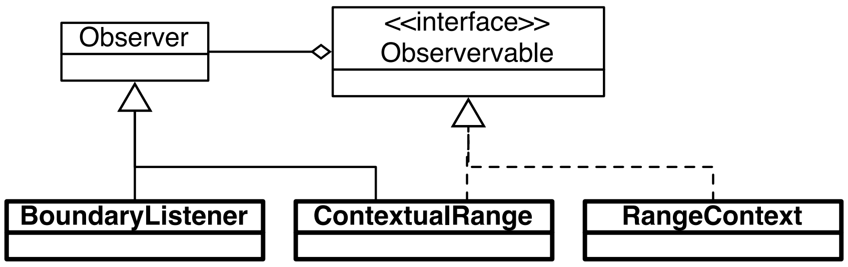


Figure 7.3: Observer Pattern for contextual Update Management

Contextual Proximity Package(contextualmap.core.proximity)

The proximity package is rather small and dedicated to the identification of contextual proximity among entity contexts. The `ProximityDetection` class implements this detection logic. It is triggered by the `BoundaryListener` class each time when a contextual update occurs (see the 2-tier observer pattern discussed above). `ProximityDetection` itself is embedded into another observer pattern. It is observed by a notifier from the mapping package, so that boundary crossings can be propagated outside the context model (see discussion on mapping package below).

Context API and Context Mapping (contextualmap.core.mapping))

The mapping package is dominated by the `Mapping` interface, which dictates the access to the contextual map. It provides initialization access for creating contextual ranges, boundaries and mapping rules.

The `Rule` interface serves for definition of `MappingRule` classes implementing such rules. Mapping rules consist of conversion rules (realized by the classes implementing the `ConversionRule` interface) that translate the entity's quantified context data (i.e. scalar values) to units used by dimensions in the contextual map. Mapping rules align those conversion rules to a mapping, so that each conversion rule is actually mapped to a particular dimension (note that a conversion rule may possess multiple input parameters). Hence, a mapping rule enables entities to update their context by committing their contextual information to a mapping rule in the `Mapping` interface. An entity's contextual data is then translated to the contextual map's dimensions according to the mapping rule's conversion rules. See figure 7.4 for a mapping rule's composition and its role in the update process.

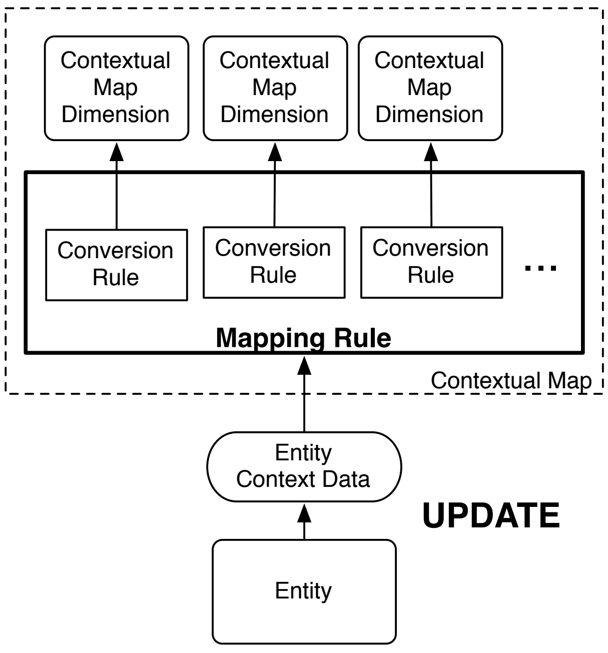


Figure 7.4: Mapping Rule during Update

Last but not least, this package includes a notifier implementing the `EnvironmentNotifier` interface that notifies the outside-world of contextual proximity detected by the contextual map model in regard to defined contextual boundaries. This particularly concerns the context-aware system that utilizes the contextual map. It is being notified by this dedicated notifier (implemented by `GeneralEnvironmentNotifier` class), which is embedded into an observer pattern with the `ProximityDetection` class, as discussed above. Since the contextual proximity detection logic is observed by the notifier, contextual proximity alerts are propagated to the notifier.

7.3 Workflow Specifications

In this section, we focus on the dynamic aspects of the prototype design. The main workflows of the prototypic context model - the initialization, contextual update and contextual proximity detection - are described as follows. It is advised to consider the graphical visualization of those workflows in the UML activity diagram provided in figure 7.5.

7.3.1 Initialization

This is the first sequence of actions, which are taken to get the contextual map up and running. It basically instantiates all of the contextual map's necessary data structures.

1. *Create entity types:* As stated in section 7.2.1, each entity is assigned an certain type (e.g. "person", "car", "place", etc.). This is necessary to allow a fine-granular definition of contextual boundaries. Those boundaries can be specified which entity types they actually should affect.

2. *Create contextual ranges:* This step encompasses the definition of the contextual map's dimensions and thus, its contextual ranges grouping those dimensions disjointly. A spatial index is created for each contextual range.

3. *Create context mapping rules:* Mapping rules, which serve as access points for entities to commit context data, are defined next. This works by first defining appropriate conversion rules for context data translation and aligning those conversion rules to mapping rules accordingly.

4. *Register contextual boundaries:* With the basic structure of the contextual map as well as context mapping set up, contextual boundaries can be defined next.

5. *Register entities:* At this point, the contextual map is initialized and ready to perform contextual proximity detection. Hence, the next step consists of registering entities that can begin committing their current context data.

Figure 7.5 visualizes this workflow as a UML activity diagram and shows dependencies among the encompassed activities.

7.3.2 Contextual Update

A contextual update that has been committed by an entity contains raw contextual data, which - in their initial state - are of no use for the contextual map. The following workflow describes how the entity's context data is stored in the contextual map.

1. *Identify entity context:* An entity update must be uniquely traceable to its originator. This is usually done by the entity ID being included in the update. Once the committing entity is identified, its context representation in the contextual map (the multi-dimensional spatial point) must be determined. If the context does not yet exist in the contextual map - meaning that the entity has never committed an update before[1] - a "plain" context object is created, i.e. no coordinates are known yet since the contextual data has not been mapped yet. In either case, a updatable context is determined, which we can apply the update to.

2. *Context mapping:* The update has been committed by the entity naming a particular mapping rule (reminder: mapping rules represent the "gateway" to the contextual map for entities committing their contextual updates). The mapping rule's conversion rules are applied on the contextual data supplied by the entity. The resultant values from the mapping rule are mapped to the according dimensions (see figure 7.4). The data resulting from the context mapping procedure is clearly assigned to the respective dimensions and quantified accordingly so that the context (the spatial point) in the contextual map can be updated.

3. *Update context:* Updating the regarded context in the contextual map (point 1) consists of applying the mapped context data (point 2) on it, so that it holds the new contextual values corresponding to the entity update. If this is the first entity update, this procedure generates a full context from the plain context mentioned earlier. The affected spatial indexes are updated as well, i.e. the correspondent entry in the index is either updated (updating existing context) or a new entry is added (creating new context) to the index.

Figure 7.5 visualizes this workflow as a UML activity diagram and shows dependencies among the encompassed activities.

7.3.3 Contextual Proximity Detection

A contextual update always qualifies for a contextual proximity check. The changed context may have caused some contextual boundaries to be crossed. However, it is not imperative to perform a contextual proximity check after each update. A contextual update just qualifies its execution. Checking if and which boundaries have been crossed, requires checking all updated contextual ranges for contextual proximity changes.

[1]hence, no data to represent the context is known

1. *Determine updated contextual ranges:* The contextual update has affected a certain number of dimensions. All contextual ranges that those dimensions belong to have to be included in the boundary check. For each of those ranges, the following procedure is conducted:

   (a) *Determine contextual vicinity:* The according range context of the updated context on the current contextual range is identified and its current range vicinity, i.e. all relevant range contexts in its spatial neighborhood, is determined. We have discussed the range vicinity's composition in section 4.4.2. In section 4.4.3, we have stated that we require the "old" range vicinity from before the update as well, so that changes in contextual proximity can actually be detected. In summary, we fetch the "old" range vicinity (before the update) from the repository associated with the range context, and we fetch the "new" range vicinity[2] (after the update) from the contextual range's spatial index. The context update has already been processed at this time, so the index includes the most current range contexts.

   (b) *Perform contextual proximity detection:* Since contextual boundaries can be applied to entire contextual ranges, individual entities and individual entity types, we have to check the updated context and all range contexts in the range vicinity (both before and after the update). For this reason, all range contexts in the current range vicinity are iterated and regarded in two additionally nested iterations:

      - iterate all boundaries regarding *updated range context* and detect proximity between updated range context and currently iterated range context regarding current boundary
      - iterate all boundaries regarding *currently iterated range context* and detect proximity between currently iterated range context and updated range context regarding current boundary

   (c) *Remember range-specific threshold crossings:* If the detected contextual proximity changes have exceeded thresholds of any checked boundaries, those threshold crossings are remembered for the currently iterated contextual range.

2. *Determine range-global boundary crossings:* The remembered threshold crossings that have been individually recorded for each range are used to determine crossings of contextual boundaries. Contextual boundaries are regarded as crossed if all of their range-specific thresholds have been exceeded in a common direction.

3. *Communicate boundary checks:* The detected boundary crossings are communicated to the notifier in the context API using the observer pattern between the boundary listener and the notifier explained in section 7.2.2.

Figure 7.5 visualizes this workflow as a UML activity diagram and shows dependencies among the encompassed activities.

