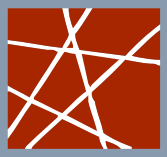


Network Architectures
and Services
NET 2012-07-2

Dissertation

Maintaining Reference Graphs in Fully Decentralized Systems

Björn Saballus

Network Architectures and Services
Department of Computer Science
Technische Universität München

TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Netzwerkarchitekturen

Maintaining Reference Graphs in Fully Decentralized Systems

Björn Saballus

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. Dr. H. M. Gerndt
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. G. Carle
	2.	Univ.-Prof. Dr. A. Polze, Universität Potsdam
	3.	TUM Junior Fellow Dr. Th. Fuhrmann

Die Dissertation wurde am 20.12.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03.06.2012 angenommen.

Cataloging-in-Publication Data
Björn Saballus
Maintaining Reference Graphs in Fully Decentralized Systems
Dissertation, July 2012
Network Architectures and Services, Department of Computer Science
Technische Universität München

ISBN: 3-937201-31-9
ISSN: 1868-2634 (print)
ISSN: 1868-2642 (electronic)
DOI: 10.2313/NET-2012-07-2
Network Architectures und Services NET 2012-07-2
Series Editor: Georg Carle, Technische Universität München, Germany

Abstract

The recent shift in processor development leads to massive parallelism in multi- and many-core processors. So, together with the wide-spreading use of cluster technologies, this development leads to huge networks of cores and processors. The development of applications that exploit this increasing parallelism is a hard task, for which only a small fraction of programmers is prepared.

One approach to support the implementation of parallel applications for distributed systems, besides others described in this thesis as well, is to offer a single system image (SSI). An SSI completely hides the underlying hardware complexity and distribution of data and threads. Thus, it can be programmed as if it is a symmetric multi-processor (SMP) system, while the SSI afterwards handles load balancing and the placement of data and threads dynamically at runtime. Hence, it allows the maintainer to seamlessly add and remove processing elements at runtime, without disturbing the execution of the running application.

This approach seems to have faded from prominence in the recent years, as the low publication rate indicates. In this thesis, I argue that the SSI approach should be examined again, in the new context of the increased on-die parallelism as well as the spreading use of cluster technologies.

As a starting point, this thesis describes one of the main requirements of a distributed system, and thus an SSI: the possibility to migrate objects between nodes, e. g. to place data and threads closer to each other, or use added resources and evade nodes, which are scheduled for shutdown. Together with such an object migration, the system must offer referencing entities a mechanism to locate and retrieve migrated objects. To allow the SSI to scale with very large systems, it requires a fully-decentralized mechanism to avoid bottlenecks and single point of failures.

The development, analysis, and evaluation of such a decentralized location and retrieval mechanism are the main topics of this thesis. It describes various possible approaches, of which the two main mechanisms are the *reactive* and the *proactive* location update approach. The reactive location update approach updates outdated references upon access. To be able to do this, the approach uses proxy objects, which remain on the object's previous location and forward all subsequent access messages to the object's current location. The proactive location update approach updates all referencing locations immediately upon object migration.

This thesis describes the design and analysis of these protocols and their prototypical implementation. Moreover, it evaluates different simulation runs with implicit and explicit object migrations. The chief insight of this evaluation is that the proactive location update approach decreases the object access latencies significantly, if a huge number of object migrations take place. Nevertheless, this fast object access comes with a significant management message overhead. The conclusion is that this overhead is only worthwhile as an optional add-on, for example, for objects where a fast access is crucial for the application performance.

Beyond that, this thesis implements and evaluates an access path optimization mechanism for the reactive location update approach. This access path optimization propagates location update

information down the proxy chain to decrease the access latency. Yet, how deep this information is propagated depends on the access characteristics of the object. Therefore, this thesis presents an analytic formula that allows the computation of the optimal update propagation depth, which minimizes the total message costs for the proxy chain update and the read access.

Besides the presented results, I describe the two BmBF projects in which I worked while conducting this research. One is the AmbiComp project, which developed embedded hardware and software for ambient intelligence systems. The other one is the J-Cell project, which has the goal to map the developed mechanisms from the AmbiComp project onto the field of high performance computing. The work in these projects continuously inspired the work described in this thesis.

Zusammenfassung

Seit einigen Jahren vollzieht sich ein fundamentaler Wandel beim Prozessorbau, der zu einer wachsenden Parallelität in Form von Multi- und Many-Core Prozessoren führt. Zusammen mit der zunehmenden Verbreitung von Computer-Clustern führt diese Entwicklung zu großen Netzwerken aus Prozessoren und Prozessorkernen. Die Programmierung von Anwendungen für diese massiv parallelen Recheneinheiten ist eine schwierige Herausforderung, auf die nur wenige Programmierer vorbereitet sind.

Ein Ansatz, der die Anwendungsentwicklung von parallelen Anwendungen für verteilte Systeme unterstützt, ist, neben anderen, die Bereitstellung eines *Single System Image* (SSI), einer einzelnen, einheitlichen Systemabbildung. Ein SSI verbirgt die gesamte Komplexität der verwendeten Hardware, die Verteilung der Daten und Threads auf einzelne Rechenknoten sowie deren Kommunikation untereinander. Dieses einheitliche Systemabbild kann wie ein einzelnes, symmetrisches Mehrprozessorsystem programmiert werden, während sich das SSI anschließend dynamisch zur Laufzeit um die Lastverteilung und die Platzierung von Daten und Threads kümmert. Hierdurch kann der Systemadministrator Komponenten entfernen oder hinzufügen ohne die laufenden Anwendungen zu stören.

Die nachlassende Zahl an Neuveröffentlichungen zeigt, dass das Interesse an diesem Ansatz offenbar in den letzten Jahren nachgelassen hat. Ich hingegen argumentiere in der vorliegenden Dissertation, dass der SSI Ansatz erneut im Kontext der zunehmenden Parallelität und der Verbreitung von Computer-Clustern betrachtet werden sollte.

Der Ausgangspunkt meiner Arbeit ist die Betrachtung einer der wichtigsten Anforderungen an ein SSI: die Möglichkeit, Objekte zwischen Knoten zu migrieren, um beispielsweise Daten und Threads näher beieinander zu platzieren oder neu hinzugefügte Ressourcen zu nutzen, beziehungsweise Knoten, die für den Shutdown vorbereitet werden, zu entfernen. Um diese Objektmigration zu ermöglichen ist es erforderlich, dass das System die Lokalisierung von – und den Zugriff auf migrierte Objekte sicherstellt. Damit das System auch mit sehr großen und wachsenden Systemen skalierbar bleibt, muss für diese Aufgabe ein vollständig dezentraler Lokalisierungs- und Zugriffsmechanismus gefunden werden.

Der Kern dieser Dissertation ist die Entwicklung, Analyse und Bewertung eines solchen vollständig dezentralen Lokalisierungs- und Zugriffsmechanismus. Hierzu stelle ich verschiedene Lösungen vor, von denen die zwei wichtigsten Mechanismen das *reaktive* und das *proaktive* Aktualisieren des Objektaufenthaltsortes sind. Der reaktive Ansatz aktualisiert den Aufenthaltsort beim Zugriff auf das Objekt. Hierzu lässt jede Objektmigration ein *Proxy Objekt* zurück, welches nachfolgende Anfragen zum derzeitigen Aufenthaltsort des Objektes weiterleitet. Der proaktive Ansatz verwendet Aktualisierungsnachrichten, welche die Information über den Aufenthaltsort des Objektes auf anderen Knoten sofort nach der Migration aktualisieren.

In dieser Dissertation beschreibe ich das Design dieser Protokolle und deren prototypische Implementierung. Ich evaluiere die verschiedenen Simulationsdurchläufen mit impliziter und expliziter Objektmigration. Die Haupterkenntnis dieser Auswertung ist, dass der proaktive Ansatz die Zugriffszeit bei einer großen Anzahl von Objektmigrationen signifikant verringert. Dieser schnellere Zugriff wird allerdings durch hohe Zusatzkosten durch die große Anzahl von Aktualisierungsnachrichten erkauft. Das Fazit ist, dass sich diese Zusatzkosten nur als zusätzliche Option lohnen, wenn der schnelle Zugriff auf ein Objekt wichtig für die Leistungssteigerung der Anwendung ist.

Darüber hinaus habe ich ein Zugriffspfad-Optimierungsverfahren für den reaktiven Ansatz entwickelt. Dieses Verfahren propagiert Aktualisierungsnachrichten die Proxykette hinab, um die Zugriffszeit zu verkürzen. Wie tief diese Information entlang der Kette propagiert wird, hängt von der Zugriffscharakteristik des Objektes ab. Hierfür habe ich eine analytische Formel entwickelt, welche es erlaubt die optimale Aktualisierungstiefe zu berechnen und damit die Gesamtkosten für die Aktualisierungsnachrichten und den Objektzugriff zu minimieren.

Neben den präsentierten Ergebnissen beschreibe ich zusätzlich die zwei BMBF-Projekte, an welchen ich während meiner Dissertation gearbeitet habe. Das eine ist das *AmbiComp* Projekt, welches eingebettete Hardware und Software für “umgebungsintelligente” (Ambient Intelligence) Systeme entwickelt hat. Das andere ist das *J-Cell* Projekt, welches die Ergebnisse aus dem AmbiComp Projekt auf das *High Performance Computing* überträgt.

Contents

1 Introduction

In recent years, distributed systems in general, and distributed and parallel computing in particular, became more and more important. General purpose, as well as high performance computing systems grow in parallelism and no longer in single core processing speed. See, for example, Figure 1 and Figure 2 in Fuller and Millett [FM11, p.33]. Among others, these figures show the historical growth of clock speed, the number of cores per chip, and the single-processor performance. Since the year 2004, the single processor performance and the clock speed have stalled, while the number of cores per chip is growing.

For example, mainstream general purpose processors are already equipped with 4 cores, while 8 core processors are available, and processors with up to 16 cores, such as the AMD Interlagos [Adv10] processor, have been announced. Another example is Intel's *Single-Chip Cloud computer* (SCC), an experimental 48-core processor [Int11], which Intel Labs have created as a "concept vehicle" for many-core software research. Furthermore, the currently fastest computer in the Top500 list [TOP11] – in November 2011 – is the *K Computer* at the RIKEN Advanced Institute for Computational Science (AICS) in Kobe, Japan. This computer consists of a network of 88 128 8-core processors, which sum up to a total of 705 024 cores [TOP11].

Today, nearly all computer systems for high-performance computing consist of homogeneous compute nodes. But the current trends in processor design lead to heterogeneous architectures, e. g. the IBM Cell processor [Pha+05].

Moreover, the compute power of embedded systems increases with every generation, which leads to the development of distributed embedded systems, for example for home entertainment, automotive systems, environmental monitoring applications etc.

Furthermore, Leen and Heffernan [LH02] stated that about 80% of all innovations in the automotive sector come from electronics, and Broy [Bro06] has seen an exponential software usage growth in this field. According to Broy, the software in a premium car has more than 10 million lines of code and runs on up to 70 small, specialized embedded devices that are connected by various bus systems. Together, these control units form a heterogeneous network, which executes one or more distributed applications, e. g. the active suspension control system, motor control, multimedia systems or the passenger compartment temperature and humidity control.

The main challenge in these systems is their programmability. This challenge is accompanied by the challenges of scalability and fault-tolerance, because these two directly influence the programming of such systems. In this context, *scalability* means *vertical scalability*, where the number of compute cores is increased, in contrast to *horizontal scalability*, where the system is replaced with a more powerful one.

scalability

Ideally, the application performance should increase linearly with the available compute power. In practice, this goal is hard to achieve. Still, an increase of compute power should at least not decrease the application performance, as it was observed by Singh, Hennessy, and Gupta

[SHG93]. The authors stated that for a fixed size problem, at some point the addition of more processors even increases the execution time. They presented a figure that shows this effect for a Barnes-Hut application, where the performance increases for up to 32 processors, but starts to decrease, if more than 32 processors are used.

The observed reasons are the limited parallelism of the fixed problem size and the increased management overhead that is needed for the parallel execution. This overhead is influenced by the growing number of potentially heterogeneous compute cores because they increase the management overhead for the hardware and have influence on the system design. During the system design, the developer has to decide which interconnect should be used and how the nodes should be placed in the network in relation to each other. This decision influences the access latency between any two cores in the network significantly. But this decision is influenced by the application too, for example, if each thread, which is executed, requires a fast access to some global memory, or in case that the application frequently ships data to a cluster of floating point units or GPGPUs (*general purpose graphics processing unit*).

Moreover, the growing number of compute cores comes with the risk of more frequent node failures. Schroeder, Pinheiro, and Weber [SPW09], for example, analyzed DRAM failures in Google's data center infrastructure. They observed 25 to 70 errors per million device hours per Mbit and detected that more than than 8% of all DIMMs (Dual Inline Memory Module) are affected every year. Schroeder and Gibson [SG10] examined systems for high performance computing at the Los Alamos National Laboratory. The authors noticed that the failure rates in a system are nearly proportional to the number of processors. Furthermore, they found a correlation between the type and intensity of the workload and the failure rate of a machine.

Altogether, frequent node failures, hardware upgrades and maintenance tasks as well as the consolidation of an application onto fewer nodes to save energy, lead to a highly dynamic network of processors and cores. Because of this frequent churn, centralized software components, such as central servers, introduce a single point of failure and do not scale with an increasing number of nodes. Thus, I advocate for the design of such networks of compute nodes as fully decentralized distributed systems.

To successfully develop parallel applications for such systems, the application programmer has to be aware of the dynamic nature of the underlying hardware architecture. This starts, for example, with the question: Which programming model must be applied to develop the parallel application for the given hardware architecture? Two common parallel programming models, for example, are message passing for distributed systems, and shared memory programming for shared memory systems, cf. Chapter 2. Moreover, the developer has to consider the demands of the parallel application, e. g. to handle the memory access latencies or the placement of data and threads right.

An alternative to handling the underlying architecture in the application is the use of an intermediate layer that provides a *single system image* (*SSI*) [Pfi98; BCJ01]. Such an SSI offers transparent access to the underlying resources, while hiding the underlying dynamic features of the system, such as node churn or frequent object and thread migrations. Thus, it allows applications to run transparently on large clusters of heterogeneous multi-core and many-core machines as well as distributed, embedded systems, cf. Section 2.2.6. Both these application domains are

large-scale distributed systems that consist of a dynamic network of potentially heterogeneous compute nodes, where nodes may join or leave at any time and the use of centralized components is not feasible.

It is the goal of the group I am part of to develop a runtime environment that provides an SSI on top of such networks of potentially heterogeneous compute nodes. This runtime system is either a distributed Java virtual machine (DJVM) that executes the code of the application, or it could be an application that uses the C library that we are developing in our group. There is no difference in the functionality between DJVM and C library with respect to the distributed execution of an application. Thus, I will assume that the envisioned system uses the DJVM for the rest of this thesis.

The target software consists of irregular, multi-threaded (see Chapter 2) applications. Such irregular application are characterized by the use of pointer-based data structures such as graphs, trees or unstructured grids, the use of irregular control structures such as conditional statements, or the exhibition of irregular and non-deterministic communication patterns, cf. [Kul+09]. Our runtime environment transparently distributes code, objects and threads of such irregular applications onto the compute resources, which may be added or removed at run-time. The threads of the application might share objects and hold references to objects that might be located on remote nodes.

Due to the dynamic features of these distributed systems, object migration is an important functionality. It allows and facilitates:

- **Maintenance**: Migrate the running threads and locally stored objects from node *A* to node *B* to shutdown node *A* for upgrade, maintenance or exchange during runtime.

- **Latency Optimization**: Migrate objects that access each other to nodes close to each other or onto the same node to decrease the access latency.

- **Replication**: Migrate object replicas to nodes that are spread throughout the network to increase the reliability in case of node failures. This also improves the access latency of accessing nodes in the close neighborhood of a replica.

- **Resources**: Migrate all threads and objects to nodes with free resources to improve resource utilization or to nodes with specialized resources such as floating-point units.

- **Energy Efficiency**: Migrate all threads and objects from node *A* to node *B* to shutdown node *A* to save energy e. g. during night time, when not the full compute power is needed.

In such a network, a scalable and fully decentralized object location and retrieval algorithm is needed. This algorithm has to ensure that any node in the system can always access all local and remote objects to which it holds a reference. Thus, the algorithm must be able to access remote objects regardless of their location.

The thesis' topic is the development and evaluation of such a decentralized object location and retrieval algorithm.

Research Methodology

The main research question of this thesis is

> *"How can I ensure that an object is reachable in a fully decentralized system that allows object migration?"*

To answer this question, I have chosen the *design research* approach to conduct my research. In this context, the term *design* means a process in which something new is created. In the design research, the researcher *creates* an artifact to gain new knowledge, cf. [Hev+04; BZ07]. Therefore, the aim is not to develop the artifact, but to analyze and understand the artifact's behavior. The goal is to gain a deeper knowledge about the research area, which might lead to the artifact's refinement.

Starting with the given research question, my first task was the analysis of the problem space, which led to a first specification of a distributed reference maintenance and location update protocol.

Afterwards, I conducted the design research in various subsequent, evolutionary research cycles. Based on the specification, I developed, tested and evaluated the protocol implementations in a network emulator as well as in a software simulation. The work on the prototype and the analysis of the results revealed new insights into the topic and allowed enhancements of the initial specification. This modified specification was the starting point of the next iteration. First, I tested the enhanced protocol implementation. Afterwards, I evaluated its results and compared them against the initial specification, which again led to new knowledge to start a new research cycle.

Overview

The remainder of this thesis is structured as follows:

Chapter 2 gives an introduction into the field of parallel programming in general. Additionally, it outlines the underlying concepts that are necessary for this thesis, e. g. the object and memory model.

Chapter 3 gives an overview over the related work of this thesis. Moreover, it summarizes different systems for distributed computing, such as distributed shared memory, distributed operating systems, and distributed Java virtual machine approaches.

Chapter 4 describes the two target environments from which the main target scenarios are derived. Namely, the ambient intelligence scenario from the AmbiComp project and the high-performance computing scenario from the J-Cell project.

Chapter 5 gives an overview over the runtime components, which are necessary for the decentralized object location and retrieval mechanism. It does not only describe the different modules of the runtime environment, but also their interactions in more detail.

Chapter 6 first gives an overview how a decentralized system can locate static objects. Afterwards, this chapter introduces and compares various approaches to locate dynamic objects.

Chapter 7 analyzes the reactive location update protocol that uses forwarding proxies to redirect messages after an object migrated to another node. Additionally, this chapter describes the access path optimization approach, which shortens the access latency in case of long chains of proxies, and depends on the access characteristics of the accessed object.

Chapter 8 analyzes the proactive location update protocol that uses proxies together with backward references. This protocol uses these backward references to immediately update the location information for the migrated object at all nodes that hold a references to the migrated object.

Chapter 9 evaluates the reactive and proactive location update protocols in an OMNeT++ network emulator. Furthermore, it evaluates the reactive location update protocol – together with the access path optimization approach – in an additional software simulation.

Chapter 10 summarizes the work of this thesis and draws a conclusion. Additionally, this chapter gives an outlook for open research questions.

Published Work

Parts of this thesis have been published:

Chapter 2, Chapter 4, Chapter 7

Björn Saballus, Stephan-A. Posselt, Thomas Fuhrmann: *A Scalable and Robust Runtime Environment for SCC Clusters*, Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, 2011.

Chapter 4

Björn Saballus, Johannes Eickhold, Thomas Fuhrmann: *Global Accessible Objects (GAOs) in the AmbiComp Distributed Java Virtual Machine*, Proceedings of the 2nd Int'l Conference on Sensor Technologies and Applications (SENSORCOMM'08), Cap Esterel, France, 2008

Chapter 4

Johannes Eickhold, Thomas Fuhrmann, Björn Saballus, Sven Schlender, Thomas Suchy: *AmbiComp: A Platform for Distributed Execution of Java Programs on Embedded Systems by Offering a Single System Image*, Proceedings of the AmI-Blocks Workshop at the European Conference on Ambient Intelligence (AmI-Blocks'08), Nuremberg, Germany, 2008

Chapter 3, Chapter 5, Chapter 7, Chapter 8, Chapter 9

Björn Saballus, Thomas Fuhrmann: *A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments*, Technical Report TUM-I1025, Technische Universität München, Munich, Germany, 2010

Chapter 7

Björn Saballus, Stephan-A. Posselt, Thomas Fuhrmann: *Brief Announcement: Fault-Tolerant Object Location in Large Compute Clusters*, Proceedings of the 13th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'11), Grenoble, France, 2011

Chapter 7, Chapter 9

Björn Saballus, Stephan-A. Posselt, Thomas Fuhrmann: *Caching Strategies and Access Path Optimizations for a Distributed Runtime System in SCC Clusters*, Proceedings of the 4th MARC Symposium, Potsdam, Germany, 2011.

2 Background

This chapter gives an overview over the background of this thesis. It starts with an introduction of the most important terminology. Afterwards, it describes different parallel programming models such as message passing and (distributed) shared memory. Finally, the chapter closes with an outline of the object and memory model.

2.1 Terminology

The following definitions introduce the most important terminology used in this thesis.

Core Following the von Neumann architecture [Neu45], a *core* is the smallest sequential execution unit of a computer, and is traditionally called *central processing unit* (*CPU*).

The CPU consists of an *arithmetic logic unit* (*ALU*) and a *control unit*. The ALU performs all arithmetic operations and comparisons, while the control unit fetches instructions from memory, decodes the instructions, fetches the operands from memory, and executes the operation, e. g. by sending the appropriate commands to the ALU or loading further data from memory. Finally, it stores the potential result, e. g. from an addition, into the destination memory.

Additionally, each core has a small, temporary storage; its registers and often a small cache.

Processor A *processor* contains a single core or multiple cores. Thus, processors are distinguished into single-core or multi- and many-core processors. The latter ones are described in more details in Section 4.3.

Besides the cores, a processor contains additional local cache memory. This cache is often split into multiple levels as a cache hierarchy, where the level 1 cache (L1) is local to a single core, while the on-die level 2 cache (L2) might be local to a single core or, if no level 3 cache is present, can be shared among all cores. If the level 2 cache is shared or a level 3 cache is present, the cores and the shared cache are attached to each other via an on-die memory bus.

Additionally, an external system bus connects the processor to the off-die main memory.

Node A *node* is a single, stand-alone entity that consists of one or more processors. All processors are attached to the main memory of the node. This memory is commonly shared among all processors and allows for *symmetric multiprocessing (SMP)*, whereby the system is said to be an SMP system.

I assume that all processors and hardware components within a node share the same fate, i. e., all processors fail together or not at all.

Various nodes can be interconnected to form a network of nodes, also called *cluster*, *Grid*, *Cloud* or *supercomputer*.

Distributed Shared Memory *Distributed shared memory* (*DSM*) is a memory architecture where all cores can transparently access the entire memory in a network of nodes. An underlying protocol handles the necessary communication and memory management among the nodes and thus, offers a common address space.

A DSM combines the advantages of *shared memory*, where all cores have direct access to the same memory, e. g. the main memory of the processor in a node, and *distributed memory*, where the physical memory is scattered across multiple nodes. In the latter case, a single node has direct access only to its local memory, and some form of communication is necessary to access the memory on another node, e. g. via message passing.

Thread/Task A *task* is a logically discrete sequence of computational work, while a *thread* is the execution of a sequence of program instructions that can be logically executed in a sequential order by a processor. The programmer has the choice to spawn a separate thread for each new task, or one thread executes a number of different tasks, one after the other.

A parallel program consists of multiple threads that can be executed at the same time on multiple cores. *Multithreading* executes multiple threads in a shared memory environment, either time-sliced on a single core, or concurrently on multiple cores, while operating systems usually do both. *Synchronization points* are used to synchronize the execution of threads. A barrier, for example, does not allow a thread in a group of threads to continue its work unless all other threads have reached this barrier as well.

Object An *object* is a chunk of memory of a given size. Our system does not make any assumptions about the internal layout of the data; this is entirely up to the using entity, e. g. our distributed Java virtual machine. Thus, an object might be an object in the sense of an object-oriented programming language, but also a C struct, an array or the execution context of a thread. It might even be the executed code itself.

Home Node An object always resides in the local memory of some node, which is called the *home node* of the object. This home node is either a *static* home node, which does not change during the lifetime of the object, or a *dynamic* home node that might change over time.

With a static home node, it depends on the chosen approach if object migrations are allowed or if the object remains, for example as a *master* object, on the static home node, cf. Chapter 3 and Section 6.2.4. If no object migrations are allowed, the static home node is responsible for the synchronization of the object access. If the system allows migration, the static home node is only responsible for the object location, not for e. g. serializing the write access to the object.

With a dynamic home node, the object is free to migrate among the nodes in the network. Hence, the location information for the object that is stored on other nodes than the home node of the object has to be managed to guarantee the accessibility of the object.

If not stated otherwise, the occurrence of the phrase *home node* means in the rest of this thesis a *dynamic home node*.

Reference A *reference* is an identifier that is used to access a particular object. References to objects are either node local identifiers, e. g. pointers (local memory addresses) into the local memory, or globally unique identifiers, e. g. tuples that address a chunk of memory on a remote node, i. e. $<$ `Node Id, Local Memory Address` $>$, or any other addressing scheme that allows the globally unique identification of a particular object.

Critical Section A *critical section* is a piece of code where multiple threads access the same shared object. Thus, this critical section must be protected from the simultaneous access of multiple threads to prevent memory corruption. For example, in a *race condition*, two or more threads update the same object at the same time, and the result depends on the scheduling of the threads. Namely, the result contains only parts of the written data of each thread, because the other parts have been overwritten by the other threads. Thus, it is necessary to synchronize the access to shared objects, so that only one thread at a time is allowed to enter the critical section in which the shared object is manipulated. Various concepts, such as *locks*, *semaphores* or *monitors*, exist to achieve mutual exclusion.

With locking, only one thread is allowed to hold the lock, while all other threads have to wait until the lock-holder leaves the critical section and releases the lock. In this way, the access to the critical section happens in a strictly sequential order, and can thus lead to an increase of the execution time of the application. Additionally, locking might lead to *priority inversion* or *deadlocks*, for example if two threads try to gain two locks to the same two critical sections at the same time but in reverse order, so that neither of them can get the lock that is held by the other. Furthermore, a node failure in a distributed system might also result in a deadlock if threads on that node possessed some locks which will never be released again and which cannot be broken in a sane way.

Transactional Memory A recent model to avoid the pitfalls of the lock-based sequential access to shared objects is to handle the concurrent access with software or hardware *transactional memory* (TM). With TM, all threads operate on local copies of the shared objects in so-called *transactions*.

A transaction is the smallest entity of execution in a transactional memory system. In our STM system, all operations within a transaction are executed strictly locally, without interactions with the surrounding environment.

At the end of a transaction the thread tries to *commit* (publish) all local changes of the shared objects. If the commit protocol does not detect any conflicts with other threads the commit succeeds and the thread continues its execution. If the commit fails, the thread has to roll back and restart the computation with the latest version of the object.

Note that I do not explicitly define the terminology from related scientific fields.

For example, one of the main topics of this thesis is distributed shared memory programming. For this reason, I did describe tasks and threads, but did not explicitly explain processes, even though I use the term later on. Similarly, this thesis assumes that there is an underlying network

with a given topology, and that nodes are interconnected by communication links having a given speed and bandwidth. Additionally, a routing protocol is present that supports end-to-end communication via multi-hop or point-to-point connections. However, I will not detail the topic of network topologies and routing protocols any further, even though these phrases are used throughout the thesis. Furthermore, I am not concerned with I/O (input/output) interfaces that allow the communication between the distributed system and the outside world, or checkpointing and fault tolerance. These are topics that are ongoing work in the group I am part of.

2.2 Parallel Programming

In the past, main-stream processors contained only a single core which was able to execute exactly one instruction at a time. Thus, traditionally, software is written as a sequential stream of instructions.

As described in the introduction, the current trends in processor design lead from increasing clock rates of single-core processors towards increasing numbers of cores in multi- and many-core processors with lower clock rates, where not each core must necessarily support the full instruction set architecture of current processors. This development requires a paradigm shift in software engineering from sequential to parallel and concurrent programming, where the different cores of the processor independently execute the instructions of multiple parallel threads.

This trend challenges software developers who have to work with legacy code that was originally targeted at single-core processors. Furthermore, most developers seem to be well accustomed to sequential algorithms, but have a hard time programming with concurrency in mind, cf. [SL05; Lee06; Luf09; SS10]. Especially, when the algorithms need to scale to a vast number of cores. According to Sutter and Larus [SL05], one of the main reasons is the human himself, who is not used to think about parallelism and concurrency.

However, the development of parallel applications is not only necessary to cope with the increasing hardware parallelism, but has also the advantage to speed up the execution time of a given computational task. I. e., the parallel execution of a multi-threaded application on multiple cores can execute a larger number of computational tasks than the sequential execution in the same execution time.

Here, the achievable speedup of a parallel application is commonly defined as the ratio of the sequential execution time T_{seq} and the parallel execution time T_{par}:

$$speedup = \frac{T_{seq}}{T_{par}} \tag{2.1}$$

Amdahl's Law

Nevertheless, *Amdahl's Law* [Amd67] states that the maximal speedup that a parallel program can achieve is bound by the sequential parts of the program. To see this, let P be the parallel parts of the program and $(1-P)$ the sequential parts. The parallel parts are executed by a number of available processors N, leading to $\frac{P}{N}$. With $T_{par} = T_{seq} \cdot ((1-P) + \frac{P}{N})$ and $N \to \infty$, the maximal speedup is:

$$speedup = \frac{T_{seq}}{T_{seq} \cdot ((1-P) + \frac{P}{N})} \leq \frac{1}{(1-P)} \tag{2.2}$$

To achieve this speedup, programmers must be well aware of the underlying hardware, the network architecture, and the chosen programming model. Here, the two major programming models for parallel programming are *message passing*, which is the predominant model for the programming of large distributed memory systems, and *shared memory* programming, also called *multithreading*, which is the predominant model for the programming of small shared-memory systems [FGK03].

2.2.1 Message Passing

Message passing environments have been around since the 1980s [OB87; Sun90; Gei+90] and the development of a common standard was started in 1992 [Wal92].

Message passing is commonly used for large systems where the memory is distributed among all nodes and where each node has direct access only to its local memory. Thus, the message passing model has to follow a 'share nothing' paradigm, where the different program entities do not share a common memory, but have to exchange messages to communicate with each other. Thus, the individual program tasks are similar to processes in an operating system.

The *message passing interface* (*MPI*) [Sni+98] is the industry standard for message passing and the dominant model for programming high-performance applications [Bas+08]. Two open source implementations of MPI are e. g. *LAM/MPI* [SL03] and *MPICH* [Arg11].

To communicate among two processes via MPI, the programmer has to specify explicitly the two-sided communication with *send* and *receive* method pairs. The MPI standard offers these two point-to-point methods as either *blocking* or *non-blocking*. In the non-blocking case, a process can call an additional method to get information about the process of the send operation.

Additionally to these two methods, the MPI-1 standard specifies some additional aspects such as MPI specific data types and collective communication, e. g. broadcast, which distributes the work to all nodes in the network, or reduce operations, which collect the results from the nodes that took part in the parallel computation.

The MPI-2 standard adds one-sided communication via *PUT* and *GET* messages, which does not involve the remote process anymore. Here, the process on one node can read data from or write data to the memory of any other node, without interrupting the execution of the processes on that node. With this feature, MPI-2 does not follow the message passing model anymore but is closer to the *remote direct memory access* (*RDMA*) [Rec+07].

Altogether, the message passing programming model is well suited for large-scale systems without a physically shared memory, i. e. a distributed memory architecture. This was shown by Balaji et al. [Bal+09], who examined the communication performance of MPI in the IBM Blue Gene/P system with 32 768 nodes, where each node consists of a 4 core processor (= 131 072 cores).

However, the major drawback of message passing is that the programmer is required to deal with the placement of and the communication among the different tasks, which makes MPI difficult to program. Furthermore, it is especially unsuited for applications with fine-grained irregular access to remote memory because of the software overhead that is required for the communication [Sha+01; Ber+04].

2.2.2 Shared Memory and Multithreading

In the *shared memory* programming model, all threads share a single common memory. Therefore, all threads can communicate via shared data and the programmer is not required to think about explicit communication. Thus, shared memory programming is for most programmers easier than message-passing [PGL94].

multithreading

Because multiple threads communicate via a shared memory, this programming model is also called *multithreading*, which is either done with *native threads* or with *green threads*. The difference is that native threads are managed and scheduled by the operating system in the kernel space, whereas green threads are managed and scheduled by a virtual machine in the user space. A common *API* (*application programming interface*) for native threads is e. g. the *Pthreads* library, specified by the IEEE POSIX 1003.1c standard [IEE04]. The Java virtual machine (JVM), for example, uses green threads that are separated from the operating system. Thus, even though the JVM can map green threads to native threads, it is still possible to execute multi-threaded application in a JVM that runs on an operating system that does not support multithreading, such as small, embedded devices. Furthermore, Java directly supports multithreading via the Java API [Lea99; Goe+06].

A shared memory programming API for C/C++ and Fortran is the industry standard *OpenMP* [Ope11]. With OpenMP, the programmer instruments the source code with OpenMP statements, e. g. to indicate a loop that should be executed in parallel by multiple threads. An OpenMP compiler translates the instrumented parts of the source code into parallel code, while a sequential compiler is not affected by these statements. I. e., the sequential compiler does not have to be extended to support the additional instrumentation. Thus, it is possible to use OpenMP to parallelize existing sequential code that is still executable on a single-core processor.

For the envisioned system, the main drawback of multithreading is that it is limited to machines that have a shared memory architecture. Moreover, the concurrent access to shared data must be synchronized to prevent memory corruptions. Here, the common approach is to use *locks*, which, in practice, proved to be hard to get right [RHW10].

2.2.3 Distributed Shared Memory

DSM

To keep the advantages of shared memory programming in distributed systems without a common shared memory, *distributed shared memory* (*DSM*) has been developed.

A DSM provides the abstraction of shared memory in a distributed memory environment by hiding the distributed nature of the memory from the user. Thereby, the DSM offers the user the transparent access to local and remote memory by hiding the underlying communication between the different nodes.

Depending on the level on that this abstraction is implemented, DSM systems can be distinguished into *hardware* DSMs, which support the shared memory abstraction on the hardware level, and *software* DSMs, which provide this abstraction within the runtime environment.

For this thesis only software DSMs are of interest because the developed system is a software and not a hardware solution. Section 3.4 gives a detailed description of various software DSM systems.

2.2.4 Partitioned Global Address Space

The *PGAS* (*partitioned global address space*) programming model is an important subclass of DSMs for this thesis because it aims at the same application domain as the envisioned system. PGAS received attention recently, especially for multi-core programming [KB09]. Unlike the envisioned system, PGAS' main objective is to allow the programmer to explicitly control and predefine the placement of data and threads. Thus, the developer can place the code and the data that the code manipulates close to each other, e. g. on the same node. Additionally, the programmer can place subsequently executed code blocks on the same node as well. **PGAS**

Similar to our memory model (see below), the local memory of a PGAS node is partitioned into a private local part and a globally shared part. All scattered globally shared parts of all nodes belong to the shared memory. This shared memory might be a physically shared memory or a simulated shared memory that is realized with a distributed shared memory and message passing. Thus, this shared memory offers a global address space with a potentially non-uniform access time.

Typical PGAS languages are *X10* [Sar+11; Cha+05], *Unified Parallel C* (UPC) [The05; CDC99], *Titanium* [Bon+06; Yel+98] and *Chapal* [Cra11; CCZ04].

X10 is an object-oriented programming language proposed by IBM. X10 follows the *multiple instruction multiple data* programming model that allows each node to execute different threads which operate on different data. X10 was derived from the Java programming language and aims at the programming of heterogeneous, non-uniform clusters. According to Charles et al. [Cha+05], it was designed to circumvent Java's drawback in multi-core systems: the tight coupling to a single, uniform heap. Furthermore, similar to other PGAS languages such as UPC, X10 makes the location of data directly visible in the code.

Charles et al. [Cha+05] argued that the transparent location of objects introduce a performance bottleneck because the programmer does not know if a data access will generate remote communication. This assumption is supported by Barton, Cascaval, and Amaral [BCA07] who analyzed the data access characteristics in UPC. They found that the performance is significantly improved if the programmer is able to decide how the shared data is distributed among the executing threads because the compiler can optimize the data access at compile time.

Unified Parallel C (UPC) is a parallel extension of the C programming language, while *Titanium* is a Java dialect. Both, UPC and Titanium, follow the *single process multiple data* programming model where all nodes execute the same thread, but each thread operates on different data. The threads are placed on the individual nodes at startup and do not migrate during the runtime of the program. The main goal of this placement is to minimize the communication overhead between two cooperating threads, and it is the responsibility of the programmer to choose a good placement.

UPC uses global pointers to access remote data, and the UPC compiler translates remote accesses into inter-processor communication. Thus, the programmer does not need to deal directly with message passing or the underlying network topology. Nevertheless, Berlin et al. [Ber+04] observed a high memory access overhead because of the translation of shared to local pointers, which results in a performance degradation of a factor of 500 compared to a direct memory access in their test setup.

The PGAS model is well suited for irregular applications because it supports fine-grain remote access, but requires a low-latency interconnect between the nodes. For this reason, PGAS is not optimal for clusters of commodity hardware that are interconnected via slow network interfaces. Another disadvantage is the predefined placement of shared data that pins an object to a particular node during run time, and thus does not allow object migration.

2.2.5 Comparison of Message-Passing and (Distributed) Shared Memory

Shan et al. [Sha+01] compared shared-memory and MPI programming for regular applications (e. g. fast Fourier transformation) and irregular applications (e. g. radix sort). All these applications have either a high communication overhead or use complex communication patterns.

The authors used a cluster of eight nodes and the shared memory and the MPI programming model were implemented in software on top of this cluster to allow their comparison. The performance analysis showed that most MPI programs outperform the shared memory programs by a factor of two, while only one had about the same performance. Nevertheless, Shan et al. concluded that the ease of shared memory programming might make up for the performance loss.

Multiple authors [CE00; SB01; RHJ09] developed and evaluated hybrid systems for clusters of SMP nodes, which combined message passing with MPI for the inter-node communication and shared memory programming with OpenMP for the intra-node communication.

Cappello and Etiemble [CE00] and Smith and Bull [SB01] observed that the pure MPI code scales better than the hybrid approach of MPI plus OpenMP. Both conclude that a hybrid approach is suitable only for some limited situations such as fine-grained, irregular memory access or load imbalances.

Conversely, Rabenseifner, Hager, and Jost [RHJ09] showed that a hybrid approach can be superior because it can reduce communication costs on clusters of today's multi-core SMP processors. They also observed that the network topology has a significant impact on the performance. Thus, they take the position that the application itself should be aware of the underlying topology and apply the adequate actions depending on the underlying hardware. Furthermore, their conclusion is that none of the three programming models (MPI, OpenMP or a hybrid approach) optimally fits for the current HPC hardware architectures. Instead, the user has to decide individually which model should be applied to a particular problem.

Berlin et al. [Ber+04] and Mallón et al. [Mal+09] both compared MPI, OpenMP and UPC with each other, whereas Berlin et al. additionally compared Pthreads. They found that programming language features can ease parallel programming, but that they cannot hide the underlying communication costs. They concluded that fine-grain shared memory programming is not suitable for current clusters of SMP nodes. Instead, course-grain shared memory accesses are a better fit for the examined system because of the high bandwidth and high latency of the interconnects between the nodes.

Patel and Gilbert [PG08] compared MPI and UPC with each other. They reported that the MPI code outperforms the UPC code and observe that the performance of UPC code depends largely on the chosen underlying memory access solutions.

Altogether, these studies showed that it is not easily decidable which of the so far discussed models fits best for a given architecture and a given task without thorough benchmarking.

2.2.6 Single System Image

An alternative to the afore mentioned approaches is the use of a *single system image* (*SSI*) [Pfi98; BCJ01]. An SSI hides the distributed and potentially heterogeneous nature of (the processors in) a compute network and the communication among the different nodes. Thus, it provides an easy means to program and handle distributed and potentially heterogeneous systems: it enables the programmer to write applications for the distributed system as if it was a single SMP system. Namely, the user can write multi-threaded applications where the threads communicate via shared memory, without the need to worry whether this memory is local or remote. SSI

This illusion allows the system to dynamically distribute processes and data among the available cores regardless of their heterogeneous architecture. Additionally, it allows the system that creates the SSI to adapt to changes in the network without the necessity for changes in the application code.

Hence, the SSI frees the programmer from the burden of having to handle the following issues explicitly:

- Which resources are available in the network?
- On which nodes are these resources located?
- Which are the different components of the distributed application?
- How do these components communicate? E. g.by remote procedure calls (RPC), remote method invocation (RMI), the method passing interface (MPI), a proprietary protocol, . . . ?
- How can components be discovered? E. g.with JINI [Arn+99] lookup service, CORBA [Obj04], OSGi Service Registry [All10], Distributed Hash Table (DHT)?

The developer of an SSI can choose to implement the SSI on different levels:

- At the hardware level as in the FLASH multiprocessor [Kus+94], which offers a hardware distributed shared memory (DSM). A hardware SSI offers the highest level of transparency, but it is inflexible if the system needs to be extended or enhanced. Furthermore, it can only be combined with other hardware that supports the same SSI.
- At the operating system level as in MagnetOS [Liu+05], which is an operating system that offers an SSI on top of ad-hoc networks, or GLUnix [Gho+98], the global layer UNIX for clusters.

  At this level, the SSI is more flexible than an SSI on the hardware level, but it must be modified for each new hardware architecture, which makes it more expensive to develop and maintain.
- At the application level as e. g. in PARMON [Buy00], where a single application offers the user of the application transparent access to all system resources or services of a distributed system.

  Here, the SSI is limited to a single application and the developer of the SSI is not supported by the hardware or the OS.

- As a runtime environment such as a distributed Java virtual machine.

  An SSI on this level is a compromise between the other SSI levels. Only the developer of the runtime environment has to deal with the implementation of the SSI on top of the available hardware or OS. A programmer who develops for this runtime environment has full, transparent access to the resources of the underlying system.

The envisioned runtime environment, which is the basis for this thesis, provides an SSI in form of a distributed Java virtual machine (DJVM). Each core in the distributed system runs an individual instance of the DJVM in its own thread. Each of these DJVM threads, again, executes one or more Java threads. Together, all the DJVM instances collaborate to create an SSI that allows one or more concurrent applications to run transparently on heterogeneous clusters of multi-core machines. The VM instances distribute code, objects, and threads onto the compute resources, which may be added or removed at run-time.

While this SSI hides the complexity of the underlying hardware, it still leaves some of the difficulties of programming parallel applications, e. g. the synchronization of the concurrent access to shared resources. Therefore, this thesis also considers appropriate synchronization mechanisms.

2.2.7 Transactional Memory

As mentioned before, the common approach to synchronize the concurrent access to shared objects is *locking*. One example of lock based synchronization is the use of a binary semaphore that guards the access to a critical section and allows only one thread to enter the critical section at a time. The thread that gains access is said to "hold the *lock*" to the critical section. All other threads have to wait until the thread that is currently in the critical section releases the lock. Thus, the access to the shared objects (as accessed from within the critical section) is strictly sequential, which increases the access latency and slows down the execution.

Another problem in distributed systems are node failures: if the failing node held some locks, these locks are never released and all other threads that want to gain access to the critical section starve. Conversely, if the lock was released upon a (presumed) node failure, a returning node might cause inconsistencies.

To prevent these problems, a promising alternative to lock-based synchronization is *transactional memory* (*TM*) [HM93; ST95]. Instead of acquiring and releasing locks when accessing shared data, these accesses to shared objects are placed in atomic blocks. Such a block is a range of code that runs as a so-called *transaction*. The observable effects of this transaction, namely the reading and writing of shared data, appear to other nodes only at the end of the transaction. Thus, modifications become only visible as if they were executed as a single, atomic instruction. This enables the TM system to follow an optimistic approach that allows multiple transactions, and thus threads to access and modify the same shared objects at the same time. Only at the end of the transaction, upon the commit of the modified objects, the TM system has to check for and resolve potential conflicts, e. g. by rolling back and restarting the transaction.

The atomicity that a TM system guarantees is distinguished between *strong atomicity*, where objects are only created and modified from within transactions, and *weak atomicity*, which allows the creation and modification of objects outside of transactions [MBL06]. Hence, weak

atomicity can only guarantee atomicity between transactions, but not between transactions and non-transactional code, while strong atomicity either does not allow non-transactional code, or handles such code implicitly as if it was executed in a transaction as well.

Another distinction of TM systems is between *hardware transactional memory* (*HTM*) and *software transactional memory* (*STM*). The main advantage of STM over HTM is that no hardware support or hardware modifications are required. Thus, an STM is applicable for any legacy processor or microcontroller.

The study of Rossbach, Hofmann, and Witchel [RHW10] revealed that programming with STM is actually easier to get right than programming with locks, even though the participants of the study claimed that programming with STM is harder than programming with coarse-grain locks, and slightly easier than programming with fine-grained locks.

Pankratius, Adl-Tabatabai, and Otto [PAO09] made a similar observation. In their study, different groups of students had to solve the same computational problem, but some groups programmed with STM while the other groups used locking. The study showed that the STM groups spent less time to develop and debug their solution than the locking groups. Furthermore, the STM implementations performed better than the locking ones. Nevertheless, the authors stated that programming with STM was still difficult and understanding the behavior of the STM application was hard for the students.

The envisioned system of this thesis offers strong atomicity and uses a multi-version software transactional memory system, *DecentSTM*, that handles the parallel access to all shared objects with the help of a fully decentralized consensus protocol [BF10].

A distributed parallel application that uses DecentSTM will be commonly programmed as a multi-threaded application, where all threads have access to the same distributed globally shared memory, cf. Section 2.3.2 for more detail. Thus, all threads can work concurrently on the *globally accessible objects* (*GAOs*), which reside in the globally shared memory. **GAO**

Logically, the application operates on mutable shared objects, while the DecentSTM represents each shared object by a list of immutable object versions, called the *version history* of the object. Whenever a transaction accesses a shared object, it first has to create a private *local object copy* (*LOC*) of the latest object version. These private LOCs – together with other local variables – **LOC** reside in the transaction's private memory and thus, the access to these LOCs is restricted to this very transaction.

At the end of the transaction, the fully decentralized consensus protocol checks and resolves all potential conflicts that might occur because of the parallel access to the same shared objects from within multiple concurrent transactions. If the consensus protocol detects a conflict that would lead to an inconsistent memory state, the commit fails and the transaction is aborted, has to roll back and re-execute its computation. If there are no such conflicts, the commit succeeds and all modified LOCs become the so-called new *head versions* of the corresponding objects, and are appended to the corresponding version history. Thus, each object consists of a current head version and a chain of multiple, outdated versions. Suppose the latest head version Y_2 of an object Y is located on node A, and the new head version Y_3 happens to be committed on node B. Then, the object Y has seemingly migrated from node A to node B. See Figure 2.1 for an example where two nodes execute multiple transactions that create new versions of an object Y.

Note that DecentSTM does not immediately delete older versions in the version history. Instead, some older versions must be kept because they are needed in case a transaction that depends on one of these older versions has to roll back.

Besides the DecentSTM algorithm, our envisioned system contains a decentralized recovery mechanism that works on well-chosen, outdated versions of data to avoid explicit checkpointing. This recovery mechanism assumes that node failures comply with the *fail stop model*, i. e. nodes either work correctly or do not respond at all. To be able to commit or roll back a transaction, the STM system stores the *read* and *write sets* of the transaction, which contain all object versions that the transaction has read, written, or newly created, in a so-called *transaction record* (*TR*). The recovery algorithm extends this TR to additionally store the meta data of each transaction. With this meta data and the knowledge where the read objects are located, the runtime system on another node is able to fully reconstruct the transaction in case it is lost because of a node failure. Thus, it is necessary to guarantee a certain degree of redundancy to ensure that another node can get hold of the lost TR. More details can be found in Posselt [Pos10], where a first study of this decentralized recovery mechanism and replication management was done.

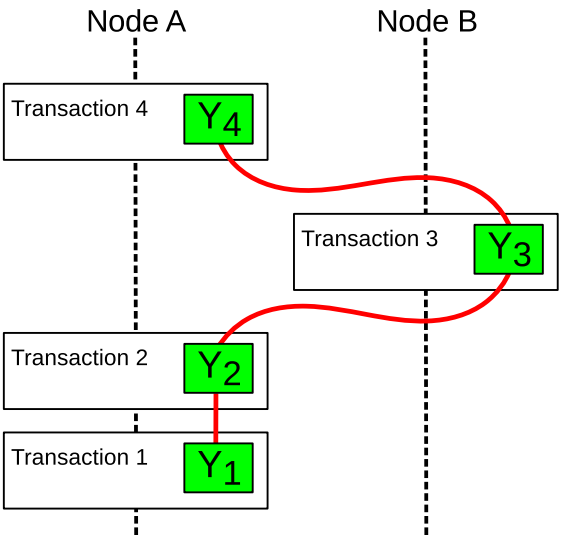


Figure 2.1: Version history of object Y with two implicit migrations from version Y_2 on node A to Y_3 on node B and back to node A with version Y_4.

2.3 Object and Memory Model

The object and memory model used in this thesis are tightly coupled. They are also influenced by the DecentSTM algorithm, which requires that a transaction can only operate on private copies of object versions, i. e. copies that are solely owned by this transaction.

2.3.1 Object Model

As stated above, the *object model* of this thesis uses the general term *object* to name all kinds of data such as object-oriented programming language data; objects and arrays, but also C structs, the program code or the execution context of a thread. In accordance to object-oriented programming languages, objects are distinguished between *dynamic* and *static* objects. Figure 2.2 shows dynamic objects, which are Java objects (instances of a class), Java arrays, or execution contexts (threads), and static objects, which are e. g. Java class variables and code objects.

To be able to access an object, the accessing entity has to hold a reference to the object. Such a reference is an identifier that uniquely identifies a particular object. Within a single core or a

SMP system with multiple cores, the reference to an object can be represented as a local memory address.

To access a dynamic object, it is necessary to possess a reference to the object. This reference may be acquired by reading a reference field of another object. However, the only way to create a new reference is by instantiating an object. Other possibilities, for example, the reception of a reference as method parameter during method invocation, are only special issues of the two basic cases. There should be no way for faulty or malicious client code to make up a valid or invalid reference - at least not easily or by chance. In particular, it should be impossible to forge an object reference, e. g. by using an integer value as an object reference.

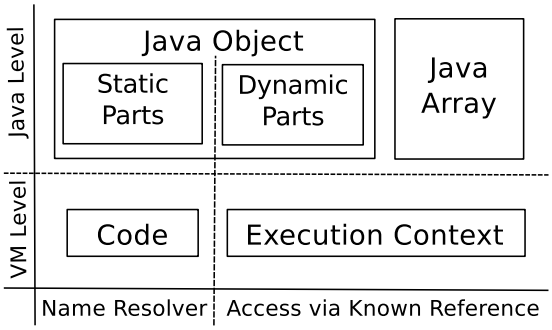


Figure 2.2: In the object model, the term *object* names all kinds of data such as object-oriented programming language data; objects and arrays, but also the program code or the execution context of a thread.

On the one hand, this is important for open distributed systems such as the AmbiComp scenario, cf. Section 4.1. In this scenario, the distributed network might expand across a single home, for example, for home entertainment, or a complete neighborhood for an automatic sun shade management. Here, a malicious node might try to gain access to private applications if it could forge references. On the other hand, it is also important for closed distributed systems such as the J-Cell scenario, cf. Section 4.3. In the J-Cell scenario, the distributed system expands, for example, throughout a data center or compute cluster. Here, a faulty node should not accidentally make up a reference to a valid object in another application and then unintendedly damage an ongoing computation.

To prevent this, the DecentVM [BEF10] for example, separates numeric values from reference values within the runtime system and stores them on separate stacks.

The objects that belong to the same application may reference each other in one way or another: a member variable of an object may hold a reference to another object; ditto for arrays. An execution context holds a reference to the code it executes, and to the objects that belong to the parameters or the local variables of the executed methods.

Altogether, these objects form the *reference graph* of the application. It is rooted in the application's *primordial execution context* (*PEC*) and evolves during the execution of the application. Upon application start, the reference graph consists only of an empty execution context – the PEC – and the associated code.

reference graph

So far, the object model was limited to systems that contain a common shared memory where a reference might be represented as a local memory address. However, in a distributed system the objects are scattered across all nodes and reside in the local memories of the independent nodes. Thus, the object model introduces the distinction between *local* objects, which reside in the local memory, and *remote* objects, which reside in the memory of a remote node.

Because the memory is distributed among all nodes, remote objects cannot be referenced with a plain memory address, because local memory addresses are only valid within the address space of the respective node. Thus, a globally valid reference is necessary that indisputable identifies a remote object.

Such a *global reference* can be either location dependent or location independent. A location dependent global reference might be a tuple of the node identifier of the node where the remote object resides plus the local memory address on this node: `<Node Id, Local Memory Address>`.

A location independent global reference representation is, for example, the use of a *globally unique identifier* (*GUID*) that suffices to uniquely identify an object within the whole system. To locate the referenced object, additional location information is needed to be able to find the node on which the object resides. This information can be provided by an underlying routing protocol, such as the ad-hoc routing protocol *scalable source routing* (*SSR*) [Fuh05] that can be used to route access requests to the current location of the remote object.

The disadvantage of a GUID is the necessary guarantee that a GUID is only assigned once for a unique object. This requires either a consensus among all nodes, or a collision-free addressing scheme that requires a large address space.

2.3.2 Memory Model

The *memory model* is derived from a *non-uniform memory access* (NUMA) architecture and distinguishes between *logical* (private or global) and *physical* (private, local, or remote) memory, cf. Figure 2.3. I describe the following memory model with respect to DecentSTM, because the envisioned system will use DecentSTM to manage the concurrent access to shared data. For this, DecentSTM manages the *logically private* and *logically global* memory. Namely, it mediates between the logically private memory, which stores the data of a single transaction, i. e. the local variables and local object copies, and the logically global memory, which stores the shared globally accessible objects.

Logically private memory typically maps to the *physically private* memory. This physically private memory is usually tightly coupled with a single core of the processor, for example the cache or registers. Thus, it has a very low latency. In Figure 2.3 the physically private memory is shown right above the individual processor cores. It is only accessible from a single core and thus, only from the transaction that this core executes.

If the physically private memory overflows, logically private objects may be offloaded to local memory, but must be logically separated from the locally stored globally accessible objects.

The logically private memory is used to store all transaction's local variables, together with all local object copies that the transaction works on. Additionally, the logically private memory stores all objects that a transaction created. Because local object copies and local variables reside in the same address space as the transaction, they are directly accessible via local memory addresses, cf. Table 2.1.

The *physically local* memory of a processor core, for example, the local RAM, and the processor core itself reside on the same node. From the perspective of a single node, this memory is shared by multiple runtime instances that reside in the same address space, e. g. on the cores of the same processor. Thus, it is possible to access this memory with local memory addresses. In

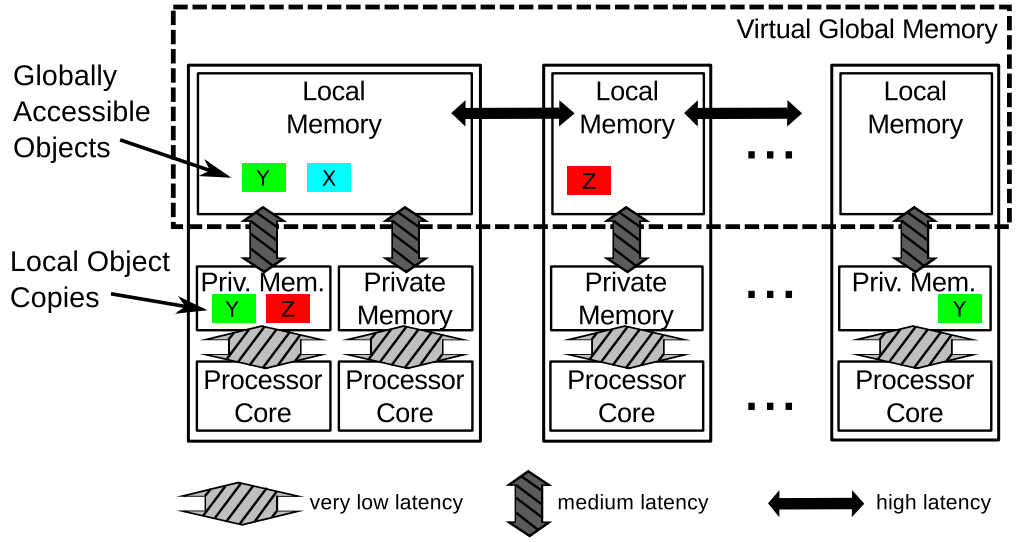


Figure 2.3: The memory model, which is derived from a NUMA architecture. The physically private memory is shown right above the individual processor cores. The physically local memory reside on the same node as the processor cores and the logically global memory consists of all the physically local memories that are available on all nodes in the network.

Figure 2.3, this memory is depicted at the top of the individual nodes and usually has a constant medium access latency.

From the perspective of a single node, the *physically local* memories of all other nodes are combined into the *physically remote* memory. It is the task of the runtime system that creates the SSI to make all these local memories appear as the *logically global* memory.

		Physical		
		private	local	remote
Logical	(private) LOC	Mem. Addr.	–	–
	(global) GAO	–	Mem. Addr.	GUID, etc.

Table 2.1: Reference Representation of GAOs and LOCs depending on their physical and logical location in memory, similar to [SPF11].

The *logically global* memory is shared among all nodes in the network. It does not have a physical representation of its own, but consists of all the *physically local* memories that are available on all nodes in the network. In Figure 2.3, this logically global memory is pictured as the dashed box that is drawn around the individual local memories of all nodes. It stores the globally accessible objects, i. e. the shared objects, namely the immutable object versions of these shared objects. If such an object version happens to reside on a remote node, it is accessed via a *global reference*. If the object version happens to reside in the local portion of the global memory, it can be accessed with a *local reference*, for example, a local memory address.

In contrast to the own, *physically local* memory, the physically remote memory has a high and variable access latency. This latency depends on the distance between the accessing node and the accessed node in the network: i. e. an access to a direct neighbor is typically faster than an access to a node that is multiple hops away.

Table 2.1 gives an overview where local object copies and globally accessible objects might be located in the different physical memory locations, together with the type of reference that might be used to access them. One can see that LOCs only reside in the logically and physically private memory, where they are accessible with a local memory address. GAOs can reside either in the physically local memory, where they are accessible via local memory addresses, or in the physically remote memory, where they must be accessed with a global reference.

3 Related Work

This chapter outlines the related work of this thesis. It starts with a description of distributed systems in general and gives an overview over mobile computing, especially over different types of mobile objects and mobile code, e. g. *mobile agents*.

Afterwards, it describes various *distributed shared memory* systems in more detail. This description is followed by a section about distributed operating systems, which gives only a brief overview over this topic, before a section on distributed Java virtual machines closes this chapter.

The general concepts to locate mobile objects that are presented throughout this chapter, e. g. centralized server, broadcasts or proxy forwards, are discussed in detail in Section 6.2. Thus, the following sections do not go into a thorough comparison of these concepts.

As defined in Chapter 2, the following sections use the phrase *home node* in different meanings. Some approaches use a *static home node* that is assigned once, e. g. when the object is created, and that does not change during the lifetime of the object. Whenever an approach in the following sections uses the phrase home node in this sense, the home node is called a *static home node*.

Other approaches use the phrase *home node* for a *dynamic home node*. Here, the object can migrate between nodes and each migration also migrates the responsibility for the object to the new node. If not stated otherwise, the following sections use the phrase *home node* in the sense of a *dynamic home node*.

3.1 Distributed Systems

A *distributed system* is a set of separate and independent compute nodes that are connected to form a network of nodes. The main goals of a distributed system are scalability, which allows to add additional resources, as well as easy maintenance, which allows to remove resources at runtime for repair or exchange.

In such a system, nodes communicate and interact with each other to achieve a common goal. For this, the application is split into parts that are executed on the different nodes. Thus, each node processes its assigned tasks and exchanges data and workload with others.

Today, different computing paradigms fall into the broad definition of distributed systems. Some keywords are e. g. *cluster*, *Grid*, *Cloud* or *high performance computing* (*HPC*).

cluster

A *cluster* is a system that consists of two or more individual compute nodes that are connected via a high-speed network [Pfi98; Buy99]. Depending on the application area, different types of clusters exist. For example:

- *high performance clusters*, which are used for HPC, e. g. heavy numerical computations such as weather forecast or particle dynamics.

- *high availability clusters*, which ensure that the user has constant access to e. g. a service application.

- *load balancing clusters*, which distribute the workload evenly on all compute resources to optimize the performance of e. g. a small cluster of only some ten nodes.

Grid

A *Grid* is according to Foster [Fos02] a system that has no centralized management. It uses open and standard general-purpose protocols and interfaces to provide the user with a nontrivial quality of service. In contrast to cluster computing, a Grid tends to be more loosely coupled and often connects commodity hardware via a conventional network such as Ethernet. While a cluster often performs a single task, a Grid offers a number of Grid services to meet the needs of the users [Fos+03].

Grids and clusters are often the foundation for HPC [DS98], where the main goal is to reduce the time a computation requires to finish. Thus, HPC enables the user to execute a larger number of computational tasks in the same execution time it takes to execute a single task on a single compute node.

Cloud

In contrast to Grids, clusters and HPC, *Cloud* computing has its main focus on scalability, and not on performance. Cloud computing ensures that the available resources grow together with the increasing demands of the user. Thus, the main focus of Cloud computing is on-demand *software as a service* [Arm+09].

According to Hayes [Hay08], the unique characteristic of Cloud computing is the shift from the local PC to a datacenter in the Internet. I. e., the user does not install a program on the local PC but has access to the software via a web frontend.

Furthermore, the cloud stores all documents that belong to the user. This has the advantage that the user can access the documents from every computer that has an Internet connection. The drawback is that the user has no direct control over the data anymore.

3.2 Mobile Objects

To increase the usability of a distributed system and support a broad range of applications it is advantageous to support the migration of objects and tasks among different compute nodes. This object migration results in *mobile objects*, where the definition of a *mobile object* depends on the application domain. Hence, a mobile object can be:

- a piece of mobile code and data, called software or *mobile agent*, that performs its task on different, remote nodes in a network. The code and data travels actively through the network to execute its operations on the different nodes, e. g. to collect data in a sensor network.

- a piece of memory, or *object*, as defined in object oriented programming languages. The object is passively moved through the network towards the code that requires the object's data for its execution.

- a physical resource, such as a device in a wireless network. People, robots or tools on a factory floor can also be considered as mobile objects. These resources are tracked or located e. g. to optimize the workflow or to bring workers and tools together. Pitoura and Samaras [PS01] give an overview and summary of location approaches for this kind of mobile units.

As seen in this bullet list, the movement of mobile objects can be either *active* or *passive*. *Active movement* is governed by the object itself; for example, by a worker who decides to go to another location, or a mobile software agent that has finished its task and moves on to the next node. *Passive movement* is managed by a separate entity that is different from the moving object. This entity can be for example the programmer, who explicitly embeds the object migration in the program, or a distributed runtime environment that implicitly migrates objects between nodes.

It depends on the application if the mobile object is only a data object, only a code object, or both. This thesis does not make a clear distinction between mobile objects and mobile code, cf. Figure 2.2, because the envisioned system can handle both, equally.

Nevertheless, the following sections describe the two terms *mobile object* and *mobile code* in more detail, but many of the presented approaches that have been developed for mobile data also work for mobile code, and vice versa. Often, the only difference is the target scenario.

3.2.1 Mobile Data

Mobile Data is a chunk of memory that is transferred between different nodes in the network.

A common target scenario for mobile data are sensor networks [Aky+02; YMG08]. Here, the mobile data can be the data that the individual sensors read, and that has to be sent to a central sink for further processing. The transfer of the data is achieved by a network protocol that handles the communication.

Message passing, cf. Section 2.2.1, and the *remote procedure call* (*RPC*) mechanism [Sri95] are other examples to transfer mobile data. Both allow a process on one node to directly transfer the control and the data to another node.

The main difference between message passing and RPC is the level on which the communicating entities are involved. With message passing, two equal processes communicate with each other via send and receive functions, while one RPC process calls a function that is offered by another remote process. Thus, the two processes in the RPC model are in a client-server relationship: The client process sends the parameters, i. e. the data, for the function call across the network, and the remote server process executes the called function on behalf of the client. In case that the called method returns a result, the client process suspends its execution until it receives the result from the server.

Some middleware systems that are based on RPC mechanisms are *CORBA* (*Common Object Request Broker Architecture*) [Obj08a], developed by the *Object Management Group (OMG)*, *DCOM* (*Distributed Component Object Model*) [EE98; Mic11] developed by Microsoft, or *Java RMI* from Sun.

CORBA *CORBA* [Obj08a] is a programming language independent, object-oriented middleware specification. Its goal is to simplify the development of distributed applications in heterogeneous networks and to enable the interoperability of different software components.

The central component of CORBA that handles the interaction between CORBA objects is the so-called *Object Request Broker* (*ORB*). The ORB is responsible for the location of objects and the handling of all communication between objects.

CORBAs communication model is client-server based, where client objects access services that are offered by server objects. The ORB is responsible for transferring the method call and its parameters from the client to the server side. The interaction between client and server objects takes place via so-called *stubs* or *proxies*. A proxy on the client side offers the same interface as the remote server, but no functionality. This is necessary because the client and the server object are located in different memory address spaces, i. e. a memory address on the client side is invalid on the server side.

How a particular CORBA implementation realizes the ORB is not specified and up to the particular implementation. Nevertheless, the specification offers some possible implementation scenarios for ORBs, e. g. a server-based ORB implementation, where the ORB is a central component, or a system-based ORB, where the ORB is implemented in the operating system and each ORB has global knowledge about the location of all objects in the system.

To make a service available to other nodes, a server object must register its service at the ORBs on another node. Therefore, each ORB manages a repository where it stores the reference and location of remote server objects. Alternatively, an ORB might act as a location service as well and answers requests for a given reference with a *location-forward* reply that contains the location of the requested object [Obj08b].

To locate remote services, the CORBA specification proposes a *naming service* in a separate specification [Obj04], which is similar to the *domain name service (DNS)* [Moc87]. This specification notes that the location of the naming server must be explicitly given to an ORB, e. g. via a configuration parameter. However, the CORBA specification itself does not require such a service.

Furthermore, CORBA supports object migrations via the optional ORB *life cycle service* [Obj02]. The life cycle service defines the creation, copy or migration, and deletion of local and remote objects. The specification requires that a reference to a migrated object stays valid after a migration, but does not make any statements how this reference mechanism should work. Therefore, according to Henning [Hen98], the different commercial and non-commercial ORB implementations that exist all handle the object migration differently.

Despite all the effort that was put into the CORBA specification, it was not widely accepted [Hen08].

DCOM Similar to CORBA, *DCOM* [EE98; Mic11] is an object-oriented middleware for distributed application. It defines a component model that describes the interaction of objects together with a corresponding infrastructure, e. g. for persistent storage of data or a unified communication model.

Chung et al. [Chu+97] showed that the architecture of DCOM and CORBA are similar, but in contrast to CORBA, DCOM is mostly limited to Windows. Even though there is a Unix implementation made by the Software AG, this implementation never gained a broader acceptance.

Java RMI *Java RMI* (remote method invocation) [Dow98] is an example for the equivalent of RPC in object-oriented programming languages. Similar to CORBA and DCOM, Java RMI allows the access to and invocation of methods on remote objects. Furthermore, it allows objects to migrate across node boundaries.

The central component of Java RMI is a centralized RMI registry that is used to locate remote objects. A node that offers a service has to register this service with the RMI registry. Conversely, a client node that wants to use a service has to look up the location of the server node at the RMI registry.