[2]this is actually *one* range vicinity at two different points in time

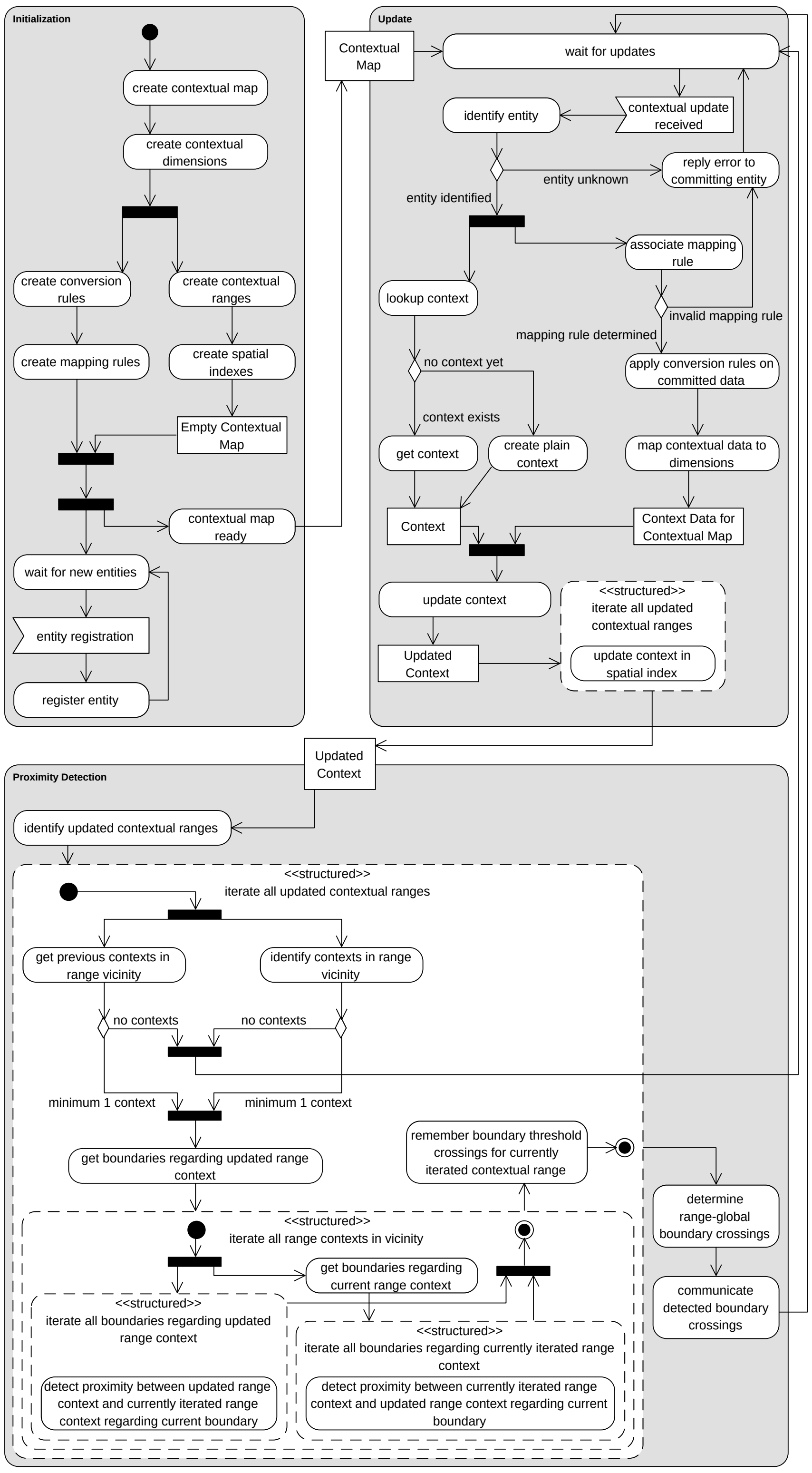

Initialization
create contextual map
create contextual dimensions
create conversion rules
create contextual ranges
create mapping rules
create spatial indexes
Empty Contextual Map
contextual map ready
wait for new entities
entity registration
register entity
Update
Contextual Map
wait for updates
contextual update received
identify entity
entity unknown
reply error to committing entity
entity identified
associate mapping rule
lookup context
invalid mapping rule
mapping rule determined
no context yet
apply conversion rules on committed data
context exists
get context
create plain context
map contextual data to dimensions
Context
Context Data for Contextual Map
update context
<<structured>> iterate all updated contextual ranges
update context in spatial index
Updated Context
Updated Context
Proximity Detection
identify updated contextual ranges
<<structured>> iterate all updated contextual ranges
get previous contexts in range vicinity
identify contexts in range vicinity
no contexts
no contexts
minimum 1 context
minimum 1 context
get boundaries regarding updated range context
remember boundary threshold crossings for currently iterated contextual range
determine range-global boundary crossings
communicate detected boundary crossings
<<structured>> iterate all range contexts in vicinity
get boundaries regarding current range context
<<structured>> iterate all boundaries regarding updated range context
detect proximity between updated range context and currently iterated range context regarding current boundary
<<structured>> iterate all boundaries regarding currently iterated range context
detect proximity between currently iterated range context and updated range context regarding current boundary

Figure 7.5: Workflows of Prototype

Chapter 8

Realization and Validation

With the prototype design of the contextual map given, we are ready to put it to work. This chapter documents a test case that utilizes the prototypic context model. In section 8.1, we illuminate some aspects concerning the implementation of the contextual map model. Subsequently, we put the context model to a test in section 8.2.

8.1 Implementation Aspects

This section briefly discusses the implementation of the context model and evaluates possible enhancements. We especially present a discussion about the performance of indexing structures. We also provide a brief discussion about the application of context clustering.

8.1.1 The Core Model

We have implemented the model from chapter 7 as a Java application running on JRE 1.6. Besides the JRE system library, we only employ a dedicated external logging component, namely log4j [3], to trace various aspects in the prototype during runtime. This yields a light-weight easily deployable system.

The entire context model is shielded by the context API allowing transparent access to the context model. The context-aware client system using the contextual map only needs knowledge about the `Mapping` interface (see figure 7.2) to access the complete functionality of the contextual map. As stated in section 7.2.2, this interface represents the context API and thus, the main access point to the context model.

8.1.2 Index Structure Selection

We have employed a very basic indexing structure provided by the JRE library. For this reason, we have neglected the focus on persistent storage and performance. All contexts are stored and queried in a J2SE `HashMap` object at runtime only. This means that upon shutdown of the context model, all data is lost. However, for our testing purposes this suffices. Our focus is put on a proof-of-concept scenario rather than on performance. This means that our emphasis is put on validation of the mechanisms presented in this work, i.e. which enable contextual proximity detection between entities. Performance is not important at this point.

Despite that approach, we have evaluated suitable indexing structures employable as persistent indexes for the contextual map. By doing so, we continue our initial discussion from section 3.3.1. Figure 8.1 shows the evolution of indexes since the late 1980s. The figure shows how newer indexes have evolved from existing ones and which indexes have outperformed others in direct performance tests (note that evolved indexes usually outperform their ancestors). This analysis is based on the publications of the respective indexes [25, 27, 28, 34, 48, 47, 55, 84, 93, 98].

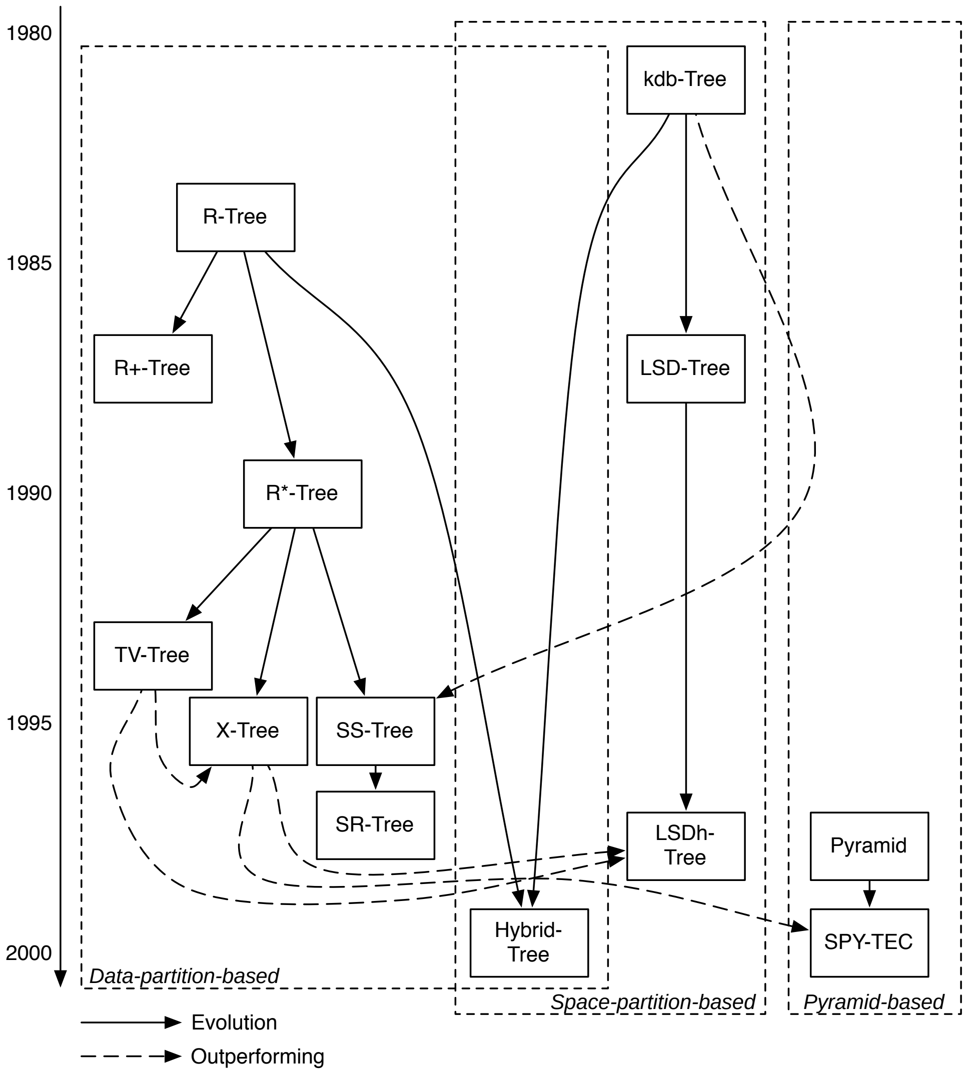


Figure 8.1: Comparison of Index Structures

As indicated by the figure, we have grouped the index structures according to their approach how to manage their data. Data-partitioning indexes (R-tree and derivatives, see figure 3.3) partition the points to be indexes according to their actual values. Space-partitioning indexes (kdb-tree and derivatives, see figure 3.4) partition the index space into disjoint parts according to the values of the points to be indexed. Pyramid-based indexes implement a special case of space partitioning by dividing the index space into pyramids (see SPY-TEC in figure 4.7).

To actually benchmark the performance of indexes, we have selected three index structures whose source-code is freely available [2, 1]:

- *SR-tree [55]:* As a quite recent index structure from the domain of data-partitioning index structures, we have expected it to perform well.
- *Hybrid-tree [34]:* This index structure is inspired by both data-partitioning and space-partitioning techniques, unifying the best of both worlds. Since it is one of the most recent indexes available (1999), we have expected this index to perform well, too.
- *R*-tree [25]:* Despite representing the most advanced index out of the R-tree family, it is outdated and serves as a reference benchmark for our tests.

We have benchmarked those three indexes using test cases with various numbers of data sets and dimensions. We have evaluated three basic operations:

- *Initialization:* Given a set of multidimensional points, this operation builds up the index. It is performed only once at the beginning before the index commences regular operation, i.e. before queries and insertions are enabled.
- *Insertion/Update/Removal:* This operation inserts, updates or removes a given data set of points into, in or from an existing index.
- *Query:* We have evaluated two types of queries on the indexes:
  - the *k-nearest-neighbor-query* yielding the k nearest neighbors of a given query point.
  - the *range query* yielding all points within a radius r of a given query point.

Our results in figures[1] 8.2 through 8.5 show that the Hybrid-tree clearly outperforms the two other candidates. It therefore represents a suitable index for indexing contextual ranges, i.e. it is capable of storing large amounts of high-dimensional range contexts.

The "curse of dimensionality" has a significant impact on the R*-tree and at least a noticeable impact on the SR-tree. However, in contrast to those two indexes, the Hybrid-tree scales quite well with increasing dimensionality.

[1]please note that those are results of the denoted fabricated test cases, which do not reflect any real-time scenario

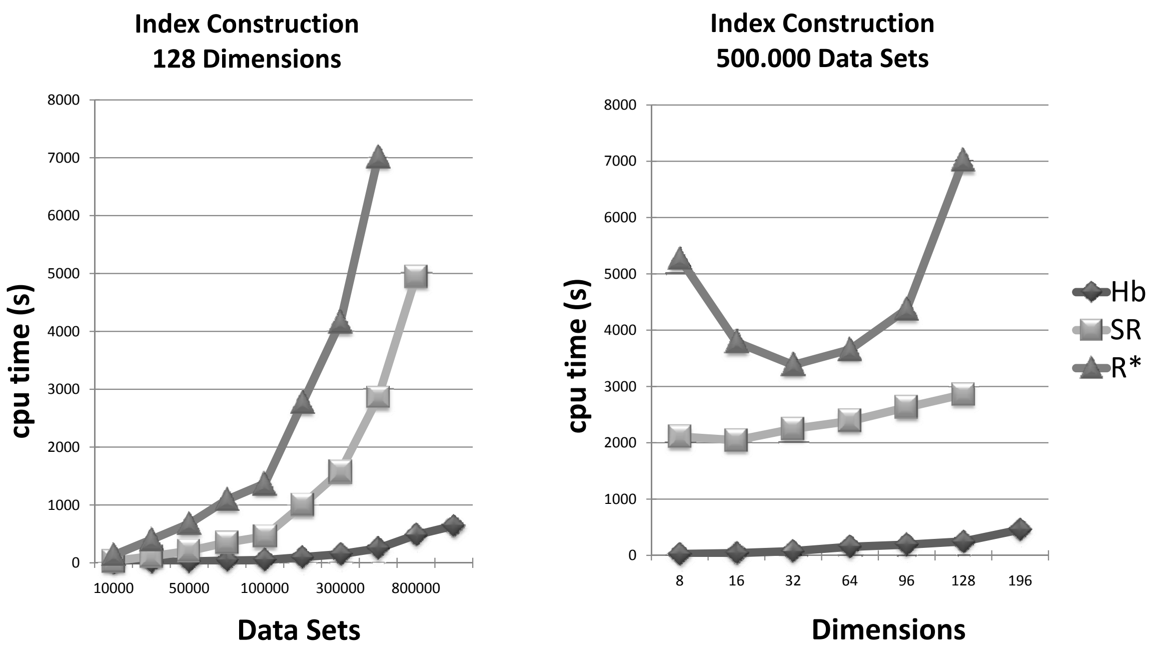


Figure 8.2: Index Construction Benchmark

Regarding the individual operations, index construction is clearly the most elaborate operation (figure 8.2). Even the Hybrid-tree performs considerably worse at index initialization than at its other operations. Inserting and querying data sets shows exponential growth with increasing numbers of inserted or queried data sets (figure 8.3 for insertion and figures 8.4, 8.5 for querying). While the exponential growth rate is extreme in case of the R*-tree, it is considerably less sized in case of the SR-tree. The Hybrid-tree performs best with a hardly noticeable exponential growth rate. Resembling figure 8.1, the most advanced indexes that where spawned by the evolutionary development processes are represented by the SR-tree (1997, data-partitioning), the Hybrid-tree (1999, hybrid), the LSDh-tree (1998, space-partitioning) and SPY-TEC (1999, pyramid-based). All other stated indexes are inferior leaving those four as the best performing indexes available and qualifying them for utilization in the contextual map. We have proved that the Hybrid tree clearly outperforms the SR-tree in any given test. This eliminates the SR-tree from the result set of suitable contextual map index structures. Concluding, SPY-TEC and LSDh-tree may also perform very well besides the Hybrid-tree. However, a detailed comparison is out of scope of this work, since we have identified an index - the Hybrid-tree - that scales extremely well with large numbers of data sets and dimensions.

8.1.3 Application of Context Cluster Detection

As stated in section 5.2, clusters of similar entity contexts can be identified with the help of the contextual map. With an efficient index structure in use, context clustering can be implemented efficiently. We have provided a survey on clustering algorithms in

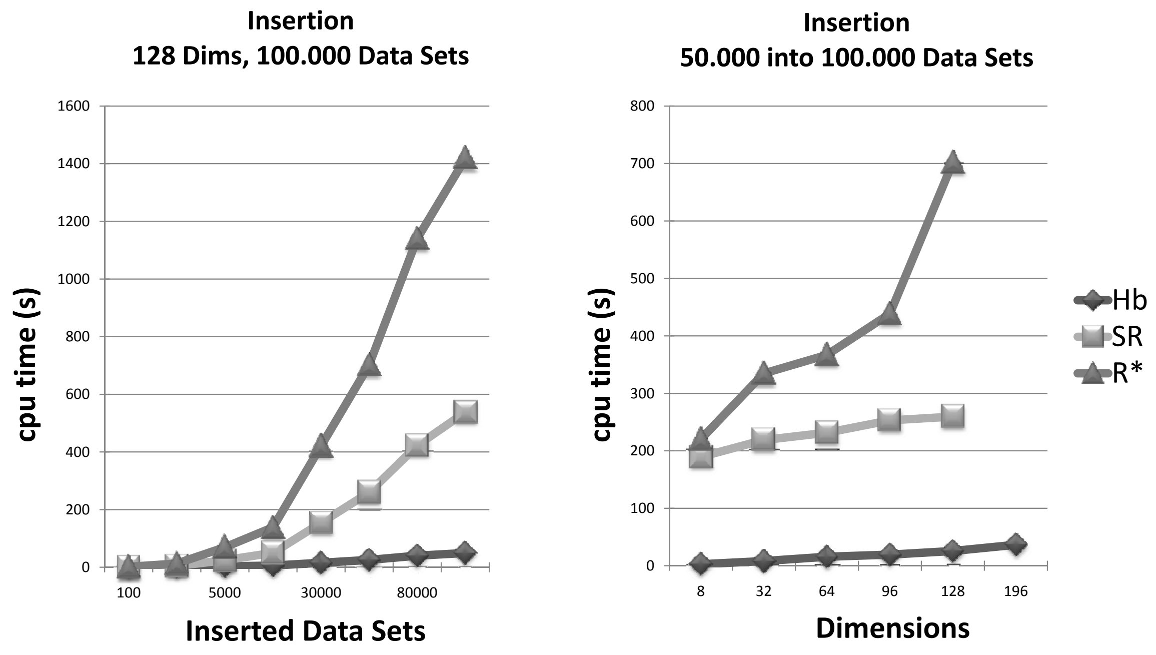


Figure 8.3: Insertion into Index Benchmark

section 5.2.1, which can be employed for determining contextual-range-specific clusters. All clustering algorithms require quickly acquirable knowledge about the exact positions of the points to be clustered. A contextual map backed by a fast-accessible index for each range can provide decently performing clustering of range contexts.

Consequently, contextual-range-specific clusters are used to derive inter-range clusters (as described in section 5.2.2), which represent the final set of clustered entity contexts provided by the context API. If entity context encompasses rich context types with many attributes (i.e. high-dimensional contextual ranges), clustering algorithms for high-dimensional spaces should be employed (see section 5.2.4). Those especially include *subspace clustering* [59] and *pre-determination of clusters using cheap metrics* [68].

8.2 Experimental Results

Testing the contextual map in a real scenario requires some preparation. We need a real-world scenario that provides context and a context-aware system utilizing it. We have set up a scenario inspired by the setting in section 4.1: Car drivers roaming the streets who want to be notified about extreme weather conditions. The subsequent subsections present a brief introduction into the scenario setting and the results of putting the scenario to work with the contextual map.

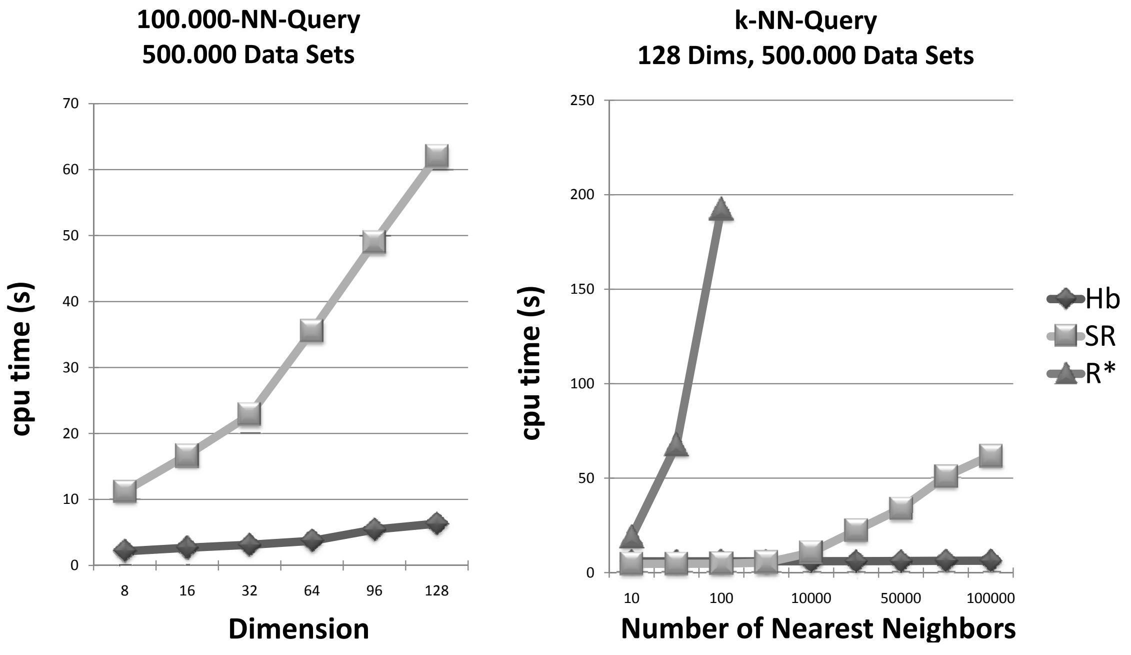


Figure 8.4: k-NN Query on Index Benchmark

8.2.1 Test Scenario Setting

As stated, the scenario mainly consists of vehicles driving on roads wanting to be notified about extreme weather. Each driver defines a custom extreme weather condition that he considers dangerous enough to be notified about. Weather stations provide current weather readings. Basically, we can state that certain contexts of a driver and a weather station get similar, if a driver approaches a weather station whose environmental readings are quite close to the driver's conception of extreme weather.