Some authors argue [PHN00; Maa+01] that Java RMI is insufficient, because it is based on slow object serialization. Therefore, Philippsen, Haumacher, and Nester [PHN00] designed a more efficient object serialization, called *UKA-serialization*, and made a re-design and re-implementation of Java RMI, called KaRMI. Their goal was to enable Java RMI for high-speed communication, e. g. for clusters of workstations that are interconnected via a non-TCP/IP network interface. As a result, the authors report that their optimizations saved on average 45 % of the runtime in an Ethernet network, and on average 85 % in a high-speed Myrinet network.

Maassen et al. [Maa+01] implemented a more efficient Java RMI in the *Manta* system. Manta is a compiler based Java system that uses a native static compiler. This compiler generates specific serializers for their Java RMI at compile-time. Furthermore, the authors designed a more efficient RMI protocol implementation. With this approach, Manta is able to push almost all RMI overhead to the compile-time and the authors report for a 32 node Myrinet cluster that they achieved a 35 times lower latency for a *null-RMI* (no parameter and no return value) than the SUN Java RMI.

Various authors, e. g. [Chu+97; PS98; MZ01], compared CORBA, DCOM and Java RMI with each other. Chung et al. [Chu+97] compared a DCOM and a CORBA implementation on various layers; namely on the communication protocol, the remoting and the programming layer. The authors goal was to give users who know one of the two architectures a quick understanding of the other architecture. As a result of their comparison, the authors came to the conclusion that DCOM and CORBA are on all three layers basically the same.

Munoz and Zalewski [MZ01] made a similar, but more thorough comparison of two CORBA implementations and Java RMI, which included performance measurements of different benchmark applications. They identified various sources for latency overhead, such as object location and parameter transformation. The authors stated that CORBA and Java RMI introduce an overhead that is about twice as high as POSIX socket calls, but that they are about three times faster than HTTP/CGI. Thus, they concluded that both, CORBA and Java RMI, are suitable architectures for the development of distributed applications, because the increased latency is still small enough to offer good results in an Ethernet network.

Plasil and Stal [PS98] compared the architectures of CORBA, DCOM and Java RMI with each other. Their focus was on the general architecture and they did not consider the topics of security or object mobility. Altogether, the authors came to the same conclusion as Chung et al.: all three architectures are basically similar and address the same problems, but their descriptions do not use the same languages.

3.2.2 Mobile Code/Mobile Agents

An alternative approach to migrate the objects to the code, e. g. via RPC, is to move the code to the data, which can significantly reduce the amount of communication. In the sensor network scenario for example, instead of shipping all data from the individual sensor nodes to a central sink for processing, the code that processes the data can travel to all sensor nodes, collect the data and perform the processing on the fly.

mobile code

mobile agents

The terms mobile code and mobile agent have different meanings in the literature. Adl-Tabatabai et al. describe *mobile code* as "any program representation that can be shipped unchanged to a heterogeneous collection of processors and executed with identical semantics on each processor " [Adl+96, p. 1]. Knabe understand *mobile agents* as "code-containing objects that may be transmitted between communicating participants in a distributed system" [Kna97, p. 1], while Pham and Karmouch see them as "self-contained and identifiable computer programs that can move within the network and act on behalf of the user or another entity" [PK98, p. 26].

For Lange and Oshima [LO99] mobile agents have a long list of advantages: They reduce the network load, overcome the network latency, can execute asynchronously and autonomously, adapt dynamically to system changes, run on heterogeneous systems and offer robustness and fault-tolerance.

An example for a programming language that was explicitly designed with mobility in mind is Java [Gos95]. Java allows the shipment of a single class file, but also of a whole application, to another node. As long as a Java virtual machine is present on that node, the code of the application can be loaded and executed, independent of the actual hardware architecture.

The next section gives an overview of some mobile agent systems that deal with the location of cooperating mobile agents. Even though this thesis does not deal directly with mobile agents, thread or process migration, the question of locating mobile agents is closely related to this thesis. Good surveys on thread and process migration can be found e. g. by Pham and Karmouch [PK98], Milojicic et al. [Mil+00] or Milanés, Rodriguez, and Schulze [MRS08].

Chen, Gonzalez, and Leung [CGL07] described the communication of cooperative agents via state variables. In this scenario, one agent leaves a state variable on the visited nodes to communicate some information to the next visiting mobile agent. Thus, state variables are not only distributed in space, but also in time, because the next agent might visit the node at an arbitrary time. This approach is very simple, but only suitable for data exchange scenarios that are not time critical.

Cao et al. [Cao+02] proposed a *mailbox* approach for mobile agents in the Internet. This approach introduces for each mobile agent an additional mailbox. This mailbox is an intermediate proxy that receives all message for the mobile agent. To receive this message, the mobile agent either has to pull them from the mailbox, or the mailbox pushes them to the mobile agent.

It depends on the user configuration whether the mailbox migrates at the same or at a lower frequency as the mobile agent, or stays put on a particular node. The latter case is similar to a home-based approach where each mobile agent has a static home node that is responsible for the mobile agent (cf. Section 6.2.4 for details). If the mailbox migrates, its migration path is bound to the migration path of the mobile agent. I. e., a mailbox can only reside on a node where the mobile agent was located at some point in history as well.

If the mailbox migrates together with the mobile agent, the agent leaves a forwarding pointer behind. This approach is similar to proxy forwarding, which is described in more detail in Section 6.2.5. If the mailbox migrates at a slower rate than the mobile agent, each mailbox migration shortens the proxy forward path. The authors stated that this forwarding pointer approach does not have an update message overhead, but they do not explain how the mailbox receives the latest location of the mobile agent.

Cao et al. [Cao+03] examined this approach in more detail and added a *time-to-live (TTL)* to each mailbox that indicates how long the mailbox was not used. When the predefined TTL expires, the mailbox is considered to be unnecessary and is removed. The main difference between the proxy approach discussed in this thesis and the message forwarding/mailbox update approach lays in the assumed communication pattern. Cao et al. assumed that a sender simply sends a message to a remote mobile agent but does not wait for a timely reply. Thus, a timely location update is not important for the sender.

Moreau and Ribbens [MR02] developed *Mobile objects in Java*, a middleware library to support the development of mobile agent systems. The communication among agents is based on a client-server model, where both, the server and the client are allowed to migrate.

Migrating agents leave a trail of forwarding pointers behind. Thus, an agent access might have to traverse a chain of proxies, for which Moreau and Ribbens investigate two routing strategies to forward an agent access request: *Call forwarding*, which is similar to the recursive proxy forwarding, where a message is redirected to the next proxy in the chain, and so-called *referrals*, which are similar to the iterative proxy approach, where a message with the new location of the agent is sent back to the requesting node, cf. Section 7.1.2. However, unlike the approach presented in this thesis, the response message travels back along the chain of proxies as well.

Furthermore, Moreau and Ribbens described two proxy update mechanisms: *Eager Acknowledgments*, which send the new agent location to all proxies in the proxy chain, and *One Acknowledgments*, which only updates the latest proxy. This thesis present a similar, but more advanced approach in Section 7.3. There, the depth to which updates are sent is computed depending on the number or read and write accesses and the length of the established proxy chain.

The benchmark results of Moreau and Ribbens showed that *Eager Acknowledgments* decrease the agent access latency and they stated that the call forwarding approach is slower than the referral approach. According to the authors, the reason is that the call forwarding approach has to traverse the chain of proxies two times, one time on the way towards the agent, and a second time when the answer is sent back. In my opinion, this second traversal is unnecessary because the answer could be sent back directly, without the indirection via the chain of proxies. Furthermore, unlike this thesis, the paper does not make any measurements of the messages overhead or the number of proxy forwards.

Bisignano, Modica, and Tomarchio [BMT03] described a two-level location approach for mobile agents. This approach uses a centralized server within a small region of the network, and peer-to-peer mechanisms within the global network. Hence, their approach splits the network into small *regions*, where each contains a centralized server where all agents within this region have to register. Before an agent can migrate to another region, it first has to de-register at the current region server. Then the agent migrates to another region and has there to re-register with the region server of its new location. To locate mobile agents across region boundaries, all regional location servers communicate among each other via the JXTA peer-to-peer mechanism [OTG02].

Even though this approach distributes a single centralized location server among smaller regions, these centralized region servers remain a bottleneck and single point of failure within each region.

3.3 Programming Languages and Middleware

The following section describes approaches that introduce either a new programming language or a middleware for distributed systems. These programming languages come with an additional runtime system that handles the communication among the different nodes.

Emerald *Emerald* [Jul+88; SJ95] is a programming language for distributed systems that offers a distributed runtime system and specialized compiler. It aims as homogeneous [Jul+88] or heterogeneous [SJ95] processor clusters of up to 100 nodes.

Emerald allows threads and objects to migrate among nodes, where the threads follow the objects when the objects are moved. Thereby, Emerald keeps threads and data co-located with each other.

To reference an object regardless of its location, Emerald uses globally unique and location independent object identifiers together with a *hash table* that maintains the object identifiers. Each global object that is referenced by a thread on a given node has an entry in this table. If the global object resides in the local memory, the object descriptor contains the local memory pointer to the object. Otherwise, it contains information about the objects' location. Thus, this table is similar to the tables described in Section 5.3.

If the location of a remote object is outdated, Emerald uses *forwarding proxies*. If this forwarding approach fails due to a node failure, the protocol falls back to a broadcast protocol. However, no further information are given on this approach. Thus, it is unclear if proxies are removed or remain in the system indefinitely.

As the Emerald system compiles a program to machine code, and not an intermediate language, the program must be re-compiled for each platform in a heterogeneous network. To be able to execute an application on a heterogeneous platform, Emerald proposes a centralized program database that stores the different code objects for the different hardware architectures. Steensgaard and Jul [SJ95] described this mechanism in more detail.

However, the main drawbacks of Emerald are that the programmer has to learn a completely new programming language, and that Emerald only aims at small to medium cluster sizes with only up to 100 nodes.

ProActive Baduel et al. [Bad+06] described *ProActive*, a Java Grid middleware that is designed as a Java library. It was developed for Grid programming with mobile agents, which are called *active* objects. Each agent runs in its own thread, which only executes methods from the agent. Additionally, each agent is associated with one or more *passive* objects, which the agent is allowed to access as well.

To communicate, ProActive agents have to pass messages among each other. They are not allowed to share the same passive objects. Thus, to pass a passive object from one agent to another, it has to be passed as a deep-copy.

Agents and their associated passive objects can migrate among nodes. To keep a migrated agent accessible, Baude et al. [Bau+00] described two approaches: The first approach uses forwarding proxies while the second approach is based on a centralized server. Baduel et al. mentioned a third, hybrid approach, which is not described in more detail.

Alouf, Huet, and Nain [AHN02] compared the forwarding approach and the centralized server with each other and evaluated both approaches with simulations and in an experimental environment. In contrast to this thesis, they examined only the communication between one source and one agent and assumed that an agent does not return to a previously visited node. Furthermore, they did not allow communication between a node and an agent while the agent migrates and made no statements about the removal of proxies or the location update process.

Linda Gelernter [Gel85] introduced *tuple spaces* as a shared memory abstraction. In this abstraction, the shared data is represented as a *tuple* of a given name and one or more additional formal or actual parameters. With this tuple, shared memory is not accessed via a known memory location, but via pattern matching queries for e. g. the name or one or more of the parameters of the tuple.

As an example, suppose a process A wants to send data to process B. For this, process A first creates a tuple and then inserts it into the tuple space, from where process B has to withdraw it. Thus, tuples are distributed in space as well as in time, because a tuple might be withdrawn at an arbitrary time from any number of arbitrary processes on arbitrary nodes. Even though the sender does not need to know which process will receive the tuple, it is possible that the sender specifies the receiving process, e. g. via a parameter that identifies this process.

Linda [ACG86] is a programming tool to develop parallel programs that are based on tuple spaces. A program in Linda is different to the common approach to develop a parallel program. The common approach is to partition the program into n tasks that depend on each other and that are executed by n processes. Instead, a program in Linda executes a number of spatially and temporally independent tasks that are inserted into the tuple space. To execute these tasks, Linda applies a so-called *replicated worker model* that replicates the program r times, where r is the number of available processors. These r independent workers search for tasks to execute in the program's tuple space.

Compared to common read and write memory access operations, the tuple space is accessed by *read*, *add* and *remove* operations. The *read* operation tries to read a tuple in the tuple space by issuing a query for its logical name. If such a tuple exists, its values are read into a local tuple and the original tuple stays in the tuple space. If more than one tuple exist, one is chosen arbitrarily.

If no such tuple exists, the reading process suspends until a matching tuple is present in the tuple space. The *remove* operation is similar to the *read* operation but removes the tuple from the tuple space. To write the values of a tuple, the tuple has to be removed from the tuple space, changed and reinserted.

The implementation of Linda's tuple space and the tuple location mechanisms are up to the developer. One implementation of Linda, for example (for the S/Net multicomputer), broadcasts all tuple space access messages to all nodes in the network, so that each node in the network stores a copy of the complete tuple space. Another implementation (for the Intel iPSC hypercube) uses a distributed hash table to store and locate tuples.

Orca *Orca* [Bal+98] is another programming language for the development of parallel programs for distributed systems. Additionally, it is an object-base distributed shared memory system. It supports mobile code as a fundamental programming construct, e. g. by forking a process that is started on a remote node. Additionally, it allows the migration of objects between nodes.

A reliable broadcast protocol handles the communication between nodes and each node caches all shared objects. To keep shared objects consistent, the node that modified a shared object broadcasts its changes so that all other nodes can update their data.

3.4 Distributed Shared Memory

As described in Chapter 2, the alternative to the 'share-nothing' paradigm of message passing is *shared memory*, where the whole memory is directly accessible from all nodes and processes in the network. Thus, a process on one node has direct access to all memory addresses of all nodes within a globally shared address space. However, such globally shared memory systems require hardware support as e. g. offered by the FLASH multiprocessor [Kus+94].

DSM

The alternative to hardware-supported globally shared memory is *distributed shared memory* (*DSM*), which is a combination of message passing and a globally shared address space. A DSM provides the abstraction of a shared memory in a distributed memory environment and hides the underlying communication between the different nodes.

DSM systems can be distinguished into hardware DSMs, where a dedicated hardware is responsible for the shared memory abstraction, and software DSMs, where the runtime environment provides this abstraction. Another dimension to classify DSMs is object-based vs. page-based memory.

In the following section, this thesis only describes object-based and page-based software DSMs, because these DSMs are closest to the envisioned system. All of these DSM systems use either centralized managers, distributed directories, static home nodes or forwarding proxies to locate and manage the access to memory pages or objects. Because these different approaches are discussed in more details in Section 6.2, the following section only lists the different DSMs, without evaluating or comparing them. As stated above, only notable concepts are described in more detail.

IVY Li and Hudak [LH89] described the first page-based DSM called *IVY*. IVY allows the migration of memory pages, but it only allows a single writer at a time to keep the memory consistent.

To locate a memory page in the distributed system, Li and Hudak examined various page location approaches: In the centralized approach a central manager maintains a table of all available shared memory pages. Pages do not have a static home node and only the central manager knows who the current owner of a page is. Moreover, the manager serializes the access to the page by locking the page for all other processors but the one to whom the central manager granted the access.

An improved centralized manager approach moves the page access synchronization from the central manager to the current owner of the page. However, the central manager is still the only entity that knows who the current page owner is and forwards all access requests to that node. To remove the central manager component, the authors proposed a distributed protocol that uses broadcasts to find memory pages.

A fixed distributed manager approach partitions the shared address space into predetermined, fixed chunks, which are distributed among all nodes. Thus, each node is the static home node of a subset of pages. To locate pages in this scenario, the authors propose a hash function that maps page addresses to nodes, similar to the DHT approach described in Section 6.2.3.

Yet another approach describes the use of forwarding proxies. These forwarding proxies are updated whenever an invalidation message is broadcasted because this message propagates the true owner of the page.

The authors evaluated these concepts in a network with eight nodes. They argued that the distributed approaches that use broadcast or proxies perform better than the centralized manager approaches, if only a small number of processors share the same page for a short period of time.

TreadMarks *TreadMarks* [Kel95; Amz+96] is a page-based DSM that uses a centralized manager to initialize the system. In TreadMarks, each page is assigned to a static home node that manages the page access. Other nodes that want to access the shared page first have to retrieve a copy of that page, while the master object remains on the static home node. To keep the different copies consistent, TreadMarks implements the *lazy release consistency* (*LRC*) [Kel95]. The idea of LRC is to propagate updated pages not immediately, but on request. Before a node modifies the page, it creates an additional copy, the so-called twin. When the node finishes its modifications on one of the copies, it computes the differences between the modified copy and the twin. Afterwards, the diff is propagated to the manager and can also be retrieved by other nodes that also modified the page, so that they can apply the diff as well.

TreadMarks' programming model allows the usage of locks and barriers to synchronize the access to shared pages. The management of a page lock is done by the static home node of the page. It is the responsibility of the static home node to track the current lock owner, i. e. , the last node that required the lock. On the contrary, the management of barriers is in the responsibility of the centralized manager.

JavaParty *JavaParty* [PZ97] is a Java runtime environment for the development of distributed applications. It uses a central runtime manager that is responsible for the management of the

distributed environment. This central manager is also the central component where all JavaParty virtual machines must register to participate in a JavaParty computation.

JavaParty defines remote objects for Java and is built on top of an optimized Java RMI, described in [PHN00]. It modifies the Java language by introducing a new class modifier *remote*. Thus, Java programs that are written for a JavaParty environment must be pre-processed to transform the JavaParty code into regular Java code with RMI hooks. Afterwards, this generated code is compiled with the RMI compiler to produce the executable for the JavaParty virtual machine.

The JavaParty runtime system is built around the central runtime manager, which knows all JavaParty instances and the location of all class objects, i. e. the host that initialized the static parts of a class. To reduce the management overhead, this information is replicated at all JavaParty instances.

JavaParty allows object migration and uses forwarding proxies that are left behind to guarantee the reachability of the migrated object. These proxies handle method calls by sending the new location of the object back to the caller.

JavaSymphony *JavaSymphony* [Fah00; FJ05; APF10] is built on top of Java RMI as well. Furthermore, it is implemented as a Java API and, in contrast to JavaParty, does not make language modifications and does not need a pre-processing step. Thus, JavaSymphony runs on a common Java virtual machine.

The main feature of JavaSymphony is the ability of the programmer to explicitly control object and thread locality for e. g. load balancing, on a high level. All the underlying mechanisms for socket communication or Java RMI calls are hidden within the Java library.

The user can select the nodes and resources that should be used to execute a JavaSymphony application. These resources form the *virtual distributed architecture (VDA)* that executes the *JavaSymphony runtime system (JRS)*. The VDA is organized in a tree-based layered hierarchy in which the lowest layer represents the individual compute nodes. Furthermore, because JavaSymphony uses Java RMI, a central registry at the top-most layer handles the remote object access and object migrations. The intermediate layers define management nodes that are responsible for sub-regions of the VDA.

Even though JavaSymphony supports automatic mapping, load balancing and object migration, Fahringer [Fah00] advocated that the user handles these tasks in the application.

Aleem, Prodan, and Fahringer [APF10] described an extension of JavaSymphony for shared memory systems such as multi-core and many-core architectures. For this, the author introduce an additional shared memory object type that is handled by a local object agent. This object agent is responsible for all locally shared memory objects, while remote objects are handled by the corresponding remote object agent. Among each others, these agents communicate via Java RMI.

Aleph The *Aleph* toolkit [Her99] offers a collection of Java packages that use remote threads to extend thread parallelism and to help with the construction of distributed shared objects. The toolkit supports push and pull communication, as well as object migration and remote method invocations.

Herlihy and Warres [HW99] described three different distributed directory services for the *Aleph* toolkit. A system that uses the *Aleph* toolkit can use one of them to keep track of moving objects and their cached copies. The first directory implementation uses a home-based protocol, the second uses the arrow directory protocol [DH98], and the third is a hybrid approach of the previous two, which uses forwarding proxies.

In the home-based protocol, each object is associated with a static home node that is responsible for this object. The static home node keeps track of the object location and of the location of all cached copies of the object. It also manages all accesses to the object, and it is not possible to reach the object other than by invoking its static home node. This scheme allows either only one object copy that is accessible by a single writer, or multiple object copies, which are accessible by multiple readers, but read-only.

The arrow protocol creates for each object a binary tree of all nodes. The protocol uses one-hop pointers, the *arrows*, that point to the direct neighbor in the tree in whose direction the object currently resides. Thus, all nodes must have one such *arrow* for every object that exists in the system. This is necessary even if the node never accesses this object during its lifetime and thus, makes the approach unsuitable for the envisioned system.

In this protocol, each remote object access follows the arrows through the network and the protocol implicitly assumes that the object migrates with each access. Thus, each node that forwards the request changes the direction of the arrow into the direction of the requesting node, i. e. towards the node from where the request came. This protocol assumes that the access always succeeds, and that the accessed object always migrates to the requesting node.

The hybrid protocol uses a dynamic home node for each object. When the object migrates to another node, the old home node stores a forwarding proxy to the new home node, which is used to forward all subsequent request messages. Because each access implicitly migrates the object, the forwarding proxy changes with each access and points after the access to the requesting node as new home node. This, again, assumes that an object access always migrates the object.

Thor Liskov, Day, and Shrira [LDS93] described *Thor*, a distributed object-oriented database system. Thor is based on a client-server model, where the servers store objects that are accessed by client application. The application scenario of Thor are large, long running systems where objects might be persistent for years.

Day et al. [Day+93] discussed references to remote mobile objects in the Thor system. Thor's distributed database stores objects on highly available servers, the so-called *object repositories (OR)*. Each object is stored in one OR and object accesses take place in atomic transactions to keep the data consistent. Furthermore, each object can migrate from one OR to another. To exploit data locality, Thor tries to avoid references that cross OR boundaries and thus, tries to place objects which reference each other in the same OR. For high-availability, each OR, together with copies of all its objects, is replicated onto different servers.

Day et al. described two types of object references: first, *location independent* references that do not change when the object migrates, and secondly *location dependent* references that do change when the object migrates.

In the location independent approach, each object is assigned with two references: A location independent reference that does not change during the lifetime of the object, and a local reference

that is only valid on the current home node (OR) of the object. The OR that created the object assigns the location independent reference, and is also the static home node that is responsible for the objects location. Upon an object migration, the new, dynamic home node of the object assigns a local reference that points to the object in the local memory, while the location independent reference is unchanged. Afterwards, the dynamic home node informs the static home node about the location change. Thus, the static home node is the central entity where all other nodes have to retrieve the object location before an access.

The location dependent approach does not use static home nodes and location dependent references only, which are tuples of `<OR ID, local memory address>`. Instead to inform a static home node, each migration leaves a forwarding proxy behind. Furthermore, each OR holds a so-called *inlist* that stores for each local object which other remote ORs reference this object. This *inlist* is used for garbage collection, but also during the object migration: Upon an object migration, the migrating object leaves a forwarding proxy at the old OR *A*. When the object arrives on its new OR *B*, it gets a new, local reference that points into the local memory of OR *B*, i. e. the local memory address. Afterwards, the OR *B* sends the reference `<B, local memory address at B>` back to *A*. OR *A* stores this tuple in the forwarding proxy, which is also a map that translates between the old and the new location dependent reference, i. e. from `<A, local memory address at A>` to `<B, local memory address at B>`. Additionally, OR *A* uses its *inlist* of the migrated object to send update messages, which contain these two references, to all referencing ORs. The referencing ORs store these two references until the next garbage collection run takes place. The next time the garbage collector runs, it takes this mapping and checks all objects for outdated location dependent references, i. e. `<A, local memory address at A>`. Whenever the GC finds this reference in an object, it replaces the old reference with `<B, local memory address at B>`. Thus, the location update depends on the frequency of the garbage collector.

This approach is similar to the incoming reference approach discussed in Chapter 8. However, the incoming reference approach in this thesis does not need to wait for the garbage collector but updates all referencing objects immediately. Furthermore, Thor does not decouple the object reference from the object location. Thus, the garbage collector has to check all references in all objects if they contain an outdated location dependent reference that must be updated.

Additionally, Day et al. did not discuss the need to update outgoing references as well to keep the incoming references consistent, cf. Chapter 8. Moreover, their discussion of these two approaches is limited to theoretical considerations about memory requirements and number of query messages. The authors neither consider the increased access latencies, the message overhead needed to update all referencing nodes, or the complexity of the protocol.

3.5 Distributed Operating Systems

Distributed shared memory systems offer the user the illusion of a shared memory system, on top of a distributed memory architecture. *Distributed operating systems* take this approach a step further and have the goal to provide the illusion of a *single system image* on top of a distributed system. Thus, a distributed OS does not only offer a shared memory abstraction, but also offers

the user a global and uniform view on all available resources such as programs and peripheral hardware. Furthermore, they offer transparent *process migration* throughout the system.

Plurix OS The *Plurix OS* [Göc+04] is a distributed operating system that aims at PC clusters. Plurix is written in Java and its main component is a page-based DSM system that creates a global address space among all nodes. A software transactional memory system deals with data consistency among nodes.

The communication between different cluster nodes is done with shared objects that reside in the DSM, and which include data and code. To locate shared objects, the DSM uses broadcast messages and is supported by the *memory management unit* (MMU) hardware: Whenever an object (and thus the page in that the object is stored) does not reside on the local node, the MMU detects a page fault that causes the DSM to broadcast the address of the missing page to all other nodes.

DEMOS/MP *DEMOS/MP* [MPP87] is a distributed operating system that uses message passing to communicate between different processors. IN DEMOS/MP, the distributed processes and kernel modules communicate with each other via *links*, which are the only way to access the services and resources of a process. Therefore, a link is the global address of a process, similar to an *object references*, that provides access to the process and to the resources of the process, such as its memory area.

Similar to object references, links can be created, duplicated, deleted and passed to other processes. The link address consists of three parts: the ID of the processor that created the process, the local process ID on that processor, and the last known location of the process.

Powell and Miller [PM83] described different approaches to support process migration in DEMOS/MP and present different mechanisms how to deal with outdated links. Their approaches are a centralized manager, a system wide name service such as a distributed hash table, or proxy forwarding. With proxy forwarding, the proxies not only forward the access message but additionally return a location update message back to the sender. This approach is necessary because DEMOS/MP also supports systems and processes that only support one-sided communication (no response or acknowledgment message).

Amoeba *Amoeba OS* [Tan+91] is a distributed operating system that is based on a *microkernel* design that hides the underlying, heterogeneous PC cluster.

The microkernel runs on each node in the network and is responsible for the management and scheduling of the different processes. Each process can execute multiple threads in the user space, but Amoeba also supports multithreading in the kernel space, but no thread or process migration.

Amoeba uses *remote procedure calls* (*RPCs*) for the communication between distributed threads. Therefore, threads are addressed by random 48 bit addresses, called *ports*. To locate or access a remote thread the first time, the sender node sends a broadcast message that contains the port of the accessed thread. The remote port responds to this broadcast and the sender caches the static home node address (e. g. the IP address) of the remote port for subsequent calls.

Tanenbaum et al. stated that the Amoeba system is suitable for clusters of a few hundred or maybe thousand nodes. But due to the usage of broadcast messages, it is not scalable for systems with much more nodes, e. g. a couple of hundreds or thousands of nodes, or even more.

3.6 Distributed Java Virtual Machines

In contrast to an operating system, which allows the execution of multiple programs and processes, a Java virtual machine (JVM), in general, only executes a single multi-threaded application. However, an advantage of Java and the JVM approach is the implicit support of heterogeneous hardware architectures: a Java application can be executed on any hardware, as long as a JVM for this architecture exists.

DJVM

Similar to a distributed OS, the main feature of a *distributed Java virtual machine* (*DJVM*) is to provide an SSI that offers the user a unified view of the system. Thereby, the user has a transparent view onto the resources in the system, without the need to know where the different resources are physically located.

The following section describes multiple distributed Java virtual machines, which are closely related to the distributed runtime environment that we develop in our group.

cJVM *cJVM* [AFT99] is a distributed Java virtual machine for homogeneous clusters that implements a *distributed heap*.

cJVM does neither support thread nor object migration, but uses remote method invocations to access remote objects. Furthermore, because objects are referenced locally by regular Java references, cJVM implements a *master-proxy* approach to access remote objects. The master object resides on the node where the object was created (the static home node); while the proxies resides on other remote nodes. This is necessary because local references are only valid in the local address space. Thus, the local access invokes the proxy, which is responsible for the communication with the master.

The first time, a reference to a local object is passed to another node, for example, as an argument of a remote operation, it is assigned with a unique global identifier, the global address of the object (GAO). Additionally, the reference contains the global address of the class (GAC). Thus, the remote node can use the tuple (GAO, GAC) to create the local proxy, which is used to access the object on its static home node.

Aridor et al. [Ari+00] proposed two optimizations for the object placement in a cJVM environment. The first optimization implements a factory method that allows the creation of an object on the node where it will be used. The second optimization allows the migration of an object, but only if this object is used by a single thread. For example, an object Y can be migrated from node A to node B if the only user of Y is a thread on node B.

JESSICA The *JESSICA* system [MWL00b; MWL00a] is an ongoing research project at the University of Hong Kong. Its DJVM runs on top of a standard UNIX operating system and offers an SSI over a heterogeneous computing cluster.

JESSICA spans a logical *global thread space* across all nodes in the cluster, which allows threads to freely move from one node to another. Unlike cJVM, JESSICA focuses on thread migration for dynamic load balancing.

The *global object space (GOS)*, which is a sub space of the global thread space, contains the globally accessible objects. To cache remote objects and to keep these cached copies in a coherent state, JESSICA relies on the cache coherence protocol of the underlying DSM system.

Initially, the GOS was implemented on top of the TreadMarks page-based DSM, which was later replaced by JUMP [CWH99]. JUMP is another page-based DSM, which allows dynamic home nodes of a memory page. In JUMP, the home node migration takes place whenever a remote node modifies a cached copy of a memory page. At this moment, the modifying node becomes the new home node of the memory page.

To prevent a node from reading an outdated page from a previous home node, JUMP broadcasts *migration notice* messages at synchronization points, and if the new home node modifies several pages, all *migration notices* are consolidated into one message.

Fang et al. [FWL02; FWL03] developed a fine-grain, object-based *global object space* for JESSICA that includes a simple object migration. Therefore, the GOS distinguishes between *node-local* objects and *distributed-shared* objects (DSO). To detect a DSO, each node examines the communication between itself and other nodes to detect object references that cross the node boundary.

The simple object migration only allows objects to migrate if there is one single writer thread at a time. In this case, the object migrates to the home node of the writer thread and leaves a forwarding proxy at its old location. If a third thread tries to access the migrated object on its previous home node, the proxy sends a location update message back to the requesting thread. Afterwards the requesting thread updates its location information about the object and sends its request to the correct new home node. However, it remains unclear if or how proxies are deleted from the system.

JESSICA2 [ZWL02; Zhu+04; Zhu05] adds a transparent Java thread migration to JESSICA. To achieve this, JESSICA2 employs Just-in-Time (JIT) recompilation that preserves the native thread execution mode and eliminates code instrumentation.

JESSICA3 [JES11] focuses on the applications that the VM executes. The main objectives are to overcome memory space limitations and to solve the problem of global thread scheduling.

JESSICA4 [JES11] aims at new parallel programming paradigms, e. g. the partitioned global address space (PGAS) programming model and transaction-based synchronization with two-way elastic atomic blocks, called TWEAK.

I did not find further information about JESSICA3 and JESSICA4 other than the project websites.

Kaffemik *Kaffemik* [And+01] is a DJVM that is based on the Kaffe VM [Kaf11] and aims at PC clusters. Kaffemik offers an SSI on top of a hardware supported, page-based DSM that exploits a memory mapped network interface that is based on the IEEE *Scalable Coherent Interface* (*SCI*) standard [IEE93].

Kaffemik dynamically distributes and manages threads at runtime, but the programmer can overwrite this automatic distribution with user defined policies. However, Kaffemik does not support object migration between nodes, only their remote creation. Instead, it relies on the fast remote memory access, and with this, on the underlying DSM and the fast interconnect between the nodes.

Hyperion *Hyperion* [MMH98; Ant+01] is a DJVM that is built on top of an object-base DSM. Hyperion uses a JIT approach that first compiles Java to C code. During this step, a customized Java-to-C compiler performs optimizations such as creating a local copy whenever a reference to a remote object is de-referenced in a loop. Thus, the whole placement of threads and objects must be predetermined at compile time. Afterwards, just before the execution, the generated C code is compiled to machine code.

Each node holds a centralized object address table that allows the access to the whole DSM. In this DSM address table, each node owns only a statically assigned portion of the address space, which is used to create local objects.

Hyperion does not allow object migrations. Instead, the node that created an object holds the master object at all times. If a remote node accesses the master object, the static home node creates a copy of the object and sends it to the remote node. If the remote node modifies the object copy, it must write these changes at Java synchronization points back to the static home node.

JavaSplit *JavaSplit* [FSS03; FSS04] is a DJVM that uses Java sockets to enable IP-based communication. It administers a pool of worker nodes that can be connected by a standard IP network.

JavaSplit uses an object-based DSM for shared objects that is similar to Hyperion. If the system detects that an object is used by more than one thread, it assigns a globally unique ID and registers it in the DSM.

Similar to Hyperion, JavaSplit does not allow object migrations and assigns each object a static home node that is responsible to manage the object access.

MagnetOS *MagnetOS* [Liu+05] provides a DJVM for distributed ad-hoc sensor networks that offers an SSI.

A MagnetOS application is composed of software modules, called event handlers, that are encapsulated as separate Java objects. These objects communicate with each other via messages, the so-called events. MagnetOS groups the different objects and migrates them onto the nodes to achieve an energy-efficient placement.

A static partitioning service is responsible for this partition. It also rewrites the communication among the components at the bytecode level. For example, a method invocation is replaced with a remote procedure call.

An object migration leaves a proxy at the former home node, and each object access updates the cached object location. Thus, MagnetOS avoids the traversal of the proxy chain for subsequent accesses. However, it is unclear if proxies are deleted at some point.