We employ the contextual map to realize this scenario. A contextual boundary enables us exactly the contextual similarity definition that has been stated above. Since we plan to perform a proof-of-concept test, the entire scenario runs as a simulation. However, we make use of a real road network and real-time weather conditions whereas vehicles and drivers are simulated.

In order to employ the contextual map model, we need to generate a proper way to access its context API. We have to define entities, contextual boundaries and mapping rules in regard to the environment that we capture the contextual data from. The remainder of this section is dedicated to the proper setting of the scenario in regard to those aspects.

Environment

The environment can be broken down to two components: geographical location and weather.

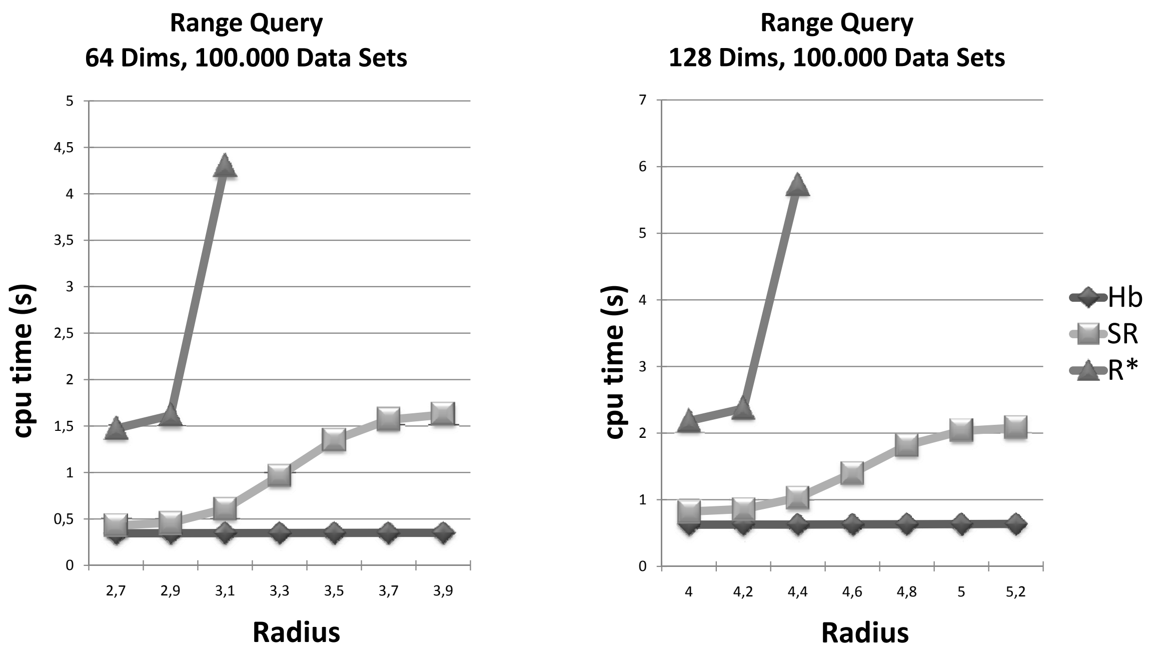


Figure 8.5: Range Query on Index Benchmark

Location Concerning location, the current position of vehicles and weather stations are relevant. Vehicles move on predefined tracks. We have extracted 50.000 kilometers of tracks in Southern Germany from freely accessible GPX tracks[2]. Those tracks correspond to the real-world road network, since they have been generated by people driving on those roads and using their GPS-devices. Vehicles move on the network spawned by the set of GPX tracks.

Weather The weather in the scenario is location-specific and bound to the coordinates of the according weather station. We employ real-time weather fetched from the YAHOO weather API [17] for 20 weather stations that exist within the coverage of our regarded road network. The weather read by a particular weather station is valid for all places to which that weather station is the closest one.

Entities

The identification of entities is straight-forward: we regard vehicles and weather stations (as we already did in section 4.1). Each entity has an ID and a set of context sensors attached that are utilized to capture context data. The acquired sensor data is then committed to the contextual map's `Mapping` interface from where it is processed and committed to the contextual map.

[2]GPX tracks are sequences of GPS-generated geographical points

Moving vehicles Each vehicle moves on one of 128 tracks that are registered in the system. The particular track corresponds to one of the GPX tracks that altogether spawn the road network used in our scenario. The vehicle is moving at constant speed and stores its current position on its assigned track as DMS coordinates. Concerning the set of attached sensors, the moving vehicle has a location sensor and a weather sensor that *reports the weather condition* at its location (thus, those conditions correspond to the environmental readings of the closest weather stations). In addition to that, each vehicle defines its individual perception of strong weather conditions. A vehicle entity considers three weather readings as possibly critical: strong wind, extreme temperature and low visibility. Thus, a vehicle's relevant context is comprised of its location and its perception about critical weather.

Weather stations A weather station stores the most recent weather conditions that have been retrieved from the YAHOO weather API [17], as well as its location in DMS coordinates. Since a weather station's location is static, a weather station has only a weather sensor *reading the current weather conditions* at the particular location. Hence, the weather station's relevant context is comprised of its static location and currently read weather.

Contextual ranges

The relevant context types that we utilize in this scenario are also inspired by the application case from section 4.1. Hence, we define two contextual ranges:

- An unbounded range R_{loc} depicting location.
- A bounded range R_{env} depicting environmental weather conditions. The context range is bound within the numeric range [0..100].

Context Mapping

We have defined the following mapping rules to map raw contextual data to the contextual map. We design the mapping function $f_{mapping}$ accordingly as defined in section 3.2. Data from two sensors is regarded[3]:

- *location:* delivers geographical DMS coordinates (latitude and longitude): (s_{lat}, s_{long}).
- *weather:* delivers weather conditions: $(s_{temp}, s_{hum}, s_{wind}, s_{vis}, s_{air})$
  - temperature s_{temp} in °C in between -50 and 50°C
  - humidity s_{hum} in per cent
  - wind speed s_{hum} in meters per second between 0 and 100 m/s
  - visibility s_{vis} in kilometers between 0 and 50 km

[3]note that *location* and *weather* are sensor IDs

– air pressure s_{air} in millibar between 880 and 1080 mbar

Thus, we can define $f_{mapping}$ as follows:

$$f_{mapping}: \begin{cases} \left(location, \begin{pmatrix} s_{lat} \\ s_{long} \end{pmatrix}\right) & \rightarrow \begin{array}{l} \Big\{ \Big(dim_x(R_{loc}), x\Big), \Big(dim_y(R_{loc}), y\Big) \ \Big| \\ x = 111.3 * \Big(s_{lat}(deg) \pm \frac{s_{lat}(min)}{60} \pm \frac{s_{lat}(sec)}{3600}\Big), \\ y = 111.3 * cos(s_{lat})\Big(s_{long}(deg) \pm \frac{s_{long}(min)}{60} \pm \frac{s_{long}(sec)}{3600}\Big)\Big\} \end{array} \\ \left(weather, \begin{pmatrix} s_{temp} \\ s_{hum} \\ s_{wind} \\ s_{vis} \\ s_{air} \end{pmatrix}\right) & \rightarrow \begin{array}{l} \Big\{ \Big(dim_t(R_{env}), t\Big), \Big(dim_h(R_{env}), h\Big), \\ \Big(dim_w(R_{env}), w\Big), \Big(dim_v(R_{env}), v\Big), \Big(dim_a(R_{env}), a\Big) \ \Big| \\ t = (s_{temp} + 50°C)\frac{1}{°C}, \\ h = s_{hum}, \\ w = s_{wind}\frac{s}{m}, \\ v = (s_{vis} * 2)\frac{1}{km}, \\ a = (s_{air} - 880)/2\frac{1}{mbar} \\ \Big\} \end{array} \end{cases} \tag{8.1}$$

Note that there are actually two different weather sensor types: one *reading* and one *reporting* sensor. The former actually *measures* and reports weather (employed by weather station entity) while the latter only *reports* weather based on the readings of the closest weather station (used by moving vehicle entity).

Contextual Boundary Setting

We need a boundary that reports if vehicles enter areas with strong weather, i.e. if they are approaching weather stations reporting such weather. Thus, we need a boundary defined on the location and weather range as follows:

- *location:* The radius defining the area for which a weather station is supposed to support its weather readings. In our scenario, we have defined it 30km equaling 30.000 units on R_{loc}.
- *weather:* The difference (in terms of contextual proximity, i.e. Euclidean distance on R_{env}) between a vehicle's critical weather perception and a weather station's weather readings. Sufficiently spaced, we choose 10 units on R_{env}, which denotes 10% of the value range of each of R_{env}'s dimensions.

Further, the boundary is only applicable to vehicles and only defines contextual proximity of those to weather stations. Thus, the *vehicle* and *weather station* entity types are regarded by the contextual proximity management procedures as follows:

- regarded entity type of updating entities: *vehicle*
- entity type to be reported as contextually converging and separating: *weather station*

Design and Implementation

The entire test setting is divided into three parts:

- *Contextual map model:* The core implementation of the context model as presented in chapter 7 providing contextual affinity management.
- *Scenario simulation:* Responsible for emulating the environment surrounding the context-aware system and thus, the contextual map. I.e., the simulation is responsible for providing entities that generate contextual data, which is dynamically altered according to real-world specifics, such as the real road network and real-time weather.
- *Context-aware client system:* This part serves as the mediator between the scenario simulation and the contextual map model. It is responsible for initializing the contextual map (contextual ranges, boundaries, mapping rules), forwarding entity updates to the contextual map and reporting boundary crossings received from the contextual map.

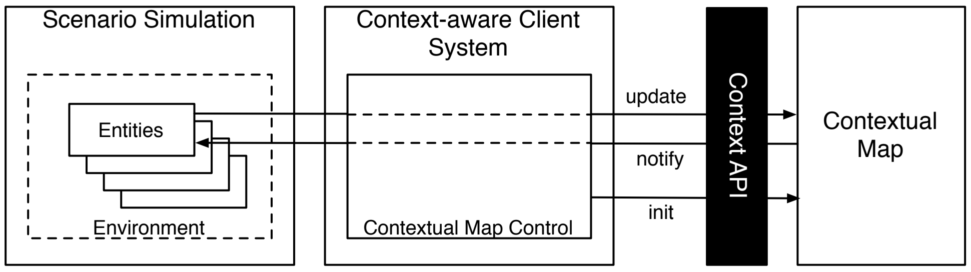


Figure 8.6: Scenario Setting

Figure 8.6 depicts this setting. At this point, we briefly discuss the architectural design of the scenario simulation and the context-aware client system. Figure 8.7 resembles the architectural sketch. The environment is driven by two controller classes: `LocationController` and `WeatherController`. Both are static and accessible by the entire system. The `LocationController` uses the `TrackController` that controls all GPX tracks represented by the `Track` class. The `WeatherController` controls weather stations and stores the measured `WeatherConditions`. The `Entity` class and its descendants `MovingVehicle` and `WeatherStation` represent the system's entities. They possess attached sensors represented by the `Sensor` class and its children `LocationSensor`, `WeatherReadingSensor` and `WeatherReportingSensor`. Entities also make heavy use of the data holding classes employed by the above-explained controllers. Entities commit their context data to the context-aware system represented by the `ContextualMapController` class, that controls the contextual map and forwards all entity updates to it.