MagnetOS uses AODV [PR99] for routing in the ad hoc network. If a link to an object is broken or lost, the system falls back to a broadcast message to search for the object.

CellVM The *CellVM* [NGF08] is a DJVM that was specially designed to run on the IBM Cell processor (see Section 4.3.1). It allows the distributed execution of multi-threaded Java applications on the various cores of the Cell processor.

The VM comes in two flavors: the ShellVM, which is executed on the Power PC core (power processing element (PPE)), and the CoreVM, which runs on the synergistic processing elements (SPEs).

The ShellVM maintains the global system resources, while the CoreVM operates on its own local storage. Moreover, each thread that is executed on the CoreVM is pinned to a single SPE and cannot migrate at runtime.

The Java heap, which is shared by all VMs, is located in the main memory. Thus, the CellVM does not have to deal with migrating objects. To access this heap, the CoreVMs need to perform a DMA transfer because this is the only way to move data into the local memory of the SPEs. Furthermore, the CoreVMs do not execute the whole set of Java bytecode because not all of the operations can be implemented and executed efficiently on the SPEs. For example, for complex memory operations, the execution is transferred from the SPE to the PPE. The creation of new objects and the execution of native methods is handled by the ShellVM on the PPE as well.

Hera-JVM Another JVM for the IBM Cell processor is the *Hera-JVM* [MS10]. It is a JVM for heterogeneous multi-core processors that hides the heterogeneous nature of the Cell processor and offers an SSI. The Hera-JVM is based on the JikesRVM [Alp+05] and thus, similar to the JikesRVM, does not interpret the Java bytecode but all methods are Just-In-Time (JIT) compiled.

The Hera-JVM allows the execution of an unmodified Java application and supports thread migration among the PPE and SPEs that is transparent to the application. This thread migration might be performed whenever another method is invoked. Hence, Hera-JVM is able to decide for each method on which core (PPE or SPE) the method should or must be executed. When a thread migrates, the bytecode of the method can be JIT compiled to one of the two different instruction sets, either the one of the PPE or the one of the SPE. This feature is mandatory to migrate a thread if the invoked method contains operations that are not executable on the current core (namely the SPE).

Similar to the CellVM, the shared heap of the Hera-JVM resides in the main memory and thus, the Hera-JVM does not need to deal with object migrations, as well.

Commercial Solutions The *Terracotta system* [Ter11a; Ter11b] is a commercial solution that allows a Java application to run on multiple DJVM instances. The goal is to scale Java enterprise applications in large business systems.

The DJVM instances run on multiple machines that are connected via a network-attached global heap. This Java heap consists of an underlying server array, to which all DJVM instances connect. Hot standby servers provide fault tolerance and take over when an active server fails.

With Terracotta, the user needs to identify and tag all Java objects in the source code that should be reachable while they reside in the network-attached heap. The bytecode of the tagged

classes is instrumented by Terracotta to allow object maintenance on the global heap [BK07]. However, Terracotta does not allow objects and threads to migrate between the machines in the cluster.

Azul Systems developed the *Zing* platform and the corresponding Zing Java virtual machine. Similar to Terracotta, the Zing platform is a commercial product that aims at better scaling Java business applications in large business systems.

The Zing JVM on the host server is only a virtualization proxy that pushes the Java stack and thus the application to the *Zing Virtual Appliance (ZVA)*. Azul states that the ZVA is a better execution stack to execute the Java application because the Zing resource controller (ZRC), a centralized management component, dynamically growths or shrinks the memory footprint of a Java application on demand.

As the Zing VM is a commercial product, little is known about the VM internals. Azul did not publish any scientific papers, but some information about the Zing VM can be found on their website [Azu11].

4 Environment

The envisioned environments of this thesis range from clusters of heterogeneous many-core machines to distributed, embedded systems. Therefore, the group I am part of develops a distributed runtime system that shall allow applications to run on both these hardware platforms and offer the user a single system image. The goal is to implement a system that provides an easy means for software engineers to implement and deploy software in such highly complex and dynamic systems, where nodes may be added or removed at run-time.

The following chapter describes the projects in which I worked during my PhD thesis, and in which I conducted my research. This chapter outlines the main properties of the projects' envisioned environments, where the two target scenarios of these projects are: A runtime environment for *Ambient Intelligence* (*AI*) applications in the *AmbiComp* project, and a runtime environment for *High Performance Computing* (HPC) in the *J-Cell* project.

4.1 The AmbiComp System

Ambient Intelligence (AI) pursues the vision that small networked computers will jointly perform tasks that create the illusion of an intelligent environment. Today, these small computers are part of everyday's life. They are contained in devices such as toasters and refrigerators, which are equipped with an increasing amount of computational power. As most of the usual tasks of these devices hardly exhaust this computational power, there is room to implement new, more sophisticated applications. This idea becomes even more interesting if all these devices are equipped with some kind of communication interface which allows to exchange data and trigger actions on remote nodes. The result is a distributed network of small, embedded devices that communicate with each other. Such networks belong to the field of ubiquitous computing and ambient intelligence, which received new attention recently, this time under the name of *cyber-physical* systems [Sha+08; Raj+10].

The research project *AmbiComp* [Eic+08] was a joint project between Universität Karlsruhe, Technische Universität München, Beecon GmbH, Hochschule der Medien (HdM), IESE Fraunhofer Institut, and Alcatel-Lucent Deutschland. It was funded from 2006 to 2009 by the German Ministry for Education and Research (BMBF). The project aimed at interconnecting everyday devices in an ad-hoc manner. Although the idea originated in the field of "digital" or "intelligent" homes, it extends to a much broader view of distributed computing in which the devices will share data and distribute their workload among each other.

The AmbiComp project was especially concerned with simplifying the software development process for a network of smart products. It aimed at providing an easy means for software engineers to develop and deploy Ambient Intelligence software in such networks. During the

runtime of the project the partners developed an AmbiComp eco-system that covered all aspects of this development process. Namely, these are:

- **Hardware**: The *ambient intelligence control unit* (*AICU*) can serve as a basic building block for smart products. An AICU consists of one or more so-called *sandwich modules* (*SMs*). SMs equip an AICU with compute power, communication interfaces, and input/output (IO) lines for analog and digital signals. At least one of the SMs in an AICU has to execute the *AmbiComp Virtual Machine* (*ACVM*).

- **Hardware Abstraction Layer**: The so-called *AmbiComp BIOS* is a small, low-level *hardware abstraction layer* (*HAL*) that offers the AmbiComp runtime environment access to the hardware resources.

- **Runtime Environment**: The *AmbiComp virtual machine* (ACVM) was specially designed for the resource-limited microcontrollers of the SMs. The ACVM is able to execute several single- or multi-threaded Java applications in parallel. Together, all ACVM instances in the AmbiComp system are able to create a single system image across the physically separated AICUs.

- **Application Programming Interface**: The *AmbiComp API (application programming interface)* is a small system library that offers a set of native Java methods. These methods offer direct access to the underlying hardware. Additionally, the AmbiComp project supports different flavors of Java APIs that offer a minimal sub-set of Java classes.

- **Integrated Development Environment**: The *AmbiComp Eclipse Plug-in* provides the developer with an easy-to-use development environment. It is realized as a plug-in for the open source framework *Eclipse* [The11].

The following sections give a brief overview over the hardware and describe the BIOS, the tool chain, and the ACVM in more detail. A detailed description of the hardware, as well as some information about the other components are given in Eickhold et al. [Eic+08] or on the project website. [1]

4.1.1 Hardware

AICU

The *ambient intelligence control unit* (AICU) is the basic building block of an AmbiComp system. A developer can use an AICU to control smart products in an ambient intelligence environment. Such environments may have vastly differing properties, and smart products may be used for a wide range of applications. The AmbiComp project takes this into account with a modular approach: AICUs consist of stackable heterogeneous *sandwich modules* (*SMs*).

SM

An inter-SM communication channel enables and supports the distributed nature of the ACVM. It allows all ACVMs in an AICU to directly access the memory on other SMs. In this manner, the whole memory of an AICU forms one uniform heap that is available to all ACVMs. This heap is

[1] www.ambicomp.org

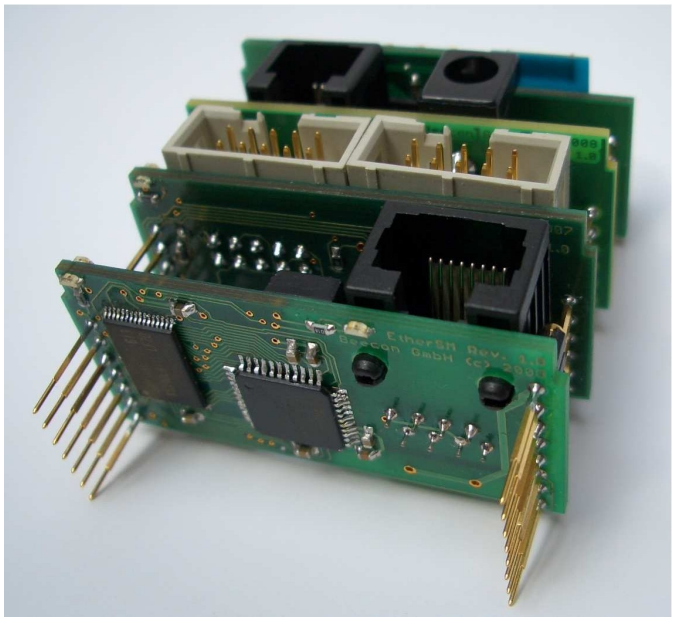


Figure 4.1: An exemplary AICU stack that consists of multiple SMs. Each SM offers a different functionality and at least one of the SMs must be equipped with a microcontroller.

used to exchange shared objects, or to allow the execution of applications that have a need for memory bigger than a single SM can offer.

Figure 4.1 shows an exemplary AICU stack, and Figure 4.2 shows all SMs of the AmbiComp project.

4.1.2 AmbiComp BIOS

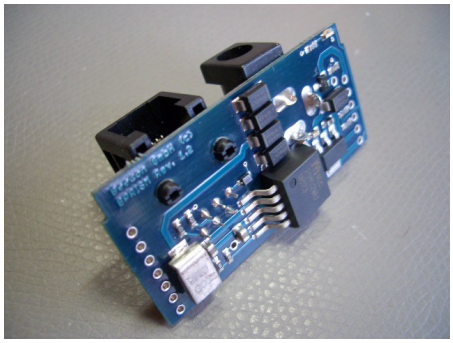
The *AmbiComp BIOS* is a small, low-level *hardware abstraction layer*. It provides generic interfaces to the hardware of the different SMs. This generic interface allows the ACVM to be independent of the underlying hardware. As a consequence, each SM must have its specially tailored BIOS that maps the functionality of the respective SM to this generic interface.

The BIOS has to meet a number of requirements to fully allow this generic hardware abstraction:

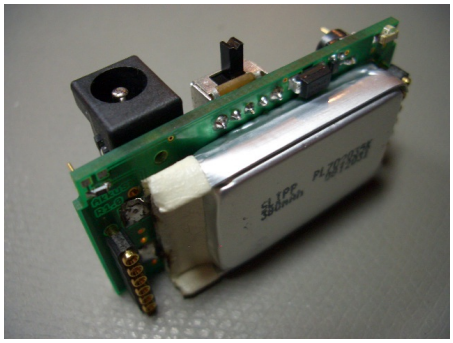
- Initialize the hardware, e. g. set IO lines into a defined state.
- Handle interrupts.
- Implement the hardware dependent parts of drivers.
- Offer a special maintenance interface, e. g. to update the firmware.

Additionally, the BIOS provides remote access to all hardware resources that are available in an AICU. This access is provided by the *fast memory access* (*FMA*) interface. FMA started as a

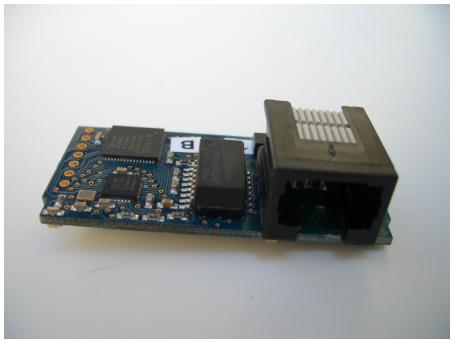
FMA



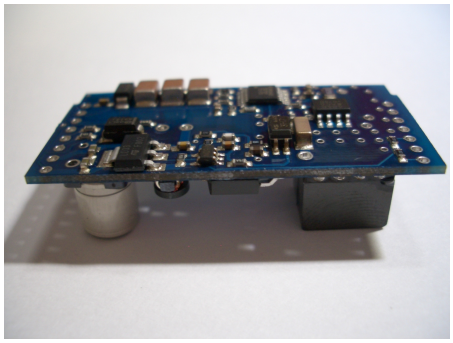
(a) Backplane Primary Supply SM.



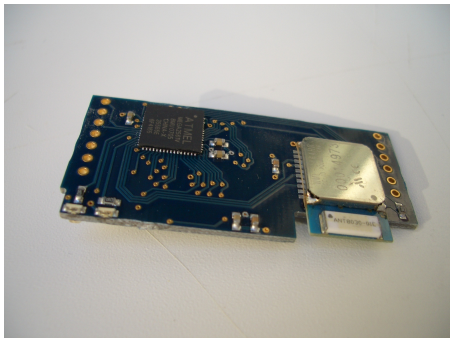
(b) Accu SM.



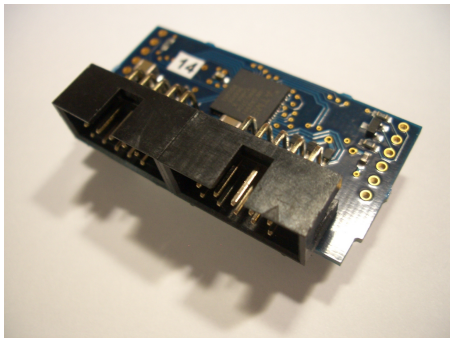
(c) Ethernet SM.



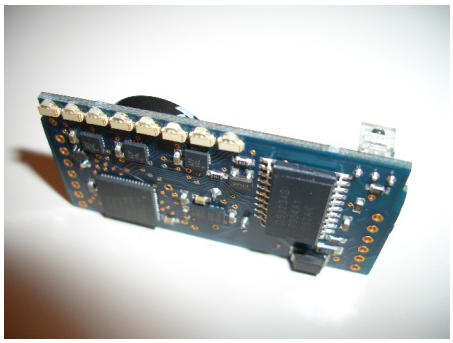
(d) Power-over-Ethernet SM.



(e) Bluetooth SM, HCI and SPP.



(f) Input-Output SM.



(g) Peri SM.

(h) E-Puck.

Figure 4.2: The AmbiComp SMs that have been developed in the AmbiComp project.

method for the remote inter-SM memory access and was later enhanced to access all hardware resources of an SM. The FMA operations access the memory on the local or on remote nodes, without interrupting the ACVM and without the need for an asynchronous request response protocol.

FMA is the foundation of the *object distribution model* (ODM), which is described in more detail in Section 4.2.

4.1.3 AmbiComp Transcoder and Tool Chain

To develop a program for an AmbiComp device, the programmer writes a common Java program and uses the AmbiComp specific user and system libraries (AmbiComp API). Afterwards, the tool chain for the AmbiComp project starts with a regular *javac* compiler that compiles the Java program into one or more *.class* files. Afterwards, the off-line *AmbiComp transcoder* takes the bytecode from the user provided *.class* files (user code and used libraries), transcodes it for the ACVMs instruction set (see below), and links it statically to the AmbiComp API. The result of this transcoding process is a so-called *Binary Large Object (BLOB)* file that contains the entire code that is needed to run the application on the target platform's AmbiComp Virtual Machine, cf. Figure 4.3. This is similar to Sun's Squawk VM [SSB03], which uses so-called *suite files*, or the DalvikVM [Dal08] for Android platforms, which uses so-called *DEX files*.

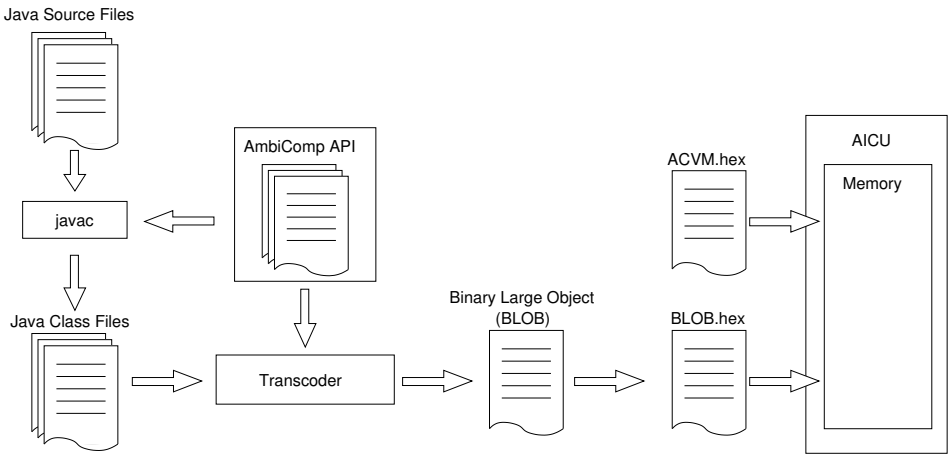


Figure 4.3: The AmbiComp tool chain. The tool chain starts with a common Java program which is compiled with a regular *javac* compiler. Afterwards, the AmbiComp transcoder takes the bytecode and transcodes it into a *Binary Large Object (BLOB)*, which is needed to run the application on the ACVM.

Because only a small part of the linked libraries will be actually used, the transcoder can eliminate a large portion of the library code. In some cases, such as e. g. getter/setter or wrapper methods, the transcoder can save code by inlining (parts of) the respective method. Furthermore, because the transcoder links the code statically, it can eliminate clear text references to class names, field names, and method names. These plaintext identifiers, e. g. `java.lang.Object`, are

contained in common Java *.class* files in the so-called *constant pool* that allows dynamic binding ("class loading"). The transcoder replaces all these names with numerical identifiers that are unique only within a BLOB.

The advantage of a reduced code size pays off in many AmbiComp scenarios. However, the execution of multiple concurrent applications benefits from dynamic binding. For this reason, the AmbiComp tool chain supports dynamic BLOB binding, too. It allows the transcoder to transcode one or more individual classes into a BLOB to create a *shared library*. This shared library reduces the code size, because common functionality can be collected in individual libraries, similar to shared libraries in the UNIX system [Arn86]. Afterwards, the library BLOB is loaded only once into an AICU and is usable by all applications.

To transcode a BLOB (library or application) that is bound against one or more library BLOBs, these library BLOBs must be given to the transcoder during transcode time to generate a list of all required BLOB files, which is stored at the head of the new BLOB. The ACVM needs this information to check if it possesses the needed BLOBs or knows how to retrieve them.

Within the AmbiComp project, this tool chain is fully integrated into the *Eclipse* software development environment, so that a programmer can develop and then directly deploy the code onto the target platform. When the BLOB is deployed and fully loaded, the ACVM starts its execution.

4.1.4 AmbiComp Virtual Machine

From its beginning, the Java programming language has been associated with the idea of cross-platform programming of embedded devices [Gos95]. Especially, the *Java Platform, Micro Edition* (Java ME) [Mic11] sets the focus on VMs that shall run on mobile devices like cell phones, PDAs etc. With the same idea in mind, the starting point of the AmbiComp project was a Java runtime environment with a small footprint, a customized Java virtual machine, called *AmbiComp virtual machine* (ACVM).

ACVM

This ACVM was specially designed for the resource-limited microcontrollers of the intelligent SMs, such as the AVR 8 bit microcontroller family. It does not require an operating system but runs on bare metal. Thus, all needed operating system functions such as scheduling or memory management are entirely up to the ACVM. Therefore, the ACVM interacts with the underlying hardware via the BIOS interface. This allows an easy ACVM development that does not have to be concerned with any specific implementation details and requirements of the various SMs.

The instruction set architecture (ISA) of the ACVM is particularly designed to less interpretation overhead and to enable a very small footprint of the ACVM binary itself, while supporting almost all SUN Java opcodes and a substantial part of the Java ME API. Thus, the ACVM can only execute BLOBs that are generated by the transcoder.

The ACVM does not allow external users to provide native code directly, but the AmbiComp API offers several low level functions such as direct access to input and output pins of the microcontroller. These low level functions are native ACVM methods that wrap the specific BIOS interface methods. Thereby, the ACVM – together with the system libraries – provides much of the functionality that is normally provided by the operating system.

4.2 Object Distribution Model

The *object distribution model* (*ODM*) describes a model for the interaction of distributed components such as different AmbiComp compliant devices from various hardware vendors. Its main goal is the description of the mechanisms for synchronous and asynchronous distributed memory management. In particular, it describes the memory hierarchy, the sharing of memory, and the remote access to distributed memory. ODM

For historical reasons, I use the abbreviation ODM for both, the AmbiComp *object distribution model* and the AmbiComp *object distribution management*. The object distribution model is the specification, while the object distribution management is a concrete implementation of the model. There exist two such prototypical implementations. One implementation was added to the AmbiComp BIOS and the ACVM. Its main goal is to provide methods for the distributed object access among remote instances of the ACVM in the same AICU. The other implementation is a middleware layer [Pep10] that allows remote data access among heterogeneous platforms and between different programming language domains. The main goal of the middleware is to support the interoperability of the ACVM with other, native non-ACVM applications, which are e. g. implemented in C. The ODM management layer allows, for example, a native C application to access remote Java objects that have been created by an ACVM. Such a mixed-language approach is especially advantageous for time-critical operations, for which the processing in Java would be too slow.

In accordance to the memory model, cf. Chapter 2, the *object distribution model* distinguishes between logically (private and global) memory, and physically (private, local or remote) memory.

A local ODM instance allocates memory from the local physical memory, or, if the node cannot provide enough memory, it requests the memory allocation from the physically remote memory on another node. The object distribution model itself does not make further assumptions about the layout of memory chunks. It is completely up to the system implementer how the memory is structured, e. g. how the ACVM organizes the field layout of a Java object within a memory chunk.

Furthermore, it is up to the system how to implement the object distribution model. For example, each node could offer a specific ODM interface through which all object accesses must pass, similar to the ODM middleware [Pep10], which uses a so-called *ODMdriver*. This ODMdriver is responsible for all remote access requests from remote nodes to the locally stored shared data. Thus, all remote entities send their requests to the ODMdriver, which has complete knowledge about the locally stored objects and can answer the requests with the correct result. On the contrary, the ACVM itself implements the object distribution model and thus, hides the local and remote memory access from the user.

ODM has its foundation in the *fast memory access* (*FMA*) offered by the AmbiComp BIOS, which allows a fast, direct access to the complete memory of all AmbiComp SMs within the same AICU.

The main goal of FMA is the atomic, synchronous read and write access to 32 bit chunks of memory. In this way, FMA is the means to access data within an AICU only.

The main reason that FMA transfers 32 bit per FMA access is the 32 bit granularity of Java fields, because the Java specification requests that the access to 32 bit object fields is atomic.

Thus, the FMA access to 32 bit chunks of memory must be atomic as well. Therefore, the ODM model introduces an additional Test-and-Set operation that prevents one FMA access from being interrupted by other cores that might access the same memory location.

If FMA transferred larger blocks, it might block other SMs that want to perform remote FMA accesses. Moreover, it would also block the accessing application, because, unlike a direct memory access (DMA), the FMA transfer is synchronous.

Next to the atomic transfer of 32 bits of memory, ODM defines an asynchronous memory access that is used for bulk transfers of chunks with a size that is a multiple of 32 bit, e. g. to transfer a whole object from one node to another. In the AmbiComp context, this bulk transfer should only be used if a copy operation cannot be avoided, in all other cases, the 32 bit FMA access should be sufficient.

4.3 Multi-Core and Many-Core Systems

In this part of the thesis, I will give an overview over the J-Cell project. J-Cell is a joint project of Technische Universität München, Universität Freiburg, and the BioSolve IT GmbH. It is funded from 2009 to 2012 by the German Ministry for Education and Research (BMBF). As such, J-Cell is the follow-up project of AmbiComp and, similar to AmbiComp, has the goal to offer an SSI that allows applications to run transparently on heterogeneous hardware. But unlike AmbiComp, J-Cell targets heterogeneous multi-core machines.

This section starts with the description of the environment and motivates the usage of a single system image on top of today's and future machines for high-performance computing. Furthermore, it outlines the hardware architecture of the two multi-core processors that the project aimed at with its prototypical runtime environment. This section shows exemplarily where the difficulties are for a software developer when facing the challenge to develop an application that should not be limited to a single processor or machine architecture. Namely, the described Cell processor from IBM and the SCC processor from Intel have a completely different memory architecture, which the software developer has to take into account during the software design.

Since the advent of electronic computing, the processors' clock speed has risen tremendously. Now that energy efficiency requirements have stopped that trend, the number of processing cores per machine started to rise. This development has led to *multi-core* systems, which have 2, 4, 8 or more cores, and *many-core* systems, which have 128, 256, 512 or more cores, where Asanovic et al. [Asa+06] even expect about 1000 cores on a single die.

multi-core

many-core

The individual cores become more specialized, and their inter-connections form complex networks, both on-chip and beyond. Thus, these systems do not follow a symmetric multiprocessing (SMP) architecture anymore, where multiple, identical cores execute multiple threads or processes, and share the same resources of the system, such as the same main memory. Instead, they form *asymmetric multiprocessing* (*ASMP*) systems, where each core or processor might offer a completely different set of functionality.

A cluster that is enhanced with GPUs (*graphics processing unit*) is one example of such an asymmetric architecture. Such heterogeneous clusters consist of general purpose CPUs, which are enhanced with a GPU co-processor. In contrast to general purpose CPUs, which are optimized

for low latency, GPUs are optimized for high throughput [LH07]. Owens et al. [Owe+07] gave a survey on research projects and applications that map general purpose computation onto GPUs, called *GPGPU* (*general-purpose computation on GPUs*).

Göddeke et al. [Göd+07] gave an overview of GPU-enhanced clusters that are used to scale *finite element method* (*FEM*) computations for e. g. weather forecast, frontal crash simulations, elasticity and structural analysis problems.

The increased compute power of these and other heterogeneous systems allows more detailed numerical computations and simulations [Owe+07], and falling costs enable even small companies to invest in multi-core systems and clusters.

However, the growing complexity might impede this growth when, for example, a software developer who is not used to parallel programming, has to write an application for a cluster of thousands of interconnected heterogeneous processor cores. Software developers would need a deep knowledge about the underlying infrastructure as well as of the data and communication dependencies in their applications to partition them optimally across the available cores. Moreover, a predetermined partitioning scheme cannot reflect failing processors or additionally provided resources.

Similar to AmbiComp, the J-Cell project has the goal to support the developer with the software development for such dynamic and heterogeneous high-performance computing systems. It facilitates systems that can dynamically include new resources into the computation or handle the failure of individual nodes. Thereby, it hides the heterogeneous and distributed nature of clusters of many-core processors from the software developer and has the goal to completely eliminating all centralized components.

4.3.1 IBM's Cell Processor

The starting point of the J-Cell project was the IBM *Cell processor*, also known as the *Cell Broadband Engine Architecture* [Kah+05; Pha+05]. The Cell processor is a heterogeneous, multi-core *system-on-chip* (*SoC*) that was developed by a consortium of Sony, Toshiba and IBM. The goal of the Cell processor development was to deliver massive floating-point processing for rich broadband multimedia applications and computation-intensive workloads.

The Cell processor consists of a general purpose, dual-threaded 64 bit PowerPC core, called *power processing element* (*PPE*), and eight co-processor cores, called *synergistic processing elements* (*SPEs*) [Fla+05], cf. Figure 4.4.

PPE

SPE

The PPE has a general-purpose Power architecture design and can thus execute a conventional operating system. It is the central processing unit and has complete control over the SPEs. It can, for example, start, stop and interrupt a whole SPE and the individual processes that are running on the SPEs. Additionally, the PPE has standard load and store operations that can directly access the local memory of the SPEs and the main memory. However, this access is not as efficient as a direct memory access (DMA).

The SPEs contain 256 KB embedded local SRAM memory for instruction and data. This SRAM memory does not work as a conventional cache, as it is not transparent to the software. Additionally, each SPE contains a register file with 128 entries, each 128 bit wide.

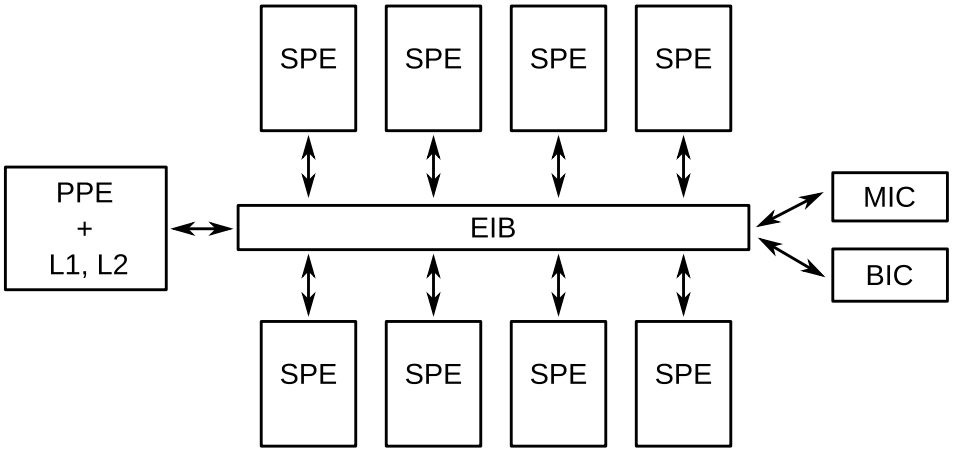


Figure 4.4: The high-level diagram of the Cell processor. It shows the *power processing element (PPE)* and the *synergistic processing elements (SPEs)* that are connected by the *element interconnect bus (EIB)*. The PPE is the central processing unit and has complete control over the SPEs.

The load and store operations of the SPEs can only access the own, local SRAM memory. All remote memory access (e. g. on other SPEs, the PPE or the main memory) requires the use of explicit, asynchronous DMA commands. These commands are handled by the DMA engine, which transfers the data to and from remote memory anywhere in the system.

All cores are equipped with a DMA engine that allows fully cache coherent DMA access via the *element interconnect bus* (*EIB*) (see description below). With this design, DMA is the central means for intra-chip data transfer, and the memory architecture is a non-uniform memory access (*NUMA*) architecture, in which the different access latencies depend on the type and location of memory. For more details about the communication network on the Cell processor see e. g. Kistler, Perrone, and Petrini [KPP06].

Additionally to the cores, a high-speed *memory interface controller* (*MIC*) offers access to the main memory. The *bus interface controller* (*BIC*) offers access to the configurable I/O interface. This on-chip I/O interface allows a dual processor configuration of two cell processors, which does not need an additional switch to connect the two chips [KPP06].

A specialized high-bandwidth circular data bus, the *element interconnect bus*, connects the PPE, SPEs, main memory and the I/O interface. The EIB consists of four rings, where two of the rings run clockwise while the other two run counterclockwise. Each of these rings can handle up to three concurrent data transfers, if their paths do not overlap.

4.3.2 Intel's Single-chip Cloud Computer

SCC

The *single-chip cloud computer* (*SCC*) is an experimental 48-core processor [Bar10; Int11a] that Intel Labs have created as a "concept vehicle" for many-core software research. The chip is organized in dual-core *tiles*, where each tile contains two X86 cores that are based on the P54C-Pentium-1 processor. Each tile contains a router that interconnects the tile to its neighbor tiles. Thus, all tiles form a 2D mesh network that is organized in an 6x4 array of tiles, see

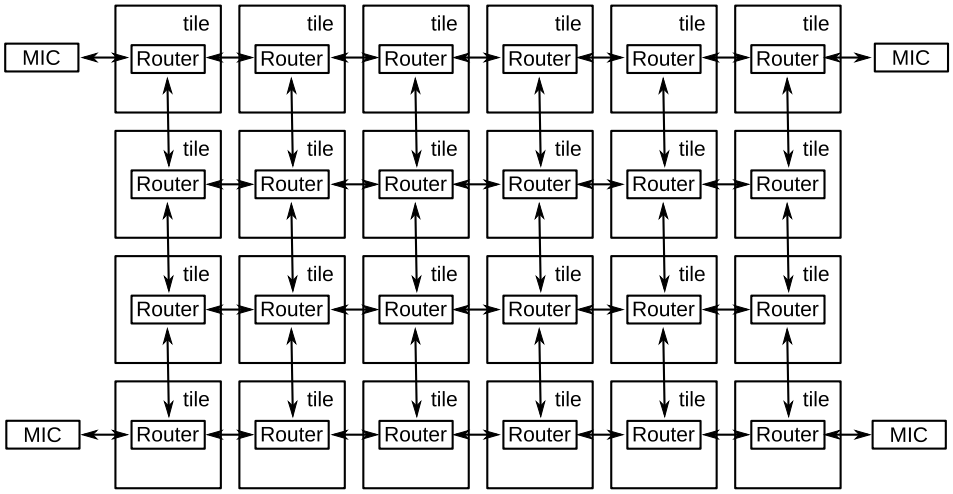


Figure 4.5: The high-level diagram of the SCC processor that shows the 6x4 array of tiles, where each tile contains two X86 cores. Additionally, the *memory interface controllers (MICs)* are shown, which connect the network to the external off-chip, but on-board, memory.

Figure 4.5. To allow on-chip message passing communication, each tile holds an additional message passing buffer (MPB), whose 16 KB SRAM memory is shared among all tiles. To provide external off-chip memory, four of the border routers are connected to a *memory interface controller (MIC)*. These MICs connect the network to the external off-chip, but on-board, *DDR3 DRAM* memory.

Because of this design, all on-chip communication, whether between different tiles or between a tile and a MIC, is done via *message passing*, where a simple *XY routing* delivers the messages to their destination.

Each MIC can address 16 GB with a 34 bit address and together, the four MICs can offer a total memory of 64 GB external memory. To address the full 64 GB external memory, the SCC uses a 46 bit address, the so-called *system address*. The usage of this memory is configurable and each core owns some of this memory as its *private memory*, while the rest of the memory is shared among all cores. To address this memory, each core uses its 32 bit *core address* that allows the addressing of 4 GB memory. Hence, it is necessary to translate between the system addresses and the core addresses to map parts of the external memory into the local address space of a core.

To perform this translation, each core holds a private *lookup table* (*LUT*) that contains 256 entries, each 22 bit wide. These entries allow to address the core's 4 GB memory space, which is divided into pages with a size of 16 MB (to match the 256 entries). The LUT is fully configurable at runtime and defines the partition of the available core-addressable memory into private and shared memory, i. e. the boundary between private and shared memory can be dynamically programmed. Thus, each LUT entry addresses either a page of private or shared memory, whereas the private memory is mapped in the memory of the closest MIC. Additionally, all message passing buffers and all configuration registers (one per tile) are addressable via the LUT as well.

The address translation from a core address into a system address is shown in Figure 4.6. The

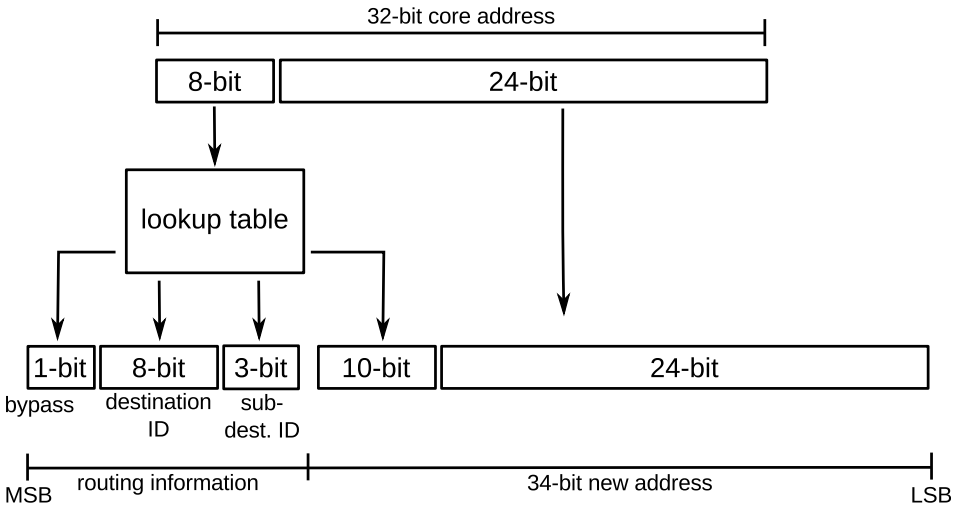


Figure 4.6: SCC core to system address translation, according to [Int11a]. The upper 8 bit of the input core address index one of the 256 entries in the lookup table, while the lower 24 bit are directly used as the lower bits of the system address.

upper 8 bit of the input core address index one of the 256 entries in the LUT, while the lower 24 bit are directly used as the lower bits of the system address. The upper 12 bit of the output system address provide the location information of the addressed memory, namely, the routing information where the mapped memory can be found, for example the XY coordinates of the corresponding MIC. The remaining 34 bit of the system address identify the actual memory address, e. g. a page in the external DRAM.

As with the Cell processor, this memory design is a NUMA memory architecture with varying memory access latencies, which depend on the type of memory and the hop distance between the accessing core and the remote memory location, see [Int11b, Table 1].

If the LUT entry targets a page of the private memory, this page is cached via the L2 and L1 caches of the accessing core. For shared memory, there is no built-in mechanism to guarantee cache coherency, but the SCC allows two types of shared memory: cacheable and non-cacheable memory.

The cacheable shared memory has a granularity of a 32 Byte cache line and cache coherency has to be implemented in software by the user. One advantage of this approach is that no hardware is needed to keep the shared memory consistent. Another advantage is that software coherency among multiple caches is dynamically reconfigurable. The disadvantage is that users have to know when to flush the cached memory and insert these flushes explicitly in their code. However, this design allows each application to define its own memory domain, in which the application can guarantee memory coherency.

The non-cacheable memory has a granularity of 1, 2, 4 or 8 Byte. A read operation to this memory bypasses both caches, and the read value is stored directly in the registers of the core.

Intel offers the *RCCE* (pronounced “rocky”) API to simplify the programming of the SCC. RCCE is a small, low-level API for message passing [MW11]. It offers a *basic* and a *gory* interface for memory management and an additional interface for power management.

The *basic RCCE interface* offers send and receive methods to pass messages between the cores. It is based on one-sided communication and the use of shared memory. When a programmer uses this API, RCCE handles all underlying details of the communication, e. g. the MPB management.

The *gory RCCE interface* allows a low-level access to the MPB, e. g. for memory allocation and flag management so that the developer has the complete control over all details of the MPB.

5 System Specification

This chapter describes the components of the envisioned distributed runtime system that are relevant for this thesis. It starts with a high level overview, and afterwards describes the individual parts in more detail.

Figure 5.1 shows the main modules of a single instance of the envisioned runtime environment. Each of these modules fulfills a specific task in the execution of a distributed application and especially in the local and remote access to migrating objects.

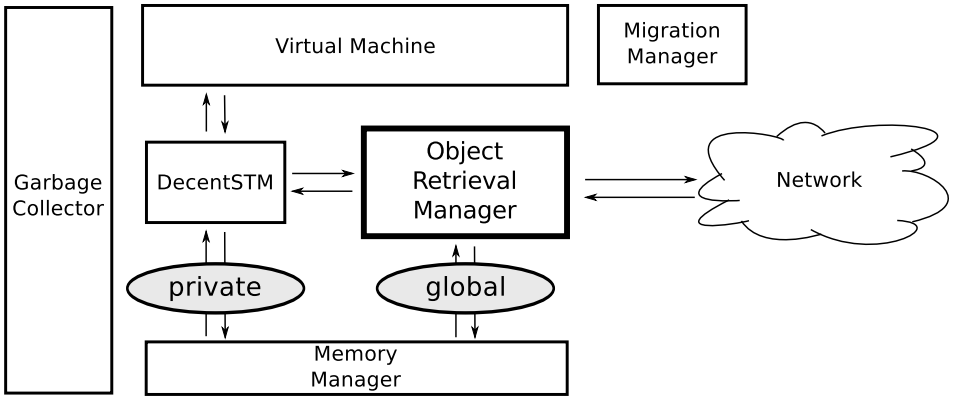


Figure 5.1: High-level view of the envisioned runtime environment and the interactions between the individual modules. The central module for this thesis is the *object retrieval manager (ORM)*. The ORM handles the translation between local and global references and is responsible for the communication with all locally referenced remote objects.

The top layer consists of (an instance of) the *virtual machine (VM)* that executes the transactional code of the application, which uses the DecentSTM. Alternatively, this could be an application using the C STM library that we are developing in our group. There is no difference in the functionality between VM and C STM library with respect to the distributed execution of an application.

The VM communicates with the underlying modules solely via the *DecentSTM* interface, e. g. via transaction control and object access operations. The DecentSTM module manages the execution of the individual transactions. It requests the creation or retrieval of local object copies, executes the distributed consensus protocol, and mediates between logically private and logically global memory. To access local object copies in the logically private memory, the DecentSTM module interfaces with the *memory manager* module, which is responsible for all physically local memory of the node. The memory manager allocates physically local memory for both, local object copies and globally accessible objects that reside on the local node. Furthermore, it creates local object copies from locally stored globally accessible objects on request.

Whenever the DecentSTM module encounters an access to an object that does not yet reside in the transaction's private memory, the DecentSTM module sends an object retrieval request to the *object retrieval manager (ORM)*. The ORM handles the translation between local and global references and is responsible for the communication with all locally referenced remote objects. When the DecentSTM module requests a local object copy, the ORM is responsible for locating the globally accessible object and retrieving a copy of this object.

The *garbage collector* and the *migration manager* are not directly involved in the object access, but fulfill some additional management and maintenance tasks. The garbage collector interfaces with the VM, the DecentSTM, and the memory manager. It is responsible for the deletion of high-level objects such as Java objects and arrays, and the collection of outdated object versions and transaction records, which are not needed by the DecentSTM or the recovery algorithm anymore. The migration manager is an additional component that can explicitly migrate threads and objects between nodes, e. g. for load balancing or system maintenance operations.

5.1 Virtual Machine

Typically, the runtime environment in the envisioned scenario will be a virtual machine (VM), e. g. a Java VM. Since the work of this thesis was inspired by a Java VM, I will use in the following the corresponding terminology.

The task of the runtime environment is the execution of one or more applications. Typically, each application consists of multiple threads that are concurrently executed. Internally, each of these threads consists of chunks of local memory. These chunks hold the execution context (program counter, stack frame, etc.) and the corresponding program code. The system handles both, execution context and code, as regular objects in the sense of the memory model. Thus, the DecentSTM module and the ORM have to create local object copies of these objects before the VM can start the execution. Furthermore, other application data may be stored on separate nodes as well. Before the VM can operate on this data, it has to initiate the creation of local object copies of this data via the DecentSTM module and the ORM.

The following bullet list describes both, a system with and a system without DecentSTM module. The reason for this separation is that the protocol descriptions that follow in Chapter 7 and Chapter 8 handle the general case of decentralized remote object access and retrieval, the DecentSTM scenario is a special case of. The DecentSTM algorithm implicitly migrates objects whenever a new head version comes into existence on another node than the one where the previous head version resides.

With respect to the decentralized object access, the only VM operations that are of interest for this thesis are the instructions that deal with the creation of or access to objects, namely:

- **NEW**: Initiates the creation of a new mutable high-level object Y, for example a Java object or array. The result of this operation is a reference to object Y.

  In the DecentSTM context, this instruction creates the object's initial object version. Until the creating transaction successfully commits, this object version only exists in the transaction's logically private memory. If the user declared the object as a shared

object, it becomes a globally accessible object when the commit succeeds. At this moment, the DecentSTM module has to copy the object from the logically private memory to the logically global memory, i. e. from the physically private memory into the physically local memory. Furthermore, in case of a dynamic object, this new object (version) is only accessible from other nodes or transactions if they have been passed a reference to the object.

- **GET <ref Z>**: Reads a reference field of an object Y to get hold of a reference to object Z. The operation returns the read reference.

  Without DecentSTM, the read object might be either local or remote. If the read object Y is located on a remote node, a read request message is sent to the home node of object Y. The remote node that holds object Y reads the requested reference field and answers the request with an access response message. This message contains the read reference to object Z, and the location information, where object Z can be found.

  In the DecentSTM context, this instruction requires by construction that the read object Y is a local object copy. To actually access object Z after the reference was read, the transaction has to hold a local object copy of object Z as well. If no such copy exists, yet, the DecentSTM module has to initiate the retrieval of such a copy at the ORM. The ORM resolves the given reference to the location information of the latest head version of object Z. If object Z's head version is located in the local portion of the logically global memory, the ORM initiates a copy operation of the data from the nodes' physically local memory to the logically private memory of the transaction at the memory manager. If object Z's head version resides on a remote node, the ORM fetches a copy of this head version from the remote node.

- **PUT <ref Z>**: Writes a reference to an object Z into a reference field of object Y.

  Without DecentSTM, neither the written object Y nor the referenced object Z need to be local. If object Y is local, the reference to Z is simply written to object Y. In the case that object Y is remote, the writing node sends a write request message that contains the reference and the location information of object Z to object Y's home node. The receiving node writes the reference to object Y and returns an acknowledgment message.

  With DecentSTM, the write operation requires that at least object Y is a local object copy in the private memory of the executed transaction. Upon the successful commit of the writing transaction, object Y's modified local object copy becomes the new head version of object Y. Thus, the DecentSTM module initiates the copy operation of object Y's modified local object copy from the private memory to the node's portion of the globally shared memory.

Scheduler The VM runs an application by executing its bytecode in one or more threads. A *scheduler* considers all runnable threads of the corresponding VM instance and assigns each thread a fixed unit of execution time at the local processor. After the assigned execution time, the scheduler puts the executed thread on hold and continues with the execution of another one.

Figure 5.2 shows the state diagram with the different states and transitions of a thread, which are concerned with object access operations. One sees that a thread alternates between the *running* and *wait* state.

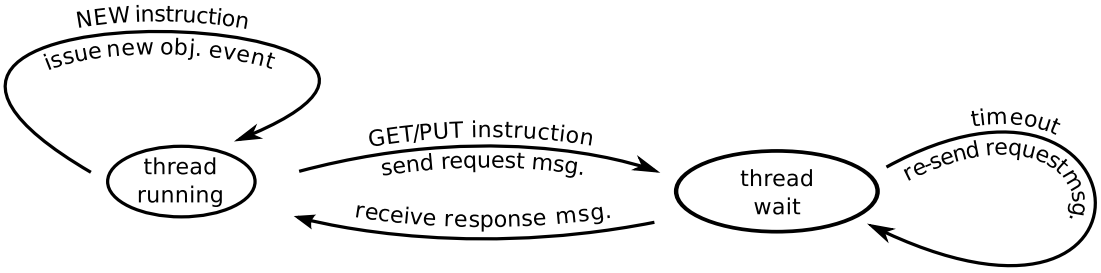


Figure 5.2: Thread state diagram with the different states and transitions that are concerned with object access operations.

While the thread is in the *running* state, it executes GET and PUT operations that access local and remote objects, which cause the transition into the *wait* state. If such an operation accesses a local object copy, the thread immediately returns from the *VM wait* state. If the accessed object is remote, the runtime system sends an access request message and the thread stalls, until either a response message is received, or a timeout occurs. Upon the reception of a response message the local thread resumes, while a timeout initiates the re-sending of the request message.

5.2 DecentSTM

The *DecentSTM* module is responsible for the management of all transactions, and especially for all shared object accesses. Because the envisioned runtime system offers strong atomicity, cf. Chapter 2, object manipulations are only allowed within transactions.

When a new transaction starts, the DecentSTM module sets up the required meta data structure for the transaction, the so-called *transaction record (TR)*. This TR contains all meta data that is required for the execution of the transaction. This includes the *read set*, *write set* and *check set* (see Bieniusa and Fuhrmann [BF10]), as well as a reference to the executed bytecode. This meta data suffices to restore a lost transaction solely from the transaction record.

Whenever the executing transaction accesses a shared object, the DecentSTM module checks if the used reference identifies a private local object copy (LOC) or a globally accessible object (GAO). If the reference identifies a GAO, the DecentSTM module has to initiate an object retrieval request at the ORM. As stated above, for a physically local GAO, the ORM initiates the copy operation of the GAO at the memory manager, and the memory manager copies the GAO from the node's local parts of the logically global memory to the logically private memory of the transaction. If the GAO resides on a remote note, the ORM initiates the retrieval of an object copy from the GAO's remote home node. When the copy arrives, the ORM passes it to the memory manager, who places the copy as a LOC in the logically private memory of the transaction.

At the end of the transaction, the DecentSTM protocol tries to publish all modified LOCs to the outside world during the commit phase. At this time, the distributed consensus protocol checks if publishing the modified objects would lead to memory inconsistencies. For this, the consensus protocol negotiates with all other transactions that read or modified the same GAOs, i. e. created and potentially manipulated local object copies of the GAO. Thereby, the involved transactions determine which of them should be allowed to successfully commit, and thus add a new head version to the objects version history. All other transactions have to roll back and restart their execution. Because transactions are represented by TRs, which are objects as well, the consensus protocol communicates with all remote transactions via the ORM, too.

5.3 Object Retrieval Manager

The *Object Retrieval Manager* (ORM) is responsible for locating, retrieving, and accessing remote objects. Therefore, it mediates between local references, which are used by the VM and DecentSTM module and global references, which point to GAOs which may be located on the local or on remote nodes.

Accessing a dynamic object requires that the accessing transaction possesses a reference to the accessed object. Such a reference can

- be read from a static object,
- be read from a remote dynamic object to which the reference is known,
- result from the migration of a local object to a remote node,
- be brought along with an object that migrates to the local node or
- result from a NEW operation that could not be fulfilled locally.

This thesis separates the identification of an object (via references) conceptually from the addressing of the object (its location in memory). This separation is comparable to a *file handle*, which is used to identify a file, and the *file name* on the hard disk, or a *socket* that identifies the connection, and the *URL* (Uniform Resource Locator), which is used to locate the resource in the network. Both, the *file handle* and the *socket*, are abstract identifiers that identify entries in some kernel data structures, which contain all the details needed to access the corresponding entity.

So, the ORM separates object references from object addresses. To reflect this separation, the ORM uses a multi staged de-referencing system shown in Figure 5.3. It uses a *ReferenceMap* and a *GaoMap*, and an optional per-object *Incoming Reference Map (InRefMap)*. While the first two maps are necessary to access and retrieve local and remote GAOs, the latter map is only used for the proactive location update approach described in Chapter 8.

This de-referencing via reference maps has similarities to the *virtual memory management* (*VMM*) in common processors. Here, the VMM separates the virtual address space from the physical memory location and uses a *memory map* to map the virtual address to the physical address. Thus, the VMM hides the underlying memory organization and offers the user transparent access to a virtual memory, which has a larger address space than the actual physical memory.

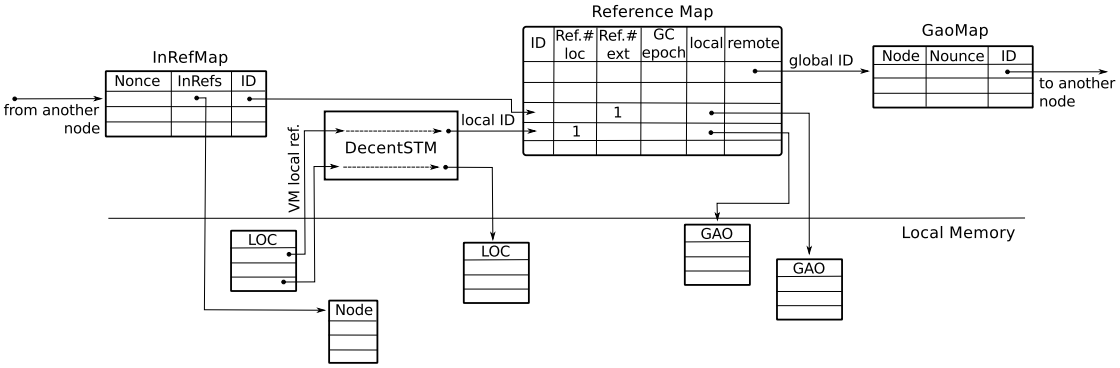


Figure 5.3: Detailed view of references, InRef and GaoMap, similar to [BEF10]. The internal translation that is performed by the DecentSTM module is not part of this thesis.

Internally, the VM uses VM-local references to identify Java objects and arrays. Upon access, the VM passes this local reference to the DecentSTM module, which translates the VM local reference either to a local memory pointer, e. g. a C pointer, that points into the private memory of the transaction, or to a *localID* that points into the ReferenceMap. The internal translation that is performed by the DecentSTM module is not part of this thesis.

The usage of VM-local references and *localIDs* shields the VM and the DecentSTM module from the knowledge about the actual object location and allows an easier implementation of both modules. For example, the VM and the DecentSTM module can use internal 16-bit for *localIDs* and local references, while the *globalID* of an object might be 160-bit or more.

Each ReferenceMap entry stores the mapping of a *localID* to either a *local memory pointer*, e. g. a C pointer that points to a GAO in the local portion of the global memory, or to a *globalID*, which points to an entry in the GaoMap. Moreover, each ReferenceMap entry contains a local reference counter that indicates how many local objects reference the corresponding GAO, and stores an external reference counter and a garbage collection epoch index for the garbage collector.

The GaoMap translates the *globalID* into the GAO's location information that is needed for the remote GAO access. To signal an authenticated access, the GaoMap contains an additional cryptographic *nonce* that is sent together with the GAO access message.

Figure 5.4 shows a simplified example of two objects on node *A*, which reference another object on node *B*. In this example, the reference counter of the ReferenceMap entry on node *A* indicates that two local objects reference a remote object. The GaoMap entry on node *A* stores the referenced object's ID and indicates that the object is located on node *B*. On node *B*, an external reference points to the InRefMap and local ReferenceMap entry, which indicates that one remote node references the local object.

The labels in Figure 5.3 and Figure 5.4 indicate that the GaoMap stores tuples of `<node ID, local ID>` to identify and locate remote objects. This representation simplifies the access to the remote object at its current location, because no additional ID translation step is required on the remote node. However, this representation is only exemplary, and the system is free to choose the reference representation and object location information. The most generic

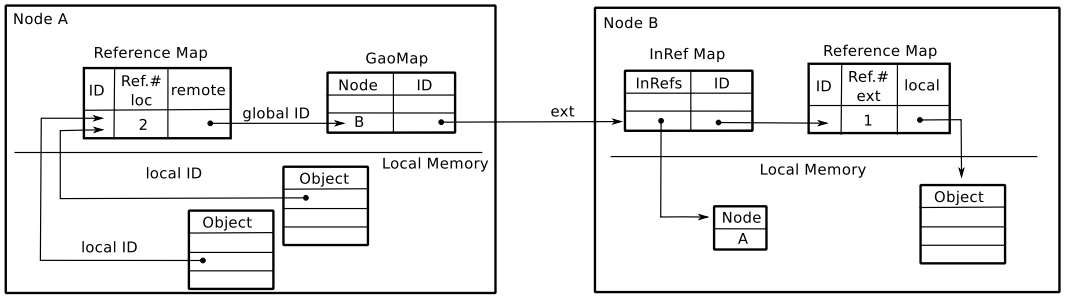


Figure 5.4: Two objects on node *A* reference a remote object on node *B*. The GaoMap entry on node *A* stores the referenced object's ID and indicates that the object is located on node *B*. On node *B*, an external reference points to the InRefMap and local ReferenceMap entry, which indicates that one remote node references the local object.

representation of a *globalID* is a *globally unique identifier* (*GUID*). With this GUID it is, for example, always possible to broadcast a request into the network, cf. Section 6.2.1.

Other location information can be, for example, any other form of routing information, such as a hop-by-hop *source route* or a virtual circuit. Which routing information is used depends on the underlying routing algorithm, which is responsible for the delivery of messages to a given location.

5.4 Memory Manager and Garbage Collector

The *Memory Manager* is responsible for the node's local memory. It allocates memory chunks for LOCs and local GAOs, and copies GAOs from the local parts of the logically global memory into the logically private memory of a transaction and vice versa.

Furthermore, the memory manager allocates and manages the memory for the internal data structures of the runtime environment, such as the ReferenceMap and the GaoMap.

Finally, the memory manager interfaces with the *garbage collector* and frees the memory of those objects that are marked for deletion, or whose reference count has dropped to zero.

The *Garbage Collector* (*GC*) is responsible for cleaning up those local objects and data structures that are not needed anymore. This includes objects on all layers in the system: in the runtime environment, where e. g. Java objects and arrays become garbage, and in the DecentSTM module, where a long version history of an object can be collected, if the older versions are not needed anymore.

The development of the garbage collector module is ongoing work in our group. A survey on (distributed) garbage collection techniques can be found, for example, in [PS95] or [JL96].

5.5 Migration Manager

The *Migration Manager* handles the explicit object migration from one node to another. Discussing details and the reason for an explicit migration is beyond the scope of this thesis, but some

potential scenarios are e. g. the need to bring two objects closer together to prevent unnecessary message traffic, load balancing, system maintenance or the need to free local memory.

The migration manager is an optional module in the envisioned system, where all object migrations take place implicitly because of the DecentSTM protocol. Namely, whenever a new head version comes into existence on another node than the node where the current head version resides. Nevertheless, the evaluation in Section 9.1 uses a simple migration manager that triggers explicit object migrations to compare the two location update protocols from Chapter 7 and Chapter 8.

In the general case, the migration manager is invoked whenever a node requests an explicit object migration. This explicit migration is performed either as *push*, *pull* or *transfer* operation, and all these operations can be aborted due to insufficient memory on the node where the object is migrated to.

In a *push* migration, the old home node initiates the migration. It takes a local object and pushes it to a remote node. The OMNeT++ network emulator described in Chapter 9 uses push migrations to simulate explicit object migrations.

A *pull* migration is initiated by the new home node of the object. The new home node *pulls* the object from the old home node and stores the object in its local memory. In the DecentSTM scenario, this pull migration is comparable to the creation of a new head version of an object on another node than the previous head version. For this reason, the OMNeT++ network emulator uses pull migrations to simulate implicit object migrations.

A *transfer* migration is initiated by a third party. It transfers an object from its old home node to its new home node without the further involvement of the initiating node.

6 Locating Objects

As mentioned in the introduction, the main research question of this thesis is

"How can I ensure that an object is reachable in a fully decentralized system that allows object migration?"

This question can also be phrased as

"How does a node locate an object in such a system after it moved from one node to another?"

Since in a fully decentralized system an object might be located on any arbitrary node in the network, an object access does not only require a valid reference that identifies the object, but also a mechanism to locate the object in the system.

This part of my thesis gives an overview over different approaches that locate objects in a fully decentralized system. Even though the research question explicitly excludes the centralized approach, I describe it to be able to compare the centralized approach against the fully decentralized ones.

The next section starts with the description of approaches that allow the location of *static* objects, e. g. via lookup mechanisms such as a distributed hash table or a central server. These lookup mechanisms are necessary, even if the system does not allow object migration, because a static object must exist only once per application. Hence, a node that accesses the static object the first time has to check whether the object already exists. Therefore, the node must be able to locate this static objects, even if the node has no further knowledge about the object's location. If the static object does not yet exist, the node has to create the static object, which must afterwards be accessible by all other nodes in the network. However, the section about static objects is kept short because the main focus of this thesis is on locating dynamic objects.

The section about static objects is followed by the description of location mechanisms for dynamic objects. To access a dynamic object, a node has to possess a valid reference to the object, which it might receive by a remote method invocation or by reading a reference field of another object. Even though such a reference can be accompanied by some kind of information about the location of the object, this information could be outdated because the object might have migrated to another node in the meantime. Therefore, the main part of this chapter gives an overview over multiple approaches that guarantee the reachability of migrating dynamic objects.

6.1 Locating Static Objects

In Java, for example, the *static fields* of an object are class variables, i. e. they exist only once per class. A common Java virtual machine (JVM) initializes the static fields during class loading.

According to the Java specification [Gos+00], this happens upon the first access to the class, i. e. upon invoking a method of that class, creating an instance of that class, or accessing a static field of that class. Afterwards, each thread of the corresponding application that accesses the class must be able to access the initialized static fields, as well. This access does not require an explicit reference; the identifier of the class suffices.

Since all the static fields of a class are initialized by the same thread, the envisioned runtime system combines them all into a so-called static object. I. e., the system does not support separate locations for the static field of the same class.

In a C programming environment that uses the DecentSTM C library, static objects correspond to global variables. Therefore, similar to static objects in Java, the C library combines all global variables in one C struct.

Note that a runtime system, such as a JVM that executes multiple applications in parallel has to shield the static objects of the applications from each other. For this separation, Liang and Bracha [LB98] described an approach to isolate user classes in the same JVM by using a separate, user defined *class loader* per application, which places all classes – and all static fields – of its application in a separate *namespace*.

The JSR-121 (*Java specification request*) [SUN06] introduced the concept of *isolates* and specified an API to isolate applications from each other.

Czajkowski [Cza00] argued that the use of a separate OS process for multiple single-application JVMs, or one class loader per application in a multi-application JVM, results in too much code duplication. His main idea is to have only one copy of the code of all user and system classes for all applications, but give each application a private set of the corresponding static fields and some private monitors.

In a distributed environment, each instance of a distributed JVM has to know if the static fields have already been initialized when a thread, which is executed by this JVM instance, first accesses a class. If it happens to be the first thread to access a class, the corresponding JVM instance initializes the class variables for that class and runs the initialization code. It must also timely publish this fact so that no other JVM instance runs the initialization again. Hence, the distributed runtime environment requires a mechanism, e. g. a resolver protocol, to store and retrieve the location of the static fields for each class.

One possible solution to ensure the uniqueness of each static object and to allow all nodes to retrieve a reference to that object is the use of a *singleton mechanism* together with a *distributed hash table* (*DHT*) that stores key-value pairs, and that is discussed in more detail in Section 6.2.3.

In case of a static object, the *key k* is, for example, the hash value of the class identifier such as the string *java.lang.Object*, or the hash value of the bytecode of the class. The stored *value v* is either the *static object* itself or the location information for the static object's current home node.

Upon class loading, each JVM instance has to perform a lookup in the DHT. If the class' static object has not yet been created, the lookup request results in a negative answer, and the requesting node creates and initializes the static object. Afterwards, it publishes (the location of) that object in the DHT. If the lookup returns a reference or location information, the node uses this information to access the static object as it does with dynamic objects. If the static object migrates to another node, this change of location must be published in the DHT.

Another approach to manage static objects is the use of a central registry, discussed in more detail in Section 6.2.2.

However, a first practical approach is to initialize all static objects of an application at startup. I. e. the JVM (or C runtime) instance that is the first to start the application, initializes all static objects and passes the reference to these objects to all local and remote threads that the application spawns afterwards. Thus, the static object can be handled in the same way as dynamic objects.

6.2 Locating Dynamic Objects

Whenever an object migrates from one node to another, the cached object location information on all other nodes, but the two that are involved in the migration, becomes invalid. Without some kind of object location and retrieval protocol, the object location is now lost and no other nodes are able to access the object anymore.

Thus, such an object location and retrieval protocol is required and either has to

- guarantee that an access message always reaches the corresponding object, even if the message is sent to an outdated location, for example, by using forwarding proxies. These proxies can be left behind by a migrating object and forward all requests to the new dynamic home node of the object.
- ensure that an object location can always be retrieved, for example, through the static home node of an object or the lookup at a centralized or decentralized lookup service.
- keep the location information of the migrated object updated on all other nodes, for example, by a proactive location update scheme that keeps all references of a migrating object updated.

As an alternative to these approaches, an accessing node can always send an access request as a broadcast message to all nodes in the network.

In the following sections, this thesis describes different object retrieval and location information maintenance approaches. These approaches ensure that a node that holds a valid reference to an object can access this object, even if the object migrates among the nodes in the network.

However, if any of these protocols (besides broadcast itself) fails for some reason, the fallback to broadcast is always possible, even though costly.

6.2.1 Broadcast

The easiest approach to locate a migrating object is to send a *broadcast* message, especially in small networks of only a couple of tenth of nodes. The use of broadcast messages is a common approach to locate objects or to inform other nodes about an object migration [LH89; Tan+91; SJ95; Bal+98; CWH99; Göc+04], see Chapter 3.

The advantage of broadcasting is that there is no need to send any update messages or to keep or maintain any additional state such as registries or proxies. The drawback is the potentially high

communication cost in large networks. However, it is a last resort in case that the other protocols fail.

For a network of n nodes that supports unicast communication, a broadcast message can be sent using $(n-1)$ unicast messages. This approach is inefficient because the same message is sent multiple times, and in case of multi-hop paths, the same information traverses some nodes more than once, especially in the direct neighborhood of the sending node.

As a result, various other broadcast approaches exist, which are additionally applicable for networks that do not support unicast, or that cannot easily keep a list of all participating nodes. One type of networks that lacks these properties are (mobile) ad-hoc networks, where nodes might join and leave the network at any time.

The underlying network of the AmbiComp scenario can be seen as a sub-class of ad-hoc networks where some nodes, such as laptops or hand held devices, frequently join and leave the network. Similarly, a network of compute nodes in the J-Cell scenario can be seen as an ad-hoc network as well. Here, the user has to expect that components might fail and have to be replaced by the network administrator, cf. for example two studies from Google data centers that examine frequent hard disk failures [PWB07] and DRAM errors [SPW09].

Williams and Camp [WC02] discussed twelve broadcast protocols for such (mobile) ad-hoc networks and classified these protocols in four categories: simple flooding, probability based, area based, and neighbor knowledge methods.

The easiest broadcast method is *flooding*, where each node transmits a given broadcast message exactly one time to all its direct neighbors. With *probability based* broadcast, a node only retransmits a message with a given probability. This saves resources in dense networks where the transmission range of multiple nodes covers a similar area. This is similar to the *area based* approaches, where a node does not retransmit a message if it is geographically close to the sending node. In *neighbor knowledge* methods, the nodes collect information about their one- or two-hop neighborhood. This knowledge is sent with each message, so that the receiving node can compare the neighborhoods and decide if a retransmission reaches additional nodes.

Dalal and Metcalfe [DM78] introduced the reverse path forwarding protocol (RFP) for broadcast packets, which is the basis for various other broadcast protocols, e. g. DVMRP [WPD88], RRB [SA83] or TBRPF [BO99]. This routing scheme forwards a broadcast message only if the interface on which the message arrived is also the interface that connects the receiving node to the source node of the message via the shortest possible path.

With respect to the task of locating, retrieving and accessing objects in a distributed network, a system can use broadcast in two different ways: either, each object migration proactively triggers the broadcast of the new location of the object, or, each node reactively broadcasts its object access request throughout the network.

Both these approaches guarantee the lowest access latency. With the reactive access broadcast, one of the $O(n)$ broadcast messages reaches the home node of the remote object on the shortest path. With the proactive update broadcast, each accessing node can send a unicast access message directly to the current home node of the object because the location information is up-to-date (at least as long as the access does not overlap with a location update broadcast, see below).

The proactive approach actively broadcasts the location update information of a migrated object to all nodes in the network. Each node that is interested in the location of a migrated object Y either caches the location information the first time or updates an already cached location information. Afterwards, each node that cached the location information is able to directly access object Y at its new home node. All nodes that are not interested in the location of Y are free to drop the information.

Broadcasting the new location of a migrated object guarantees the lowest object access latency that requires $O(1)$ access messages, i. e. a single request message and a single response/acknowledge message. Nevertheless, in a network with n nodes, a single object migration requires $O(n)$ messages to update the object location on all other nodes. This is especially expensive for all objects that are only accessed by a small subset of nodes, and for all objects that are seldom accessed but frequently migrated.