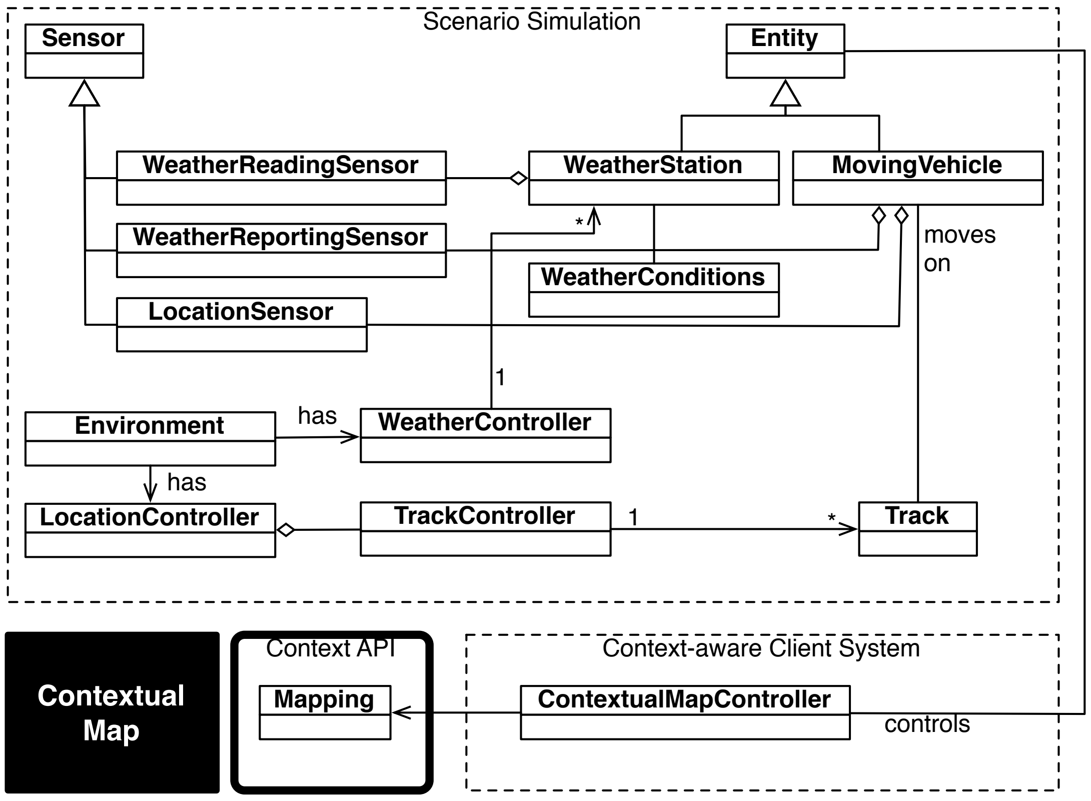


Figure 8.7: Scenario Simulation Architecture

8.2.2 Simulation Results

With moving vehicles and weather stations present in our simulated environment, we can perform a test run and observe how contextual proximity changes occur. Vehicles and weather stations have their context represented as multi-dimensional points in the contextual map. The weather station's context representation in the contextual map is straight-forward: its location and currently measured weather conditions are mapped onto the dimensions as depicted in equation 8.1. Deriving the context representation of moving vehicles is a little more elaborate. While the representation of its location is straight-forward, the definition of its critical weather perception follows the subsequent mechanism. The critical weather perception is dependent on the currently reported weather at the vehicle's current location (provided by its weather sensor). If at least one of the relevant readings (wind, temperature, visibility) *gets close* to a critical value, the vehicle defines its overall critical weather perception as follows. The currently reported weather conditions are copied and defined as its critical weather conditions. The problematic readings in the critical condition perception are then set extreme. E.g., the wind is getting stronger at the weather station in the city of Ingolstadt. Other readings, such as temperature and visibility remain relatively unchanged. A vehicle passes by the city on the freeway and receives those weather conditions. It adapts its critical weather perception by taking the reported weather conditions for Ingolstadt and replacing the real strong wind readings with the extreme wind value that is regarded as dangerous by the vehicle driver. Following this determination mechanism, the vehicle's critical weather perception is mapped adequately in the contextual map (as dictated by equation 8.1).

General observations

To make use of the depicted scenario setting, we have defined a contextual boundary that denotes contextual proximity between vehicles and weather stations exactly then when a vehicle approaches a station that is reporting extreme weather. Running the scenario at days with normal weather, nothing occurs. Once a weather station starts reporting readings that get closer to extreme values, large numbers of contextually converging boundary crossings occur in a short period of time. Because of the worsening weather conditions, all vehicles in the geographical vicinity suddenly become contextually proximate to the weather station reporting the problematic weather. Contextual proximity regarding the defined boundary is given (for the affected vehicles and the weather station), since the vehicles are closer than 30 kilometers - 30.000 units on R_{loc} - to the weather station and the measured weather conditions on R_{env} are no farther than 10 units from any of the affected vehicles critical weather perceptions (note that the distance statements are affecting the corresponding range contexts of vehicles and weather station). This observation validates the statement made about contextual proximity in equation 4.5 in section 4.1. It is to be noted that the change of weather usually isn't a geographically local phenomenon. It occurs in larger areas so that multiple weather stations and even more vehicles (geographically close to those weather stations) are affected. In summary, the changes on R_{env} cause the contextually converging boundary crossings.

After those "initial" boundary crossings, additional boundary crossings are rather sporadic. They occur when vehicles geographically enter the area with bad weather (contextually converging boundary crossing), or if vehicles leave that region (contextually converging boundary crossing). Those boundary crossings are triggered because of changes on R_{loc}. A vehicle's contextual (and in this case also geographical) approach to the weather station on R_{loc} causes contextual proximity in regard to the defined boundary, thus triggering a contextually converging boundary crossing. Separating from the weather station on R_{loc} causes the contextual proximity not to be given anymore triggering a contextually separating boundary crossing.

Once a weather station starts reporting better weather conditions, a large number of contextually separating boundary crossings occurs. Logically, this is because the weather conditions have improved so significantly, so that they do not exhibit contextual proximity to any vehicles on R_{env} anymore. This means that the Euclidean distance between range contexts of vehicles and the weather station on R_{env} exceeds 10 units. The contextual separations caused in this scenario validate equation 4.6 in section 4.1.

Performance and Scalability

We have found out that the scenario simulation controlling the simulated environment requires only a marginal amount of resources. It scales very well in regard to the number of employed entities. Updating several thousands of entities takes only a few seconds. The contextual map's performance, however, scales worse. It exhibits exponential cost with increasing numbers of entities and contextual ranges. In the proof-of-concept prototype, it takes several minutes to update entity contexts in the contextual map and to perform boundary crossing checks, whereas it takes only a few seconds to just update contexts of the same number of entities without utilizing the contextual map.

This observation shows clearly that utilization of the contextual map imperatively requires efficient data structures and well performing platforms when being applied to large-scaled systems. We have further observed that a large portion of the performance drain comes from the utilization of the index structure. We have employed simple indexing structures with no regards to performance, since it would have not contributed to the contextual map's functional validation. However, there is clearly lots of improvement potential given when using well performing index trees (see section 8.1.2). However, although efficient indexes may have decisive influence on the contextual map's performance, they are only one possible aspect of tackling the performance issue. However, further considerations about improving performance (besides index selection) are out of scope of this work.

Chapter 9

Conclusion and Outlook

With the contextual map introduced both in theory and practical application, we close the documentation on this context model in this section. We summarize our work and outline the applicability options of the contextual map. We close by outlining the aspects that have been covered by this work in theory, but which have not been put to practice yet, and by sketching a few ideas for future work on contextual proximity management using the contextual map.

9.1 Summarizing the Contextual Map Concept

With the contextual map, we have proposed a novel context model concept specializing on determining contextual similarities among individual entities. Settling the model in Euclidean space, we are able to exploit Euclidean distance calculations to make those determinations.

Contextual proximity The notion of *contextual proximity* is borrowed from the location-awareness domain. Geographical proximity denotes (at least) two entities being geographically close to each other. We have leveraged that perception into the domain of context-awareness, where entities are contextually proximate, if their respective contexts (or particular pieces of their respective contexts) are similar to each other. This approach yielded the conceptual map, which is primarily conceptualized to handle such contextual proximity.

Cartesian map model We have determined that mapping entity contexts into a Cartesian map model requires contextual information to be partitioned according to context types, so that each piece of information belongs to a particular context type. Contextual information of each context type is assigned a dedicated Euclidean space, where all contextual information of that type is mapped to in the form of scalar attributes. We call those spaces contextual ranges on which each entity has its typified context represented as a multi-dimensional point. The Euclidean distances between those points denote contextual similarity, i.e. contextual proximity. The set of all contextual ranges comprises the contextual map, so that contextual ranges represent dimensional subspaces of the contextual map in turn. In summary, as a context model, the contextual map represents the contexts of all known entities and thus, a global view on the overall contextual situation. Chapter 3 has documented the contextual map model.

Contextual similarity definition We have introduced the notion of contextual boundaries to enable the definition of contextual similarity degrees between different entity contexts. Contextual boundaries define a scalar similarity degree for each contextual range, thus defining a similarity degree for each context type. Regarding all contextual ranges affected by a boundary, we are able to track specific alikeness of contexts, as discussed in chapter 4. Concluding, contextual boundaries allow the triggering of contextual proximity and separation alert on application level, thus reporting which entities have been become proximate (or the opposite) to any context-aware application using the contextual map.

Monitoring and updating We have discussed update semantics for entities committing their most current contextual information as rarely as possible but keeping their context in the contextual map as current as possible at the same time. Monitoring entity contexts and determining if they have become contextually proximate is done by checking current entity contexts against contextual boundaries. Entities become contextually proximate in regard to a particular boundary (i.e. the similarity definition), if the Euclidean distances between their contexts fall below all the boundary's thresholds defined on the relevant contextual ranges (i.e. one threshold per range). Contextual separation exists if this condition does not hold anymore. The context monitoring is conducted by employing a suitable indexing structure that stores all multi-dimensional entity contexts. We have discussed two fundamentally different structures: index trees, which are usually employed to store high-dimensional data, and a spatial grid that partitions Euclidean space into equally sized disjoint areal subspaces[1] (see section 4.5).

Contextual proximity utilization With its contextual proximity management, the contextual map enriches the context API (the interface for utilizing the contextual map, see figure 2.5) for context-aware applications with four access options - two querying methods, one notifier and one setter:

- The *contextual similarity query* yields a set of entities with similar contexts compared to the context of a particular reference entity. The query is highly scalable in terms of how similarity is actually quantified (i.e. specified), and in terms of which contextual information is to be included in the similarity comparison. We have described the contextual similarity query in section 5.1.
- *Context clusters* depict the second major query method for the context API. It yields clusters of entities with similar contexts. Such a cluster corresponds to a set of entities that a context-aware system can regard as one. Since all of those

[1]not to be confused with dimensional subspaces (defined by subset of dimensions)

entity contexts are similar to each other, a context-aware system can concentrate on serving a *single* contextual representation (the cluster's "mean context") instead of adapting to each single entity context. Hence, context clusters facilitate the adaptation process for context-aware applications. The loss of accurately matching an entity's context remains low, since the context matched by the context-aware system is always similar to the respective entity context. Context clusters are also scalable in size. We have provided an extensive discussion on context clusters in section 5.2.

- *Contextual notifiers* report that specified degrees of contextual proximity have been reached between entity contexts. Those notifiers inform any context-aware application on top of the contextual map about the proximity changes. The specification of the "similarity degrees" are directly defined by contextual boundaries. The notifications can be used to trigger further application-specific actions.

- *Contextual realms* are areal subspaces in the contextual map, which can either be derived from the space occupied by a context cluster, or defined arbitrarily. A contextual realm defines a set of particular situations that an entity can be in. If an entity context is included in a contextual realm, additional reasoning about the entity context can be made (see section 5.3). This is because the context of the entity fits a situation defined by the contextual realm.

Employable Architectures We have designed possible system architectures for distributed deployment of the contextual map based on multiple architectural paradigms, including client-server systems, peer-to-peer networks and wireless sensor networks. We have identified the problem that global context actuality (the ability to know every entity's *current* context) cannot always be guaranteed. In fact, only client-server systems provide a satisfactory view on global context. All other architectures only allow approximation of global context. Section 6.1 has discussed the issues of a distributed contextual map.

Heterogeneity As in the entire domain of context-awareness, heterogeneity has had a tremendous impact on the conceptualization of the contextual map. Identifying relevant entity context and mapping it to the contextual map is an application-specific task when employing the contextual map. Although not being in our main focus, we have provided a brief discussion about the issue in section 6.2.

Validation The basic idea of the contextual map - applying location-based proximity management to context-aware computing - has been successfully validated. The client-server-based prototype implementation from chapter 7 has been employed by a real-world inspired scenario that employs both location and additional context. The proximity management mechanisms from the location awareness domain have been successfully applied on non-location context, thus leveraging the notion of *proximity* to context awareness. We have shown that *contextual proximity* is a suitable notion for identifying and managing entities with similar contexts, thus entities that are in similar situations at a particular point in time. Euclidean distance in $\mathbb{R}^n$ has proved to be a good metric for expressing contextual proximity. We have validated contextual boundaries as a suitable data structure for defining contextual proximity. Further, the contextual map is universally applicable. Its mechanisms for detecting and exploiting contextual proximity can be applied to any application case if context mapping and the contextual boundaries are set application-specifically. Our test scenario demonstrates such a real-world-inspired application case exploiting contextual proximity. It is documented in chapter 8.

9.2 Applicability of the Contextual Map

With our work summarized in the previous section, we briefly discuss two of the contextual map's applicability options.

9.2.1 Context Model Augmentation

As stated, the contextual map model turns out to be very suitable for tracking contextual affinity. Our prototypic scenario from chapter 8 has demonstrates the contextual map's applicability as a stand-alone context model. However, its suitability concerning context-aware systems, which provide other context-aware services than managing contextual similarity, is rather small. Out of many proposed context models (see section 2.3), recent research on context-aware systems has identified ontologies as most efficient context models [39, 95]. Despite performing excellently in representing and inferring complex contextual information, ontologies lack the ability to detect and manage contextual similarity as simply as the contextual map.

This situation suggests a hybrid approach. Well proven context models can be augmented by the contextual map model to extend their expressiveness. In order to realize this approach, the contextual map needs to be tied in appropriately into existent architectures. We are going to base our argumentation on the generic architecture depicted in figure 2.5. Being an auxiliary component to the core context model, the contextual map needs to be connected to the context model of choice via a fine-granular interface. Since the contextual map issues alerts upon detecting critical changes in contextual affinity, a connection to the *context API* seems reasonable, too. This enables direct access to those alerts for context-aware applications. Contextual proximity and separation alerts may also be important for inferring new context from existing context. That is why the contextual map should be linked to the *context inference engine*. This allows such alerts (triggered by the contextual map) to infer new context immediately. Just as it seems reasonable to establish links to the context API and the inference engine, it is intimating laying a link to the information acquiring gateway, the *context capturing interface*, as well. Contextual information gathered from context sources and refined by the context capturing interface can be committed to both the context repository and the contextual map where it is processed individually. To arrange the contextual map into the architecture of context-aware systems, we recall figure 2.5 depicting an abstract context-aware system. As an implementation of a context model, the contextual map is to be situated on the reasoning level, together with the context repository. In this setting, the context repository implements a well-proven context model of choice, whereas the contextual map provides contextual proximity management. Figure 9.1 visualizes the integration of the contextual map into the generalized context-aware architecture depicted in figure 2.5.

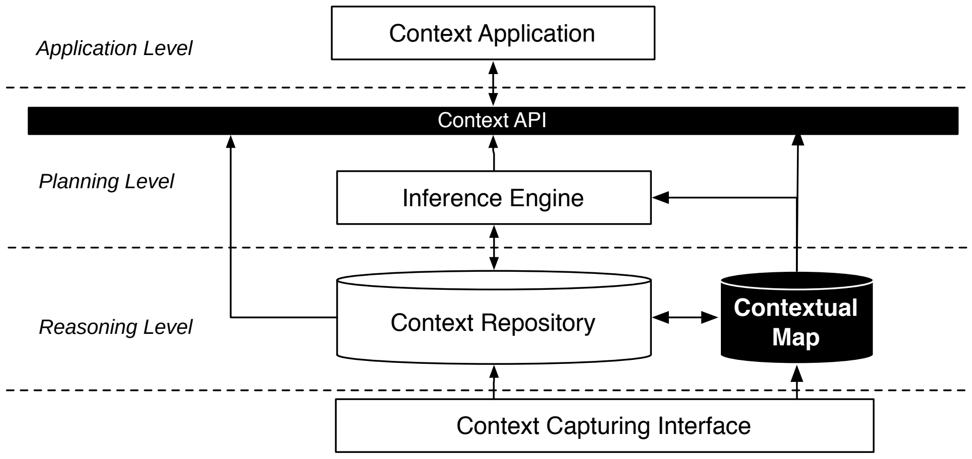


Figure 9.1: Hybrid Context Model

9.2.2 Contextual Proximity Management in distributed Systems

It is to be noted, that the architectures in both figures 2.5 and 9.1 are general in nature. However, many context-aware systems are based on a distributed system design that includes mobile nodes. With mobile devices, there are much more possibilities to acquire context. Generally, mobile devices allow versatile utilization to acquire contextual information wherever necessary. This includes sensor readings on the one hand, and user input on the other.

The sketches in both figures do not make any statement about distribution or mobility. For this reason, we sketch the contextual map's applicability on distributed and mobile systems. In section 6.1, we have elicited how the contextual map can be deployed into distributed architectures. As stated, a current global view on the overall contextual situation is only possible with a centrally located server coordinating the context-aware system. E.g., a cellular GSM network would qualify for that. In such a setting, the server is provided with current data (via GPRS or an equivalent technology) that has been acquired from the mobile device's sensors and user input. Mobile devices do not need to be equipped with a contextual map (they can be, however, to optimize update semantics). This exemplary setting would theoretically allow a globally GSM-driven context-aware system with billions of mobile devices and users.

In mobile peer-to-peer networks, mobile devices generally need to be equipped with a contextual map (exception: H-P2P networks). E.g., mobile devices could be interconnected with a local area wireless technology, such as 802.11 or Bluetooth. In this setting, each device manages its own view on the overall contextual situation while keeping it as updated as possible by sending and receiving contextual updates top/from neighboring devices in reach (again, data acquired from sensors and user input).

Concerning wireless sensor networks, employed mobile devices are usually very resource-constrained and employed for environmental monitoring. Thus, transmission ranges are small, transmission intervals long, and capabilities concerning storage and computation minimal. Since the centrally located base station cannot expect its associated mobile devices to regularly commit contextual updates, its view on the overall contextual setting remains not current.

In summary, employing the contextual map to distributed systems results in drawbacks regarding the view on the overall contextual situations.

9.3 Possible Future Work

The given concept and prototypic implementation of the contextual map leaves several possible aspects for further enhancements.

High-level applicability In this work, we have provided a basic prototype that provides the contextual map's core functionality: contextual proximity detection. Beyond that, we have provided an in-depth discussion on possible application techniques for contextual proximity utilization: the contextual similarity query and context clustering. Besides that, we have introduced contextual realms as an additional structuring criterion for the contextual map. The basic prototype from chapter 7 model lacks support for those three functionalities. Hence, the next reasonable step is to extend the prototype to support those. This logically implies that the context API supports those mechanisms as well: querying entity contexts that are sufficiently similar to a query context, determining context clusters, and deriving or defining contextual realms, i.e. a particular state space of possible entity contexts. As an example, the scenario from chapter 8 is to be regarded. A suitable context cluster may be represented by a location-independent set of vehicles with similar attributes, such as driving safety, driver characteristics, passengers, etc. (location may be included as well, we have chosen to omit it to emphasize non-location context). Defining contextual similarity queries and contextual realms may yield similar application cases.

Index integration The basic prototype also neglects the utilization of a well performing indexing structure. Since the contextual map concept is potentially applicable to large numbers of entities possessing high-dimensional context (in the contextual map), integrating a well performing index represents another reasonably enhancement. We have already performed a benchmark on suitable index trees in section 8.1.2. Besides the index trees, however, the spatial grid approach represents a worthwhile candidate for an index.

Context mapping Throughout this work, we have accented that context-awareness excels heterogeneity in many aspects. For the contextual map, context mapping is of pivotal importance. So far, we have tailored the mapping function, which maps contextual data onto the contextual map's dimensions, application-specifically. Hence, the mapping function is application-dependent and heterogeneous. For this reason, identifying generally applicable principles for context mapping remains an unsolved issue.