The reactive approach shifts the broadcast operation from the object migration to the object access. Here, each node that accesses a remote object broadcasts its access request into the network, which is answered by the current home node of the accessed object. Thus, each object access in a network with n nodes requires $O(n)$ messages, while an object migration requires $O(1)$ messages: one message that transports the migrating object, and one message that acknowledges the migration and indicates its success.

To reduce this access overhead each node can cache the location information that it receives after the broadcast together with the response message. Afterwards the node sends all access message directly to the cached home node of object Y. This continues until Y migrates to another node with the result that the next access request fails and the accessing node has to broadcast the access request again.

This is advantageous for all objects that never or seldom change their home node, but increases the access latency for all objects that frequently migrate.

Both these approaches are straight forward but have to handle object accesses that overlap with an object migration. If an access overlaps with a migration, the previous home node of the migrating object might either drop the received access message or optimistically forward the access message to the assumable new home node. In any case, the previous home node must not response with a location update message because the previous home node must not make any authoritative statements about the success of the ongoing migration.

In case that the migration succeeds, the forwarding of the access message can shorten the access latency. Thus, this approach should be taken for the reactive broadcast, because it might prevent the sending of an additional number of $O(n)$ messages for a re-send access request.

Besides the individual drawbacks of the discussed broadcast approaches the general scalability problem of broadcast remains because either the access or the migration requires $O(n)$ messages. Nevertheless, broadcast is the fallback solution for the case that the location of a remote object cannot be obtained otherwise.

6.2.2 Central Registry

A *Central Registry* stores the location information for all objects present in the system and is a common approach to locate or track mobile objects [Ste+98; BMT03; Bha+07], cf. Chapter 3.

A similar approach is taken by the *domain name system* (*DNS*) [Moc87a; Moc87b]. DNS is a distributed, hierarchical lookup service for the Internet that uses centralized *DNS name server*. The local *DNS resolver* at the client side is responsible for the resolution of a given name to the corresponding IP address. Therefore, the resolver sends a query to its pre-configured name server, where the DNS query is either recursively or iteratively handled.

In the iterative approach, the name server either responses with the requested *resource record* or returns the address of another name server. In the latter case, the client's resolver re-transmits the query to this new name server. This process is repeated until a name server answers the request with a valid *resource record*.

A recursive query is always answered by the queried name server. In case that this server cannot answer the request directly, it queries other name servers until it gets an answer that is sent back to the client.

The concept of centralized registries is used in other application domains as well. For tiered sensor networks, Bhattacharya et al. [Bha+07] describe a flexible, hierarchical location directory service, called *Multi-resolution Location Directory Service (MLDS)*. The goal of MLDS it to locate and track physical objects, such as tools or employees. MLDS has a tiered hierarchy with a central registry at the top most layer and central servers for each sub-region.

Similarly to MLDS, Steen et al. [Ste+98] and Bisignano, Modica, and Tomarchio [BMT03] use a hierarchy of central location servers for mobile agents with a central server at the top-most layer and for each sub-region a central servers, as well. All agents within one sub-region have to register their location with the sub-region's central server. If an agent migrates to another sub-region, it first has to de-register itself in the current sub-region and re-register in the new sub-region.

RPC middleware systems such as CORBA or DCOM also use a centralized server as a central object and service location registry that allows nodes the lookup of remote services, see Section 3.2.1.

If a central registry is used to maintain the location of migrating objects each successful object migration must propagate the new object location to the central registry. For the first object access, the accessing node has to obtain the object location from the central registry before the actual access can take place. Thus, this object access requires four messages: two messages (request and response) for the lookup of the object location and two messages for the actual access. As an alternative, the central registry could forward the access request to the current object location. This decreases the number of messages from four to three.

When the accessing node receives the response, it can cache the obtained object location and use it for all subsequent accesses. Thus, the node reduces the message overhead until the first access fails and requires a new registry lookup.

The main problem of the central registry approach is its lack of scalability and fault tolerance. If the registry is stored on a single node, this node is a single point of failure, for example, if the node physically fails, its storage capacity is exceeded, or the node cannot handle the load of frequent lookup and location update messages. Furthermore, the latency to access a single central server increases with the size of the network, i. e. with the maximal possible distance between a node and the central server.

To avoid the single point of failure, the central registry can be fully replicated on different nodes. This has the additional advantage that the lookup requests can be distributed among different nodes as well. Thus, the access latency in a large network is decreased because a node can contact that registry node that is latency-wise closest to itself. However, an additional consistency protocol must ensure that all replicated registries have consistent object location information.

Even if the introduction of intermediate layers reduces these problems, they still introduce additional management and message overhead, e. g. to keep replicated registries consistent or to locate and access objects that are only reachable by traversing multiple layers in the hierarchy.

Another drawback is that a centralized server must manage all objects in the system to make them available to all nodes in the network. As stated before, this is unnecessary for the envisioned system were objects are commonly accessed from a small subset of nodes.

6.2.3 Distributed Hash Tables

An alternative to a centralized registry is a decentralized and distributed registry. A common implementation of such a registry is a *distributed hash table* (*DHT*). Some DHT implementations, which were mainly developed for peer-to-peer systems, are the *Content Addressable Networks (CAN)* [Rat+01], Chord [Sto+01], Kademlia [MM02] or Pastry [RD01].

A DHT is a distributed data structure that maps identifiers, called *keys*, to associated *values*. The number of distinct values that a DHT can store is determined by the DHT's *keyspace*, which is defined by the hash function used for the DHT. To distribute the database, the virtual keyspace is split into non-overlapping parts, according to a given metric. The management of these keyspace chunks is distributed among all nodes in the network. I. e. each node in the network is responsible for a given chunk of the keyspace and thus for the values that are associated with the keys. Thereby, a DHT implicitly provides load balancing.

To allow the access to the DHT, all participating nodes form a structured, virtual overlay network on top of the physical network topology. Kademlia, for example, forms a binary tree, while Chord forms a virtual ring, where each node stores a link to those nodes that are responsible for the successor and predecessor keyspace chunk. At both ends of the keyspace interval these links are wrapped around. To allow a faster access to the DHT, namely, to a value that is stored in the DHT, each node in a Chord network stores additional shortcut links within the ring. These shortcuts are chosen with increasing virtual distance in the ring, so that each node in the network with n nodes can reach any other node in $O(log(n))$ hops in the virtual overlay [Sto+01].

These properties support *Key-Based Routing* (KBR) within the DHT, which allows a node to access each key without the need to know the exact location of this key. Instead, it is sufficient to know a node that is, with respect to a given distance metric, virtually *closer* to the key. With this knowledge, the accessing node routes a message "towards" the key, i. e. it sends the access message to that known node that is virtually *closest* to the accessed key. There, the node is either responsible for the key and answers the request, or the node knows yet another node that is again virtually closer to the key and thus, forwards the message. This process continues until the node that is responsible for the key is reached.

KBR

To use a DHT to locate and access migrating objects in a distributed system, the system must assign each object (value) a globally unique identifier (key) from the DHT's keyspace. This key determines which node in the system is responsible for the object, and thereby for the object's location information.

To access an object via the DHT, the accessing node uses KBR to route the access request message towards the globally unique identifier of the accessed object. When the message finally reaches the responsible DHT node, this node either forwards the message to the current home node of the accessed object, or it returns a message that contains the current location of the object. (This thesis assumes that the message is always forwarded to the current home node of the object.) When the accessing node receives the response message, it caches the location information that is contained in the response message and uses this location information for all subsequent accesses. When the object afterwards migrates, the next access will fail and requires a new lookup of the object location in the DHT. Therefore, it is necessary that an object that migrated to another node announces its new location to the responsible node in the DHT overlay via a KBR message.

Because of the shortcut links, each object access and each location update after a migration requires $O(log(n))$ messages. Besides these access and location update messages, the DHT requires a certain amount of management messages to keep its structure consistent in case that nodes join or leave the network. The Chord ring for example requires that each node frequently checks its successor, predecessor and all its shortcut links.

If only one node is responsible for a given chunk of the keyspace and this node fails, the location information of the corresponding objects is lost. Since its neighbors in the DHT divide the lost keyspace among themselves, the next object migration routes a location update message via KBR through the DHT, and the lost information is built up again. However, the next object access via the DHT to one of these objects will fail, if the corresponding object has not yet updated the DHT. To resolve this situation, the DHT node responsible for a key broadcasts a request for the object's location when the first access request arrives. Here, the probability that an access request requires that the DHT node sends a broadcast message depends on the migration and access rates of an object. It depends on this probability if the DHT node should drop a given number of access requests and wait for a location update to arrive, before it broadcasts the location update request. This approach is advantageous for example, if the migration rate of the object is high because it avoids the costs of the broadcast.

Similar to the centralized registry approach, an alternative solution is the use of multiple nodes that are responsible for the same keyspace chunk, which again requires an additional consistency protocol.

Altogether, the DHT approach has the advantage that it alleviates the problem of a single point of failure, but it comes with the overhead to maintain the overlay structure of the DHT. Compared to the centralized registry, one of the main advantages of a DHT is that it distributes the knowledge about the location of objects among all nodes in the network. However, this is also its main drawback. As stated before, we expect large networks of compute nodes in which objects are only accessed by a small subset of nodes. But even if objects are only accessed by a few nodes, the DHT spreads the resulting traffic across the entire network. Here, the proxy

approach is advantageous because the proxies confine this traffic to only some nodes in the network, cf. Section 6.2.5.

6.2.4 Static Home Nodes

Another common approach to track migrating objects is to assign each object a *static home node* [ZIL96; CWH99; Her99; AFT99; HST01; Mor01; ZWL02; FWL03; FSS06], cf. Chapter 3. This static home node is responsible for the object during the object's whole lifetime.

Some home node protocols do not allow objects to migrate. Here, the static home node holds the *master object* and manages all accesses to this master [AFT99; HST01; FSS06]. Thus, a remote node that wants to read or write the object has to retrieve a copy of the object from its static home node. The static home node keeps track of all copies and is responsible to keep the master and all its copies consistent. Therefore, all modifications of a copy must be propagate back to the static home node at a given time, e. g. at a predefined synchronization point. Besides synchronizing the object access, some home node approaches, such as cJVM [AFT99], use method shipping to execute a method of the object at the object's static home node.

However, the envisioned system requires object migrations. Furthermore, it does not require the management of object accesses at a single home node to keep the object and its copies in a consistent state. Instead, the consistency of concurrent object accesses is guaranteed by the DecentSTM protocol.

Other static home node approaches allow object migrations and the static home node only keeps track of the object location. This approach is comparable to a decentralized and distributed registry because the location information of all objects, together with the management of this location information, is spread throughout the system.

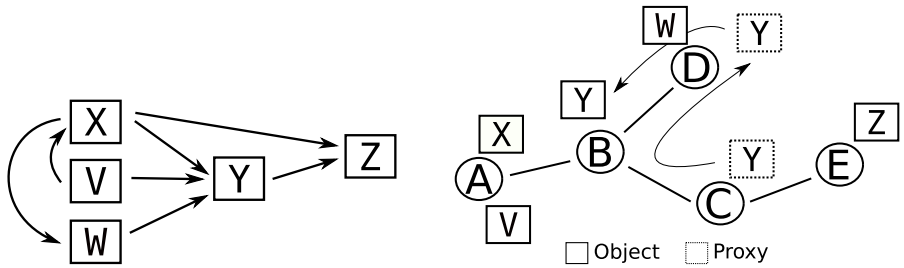
Nevertheless, a static home node is also the central registry node for the corresponding object. Thus, the object access and object migration is handled similar to a central registry: Each object migration has to propagate the new location of the object to the static home node. Nodes might cache the current location of an object, but after the accessed object migrated, the next access fails and the new location information must be retrieved from the static home node.

The drawbacks of a static home node are similar to those of the previous approaches: The single home node of an object is also a single point of failure. If a node leaves the system ungracefully, a redundant home node scheme is necessary, which introduces an additional management overhead. Such a scheme would be mandatory in the envisioned system that allows nodes to leave and join at any time.

Unlike with the DHT approach and without a redundant home node scheme, there is no alternative node that could take over the part of the failed static home node. Therefore, each subsequent object migration invalidates all existing references to the object. The only chance to repair these references after each such migration it to repeatedly use broadcast to regain the object location.

6.2.5 Forwarding Proxies

Forwarding proxies are another common approach to ensure the reachability of migrating objects [LH89; SJ95; PZ97; FWL03; Liu+05], cf. Chapter 3. To my best knowledge, Fowler [Fow86] was the first who introduced forwarding proxies to locate mobile objects in distributed systems. In his approach objects reside on dynamic home nodes and can migrate among the nodes in the network. With each migration the migrated object leaves a forwarding proxy on the old home node that stores the location information of the new home node. Each proxy knows only this next node, and the only purpose of the proxy is to recursively forward all incoming requests for the object to this next location. An alternative approach is to iteratively reply with a message that contains the next location of the object, but the following sections assume that proxies recursively forward all messages.



(a) Object Reference Graph. (b) Object Location in Distributed Network.

Figure 6.1: Proxy chain of object Y, which migrated from node C to D and afterwards further to node B. Because of this migration, the location information on node A is outdated and an access from object X on node A to object Y has to traverse the proxy chain. Node A updates the outdated location information as soon as the response (read access) or acknowledgment (write access) message from node B arrives.

Leaving a proxy after each migration leads to a *chain of proxies* that the migrating object creates while moving through the system. The length of the proxy chain is potentially unbound because each migration adds a proxy, and the following chapters describe approaches to remove proxies eventually. As with the previous approaches, all nodes cache the location information for all remote objects that are referenced from local objects. This location information is created when the corresponding reference is established. If the location information is outdated due to an object migration, the next access message has to traverse the chain of proxies to reach the actual object. Afterwards, the response message transports the current location of the object and the accessing node uses this location information to update the cached location information. Thus, the number of proxy forwards, and with this, the average length of the proxy chain ℓ that an access message must traverse, depends on the average migration rate m of the object and the average access rate a of the accessing thread and can be computed as $\ell = \frac{m}{a}$. To see this consider an interval that contains m migrations and a accesses. Assuming a uniform distribution, the expected value of migrations between two consecutive accesses is $\frac{m}{a}$.

Here, each migration increases the proxy chain for the next access by 1-hop, while the object access updates the outdated location information and thus, shortcuts the proxy chain.

As an example of a *chain of proxies* see Figure 6.1 were object Y migrated from node C to D and afterwards further to node B.

The main advantage of the proxy approach above all others is that it confines the message and maintenance overhead to only a few nodes that are actually linked with the migrating object, i. e. the previous home nodes of the object. The disadvantage is the increased access latency that depends on the length of the proxy chain ℓ.

Chapter 7 gives a detailed description of the proxy approach and the challenges and design decisions of a reactive location update protocol that uses forwarding proxies. Section 7.3 describes an access path optimization that shorten the length of a proxy chain and thus decrease the access latency.

Table 6.1 compares the presented approaches with respect to the costs for an object access, an object migration, and the average load on a single node. Furthermore, the approaches are rated with respect to the maintenance overhead, fault tolerance, and the object locality, i. e. how good a single access or migration is confined to only a small number of nodes. Thus, the table summarizes the different advantages and disadvantages of the different approaches, which have been described in more detail in the previous sections.

The parameters of the table are the network size n, the total number of object accesses a, the total number of object migrations m, and the average length of a proxy chain ℓ. All complexity assumptions in the table assumes a uniform distribution of all objects o among all nodes, leading to $\frac{o}{n}$ objects per node, and a uniform access and migration characteristic for each object that leads to the average length of a proxy chain $\ell = \frac{m}{a}$.

Approach	Msg. Costs Access	Msg. Costs Migration	Max. Load on single node	Mgmt. Overhead	Fault Tolerance	Locality
Mig. Broadcast	$O(1)$	$O(n)$	$O(m)$	$\oplus$	$\oplus$	$\ominus$
Acc. Broadcast	$O(n)$	$O(1)$	$O(a)$	$\oplus$	$\oplus$	$\ominus$
Central Registry	$O(1)$	$O(1)$	$O(a+m)$	$\bigcirc$	$\ominus$	$\bigcirc$
Static Home Node	$O(1)$	$O(1)$	$O(\frac{a+m}{n})$	$\bigcirc$	$\ominus$	$\bigcirc$
DHT	$O(log(n))$	$O(log(n))$	$O(\frac{a+m}{n})$	$\ominus$	$\bigcirc$	$\ominus$
Proxy	$O(\ell)$	$O(1)$	$O(\frac{a \cdot \ell}{n})$	$\oplus$	$\oplus$	$\oplus$

Network Size: n, # of Accesses: a, # of Migrations: m, Average Proxy Chain Length: ℓ

Table 6.1: Comparison of the different object location approaches. Altogether, the comparison of these approaches shows that for the envisioned scenario only the proxy approach is applicable.

The second column shows the message costs of a remote object access. This number is constant for the migration broadcast approach, the central registry and the static home node approach. The reason is that at least two messages are necessary for an access in the migration broadcast approach, and at most four messages for an access in the other two approaches. Here, an access requires two messages to retrieve the latest object location from the static home node or central registry, and two messages to access the object afterwards. The key-based routing access of the DHT approach requires $O(log(n))$ messages, while the access broadcast approach requires $O(n)$ messages, because the object location is unknown and a broadcast message must be sent. The access latency of the proxy approach depends on the average length of the proxy chain ℓ,

which itself depends on the access-to-migration ratio. The worst case access latency of the proxy approach requires $O(n)$ messages; in the case that the object migrated to all other $n-1$ nodes between two accesses.

The third column lists the message costs for an object migration. Here, only the migration broadcast approach has a major overhead due to the location update broadcast that requires $O(n)$ messages. The DHT approach requires again $O(log(n))$ messages because of the key-based routing characteristics. For all other approaches, an object migration only requires $O(1)$ messages, because the central registry, the static home node, and the proxy approach all require only two messages. The central registry and the static home node approach both need one message that informs the responsible node about the new location, and one message that acknowledges the location update. The proxy approach needs one message that migrates the object, and one message that informs the previous home node about the success of the migration.

The fourth column shows the load on each node. This load depends on the total number of access and migration operations. It is seen that the highest load occurs in the central registry approach because a single node has to bear the complete location information management for all objects. In the worst-case, each access and each migration operation has to contact the central registry node to either retrieve or update the location information of an object. In the static home node and DHT approach this load is the same, but the approaches distribute the load evenly among all nodes in the network. This uniform load distribution is artificial for the static home node approach, because I assumed a uniform distribution of objects and access and migration operations. In the DHT approach, the load distribution depends on the chosen hash function, the chosen metric that divides the keyspace and the assignment of the globally unique object identifier that are used as the keys in the DHT.

The load in the broadcast approaches depends on the reason why the broadcast message is sent: If each migration sends a broadcast, all nodes have to handle this message and the load is for a single node $O(m)$, while a broadcast upon each access causes a load of $O(a)$ on every node.

The lowest load on each node is in the proxy approach. Here, each node has to handle $O(\frac{a \cdot \ell}{n})$ messages. The reason is that I assume that each node has to handle $\frac{a}{n}$ object accesses and that a single access has to traverse ℓ proxies.

The fifth column rates the management overhead of the different approaches. It is seen that the DHT approach comes with the worst management overhead, because not only the location information must be managed, but also the overlay structure of the DHT itself. This overhead is absent in the central registry and static home node approach, whereas these two have only an average management overhead that requires the management of the location information on the central registry node or on the static home node. The two broadcast approaches have no management overhead at all, while the proxy approach only has to store the additional proxies and potentially forward access messages.

The picture is similar for the sixth column where the fault tolerance is rated. Here, fault tolerance does not mean the fault tolerance due to an arbitrary node failure that can occur in any system. Instead, my model assumes an adversary that picks a node that is necessary for a given approach to work.

Under this condition, the two broadcast approaches have the best fault tolerance characteristics, because there is no component that has a special responsibility in the network. The object access is only affected if the adversary picks the node on which the accessed object currently resides. This case breaks all approaches because the object is lost. In all approaches described below, the only way to prevent the application to fail if an object is lost, is to create multiple object replicas on different nodes, or to use checkpoints, from where the application can restart. Both, object replication and implicit checkpointing are ongoing work in the group I work with and I will not go into more detail in this thesis. Note that each node that fails in the following descriptions might hold objects, as well.

A similarly good fault tolerance characteristics as the broadcast approaches has the proxy approach, where the worst case is that the adversary picks a node that holds a proxy object. However, the failure of one of the nodes that holds a proxy does not affect all objects that reference the corresponding object, because the entry point in the proxy chain will likely be different for all referencing objects. Thus, only a small subset of nodes is affected by the loss of a proxy, and only these nodes have to use broadcast to retrieve the latest object location. The DHT approach has an average fault tolerance, because the keyspace of a lost DHT node is divided among its neighboring nodes in the overlay. Furthermore, the lost location information is re-built with each subsequent object migration, or if the DHT node is required to send a broadcast because of an access request message. The central registry has the worst fault tolerance because the failure of the registry node causes the loss of all object location information, while the failure of a single static home node only looses the location information that was stored on the corresponding node. Even though the objects are accessible with the cached location information at the individual nodes, the next object migration invalidates all this cached location information. Without a failover node it is impossible to re-build this information and after each subsequent object migration all nodes that want to access the object have to send a broadcast message to retrieve its new location.

Finally, the seventh column rates the locality of the different approaches. In this context, locality means the number of nodes that are involved in an access or migration operation. Furthermore, it rates how far the location information is spread throughout the network. Here, the proxy approach has the best locality because all access messages are confined to those nodes that take part in the access and the proxy chain. The central registry and static home node approach have an average locality because access and location update messages are only sent among the corresponding nodes and the static home node or central registry node. The worst locality comes with the broadcast approaches and the DHT, because these approaches spread the access and/or migration messages among all nodes in the network.

Altogether, the comparison of the different object location approaches shows that for the envisioned scenario only the proxy approach is applicable. It has of all approaches the best ratings in the last three columns and altogether the lowest message costs and load burden on a single node. Therefore, the following chapters only deal with the proxy approach and its optimizations, such as the decrease of the access latency if long proxy chains are established, or the task to delete unnecessary proxies.

7 Reactive Location Updates with Migration Proxies Protocol

This chapter describes the *reactive location update* protocol for outdated object location information, which uses forwarding proxies to keep objects reachable after their migration, cf. Section 6.2.5.

At the beginning, it discusses the general access to migrating objects that left a proxy behind, and examines some basic prerequisites, which must be fulfilled for the protocol to work. Afterwards, this chapter introduces the state diagram for a migrating object and discusses the different state transactions in more detail.

Whenever an object migrates to another node, it leaves a proxy behind that forwards all messages to the new location of the object. Hence, I define a proxy as follows:

Definition 1 *A* **proxy** *is a data structure that contains a* forwarding pointer *that redirects messages for a migrated object to another node in the network. This node may be the current home node of the object, cf. Definition 2, or another proxy node, cf. Definition 3.*

Whether the proxy is implemented as an actual object or only represented by an entry in a local data structure depends on the implementation. In the system described in Chapter 5, a proxy is represented as a forwarding pointer entry in the *GaoMap*, together with a migration sequence number (see description below).

Furthermore, the following chapter uses the terms *home node* and *proxy node*.

Definition 2 *The* **home node** *of an object is defined as the node that currently stores the object (or head version of the object) in its local memory. The home node of an object is the authorized node, which is allowed to perform actions on this object, such as reading or writing fields.*

An object changes its home node with each migration.

Definition 3 *The* **proxy node** *is the node that stores a proxy for a given object in its local memory. Hence, a proxy node must be a former home node of the corresponding object.*

A proxy node is only authorized to forward requests for the actual object to the node, the forwarding pointer points to.

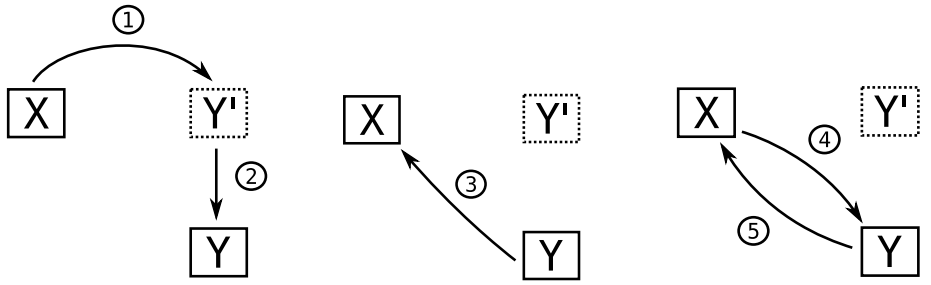
Unlike objects, proxies cannot change their proxy node. They remain on the proxy node until they are deleted.

In the following, objects are indicated by plain letters, e. g. Y, proxies are indicated by a dash, e. g. Y', and pending objects (described later) are indicated by a star, e. g. Y^*. Moreover, the home node of object Y is indicated by N_Y, the proxy node of proxy Y' is indicated by $N_{Y'}$, and the node that stores the pending object Y^* is indicated by N_{Y^*}.

7.1 Object Access

The reactive location update protocol leaves a *proxy* on the former home node after each successful object migration. Whenever the proxy node afterwards receives an access request for the migrated object, it forwards the request to the node, the forwarding pointer points to. Thus, an object is still reachable, even if the cached location information for the object is outdated.

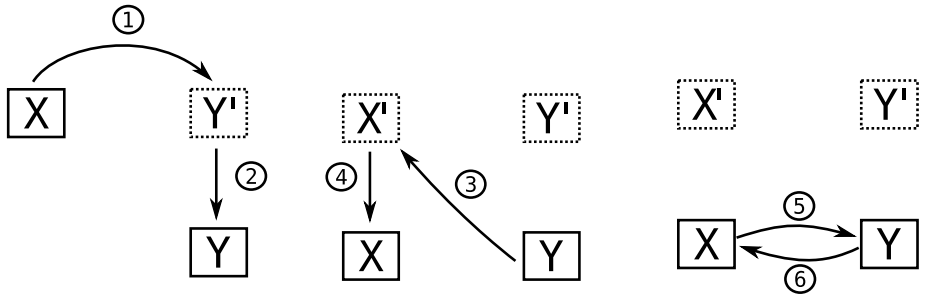
Figure 7.1 shows the simple object access along (a chain of) proxies. Figure 7.1(a) shows the state, where the object Y is migrated to another node and left behind the proxy Y'.



(a) Access Request and Forwarding. (b) Access Response. (c) Subsequent Accesses.

Figure 7.1: Simple object access from object X on node N_X to object Y along one intermediate proxy Y' that forwards the object access message to N_Y.

At the access process' beginning, N_X holds an outdated reference to Y. To access Y, N_X sends an access request ① to the outdated location $N_{Y'}$, which forwards the access request ② to N_Y. Figure 7.1(b) shows that N_Y answers the request by sending the response message ③ back to N_X. When N_X receives this response, it updates its location information for Y. Afterwards, Figure 7.1(c) shows that N_X uses this location information for all subsequent communication with N_Y, ④ and ⑤, without the indirection via $N_{Y'}$.



(a) Access Request and Forwarding. (b) Access Response. (c) Subsequent Accesses.

Figure 7.2: Simple object access from object X on node N_X to object Y. This time, not only Y migrated to another node, but X migrates as well after N_X sent the access message.

Figure 7.2 shows the same process, this time with the additional migration of object X. Again, in Figure 7.2(a), the access to Y is initiated on N_X ①, and the request is forwarded from $N_{Y'}$ to N_Y ②. Afterwards, X migrates and leaves X' on its former home node, cf. Figure 7.2(b). As a next step, N_Y answers the request and sends the response ③ to $N_{X'}$, which forwards the response ④ to N_X, where the node processes the information and updates the location information of Y. Now, all location information are up-to-date and the subsequent access to Y involves N_X and N_Y only, ⑤ and ⑥ in Figure 7.2(c).

7.1.1 Proxy Deletion

With the so far described proxy forward protocol, the proxies remain in the system indefinitely, because the proxy node cannot determine if there are remote references that still point to the proxy. Because of this, the protocol requires the deletion of all un-referenced proxies during a (distributed) garbage collection run of e. g. a common distributed mark-and-sweep *garbage collector* (*GC*). Such a garbage collector starts at a root object, for example, the primordial execution context, and follows all references to all reachable objects. During this run, all visited objects are marked with the timestamp of the current GC period, until an object with no further references is reached, cf. [PS95]. Afterwards, all objects that are marked with an older timestamp than the one of the current GC period will be deleted in the next GC run.

While the GC follows all references to all reachable objects, it also visits all proxies that are encountered during this travel. When the GC reaches the object and traversed (a chain of) proxies, it sends an location update message back to the referencing node, so that the referencing node can update its location information for the referenced object. Because of this update, none of the proxies is used anymore. Thus, the garbage collector removes them in the next GC run.

7.1.2 Reactive Location Update

Forwarding proxies can handle object access messages in two different ways: either, they forward the request message to the new object location, or they respond with a location update message that contains the new object location. Thus, the location update process is either *recursive* or *iterative*, similar to the two modes of operations in the *DNS* protocol [Moc87a; Moc87b].

In both cases, the costs for an access message depend on the length of the proxy chain ℓ, which is defined as the number of proxies that are part of the corresponding proxy chain.

For the following sections, suppose an object X on N_X wants to access an object Y on N_Y. Therefore, N_X sends the *request message* to its cached location information for Y, e. g. $N_{Y'1}$, which is the first proxy node in the proxy chain of Y.

Recursive Approach In the recursive location update approach, each $N_{Y'}$ recursively forwards the *request message* according to its forwarding pointer information to the next node. This process continues, until the message reaches N_Y. N_Y answers the request with a *response message* that is sent back to N_X, without traversing the chain of proxies. This *response message* implicitly transports the new location information for the accessed object. Thus, N_X updates its outdated location information reactively on the fly and sends all subsequent requests directly to N_Y.

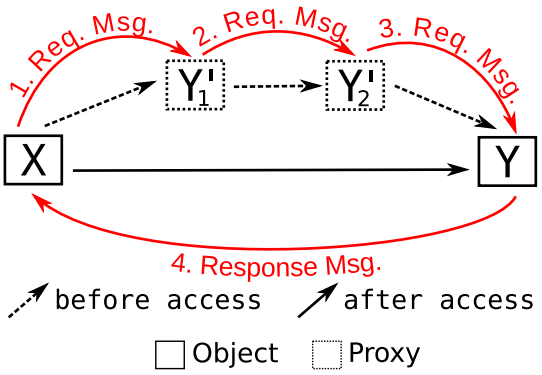


Figure 7.3: Reactive and recursive update of proxy chain. Each $N_{Y'}$ recursively forwards the *request message* and N_Y answers with a *response message* without traversing the chain of proxies.

After receiving the response, N_X might send an optional *update message* along the chain of proxies, similar to the approach of Moreau and Ribbens [MR02]. On one hand, this shortens the access for all other objects that use one of these proxies, on the other hand, this update message increases the traffic overhead.

Furthermore, Moreau and Ribbens used this update mechanism only upon access, not upon the creation of a new proxy. As a result, a long chain of proxies might have been established in the meantime. For this reason, this thesis describes an access path optimization in Section 7.3, which reduces the length of the traversed proxy chains upon object migration.

Without the optional update messages, the message costs c_{acc} are computed as the length of the proxy chain ℓ, counted in intermediate proxies, plus the two messages for the access, i. e. the initial *request message* from N_X to $N_{Y'1}$, and the *response message* from N_Y back to N_X. Hence, the message costs c_{acc} are computed as:

$$c_{acc_{recursive}} = \ell + 2 \tag{7.1}$$

For an example, see Figure 7.3, which shows a proxy chain before and after an access request from N_X to N_Y. The labeled arrows indicate the travel of the *request message* and the *response message*. The labels not only indicate the message type, but also count the number of messages as well. The unlabeled, dashed arrows indicate the outdated location information before the access, and the forwarding pointers, respectively. The unlabeled solid arrow indicates the updated location information after the access.

In this example, the proxy chain has a length of $\ell = 2$ proxies and thus the costs are $c_{acc_{recursive}} = 4$ messages.

Iterative Approach In the iterative location update approach, each $N_{Y'}$ answers a *request message* with an *update message* that is sent back to N_X. This update message contains the information of the proxy's forwarding pointer, i. e. the next node where the request should be sent. In the example, $N_{Y'1}$ returns the location information for $N_{Y'2}$. Thus, N_X iteratively re-sends the request to each proxy in the proxy chain, until the request reaches N_Y.

Figure 7.4 shows this process with the chosen example. Like before, the labeled arrows indicate the *request*, *update*, and *response messages* and additionally count the messages. It is seen that N_X exchanges two messages with each $N_{Y'}$ in the proxy chain of length ℓ plus two additional

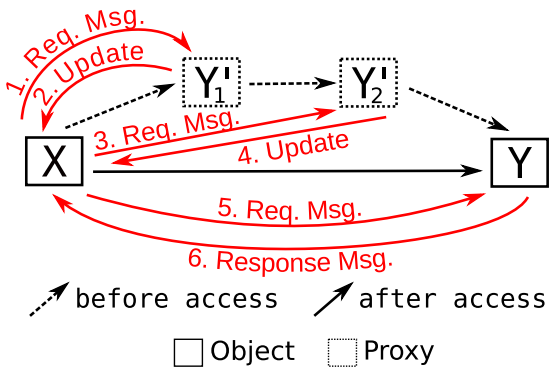


Figure 7.4: Reactive and iterative update of proxy chain. Each $N_{Y'}$ answers a *request message* with an *update message* that is sent back to N_X, which iteratively re-sends the request to each proxy in the proxy chain, until the request reaches N_Y.

messages with N_Y. Hence, the message costs c_{acc} for the iterative location update approach can be computed as

$$c_{acc_{iterative}} = 2 \cdot (\ell + 1) \tag{7.2}$$

In the example, these costs are $c_{acc_{iterative}} = 6$ messages.

As seen in Equation (7.1) and Equation (7.2), the message costs of the iterative location update approach are about twice as high as the costs for the recursive approach. Therefore, I decided for my hereafter presented reactive location update protocol to use only the recursive message forwarding approach.