Real-world application So far, we have tested the contextual map in a simulation using real-world data (road network, real-time weather). Using the contextual map in a large-scale real-world context-aware application may yield valuable results. As stated before, the contextual map is very specialized in nature by having a narrow focus on contextual proximity only (although it is universally applicable in this particular domain). Thus, augmenting a large-scaled context-aware application, which uses a powerful context model (in terms of encompassing many facets of context), with the contextual map as an auxiliary component, would enrich the application with contextual proximity management. This especially accounts for a distributed architecture with mobile nodes. An obvious option for such a setting is to embed such an application into existent cellular GSM-networks, where many mobile devices are included, which capture large amounts of contextual information. For this purpose, it is of decisive importance to employ a well-scaling implementation with good performance. A rising number of both dimensions and entities causes exponential computing costs. Such a largely scaled application would provide additional experiences in the domain of contextual proximity management, as we have focused on rather fundamental aspects in our documentation of the contextual map.

Glossary

crc-Operator	Function that maps a complete inter-range *Context* in the *Contextual Map* to a particular contextual-range-specific *Range Context* when given the particular contextual range to map to.	77
k-Nearest-Neighbor Query	A query yielding the *k* nearest neighbor of a point in a multi-dimensional Cartesian space.	112
rcc-Operator	Function that maps a contextual-range-specific *Range Context* to its complete inter-range *Context* in the *Contextual Map*.	76
Bounding Hypercube	A spatial form for determining a context's *Range Vicinity* resembling a hypercube.	113
Bounding Hypersphere	A spatial form for determining a context's *Range Vicinity* resembling a hypersphere.	109
Context	In general, context is any information used to characterize the situation of an *Entity*. Basically speaking, context is the information that an entity's surrounding situation is comprised of.	19
Context	In the *Contextual Map*, we use the term *context* to depict an entity's context representation in the *n*-dimensional Cartesian map model as an *n*-dimensional point in the map. Thus, a context is made up of the values on the *n* dimensions of the contextual map.	75
Context Cluster	Sets of *Contexts* sharing *Contextual Similarity*. A context cluster is parameterized by a set of *Contextual Ranges*, which should affect the context cluster, and a maximum Euclidean distance that defines contextual similarity on each contextual range.	138, 150

Context Mapping	The process of mapping unrefined contextual information into the *Contextual Map*. Data captured from context sources is mapped to the respective dimensions in the contextual map according to the rules of a *Mapping Function*.	78
Context Model	A data structure that implements particular mechanisms to represent *Contexts* of *Entities*. It thus provides persistent storage of those contexts and allows applications to utilize them.	28
Context Type	A notion denoting a particular part of the *Context* of an *Entity*, thus denoting contextual information of a particular type (e.g. contextual information depicting weather conditions). It is used to structure information covered by a context. In the *Contextual Map*, a set of *Dimensions* may describe contextual attributes of one context type and resemble a *Contextual Range*.	21, 76
Context Updates	The process of an *Entity* committing its most current context data elsewhere for further processing, i.e. to a context-aware system, such as the *Contextual Map*.	103
Context Vicinity	The set of *Contexts* in the spatial neighborhood of a particular context C on inter-range level. It is the union of all complete contexts determined from all of C's *Range Vicinities* via the *rcc-Operator*.	105, 108
Contextual Affinity	see *Contextual Similarity*	92
Contextual Boundary	Defines the degree of *Contextual similarity* among different entity contexts by defining multiple thresholds of which each defines a particular Euclidean distance that denotes similarity on a particular *Contextual Range*.	92
Contextual Map	A *Context Model* specialized on detecting similarity among *Contexts* of *Entities* by mapping contextual information into Euclidean space and using Euclidean distance as a similarity metric.	11

Contextual Proximity	see *Contextual Similarity*. The term *proximity* is used to emphasize that contexts, which are *proximate* to each other in the *Contextual Map* in regard to the Euclidean distance between each other, are actually similar to each other.	11, 92
Contextual Range	A set of *d Dimensions* in the *Contextual Map*, thus representing a subspace of the contextual map. Those grouped dimensions usually correspond to contextual attributes of a dedicated *Context Type*.	76
Contextual Range Grid	A partitioned *Contextual Range*, which is completely split up into disjoint, equally sized subspaces. Serves efficient querying of *Range Contexts* inside the affected contextual range.	89
Contextual Realm	Spatial region in the *Contextual Map*. A contextual realm is either defined arbitrary or by the space occupied by a *Context Cluster*. Thus, a contextual realm is the union of subspaces that are individually defined on multiple *Contextual Ranges*.	158
Contextual Similarity	Expresses the fact that entity contexts are similar to each other. See *Contextual Proximity*.	10, 92
Contextual Similarity Query	Given a query context C, the similarity query delivers a result set containing all *Contexts* that are similar to C in terms of *Contextual Similarity*. The contextual similarity query can be defined using a *Contextual Boundary* or arbitrary parameters.	135
Dimension	Each dimension represents one particular contextual attribute in the real-world *Context*. All dimensions together form span the *Contextual Map* enabling it to encompass all relevant context attributes and thus, all possible context.	11, 75
Entity	Represents the context carrier, which can be a person, place or physical or computational object.	19
Global Map	An instance of the *Contextual Map* encompassing all known entity contexts.	167

Individual Cluster Updating	A dynamic clustering procedure that determines changes in the current *Context Cluster* configuration after each *Context Update*. It is contrasted by the *Iterative Clustering* procedure.	152
Iterative Clustering	A dynamic clustering method that is based on the iterative adaptation of the cluster's supporting data structures, i.e. minimum spanning trees and hierarchies. It is contrasted by the *Individual Cluster Updating* procedure.	151
Local Map	An instance of the *Contextual Map* stored locally at an *Entity* encompassing only its own *Context*.	167
Mapping Function	Defines the rules how unrefined context data from context sources is mapped to the *Contextual Map*'s dimensions.	75, 78
Nearest-Neighbor Query	see *k-Nearest-Neighbor Query.*	112
Range	see *Contextual Range.* Not to be confused with numeric and Euclidean spatial ranges.	76
Range Context	d-dimensional representation of an n-dimensional *Context* in the *Contextual Map* on a particular *Contextual Range* (with $d < n$). Thus, a sub-context of the complete context on a particular range.	76
Range Query	A query yielding the points in particular range of a multi-dimensional Cartesian space. Thus, the range defines a spatial part that the query applies to.	110, 129, 137, 201
Range Vicinity	The set of *Range Contexts* in the spatial neighborhood of a particular range context C on a particular *Contextual Range.*	108
Situation	The currently state of an *Entity* at a particular point in time. The information that is associated with the entity at that time resembles its *Context.*	9, 19
Super Situation	A set of multiple *Situations* (or *Contexts*) that an *Entity* can be in. See *Contextual Realm.*	10, 158

Update Semantics	Define the mechanisms when and how *Context Updates* are committed by an entity. The process of an update makes an *Entity* send its current contextual data elsewhere for further processing.	98
Update Vector	An n-dimensional vector constructed locally at an *Entity* and used as carrier for *Context Updates* sent by entities. Each of the vector's entries corresponds to one of the *Contextual Map*'s n dimensions and includes the entity's most current context data for that dimension.	104, 176
Validity Vector	An n-dimensional boolean vector stored locally at an *Entity* and used to construct *Update Vectors* for *Context Updates.* Each entry defines if a current contextual attribute has been committed by an update to the n-dimensional *Contextual Map.* Each entry corresponds to one of the contextual map's n dimensions.	103, 175
Vicinity Map	An instance of the *Contextual Map* stored locally at an *Entity* encompassing *Context* of entities in its logical and/or physical neighborhoods.	167

Bibliography

[1] Source code - hybrid-tree. http://www-db.ics.uci.edu/pages/software/htree.shtml, March 1999.

[2] Source code - sr-tree and r*-tree. http://research.nii.ac.jp/˜katayama/homepage/research/srtree/English.html, April 2003.

[3] Apache log4j. http://logging.apache.org/log4j/, May 2010.

[4] Apple iphone os dev center. http://developer.apple.com/iphone/, May 2010.

[5] Composite capabilities / preferences profile. http://www.w3.org/Mobile/CCPP/, May 2010.

[6] Google android. http://developer.android.com/, May 2010.

[7] The java me platform. http://java.sun.com/javame/, May 2010.

[8] Microsoft windows phone. http://www.microsoft.com/windowsmobile/, May 2010.

[9] Object role modeling. http://www.orm.net, May 2010.

[10] Palm webos - mojo application framework. http://developer.palm.com/, May 2010.

[11] Series 60 platform for symbianos. http://www.forum.nokia.com/s60, May 2010.

[12] Standard generalized markup language (iso 8879:1986). http://www.w3.org/MarkUp/SGML/, May 2010.

[13] Symbian developer network. http://developer.symbian.com/, May 2010.

[14] Unified modeling language. http://www.uml.org, May 2010.

[15] User agent profile. http://www.openmobilealliance.org/Technical/release_program/uap_v2_0.aspx, May 2010.

[16] Xsl transformations (xslt). http://www.w3.org/TR/xslt, May 2010.

[17] Yahoo! weather rss feed. http://developer.yahoo.com/weather/, May 2010.

[18] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[19] Giuseppe Amato, Giuseppe Cattaneo, and Giuseppe F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 314–323, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[20] Arnon Amir, Alon Efrat, Jussi Myllymaki, Lingeshwaran Palaniappan, and Kevin Wampler. Buddy tracking - efficient proximity detection among mobile friends. *Pervasive Mob. Comput.*, 3(5):489–511, 2007.

[21] C.B. Anagnostopoulos, Y. Ntarladimas, and S. Hadjiefthymiades. Situational computing: An innovative architecture with imprecise reasoning. *Journal of Systems and Software*, 80(12):1993–2014, December 2007.

[22] Christos B. Anagnostopoulos, Athanasios Tsounis, and Stathes Hadjiefthymiades. Context awareness in mobile computing environments. *Wirel. Pers. Commun.*, 42(3):445–464, 2007.

[23] V. Baousis, E. Zavitsanos, V. Spiliopoulos, S. Hadjiefthymiades, L. Merakos, and G. Veronis. Wireless web services using mobile agents and ontologies. In *2006 ACS/IEEE International Conference on Pervasive Services*, pages 69–77, 2006.

[24] Christian Bartelt, Thomas Fischer, Dirk Niebuhr, Andreas Rausch, Franz Seidl, and Marcus Trapp. Dynamic integration of heterogeneous mobile devices. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[25] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[26] Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli. A mobile computing middleware for location- and context-aware internet data services. *ACM Trans. Inter. Tech.*, 6(4):356–380, 2006.

[27] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 142–153, New York, NY, USA, 1998. ACM.

[28] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[29] Gregory Biegel and Vinny Cahill. A framework for developing mobile, context-aware applications. *Pervasive Computing and Communications, IEEE International Conference on*, 0:361, 2004.

[30] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[31] Luciano Bononi, Michele Bracuto, Gabriele D'Angelo, and Lorenzo Donatiello. Proximity detection in distributed simulation of wireless mobile systems. In *MSWiM '06: Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*, pages 44–51, New York, NY, USA, 2006. ACM.

[32] Otakar Boruvka. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Spolecnosti*, 3:1, 1926 (in Czech).

[33] Ying Cai, K.A. Hua, and Guohong Cao. Processing range-monitoring queries on heterogeneous mobile objects. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, 1:27–38, 2004.

[34] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering*, pages 440 –447, mar 1999.

[35] Ellick Chan, Jim Bresler, Jalal Al-Muhtadi, and Roy Campbell. Gaia microserver: An extendable mobile middleware platform. *Pervasive Computing and Communications, IEEE International Conference on*, 0:309–313, 2005.

[36] Eleni Christopoulou, Christos Goumopoulos, and Achilles Kameas. An ontology-based context management and reasoning process for ubicomp applications. In *sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence* [37], pages 265–270.

[37] Eleni Christopoulou and Achilles Kameas. Gas ontology: An ontology for collaboration among ubiquitous computing devices. In *International Journal of Human-Computer Studies* [36], pages 664–685.

[38] Marcello Cinque, Domenico Cotroneo, and Stefano Russo. Achieving all the time, everywhere access in next-generation mobile networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(2):29–39, 2005.

[39] Ricardo Couto A. da Rocha and Markus Endler. Evolutionary and efficient context management in heterogeneous environments. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[40] Nguyen Ngoc Diep, Sungyoung Lee, Young-Koo Lee, and HeeJo Lee. Contextual risk-based access control. In *SAM '07: Proceedings of the 2007 International Conference on Security and Management*, 2007.

[41] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[42] Hans W. Gellersen, Albercht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mob. Netw. Appl.*, 7(5):341–351, 2002.

[43] Oleksandr Grygorash, Yan Zhou, and Zach Jorgensen. Minimum spanning tree based clustering algorithms. In *Tools with Artificial Intelligence, 2006. ICTAI '06. 18th IEEE International Conference on*, pages 73–81, Nov. 2006.

[44] Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM.

[45] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[46] Erkki Harjula, Mika Ylianttila, Jussi Ala-Kurikka, Jukka Riekki, and Jaakko Sauvola. Plug-and-play application platform: towards mobile peer-to-peer. In *MUM '04: Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, pages 63–69, New York, NY, USA, 2004. ACM Press.

[47] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[48] Andreas Henrich. The lsdh-tree: An access structure for feature vectors. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 362–369, Washington, DC, USA, 1998. IEEE Computer Society.

[49] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 167–180, London, UK, 2002. Springer-Verlag.

[50] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[51] Mohammad Rezwanul Huq, Nguyen Thi Thanh Tuyen, Young-Koo Lee, Byeong-Soo Jeong, and Sungyoung Lee. Modeling an ontology for managing contexts in smart

meeting space. In *SWWS '07: Proceedings of the 2007 International Conference on Semantic Web and Web Services*, 2007.

[52] Carsten Jacob, David Linner, Ilja Radusch, and Stephan Steglich. Loosely coupled and context-aware service provision incorporating the quality of rules. In *ICOMP 07: Proceedings of the 2007 International Conference on Internet Computing*. CSREA Press, 2007.

[53] Finn V. Jensen. *Bayesian Networks and Decision Graphs.* Springer, Berlin (ISBN 978-0387952598), 2002.

[54] Hassan A. Karimi and Xiong Liu. A predictive location model for location-based services. In *GIS '03: Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 126–133, New York, NY, USA, 2003. ACM Press.

[55] Norio Katayama and Shin'ichi Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 369–380, New York, NY, USA, 1997. ACM.

[56] Mehdi Khouja, Carlos Juiz, Isaac Lera, Ramon Puigjaner, and Farouk Kamoun. An ontology-based model for a context-aware service oriented architecture. In *SERP 07: Proceedings of the 2007 International Conference on Software Engineering Research and Practice*, 2007.

[57] Wolfgang Kieß, Holger Füßler, Jörg Widmer, and Martin Mauve. Hierarchical location service for mobile ad-hoc networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 8(4):47–58, 2004.

[58] Michal Kratky, Vaclav Snasel, Jaroslav Pokorny, and Pavel Zezula. Efficient processing of narrow range queries in multi-dimensional data structures. In *IDEAS '06: Proceedings of the 10th International Database Engineering and Applications Symposium*, pages 69–79, Washington, DC, USA, 2006. IEEE Computer Society.

[59] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1–58, 2009.

[60] Jr. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[61] Axel Küpper and Georg Treu. Efficient proximity and separation detection among mobile targets for supporting location-based community services. *SIGMOBILE Mob. Comput. Commun. Rev.*, 10(3):1–12, 2006.

[62] Dong-Ho Lee, Shin Heu, and Hyoung-Joo Kim. An efficient algorithm for hyperspherical range query processing in high-dimensional data space. *Information Processing Letters*, 83(2):115–123, July 2002.

[63] Ken C. K. Lee, Wang-Chien Lee, and Hong Va Leong. Nearest surrounder queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 85, Washington, DC, USA, 2006. IEEE Computer Society.

[64] Ken C.K. Lee, Josh Schiffman, Baihua Zheng, Wang-Chien Lee, and Hong Va Leong. Round-eye: A system for tracking nearest surrounders in moving object environments. *Journal of Systems and Software*, 80(12):2063–2076, December 2007.

[65] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, Mar 1982.

[66] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-trees: Theory and Applications*. Springer Verlag, 2006.

[67] Teddy Mantoro and Chris Johnson. Location history in a low-cost context awareness environment. In *ACSW Frontiers '03: Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 153–158, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[68] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, New York, NY, USA, 2000. ACM.

[69] Martin Modahl, Ilya Bagrak, Matthew Wolenetz, Phillip Hutto, and Umakishore Ramachandran. Mediabroker: An architecture for pervasive computing. *Pervasive Computing and Communications, IEEE International Conference on*, 0:253, 2004.

[70] Mohamed F. Mokbel, Xiaopeing Xiong, and Walid G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 623–634, New York, NY, USA, 2004. ACM.

[71] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 634–645, New York, NY, USA, 2005. ACM.

[72] Jonathan P. Munson and Vineet K. Gupta. Location-based notification as a general-purpose service. In *WMC '02: Proceedings of the 2nd international workshop on Mobile commerce*, pages 40–44, New York, NY, USA, 2002. ACM Press.

[73] Jin Nakazawa, H. Tokuda, W.K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 3–3, 2006.

[74] Nesetril, Milkova, and Nesetrilova. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233:3–36, 2001.

[75] A. Padovitz, S.W. Loke, and A. Zaslavsky. Towards a theory of context spaces. *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, 1:38–42, March 2004.

[76] A. Padovitz, A. Zaslavsky, Seng Wai Loke, and B. Burg. Maintaining continuous dependability in sensor-based context-aware pervasive computing systems. *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, 1:290a–290a, Jan. 2005.

[77] Amir Padovitz, Seng Wai Loke, Arkady Zaslavsky, and Bernard Burg. Stability in context-aware pervasive systems: A state-space modeling approch. In *Proceedings of the 1st International Workshop on Ubiquitous Computing*, 2004.

[78] Amir Padovitz, Seng Wai Loke, Arkady Zaslavsky, and Bernard Burg. Towards a general approach for reasoning about context, situations and uncertainty in ubiquitous sensing: Putting geometrical intuitions to work. In *2nd International Symposium on Ubiquitous Computing Systems (UCS2004)*, 2004.

[79] Luca Passani and Andrea Trasatti. Wireless universal ressource file. http://wurfl.sourceforge.net.

[80] S. Prabhakar, Yuni Xia, D.V. Kalashnikov, W.G. Aref, and S.E. Hambrusch. Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. *Computers, IEEE Transactions on*, 51(10):1124–1140, Oct 2002.

[81] Paul Prekop and Mark Burnett. Activities, context and ubiquitous computing. *Computer Communications*, 26(11):1168 – 1176, 2003. Ubiquitous Computing.

[82] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[83] Iulian Radu and Son T. Vuong. Nemos: Mobile-agent based service architecture for lightweight devices. In *SWWS '07: Proceedings of the 2007 International Conference on Semantic Web and Web Services*, 2007.

[84] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, New York, NY, USA, 1981. ACM.

[85] Gruia-Catalin Roman, Christine Julien, and Qingfend Huang. Network abstractions for context-aware mobile computing. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 363–373, New York, NY, USA, 2002. ACM Press.

[86] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 71–79, New York, NY, USA, 1995. ACM.

[87] Ichiro Satoh. A location model for pervasive computing environments. *percom*, 00:215–224, 2005.

[88] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 85–90, Dec 1994.

[89] Albrecht Schmidt, Michael Beigl, and Hans-W. Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893 – 901, 1999.

[90] Robert Schmohl and Uwe Baumgarten. Mobile services based on client-server or p2p architectures facing issues of context-awareness and heterogeneous environments. In *PDPTA '07: Proceedings of the 2007 international conference on parallel and distributed processing techniques and applications*, pages 578–584. CSREA Press, 2007.

[91] Robert Schmohl, Uwe Baumgarten, and Lars Koethner. Content adaptation for heterogeneous mobile devices using web-based mobile services. In *Proceedings of the 5th International Conference in Computing and Multimedia (MoMM2007)*, pages 77–86. Oesterreichische Computergesellschaft, 2007.

[92] Wieland Schwinger. Exploring model engineering techniques for the integration of heterogeneous context models. In *Proceedings of the 5th International Conference in Computing and Multimedia (MoMM2007)*, pages 65–76, 2007.

[93] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[94] Zhexuan Song and Nick Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, London, UK, 2001. Springer-Verlag.

[95] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Proceedings of the Workshop on Advanced Context Modelling, Reasoning and Management associated with the 6th International Conference on Ubiquitous Computing (UbiComp), Nottingham.*, 2004.

[96] Olga Volgin, Wanda Hung, Chris Vakili, Jason Flinn, and Kang G. Shin. Context-aware metadata creation in a heterogeneous mobile environment. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 75–80, New York, NY, USA, 2005. ACM.

[97] Roy Want and Trevor Pering. System challenges for ubiquitous & pervasive computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 9–14, 2005.

[98] David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.

[99] Shiow-Yang Wu and Kun-Ta Wu. Effective location based services with dynamic data management in mobile environments. *Wirel. Netw.*, 12(3):369–381, 2006.

[100] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.

[101] C. Yu. High-dimensional indexing. *Springer LNCS 2341*, 1:9–25, 2002.

[102] C.T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *Computers, IEEE Transactions on*, C-20(1):68–86, Jan. 1971.

[103] Hai Zhou, Narendra Shenoy, and William Nicholls. Efficient minimum spanning tree construction without delaunay triangulation. In *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 192–197, New York, NY, USA, 2001. ACM.