7.1.3 Cyclic Routing

In the protocol described so far, object migrations and different object access patterns can lead to inconsistent object location information in the network. Even though newer information would be available, these inconsistencies result from the usage of outdated location information. For example, suppose a node A has the information that an object X is located on node B. If node A now receives a message that contains the information that object X is located on node C, it is impossible for node A to distinguish without further actions, which information is correct. If node A chooses the wrong information, the result could be an infinite loop (see [SF10] for an example), which is similar to the "count to infinity" problem in distance vector routing protocols, cf. [Tan02].

To prevent this problem, my proposed protocol assigns each object with a per-object migration sequence number. This migration sequence number is incremented with each migration of the object. Additionally, this migration sequence number is stored together with each remote reference. In the envisioned system, this corresponds to an entry in the *GaoMap* that indicates for which migration step of the referenced object the entry was created. Thus, node A in the example can compare the migration sequence numbers of the location information for object X and choose the most recent one.

A global timestamp, which would require time synchronization among the nodes, is unnecessary because the migration information of a single object is unique to this object.

7.2 Object State Diagram

Figure 7.5 shows the state diagram and the possible state transitions for my developed reactive location update protocol. The following sections describe these transitions in more detail, while the complete description is given in [SF10].

The arrows in the state diagram depict the transitions. The labels above the arrows indicate the cause for the transition. The labels below the arrows illustrate the effect that the transition has on the node or on the corresponding object.

In the following, I describe the general object access process without considering the prerequisites of the DecentSTM protocol. Therefore, the description does not distinguish between local object copies and globally accessible objects, but handles all objects as globally accessible. However, the description contains some remarks where the DecentSTM protocol influences the state transitions.

An object on a given node, for which the access is managed by the reactive location update protocol, can assume five different states during its lifetime:

- *initial* state: The initial state describes the absence of an object. The system leaves the initial state when the object comes into existence, because it is newly created or migrated to that node.

- *regular* state: The object is a regular object and can be used in the regular way. The node can give authoritative answers or perform authoritative actions such as reading and writing fields.

- *pending* state: The object is currently migrating to another node, but the migration is not finished yet. A node that holds an object in the pending state is not its home node anymore, and thus, is not authorized to perform any actions on the object. Furthermore, the node must not make any assumptions about the current state of the object.

- *forwarding* state: When the migration has completed successfully, the object changes from the *pending* to the *forwarding* state. Thereby, a proxy comes into existence and the node becomes a proxy node that forwards request messages to the new home node of the object. However, the proxy node must not perform further actions on the object.

- *finished* state: The finished state indicates that the lifetime of an object or a proxy on this node ends. An object enters this state e. g. if its reference count drops to zero and the garbage collector deletes the object. A proxy enters this state after a garbage collection run where all outdated location information were updated. Afterwards, the proxy is not used anymore and the next GC run will remove it.

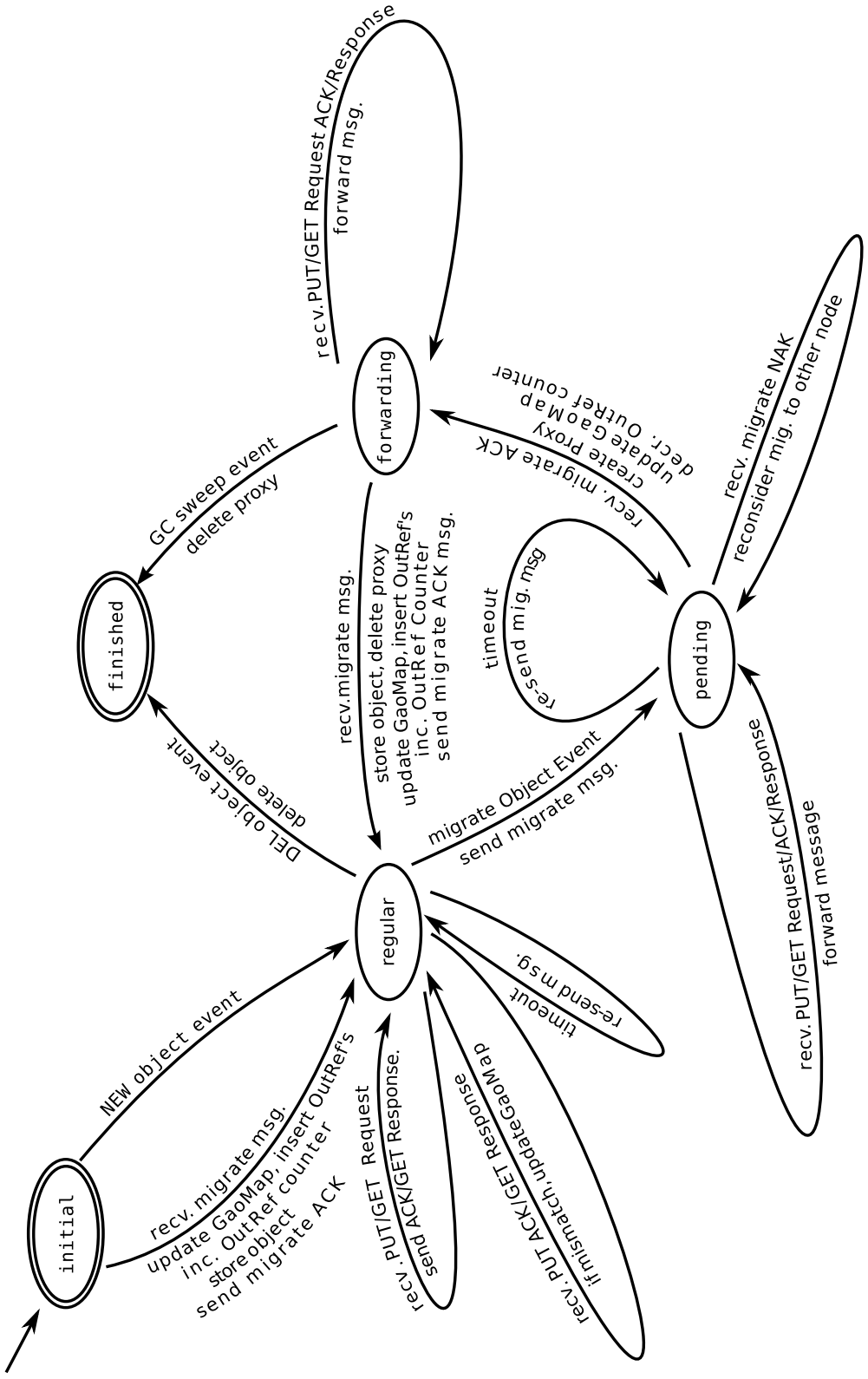


Figure 7.5: State diagram for the reactive location update protocol with migration proxies. The start is at the *initial* state where an object becomes a local object either, because it has been created on this node or because it has been migrated to this node.

In the following, I describe the *regular*, *pending*, and *forwarding* states in more detail. Because the *initial* and *finished* state are only the start state and the final state, they are discussed together with the other three states. Before I describe these states and their transitions, I make some prerequisite considerations on the underlying network.

First, I only consider proxy forwards. Hence, I am not concerned with a particular underlying routing algorithm or a particular network topology. Secondly, I suppose that neither the underlying network nor the routing protocol is reliable, and that for this reason messages might get lost. Therefore, the system uses a timeout period in which a node expects a response or acknowledgment message. If a message is not acknowledged within this time period, the message will be re-send. However, the locally stored object location information and thereby the destination for re-sending the message, might have changed during the given time period. Thus, the node that re-sends the message has to check the currently stored object location and potentially adapt the message's destination. Thirdly, checkpointing and redundancy are the topic of the PhD thesis of one of my colleagues. Thus, I will not consider fault tolerance in my protocol description. Fourthly, I assume that the system that executes the reactive location update protocol is compliant with the system specification from Chapter 5. Thus, I assume that nodes in the system use a ReferenceMap and a GaoMap to store local and remote references. However, the system is not required to use the DecentSTM protocol and thus, I describe both cases: a system with and a system without DecentSTM.

7.2.1 Regular State

The *regular* state indicates that the object Y is located in the local memory of the node and thus, the node is the home node N_Y. An object is local either, because it has been created on this node or because it has been migrated to this node.

In the first case, the invocation of a NEW operation triggers a *NEW object event*, which invokes the local ORM and memory manager to create the new object. In the second case, the node receives an *object migration message* and if the node accepts the migration, the migrated object is added to the local memory. If the migrated object holds additional *outgoing references* to other objects, all these references and their corresponding location information are added or updated in the corresponding ReferenceMap and GaoMap. If an entry is only updated (rather than being created), the runtime environment checks the migration sequence numbers to decide which information (the present or the new one) is kept. Afterwards, N_Y sends a *migration acknowledgment (ACK) message* back to the sending node. If the node cannot fulfill the migration request, for example because of insufficient local memory, the node sends a *NAK (not acknowledged) message* (I assume in this thesis that a node always accepts migrating objects).

A node N_Y that receives a PUT or GET *request message* for Y answers the request with the corresponding *response message*. For a PUT request, the node writes the received reference to the corresponding reference field of Y and acknowledges the access with an ACK message. For a GET request, the node reads the corresponding reference field and sends the read reference – together with the latest known location information of this reference – within a response message back to the accessing node.

An *object migration event* initiates the transition into the *pending* state, and the migration of Y to another node. This migration requires that the current N_Y collects all information about all outgoing references of Y from the ReferenceMap and the GaoMap and includes them in the migration message. If there is a local reference entry for Y in the ReferenceMap, this reference is changed from local to remote and the location information is set to the new home node of the object. This is an optimistic approach, which assumes that the migration will succeed. If the migration fails, the object stays on the local node and the references are changed back to local ones.

This process differs, if the DecentSTM protocol is used. Here, a remote node first pulls an object copy from N_Y, which does not change the local state of the object. Only when the remote node has modified its copy and successfully committed, an implicit migration takes place. Namely, the remote node that modified the object and successfully committed a new object version sends an update message to the local node to inform it about the new head version. Hence, the local object changes directly from the local state to the proxy state because no explicit migration, and thus transition via the *pending* state, is necessary.

The garbage collector triggers the *delete object event*, if the object is not needed anymore. When the object is finally deleted, the node decrements the reference counters of each outgoing reference that the deleted object held. Afterwards, the object changes into the *finished* state and ceases to exist.

7.2.2 Pending State

An object that is currently migrating from the local node to a remote node is in the *pending* state Y^*. With the beginning of the migration, the local node ceases to be the home node of Y and becomes N_{Y^*}. Thus, it is not authoritative anymore and must not perform any actions on Y.

However, N_{Y^*} can forward all *request messages* for Y to its supposed new home node N_Y. This is an optimistic approach, which assumes that the migration succeeds. In the best case, the forwarded message decreases the access latency, because N_Y accepted the migration and is now authorized to answer when the forwarded message arrives. In the worst case, the assumed new home node must drop the forwarded message, because it is not the authoritative home node, yet and the accessing node must re-send the request message after the timeout period.

Hence, there is no authoritative node during the migration takes place.

The simulations in the OMNeT++ emulator, evaluated in Chapter 9, showed, it can indeed happen that a migration does not succeed before the forwarded access message reaches the node. This is e. g. the case if the migration message was sent along a longer path than the forwarded access message. This happens for example when route changes occur in-between the migration and the reception of the access request message. However, this case happens very rarely and thus, does not significantly influence the message costs.

Because there are no authoritative home nodes for the object during its migration, an alternative pessimistic approach could let a node wait for the success of the migration and drop all messages for the pending object until then. If this is the case, the access latency is increased by at least the timeout period plus the time that the re-send message needs to traverse the proxy chain to reach the object at its new location.

Another approach is to leave the old home node in authority for the object until the migration succeeded. Thus, the home node performs all requested actions on the object, but has to publish all changes to the new home node after the migration is completed. This requires a consistency protocol, which needs to send additional messages after each migration.

During the simulations of the protocol it was revealed that it rarely happens that the optimistic approach does not succeed. Hence, an additional consistency protocol that requires additional messages is more costly than a timeout period. For this reason, neither the message dropping, nor the consistency protocol approach seemed to be worthwhile and I decided to use the optimistic approach for the protocol implementation. Furthermore, in the envisioned system, which uses DecentSTM and implicit migrations only, this consistency protocol is implicitly integrated into the commit protocol of the DecentSTM algorithm. Here, the commit protocol decides, which node is the new home node of the head version of the object. Until then, the old home node remains the authoritative node for the object.

If the migration is aborted, e. g. due to insufficient memory on the remote node, N_{Y^*} receives a *migration NAK message*. Depending on the migration policy, the node can either choose another node to migrate the object to, invoke the garbage collector on the defeating node to try to free memory for the migrating object, or cancel the migration completely and become N_Y again.

However, the OMNeT++ simulation environment assumes that there is always enough memory for a migrating object. Thus, no *migration NAK messages* occur in the simulation. I do not consider this a drawback because a *migration NAK message* has no influence on the reactive location update protocol.

The reception of the *migration ACK message* triggers the transition into the *forwarding* state. At this time, the proxy Y' is created and the reference counter of the object's outgoing references is decremented. This decrement must not happen before the migration succeeded, to prevent the premature deletion of the entries in the location maintenance maps, which are needed if the migration aborts.

If N_{Y^*} does not receive the *migration ACK message* during the timeout period, it re-sends the *migration message*.

7.2.3 Forwarding State

The *forwarding* state indicates that the object Y successfully migrated to another node and a proxy Y' comes into existence on the local node. Hence, the local node becomes the proxy node $N_{Y'}$ and is authorized to forward all messages for the object to N_Y.

If the object re-migrates back to this node, the proxy Y' is deleted and a potentially remote reference to Y in the *GaoMap* is changed to a local reference. Object Y is stored in the local memory and all outgoing references and their corresponding location information are added to the corresponding maps. This is the identical behavior as described for the reception of a *migration message* that results in the transition from the *initial* to the *regular* state.

With this process, the proxy forwarding algorithm inherently cuts out loops in a proxy chain, whenever an object returns to a previously visited node, because the next migration creates a new forwarding pointer.

Finally, Y' is deleted after the distributed garbage collection traversed all proxy chains and marked all proxies for deletion. Then, the sweep phase will delete all remaining proxies, which triggers the transition from the *forwarding* state into the *finished* state and Y' ceases to exist.

7.3 Access Path Optimization

The protocol described so far leaves chains of proxies, which lead to potentially high access latencies if an object frequently migrates but is seldom accessed. This observation was supported by the OMNeT++ emulator and the software simulation from Chapter 9.

Therefore, I developed an access path optimization approach that shortens the access latency to a frequently migrating object that left a long chain of proxies behind. This optimization approach is especially suited for a system that uses the DecentSTM protocol, where each write operation potentially results in an implicit migration of an object from one node to another.

The so far described reactive location update protocol establishes a chain of proxies for each migrating object. So, each proxy Y'_i in a chain of proxies holds a forwarding pointer to the next proxy Y'_{i+1} in the proxy chain. Additionally, for each migration, DecentSTM implicitly creates a backward reference between the object's head version and its previous version, i. e. from an object to its previous proxy. Thus, each proxy Y'_{i+1} also knows its predecessor, the node that stores proxy Y'_i. The reactive location update protocol does not need these backward pointers, but the *DecentSTM* protocol has to be able to walk the history of an object.

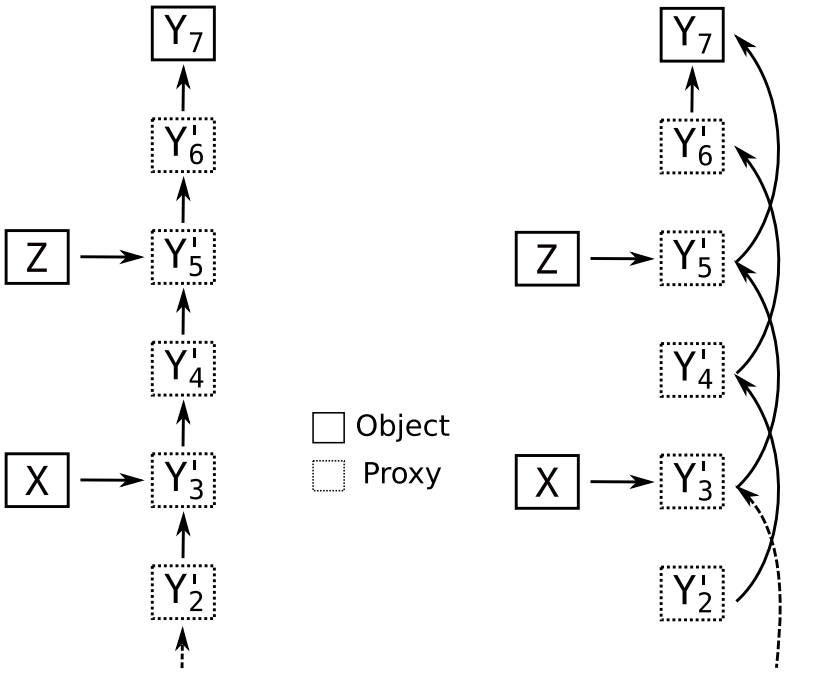
The access path optimization approach uses these backward pointers to propagate updated object location information backwards along the proxy chain, from the target to the source.

To describe this update mechanism, suppose that an object Y comes into existence with a migration sequence number of $m = 0$ and migrates m times (with DecentSTM, m is equivalent to the number of implicit migrations, i. e. the number of write operations that create a new head version on another node than the previous home node). If the system wants to update the forwarding information of all previous proxies, it needs to send the update message along m hops, one hop for each step along the chain.

Suppose two additional objects, X and Z, which received a reference to Y at the migration steps $m = 3$ and $m = 5$, respectively. After $m = 7$ migrations, the established proxy chain looks like the one presented in Figure 7.6(a). This figure shows that the length ℓ of the proxy chain changes with the viewpoint and depends on the time when the reference to the migrated object was created, i. e. how deeply in the proxy chain the accessing object begins its way up to N_Y. In the example, the length ℓ is for object X, $\ell = 4$, and for object Z, $\ell = 2$.

Now, suppose the system sends the location update messages for Y instead of m hops, only k hops down the proxy chain. As a result, the update overwrites the location information of object Y at the k previous proxies. [2]

[2] Note that DecentSTM requires that each proxy at least keeps the location information of its successive proxy to be able to walk the complete version history of an object. Furthermore, a proxy can keep the location information of all its k successor proxies. This adds additional paths, along which the referenced object can be accessed, and thus, hardens the proxy chain against node failures. However, reliability and fault tolerance is not in the scope of this thesis.



(a) Depth $k = 1$. (b) Depth $k = 2$.

Figure 7.6: Update propagation example with a propagation depth of $k = 1$ and $k = 2$.

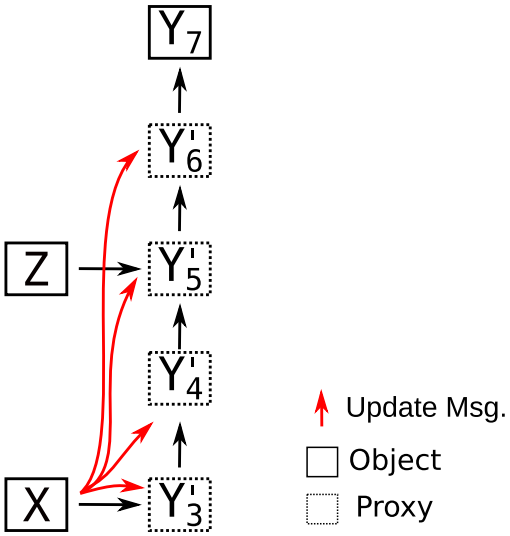
The result is that the number of hops needed to access an object along a proxy chain of length ℓ decreases from $\ell + 1$ to $\lceil \frac{\ell}{k} \rceil + 1$ hops. See, for example, Figure 7.6(b), which shows the result of a propagation depth of $k = 2$. In this example, object Z reaches Y after 2 hops (instead of 3 hops with $k = 1$), and object X reaches object Y after 3 hops (instead of 5 hops with $k = 1$).

The optimal propagation depth k depends on the access characteristics of the corresponding object. Namely, on the ratio of the number of read accesses R, the number of migrations M, and the established proxy chain's length ℓ, that a read message has to traverse to reach the home node of the object. This length depends on both, the read-to-write ratio and the garbage collection rate, because a garbage collection run removes all proxy chains present in the system. Hence, to optimize the costs of an object access, the number of update message hops per migration plus the number of access message hops per read, or $kM + \frac{\ell}{k}R$, must be minimal, i. e. $M - R\ell/k^2 = 0$ or

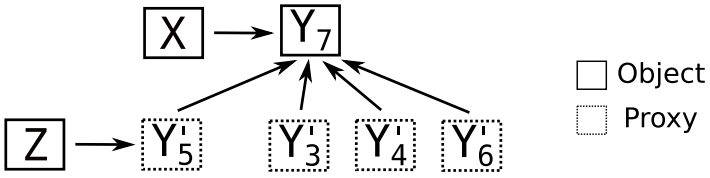
$$k = \sqrt{\frac{R \cdot \ell}{M}} \tag{7.3}$$

Moreau and Ribbens [MR02] developed a middleware for mobile agents that uses chains of proxies as well. The authors describe an *Eager Acknowledgments* mechanism that propagates the new location of a mobile agent down the proxy chain to all previous proxies. Thus, their approach can be seen as a special case of my approach with $k = M$. However, Moreau and Ribbens do not consider the access characteristics of a mobile agent at all.

Fowler [Fow86] describes another access path optimization algorithm that updates a chain of proxies after a successful object access. Here, a node that learns a new location of an object *Y* because of the object access, propagates this knowledge upwards in the chain of proxies. Thus, all proxies in the chain learn the new location, and all subsequent accesses from all objects that reference object *Y* profit from this update, cf. Figure 7.7.



(a) Updates After Access from Obj. X.



(b) Proxies After Update Messages.

Figure 7.7: Path optimization after Fowler [Fow86]: A node that learns a new location of an object *Y* because of the object access, propagates this knowledge upwards in the chain of proxies.

Unlike my approach, Fowler always updates all proxies in the proxy chain after each object access that traversed a chain of proxies. Furthermore, Fowler updates the proxies upon object access, while I update the proxies upon object migration. Thereby, my approach avoids that the first access after a number of migrations must traverse the whole chain of proxies, and additionally, avoids the case where multiple nodes traverse and try to update the chain of proxies at the same time, as it might happen in Fowlers approach.

8 Proactive Location Update with Incoming References Protocol

The reactive location update approach from Chapter 7 has two potential drawbacks.

First, the length of proxy chains is potentially unbound because each migration adds a proxy. Even with garbage collection it might happen that an object access has to traverse a long chain of proxies, e. g. if the object is seldom accessed, but frequently migrates, and even though the length of a proxy chain is decreased by the access path optimization from Section 7.3, the basic problem remains unresolved.

Secondly, the system cannot delete proxies immediately when they are not needed anymore, but has to wait for the garbage collector: If objects are accessed frequently and the location information is almost always up to date, the un-referenced proxies still remain in the system and occupy memory until the next garbage collection run.

The hereafter presented *proactive location update* protocol solves these problems. It is based on the basic reactive location update protocol, but adds enhancements that update invalid location information directly after an object migration.

In practice, it is impossible to ensure the timely update of location information in a distributed system. Thus, one wants to use a combination of both: proxies to forward requests and update messages to eliminate the proxies eventually.

Similar to the access path optimization, this approach requires additional backward references, the so-called *incoming references* (*InRefs*) of an object. In contrast to the backward references that the DecentSTM protocol uses to reach the previous object version (or previous proxy), these incoming references are the backward references that belong to regular object references. Thus, unlike with the access path optimization, the number of backward references is potentially unbound.

An *incoming reference (InRef)* is the backward reference of an *outgoing reference (OutRef)*: If object X contains a reference to object Y, the backward reference points from the referenced object Y back to the referencing object X.

Suppose an object Y on node $B = N_Y$ is referenced by multiple objects that reside on node A, for example, $A = N_{X,Z}$. Thus, object Y would contain two incoming references, one per object, cf. Figure 8.1(a).

Because node A consolidates all outgoing references to the same object in one GaoMap entry, it is sufficient when the proxy node $B = N_{Y'}$ informs the **node** A in case Y migrates, and not the **objects** X and Z, cf. Figure 8.1(b). This is sufficient because node A only has to adapt its GaoMap entry for Y, without the need to touch X or Z. Therefore, incoming references are the counterparts of GaoMap entries. Furthermore, each object stores its incoming references on a per-node basis, and not on a per-object basis, in its so-called *incoming reference list*, cf. Section 5.3. Thus,

(a) One incoming reference per Object. (b) One incoming reference per Node.

Figure 8.1: Incoming references per object or per node. In Figure 8.1(a) object *Y* contains two incoming references, one per referencing object, while in Figure 8.1(b), object *Y* contains only one incoming references per referencing node.

whenever an incoming reference is in the following added to an object *Y*, it is stored in this incoming reference list.

The main idea of the proactive location update approach is that the proxy node sends update messages along all incoming and outgoing references of the migrated object immediately after the migration is successfully completed. This approach has some advantages: all location information are (almost) always up to date and access messages are sent directly to the migrated object with no (or only a few) proxies in between. Moreover, when the proxy node knows that the proxy is not referenced anymore, the proxy node can delete it. The disadvantage of this approach is the increased management and message overhead that is needed to update the remote location information.

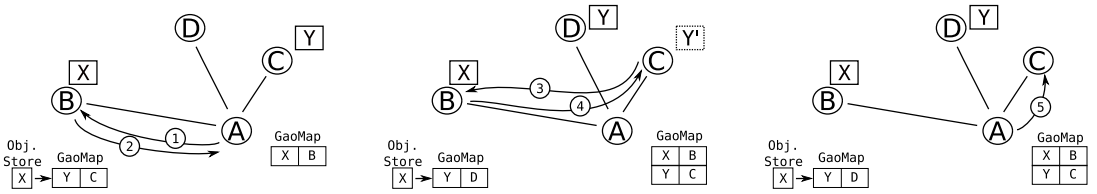
Day et al. [Day+93] proposed a similar update mechanism for their object-oriented database system *Thor*, cf. Section 3.4. Thor does not separate the reference to an object from the location of the object as my approach does. Hence, Thor does not introduce a mapping such as the GaoMap presented in this thesis, and all objects hold location dependent references.

Upon an object migration, a Thor proxy node sends the new location dependent reference as update messages to all nodes that hold objects that reference the migrated object. These updates are collected until the next garbage collection run, during which the garbage collector updates all outdated references. This has two disadvantages I avoid in my approach: first, the garbage collector has to examine all objects to identify the outdated references, and secondly, the location update depends on the frequency of the garbage collector runs.

Day et al. concluded that this update scheme is applicable whenever it is important to reach an object quickly, even though this depends on the GC frequency. However, they did not measure the maintenance message overhead of this scheme, as done in this thesis.

8.1 Object Access and Proactive Location Update

The proactive location update protocol tries to delete proxies as soon as possible, which can lead to inconsistent location information if the update process overlaps with ongoing PUT and GET operations. In this case it can happen that a newly created incoming reference is not established in time, and the corresponding proxy is deleted too early. Figure 8.2 shows an example of this process. (The used *InRef Establish* and *InOut Notify* messages are described in more detail later on.)



(a) Node B/A: GET Reference operation.

(b) Node C: InRef notification message.

(c) Node B: InRef establishment message.

Figure 8.2: Reason for a failed InRef establishment. Because the location update process overlaps with an ongoing GET operations, the newly created incoming reference from node *A* is not established in time, and the corresponding proxy Y' is deleted too early.

In this example, object X on node B, represented as tuple (X,B), holds a reference to an object Y on node C, represented as tuple (Y,C). In the first step, shown in Figure 8.2(a), node A reads the reference to Y from X with a GET Request message ①. Node $B = N_X$ responds with the reference ② and the current location information (Y,C). Before node A sends the required *InRef Establish* message ⑤ to node $C = N_Y$, node C migrates Y to node D and becomes $C = N_{Y'}$, see Figure 8.2(b). Moreover, C sends an *InOut Notify* message ③ to node B. When the acknowledgment message ④ from B arrives at node C, C deletes Y'. When the *InRef Establish* message ⑤ from node A finally arrives at node C, this establishment fails and node A has lost the location information of Y.

To prevent this failure, I introduce the *triangular access approach*. To clarify the usage of the different messages that are used in the following sections, I describe the layout of the different message fields. The common invariant of these messages are the first three fields:

```
Msg{Source Node,
    Destination Node,
    Message Type,
    ....
   }
```

These are followed by the individual fields of the corresponding message class.

8.1.1 Triangular Object Access

The regular triangular access approach uses *Triangular-GET* and *Triangular-PUT* messages.

In contrast to regular GET operations, which follow a simple request-response protocol, Triangular-GET operations follow a three step protocol that introduces an additional message forwarding. For this, the *GET Request* message is split into two parts, a *GET1 Request* and a *GET2 Request*: First, node A sends a *GET1 Request* message to node B, cf. Figure 8.2(a):

```
Msg{Src = A,
    Dst = B,
    Type = GET1 Request,
    Operation = Read reference field from object X
   }
```

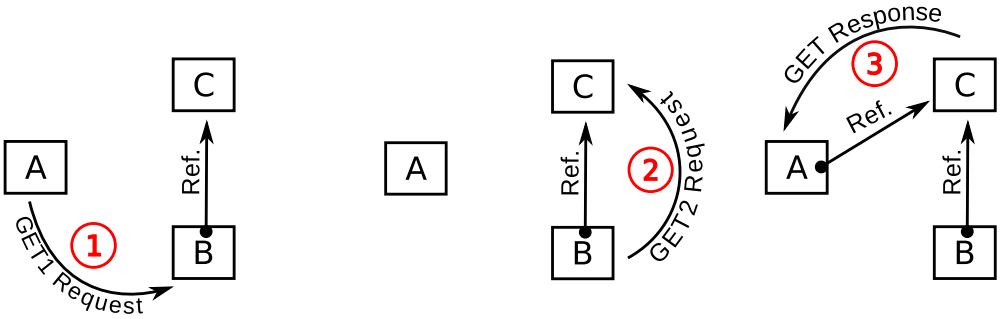
In the example, node *B* does not answer the *GET1 Request* message ① (Figure 8.2(a)) directly, but forwards the message as a *GET2 Request* message to node *C*:

```
Msg{Src = B,
    Dst = C,
    Type = GET2 Request,
    Operation = Add incoming reference for node A to object Y
                send reference Y as GET Response back to node A
   }
```

This message notifies node *C* to add a new incoming reference to object *Y* and to send the response message that contains the reference to *Y* back to node *A*.

Thus, node $C = N_Y$ can timely add the new incoming reference for node *A* to *Y*. In the next step, *Y* migrates to node *D* and node *C* becomes the proxy node $C = N_{Y'}$ and sends the *InOut Notify* message ③ (Figure 8.2(b)) to *B*. Now, suppose the acknowledgment message ④ (Figure 8.2(b)) from node *B* arrives at node *C*. Hence, node *C* deletes the incoming references for node *B* from Y' but does not delete Y' because there is the additional, not yet updated, incoming reference from node *B*.

Furthermore, this has the advantage that node *A* receives an updated location information not only for object *Y*, but also for the read object *X*, which resides on node *B*.



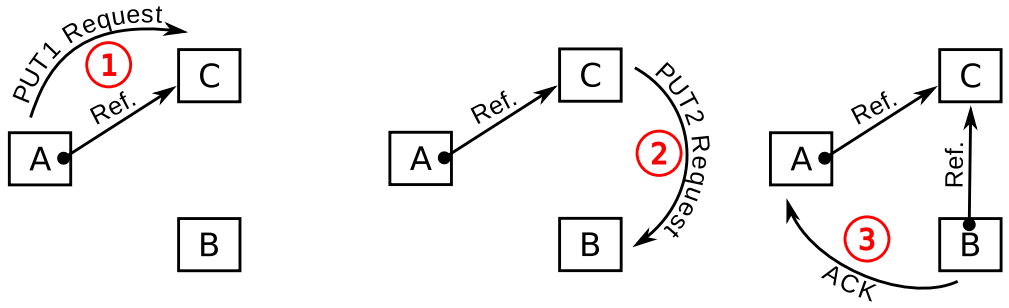
(a) Send GET1 Request.

(b) Send GET2 Request to Referenced Object.

(c) Referenced Object sends GET Response to Reader.

Figure 8.3: Triangular-GET Operation, which follows a three step protocol that introduces an additional message forwarding. The *GET Request* message is split into two parts, a *GET1 Request* and a *GET2 Request*. Node *B*, where the reference is read from an object, forwards the *GET1 Request* as a *GET2 Request* message to the node where the referenced object resides.

Figure 8.3 shows the timing sequence and message flow of this Triangular-GET access. The advantage of this process is that the requesting node not only receives the read reference but also the latest location information of the corresponding object. The disadvantage is that the access latency is increased.



(a) Writer send PUT1 Request to References Object Y.

(b) N_Y sends PUT2 Request to Home of Written Object N_Z.

(c) N_Z sends ACK Message back to Writer.

Figure 8.4: Triangular-PUT operation where the *PUT1 Request* is not send to node B where the reference is written, but to node C where the referenced object resides.

The Triangular-PUT is similar: Suppose an object Y is located on node $C = N_Y$, and node A holds a reference to Y, that it writes into an object Z on node $B = N_Z$. Instead of node A sending the *PUT1 Request* message to node B, the writing node A sends the *PUT1 Request* message to node C, cf. Figure 8.4(a):

```
Msg{Src = A,
    Dst = C,
    Type = PUT1 Request,
    Operation = Add incoming reference for node B to object Y
                send PUT2 Request to node B to write reference
                to object Y into object Z
   }
```

Node C adds the new incoming reference from node B to Y and sends the *PUT2 Request* message to B:

```
Msg{Src = C,
    Dst = B,
    Type = PUT2 Request,
    Operation = Write reference to object Y into object Z
                Send ACK to node A
   }
```

Upon the reception of this forwarded *PUT request* message, node B writes the reference to Y into Z, cf. Figure 8.4(b), and stores the latest location information of Y. Afterwards, node B sends the ACK message back to the writing node A, see Figure 8.4(c).

Even though this Triangular access ensures the timely creation of incoming references, the main disadvantage is the increase access latency which is due to the indirection of the request messages. Therefore, I developed the enhanced triangular access approach.

8.1.2 Enhanced Triangular Object Access

The enhanced triangular access approach circumvents the increased access latency of the regular triangular access approach.

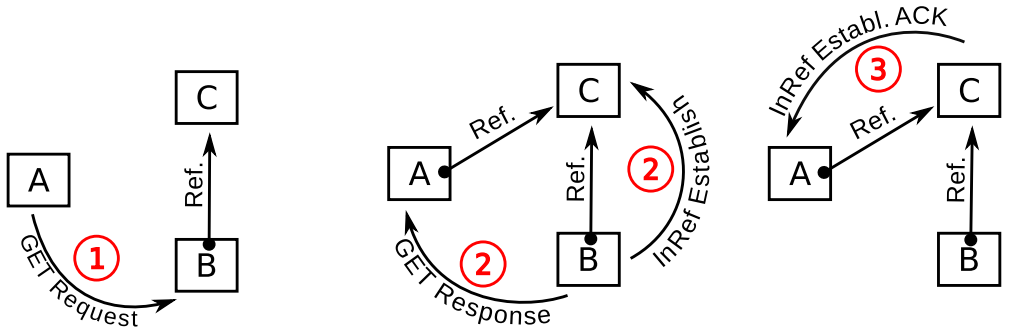
Suppose a node B holds an object X, $B = N_X$, that references an object Y on node $C = N_Y$, cf. Figure 8.5. In the enhanced Triangular-GET approach, the accessing node A sends the *GET Request* message to node B to read a reference field of X, cf. Figure 8.5(a):

```
Msg{Src = A,
    Dst = B,
    Type = GET Request,
    Operation = Read reference field from object X
   }
```

As a response, node B sends two messages: A regular *GET Response* message back to node A, which contains the read reference to Y together with Y's location information, and an additional *InRef Establish* message to node C (described in more details in 4), cf. Figure 8.5(b).

Node C receives the *InRef Establish* message and adds the new incoming reference to Y. Afterwards, node C acknowledges the reception, but sends the ACK message to node A, see Figure 8.5(c). Thus, a potential proxy Y' on $C = N_{Y'}$ cannot be deleted too early, because there exists at least the newly established incoming reference from node A that must be updated before the deletion.

The ACK message informs node A about the successful *GET Request* message. If node A does not receive this ACK message within a given timeout period, it has to re-send the request message to ensure that the new incoming reference is correctly established, but the execution of the application on node A may proceed immediately when the *GET Response* message is received.



(a) Send GET Request.

(b) Send GET Response and InRef Establish.

(c) Send InRef Establish ACK.

Figure 8.5: Enhanced Triangular-GET operation. Node B, where the object is read, not only sends a regular *GET Response* message back to node A, but also an additional *InRef Establish* message to node C to timely establish the new incoming reference.

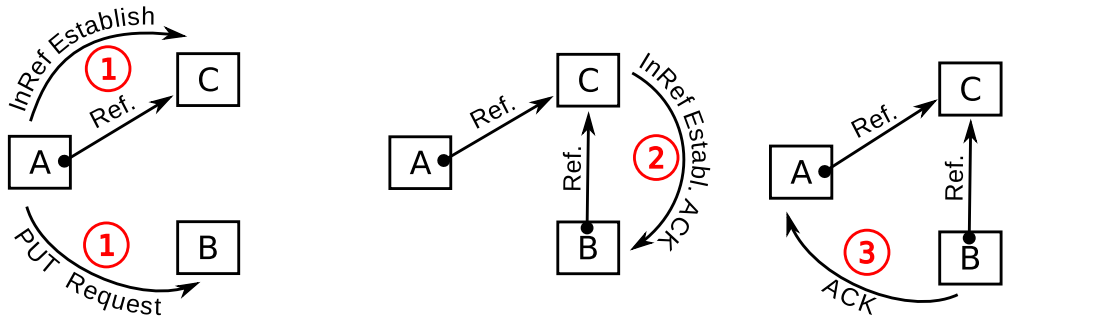
The enhanced Triangular-PUT approach is similar. Suppose again, that node A holds a reference to Y on node $C = N_Y$ that is written to an object X on node $B = N_X$. This time, node A not

only sends the *PUT Request* message to node *B*, but also the *InRef Establish* message to node *C*, cf. Figure 8.6(a):

```
Msg{Src = A,
    Dst = B,
    Type = PUT Request,
    Operation = Write reference to object Y into object X
   }
```

When node *B* receives the *PUT Request* message, it writes the reference to *Y* into *X*, but does not acknowledge the write operation. When node *C* receives the *InRef Establish* message, it adds the new incoming reference to *Y* and sends the *InRef Establish ACK* message to node *B*, cf. Figure 8.6(b). Upon the reception of this *InRef Establish ACK* message, node *B* acknowledges the success of the *PUT Request* message by sending the acknowledgment message to node *A*, cf. Figure 8.6(c). If node *A* does not receive this ACK message, it must restart the PUT operation.

Compared to the regular triangular approach, the advantage of this enhancement is the decreased access latency, because the *GET Response* message is sent back immediately and the *PUT Request* message reaches the accessed object immediately. The disadvantage is the increased cost for the additional *InRef Establish* messages.



(a) Send PUT Request and InRef Establish.

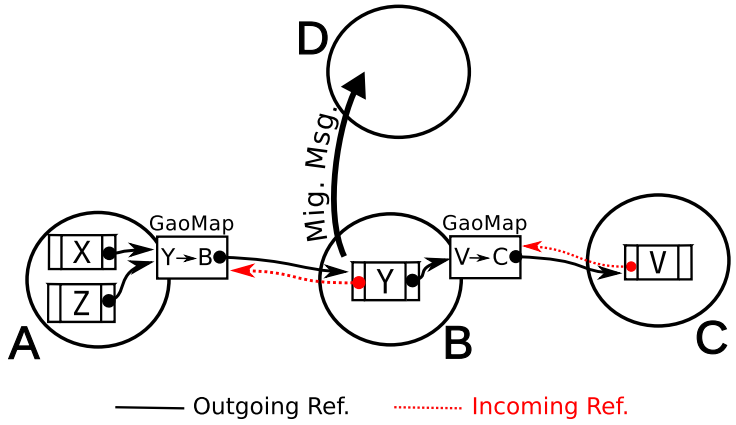
(b) Send InRef Establish ACK.

(c) Send PUT ACK back to Writer.

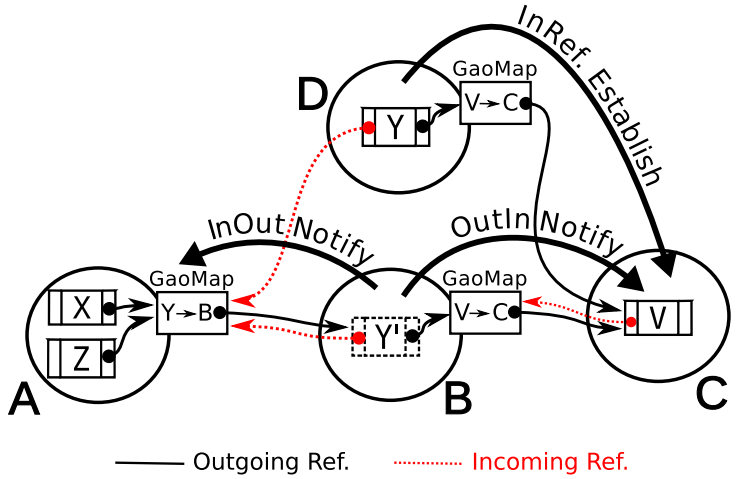
Figure 8.6: Enhanced Triangular-PUT operation. The writing node *A* not only sends the *PUT Request* message to node *B*, but also the *InRef Establish* message to node *C* to timely establish the new incoming reference.

8.2 Differences to Reactive Location Update Approach

Similar to the reactive location update protocol, the proactive location update protocol uses proxies to forward all messages to the new home node of an object. However, the proxy node not only forwards messages to the home node, but additionally updates all outdated location information in the system that still point to the proxy. When all these references are updated, the proxy is not used anymore and the proxy node can delete it. Therefore, the definition of a proxy node is different for the proactive location update protocol:



(a) Object Y migrates to node D and leaves proxy Y'.



(b) Proxy Y' sends Notify Msg's.

Figure 8.7: After object Y migrated from node B to node D, the remaining proxy Y' sends the InOut and OutIn Notify messages along the corresponding incoming and outgoing references.

Definition 4 *The* **proxy node** *is authorized to forward all messages to the node to which the proxy's forwarding pointer points. Furthermore, the proxy node is authorized to propagate the new location information of the object when the migration has succeeded, and also upon the subsequent reception of access or maintenance messages.*

Therefore, the proxy node updates all incoming and outgoing references that the migrated object contained at the time of the migration, and all references that are established in-between the moment when the proxy came into existence and the moment the proxy is deleted. For this update, the proxy node has to hold additional state information about all incoming and outgoing references of the migrated object. This state is necessary to keep track of the update status of the corresponding references, i. e. if the sent notification message was already acknowledged or is still pending.

According to this definition, $N_{Y'}$ handles the reference maintenance, and not N_Y, even though N_Y could perform the reference maintenance tasks as well, and afterwards signals $N_{Y'}$ that Y' is not needed anymore.

However, N_Y is not authorized to make assumptions about N'_Y. Furthermore, $N_{Y'}$ is the node where other nodes with outdated location information for object Y send their messages to. Thus, $N_{Y'}$ knows first about such outdated location information, and can immediately respond with an update message. Moreover, the goal is to delete Y' as soon as possible, which is best done when $N_{Y'}$ keeps track of all outdated references that still lead to $N_{Y'}$.

To bring a proxy node into the position to update incoming and outgoing references, the proactive location update protocol defines two additional classes of *reference maintenance* messages.

Reference Maintenance Message Classes

This section describes the additional classes of *reference maintenance* messages that are needed for the proactive location update protocol. It starts with the description of the messages that are needed for the incoming reference establishment and removal. Afterwards, I describe the messages that a proxy node $N_{Y'}$ has to send to inform all incoming and outgoing references about the migration of an object.

I will use the example from Figure 8.7 to explain first, the incoming reference establishment and removal, and secondly, the different messages that a proxy node sends when an object Y migrates from node $B = N_Y$ to node D.

InRef Establishment and Removal The first message class contains the *InRef Establish* message, which announces the establishment of a new incoming reference, and the *InRef Remove* message, which announces the removal of an incoming reference. Both messages address a particular object V on node $C = N_V$ to either add or remove an incoming reference to/from V.

An *InRef Establish* message is sent from node D to node C for two reasons: first, in the enhanced triangular GET approach described above, when node D receives a *Get Request* message that reads a reference field from a local object, for example, from object Y.

Suppose than in Figure 8.7(b) the node A reads the reference field of object Y on $D = N_Y$, which contains a reference to V that is located on node $C = N_V$. In this example, node D would send the following *InRef Establish* message to inform V on node C about the new incoming reference from node A:

```
Msg{Src. = D,
    Dst. = C,
    Type = InRef Establish,
    Operation = Add incoming reference for node A to object V
    Send ACK to node A
   }
```

Secondly, when node D in the example of Figure 8.7(a) receives the migrating object Y and becomes $D = N_Y$. Suppose Y contains a new outgoing reference to object V on node $C = N_V$. Then, D sends the following *InRef Establish* message to node C:

```
Msg{Src. = D,
    Dst. = C,
    Type = InRef Establish,
    Operation =  Add incoming reference for node D to object V
   }
```

Here, the *InRef Establish* message is optional, because the proxy node $B = N_{Y'}$ has sent an *OutInNotify* message (see description below), which already informed node C about the new incoming reference. However, node D might send this message to ensure a faster update of its location information for V, in case V migrates to another node. In this case, $C = N_{V'}$ would send the location update message directly to node D, which otherwise would first have to traverse node B (see *InOut Notify* messages below).

When one of these *InRef Establish* messages reaches node C, and the incoming reference contained in this message is new to object V, node C adds the incoming reference to V and sends an acknowledgment. If the incoming reference has been established already, node C only sends the acknowledgment.

An *InRef Remove* message is sent, for example, from node B to node C to inform node C that the corresponding incoming reference from node B can be deleted from e. g. object V. Node B sends this message because the reference counter for the outgoing reference to object V dropped to zero. This happens, for example, if the reference field in an object Y, which contained the last outgoing reference to V, is overwritten with a new reference, or if object Y is deleted on node B.
In both cases, node B creates the following *InRef Remove* message and sends it to node C:

```
Msg{Src. = B,
    Dst. = C,
    Type = InRef Remove,
    Operation = Remove incoming reference for node B from obj. V
   }
```

If object Y migrates and takes the last outgoing reference to another node, node $B = N_{Y'}$ handles the incoming reference removal with an *OutIn Notify* message (see description below).

InRef and OutRef Update Notification The second message class is used to maintain the outdated incoming and outgoing references when the proxy Y' comes into existence on e. g. node B. These reference maintenance messages transport the same information as the *InRef Establish* and *InRef Remove* messages. However, they are an individual class because the reason why they are sent differs. Furthermore, a node has to handle the reception of these reference maintenance messages differently, as well, which will be seen in the description of the state diagrams below.

When the migration of Y to node D in the example of Figure 8.7(b) is completed, node B becomes the proxy node $B = N_{Y'}$. Afterwards, node B is authorized to forward all messages for Y

to node D and to update all incoming and outgoing references that object Y held at the time of the migration. Furthermore, node B is authorized to update all new references that are announced to node B due to outdated location information.

For this update, node B sends *InOut Notify* and *OutIn Notify* messages, where the prefix defines the direction in which the update message is sent: either from an incoming reference entry to the corresponding outgoing reference (*InOut Notify*) or vice versa.

An *InOut Notify* message informs a **node** A about the new location of an object that is referenced from node A by one or more objects.

In the example from Figure 8.7(b), node B sends the *InOut Notify* message along the incoming reference to node A to notify A that Y has migrated to node D. Node A takes this information and adapts the location information of the corresponding outgoing reference entry in its GaoMap, cf. Figure 8.7(b). For this update notification, node B sends the following *InOut Notify* message:

```
Msg{Src. = B,
    Dst. = A,
    Type = InOut Notify,
    Operation = Object Y now located on node D
                update location information
   }
```

The *OutIn Notify* message exists in two versions, which both inform an **object** V on node $C = N_V$ about a new incoming reference from a node D. The regular *OutIn Notify* message only announces the new incoming reference and looks like this:

```
Msg{Src. = B,
    Dst. = C,
    Type = OutIn Notify,
    Operation = Add incoming reference for node D to object V
   }
```

The enhanced *OutIn Notify Remove* message additionally announced the deletion of an incoming reference. This removed incoming reference indicates that the migrating object took the last outgoing reference to the corresponding object to the new home node during the migration. Therefore, a node sends the following *OutIn Notify Remove* message:

```
Msg{Src. = B,
    Dst. = C,
    Type = OutIn Notify Remove,
    Operation = Add incoming reference for node D to object V
                Remove incoming reference for node B from obj. V
   }
```

In the example from Figure 8.7(b), node $B = N_{Y'}$ sends an *OutIn Notify Remove* message to node $C = N_V$, so that C adds the new incoming reference from node D to V. Furthermore,

node *C* deletes the incoming reference from node *B*, because *B* does not hold further objects that reference *V*.

In pseudo code the process on node *B* looks like this:

```
success = migrate(Y -> D)
if ( success ) {
  send Msg{B, A, InOut Notify, Obj. Y on node D, update}
  send Msg{B, C, OutIn Notify Remove,
           Add InRef D to V
           Remove InRef B from V}
  wait for ACKs
}
```

Both, the *InOut Notify* and the *OutIn Notify Remove* messages are acknowledged from node *A* and node *C*, respectively. When the proxy node $B = N_{Y'}$ receives the last of these acknowledgments, it can delete Y' because it is not needed anymore.

As mentioned above, Figure 8.7(b) shows the optional *InRef Establish* message that node *D* sends to node *C*. As stated above, this *InRef Establish* message transports nearly the same information as the *OutIn Notify Remove* message sent by node *B*, but ensures the faster update of the GaoMap entry on node *D* in case that *V* migrates to another node.

8.3 Object State Diagram

The following section describes the state diagrams for the proactive location update protocol. Because this protocol is based on the reactive location update protocol, the state diagram from Figure 7.5 stays valid. The following sections only present the additional state diagrams and transitions that are new for the proactive location update protocol.

Again, the arrows in the state diagram depict the transitions, while the labels above the arrows indicate the cause for the transition, and the labels below the arrows indicate the effect that the transition has on the node or the corresponding object.

The state diagrams are divided into three parts: Figure 8.8 shows the transitions for runtime operations, Figure 8.9 shows the transitions for the object migration, and Figure 8.10 shows the transitions for the reference management.

Note that I combined the two triangular access approaches in the state diagrams. The regular triangular access is mainly handled in the state diagram for the runtime operations (Figure 8.8). Conversely, the reference management state diagram (Figure 8.10) shows the *InRef Establish* messages, which are mainly sent during the enhanced triangular access.

8.4 State Diagram: Runtime Operations

The state diagram for the runtime operations describes the handling of remote GET and PUT messages, while some of the transitions influence the reference management. For example, does a PUT operation that overwrites a reference field result in the sending of an *InRef Remove* message.

Furthermore, a GET2 or PUT1 message causes a proxy to send an *InOut Notify* message (see description below).

8.4.1 Regular State

This section describes the different state transitions of an object in the *regular* state.

The creation of a new object with a *NEW Object Event* causes the home node to add an initial incoming reference from the local node to the object. This incoming reference indicates that the local execution context holds a reference to the object on its stack.

A node N_Y that receives a *GET1 Request* from node A reads a reference field from the accessed object Y, which I suppose, contains a reference to object V. If N_Y is equal to N_V, N_Y sends the *GET Response* message back to node A. If N_Y and N_V are not equal, N_Y forwards the request as a *GET2 Request* to node N_V. When N_V receives this *GET2 Request* message, it inserts the new incoming reference from the accessing node A to object V and sends the *GET Response* message back to node A.

Node A either adds the received outgoing reference to its maintenance maps or, if the outgoing reference is already present, increments the reference counter. If the received location information of object V is newer than the one in the *GaoMap*, node A updates its *GaoMap* entry. Afterwards, the execution of the currently blocked thread on node A, which read the reference, is resumed.

A node N_Y that receives a *PUT1 Request* adds the new incoming reference for the announced node N_V to the incoming reference list of Y. If N_Y is not equal to N_V, N_Y forwards the *PUT1 Request* as a *PUT2 Request* message to N_V.

When node N_V receives this *PUT2 Request* message, it writes the new reference to Y into the corresponding reference field of the accessed object V and sends the *PUT ACK* message back to node A, which is the node that initiated the PUT operation. If a PUT operation overwrites a reference field, it might happen that the reference counter of the overwritten reference to e. g. object Z, drops to zero. When N_V is not equal to N_Z, N_V sends an additional *InRef Remove* message to N_Z.

Upon the reception of the *PUT ACK* message, node A checks if the message contains newer location information for the objects Y or V and potentially updates its GaoMap. Afterwards, the PUT operation is finished.

The transition of an object Y from the *regular* state into the *finished* state has the result that the reference counter of all of Y's outgoing references are decremented. If one of the corresponding reference counters for a reference to e. g. object X drops to zero, N_Y sends an *InRef Remove* message to N_X and waits for the acknowledgment (handled in Figure 8.10).

8.4.2 Pending State

Similar to the reactive location update approach, an object in the pending state Y^* causes the optimistic forwarding of all access messages to N_Y.

Furthermore, each *PUT1 Request* and *GET2 Request* message causes N_{Y^*} to add a new incoming reference to Y^*. This incoming reference is necessary because the access message

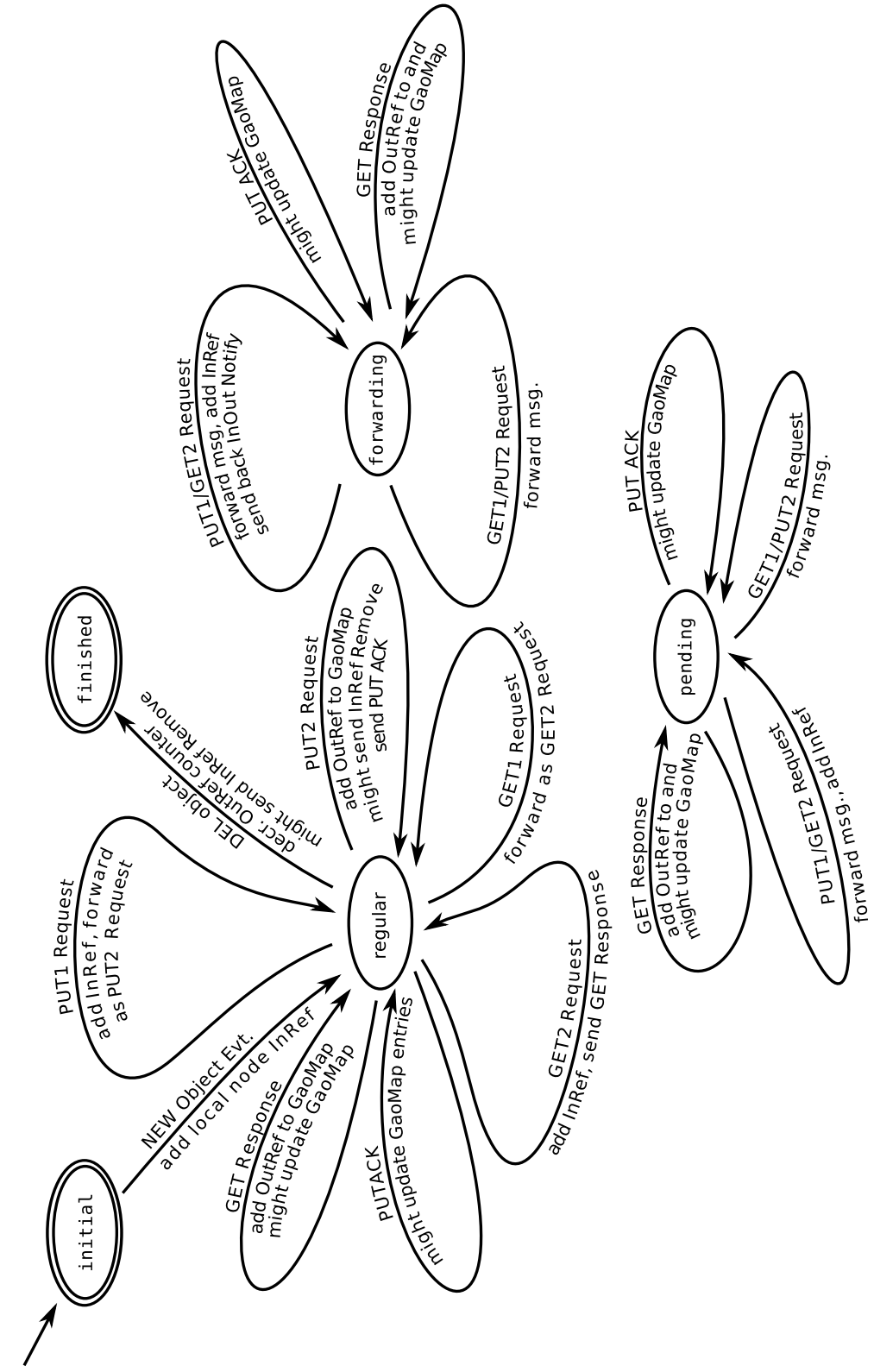


Figure 8.8: State diagram for the runtime operations that describes the handling of remote GET and PUT messages, while some of the transitions influence the reference management.

indicates that there will be an additional object that holds a new outgoing reference to Y^*, and which must be informed when the migration succeeded. However, N_{Y^*} is not allowed to send an update message, because the migration is not yet completed.

The envisioned system is currently supposed to not support thread migration. Hence, *PUT ACK* and *GET Response* messages are not forwarded because only an execution context can initiate PUT and GET operations and wait for the result. Thus, response and acknowledgment messages always terminate on the same node that initiated the request. They do not need to be forwarded because such a situation cannot occur.

If thread migration was supported, a migrating thread would also leave a pending object behind, which forwards these messages to the new location of the thread.

8.4.3 Forwarding State

The node $N_{Y'}$ always forwards all access messages to N_Y. Similar to the *pending* state, it is not sufficient to only forward *PUT1 Request* or *GET2 Request* messages, but $N_{Y'}$ has to add a new incoming reference from the sending node A to Y'. This additional incoming reference is necessary to prevent the heady deletion of Y', because the received message indicates that there is a node A in the system that currently holds outdated location information for object Y. However, in contrast to the *pending* state, $N_{Y'}$ is authorized to immediately send an *InOut Notify* message back to the requesting node A to announce the new location of Y, similar to the process shown in Figure 8.7(b). When node A acknowledges this message, $N_{Y'}$ marks the corresponding incoming reference as updated.

This procedure results in the sending of more than one location update message if the access message traverses multiple proxies on its way to N_Y, because each $N_{Y'}$ that is encountered on the way sends an *InOut Notify* message. This process increases the message overhead of the protocol, but is necessary to keep the location information consistent, to prevent the premature deletion of proxies, and to update the outdated location information that was used to send the request message. However, the number of *InOut Notify* messages is limited because proxy chains in this protocol are short, as we will see in Chapter 9.

Similar to the *pending* state, *PUT ACK* and *GET Response* messages are not forwarded as long as no thread migration is supported. See above: This cannot occur.

8.5 State Diagram: Migration

The state diagram for the migration process, see Figure 8.9, describes the different transitions that an object can encounter while a migration process takes place. It is the only state diagram in which transitions between the three states *regular*, *pending* and *forwarding* occur.

8.5.1 Regular State

The migration process starts when an object Y is scheduled for an explicit object migration, or if an implicit object migration occurs because the DecentSTM protocol created a new object version on a remote node.

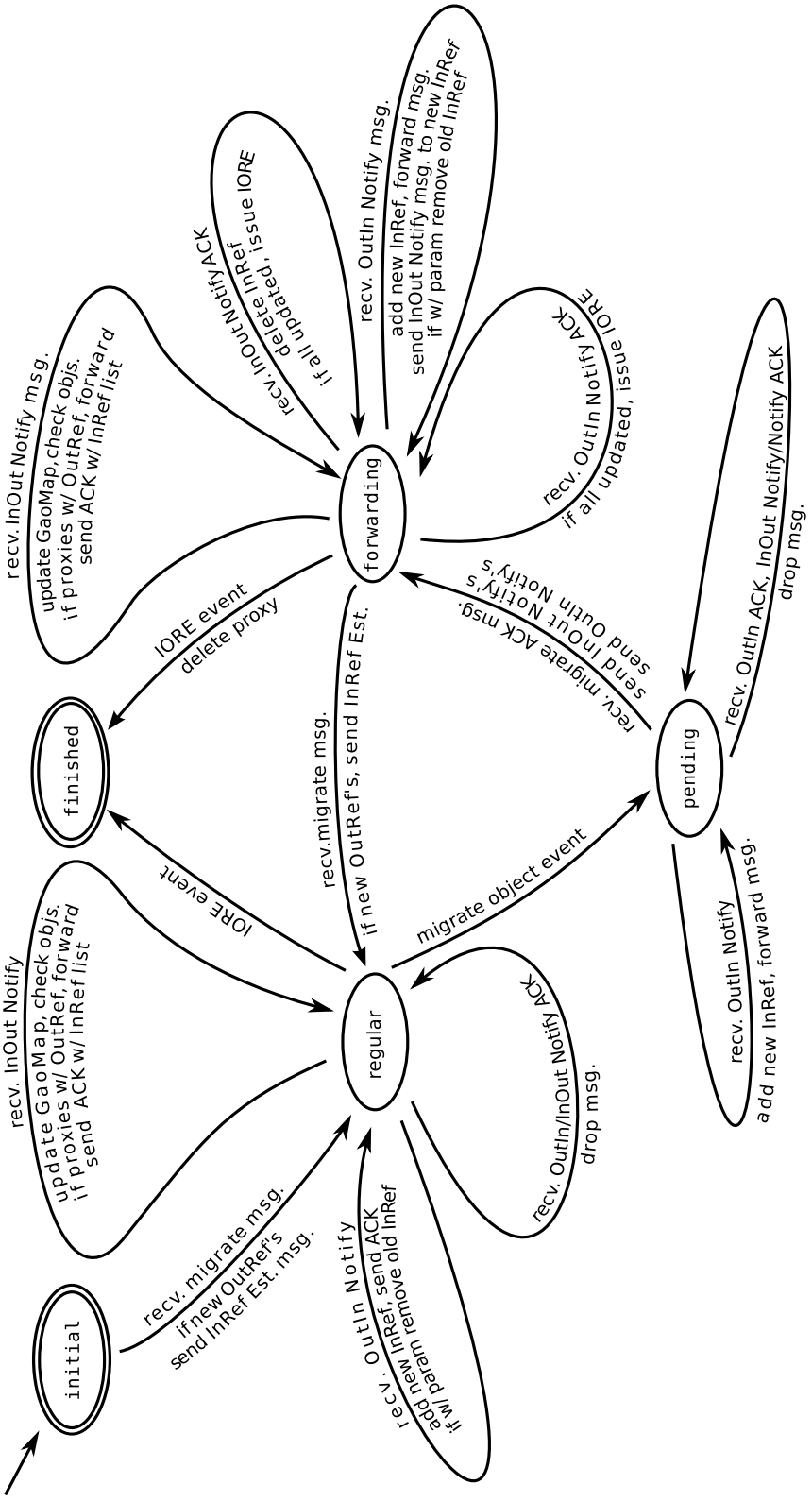


Figure 8.9: State diagram for the migration process, which is the only state diagram in which transitions between the three states *regular*, *pending* and *forwarding* occur.

With both, implicit and explicit migration, a new object Y comes into existence on some node A. If node A accepts this migration, it becomes the new home node $A = N_Y$. Hence, node A adds the object Y to its local memory and inserts all outgoing references that Y contains, e. g. to another object V on node C, to the management maps, or updates the already existing entries. As described above, node A might send an optional *InRef Establish* message to $C = N_V$ if the corresponding outgoing reference is new.

If there is currently a proxy Y' on node $A = N_{Y'}$, this proxy is replaced by object Y before node A becomes $A = N_Y$.

If a node A receives an *InOut Notify* message from a proxy node $B = N_{Y'}$, A updates the corresponding outgoing reference entry for Y in its *GaoMap*. If all objects on A, which hold an outgoing reference to object Y, are in the *regular* state, a plain *InOut Notify ACK* message is sent back.

If there is at least one Z^* that holds an outgoing reference to Y, node A must drop the *InOut Notify* message, because the state of the migrating object Z is unclear and node $A = N_{Z^*}$ must not make any authoritative statements about the success of this migration.

If there are one or more objects, e. g. X' and Z', on node A that hold an outgoing reference to object Y, $A = N_{X',Z'}$ forwards the *InOut Notify* message to $C = N_X$ and $D = N_Z$. Afterwards, the *InOut Notify ACK* message is sent back to node B, which contains the nodes C and D to announce the additional incoming references to Y':

```
Msg{Src = A,
    Dst = B,
    Type = InOut Notify ACK,
    Operation = Add incoming references
                for node C and node D to proxy Y'
   }
```

When node $B = N_{Y'}$ receives this *InOut Notify ACK* message with these new incoming references, it takes these references and adds them to proxy Y'. Because the *InOut Notify* message was already forwarded to the nodes C and D, node B only forwards the message to N_Y so that N_Y can add the additional incoming references to Y, too. However, if these forwarded *InOut Notify* messages are not acknowledged from N_X and N_Z within the given timeout period, node B has to send an *InOut Notify* message to these nodes.

If a node N_V receives an *OutIn Notify* message from a node $B = N_{Y'}$, this message contains an additional incoming reference to another node $D = N_Y$ (see Figure 8.7). Thus, N_V adds the incoming reference for node D to V. If the message was an *OutIn Notify Remove* message, N_V additionally deletes the incoming reference of node B from V.

8.5.2 Pending State

As long as an object is in the *pending* state Y^*, the node N_{Y^*} only handles *OutIn Notify* messages for Y^*. The node optimistically forwards this message to the assumed new home node N_Y and adds a new incoming reference to Y^*. However, it does not respond with an acknowledgment

message. This additional incoming reference, again, prevents the heady deletion of the proxy Y' after the transition from the *pending* to the *forwarding* state.

If the message was an *OutIn Notify Remove* message, Y^* does not delete the contained incoming reference from Y^* to prevent an inconsistent object state.

8.5.3 Forwarding State

During the transition from the *pending* into the *forwarding* state, the proxy Y' comes into existence and $N_{Y'}$ sends the *InOut Notify* and *OutIn Notify* messages along the incoming and outgoing references of Y'.

An *InOut Notify* message that announces the new location of an object V is handled as described in the *regular* state. Namely, if Y' references the announced object V, $N_{Y'}$ forwards the message to N_Y.

If $N_{Y'}$ receives an *OutIn Notify* message for the object Y, it adds the transported new incoming reference for node C to Y' and forwards the message to N_Y. To mark this incoming reference as updated, $N_{Y'}$ sends an additional *InOut Notify* message to node C. If the message was an *OutIn Notify Remove* message, $N_{Y'}$ additionally deletes the incoming reference for the sending node from Y'.

If $N_{Y'}$ receives an *OutIn Notify ACK* message for Y', the corresponding outgoing reference in Y' is marked as updated. The reception of an *InOut Notify ACK* message causes $N_{Y'}$ to mark the corresponding incoming reference in Y' as updated.

When all incoming and outgoing references of Y' are updated, $N_{Y'}$ can delete Y'.

8.6 State Diagram: Incoming Reference Management

The incoming reference management is tightly coupled to both, the regular operations and the migration process, so that some effects in the other state diagrams cause transitions in this state diagram.

8.6.1 Regular State

The handling of incoming reference management messages is straight forward for an object Y: an *InRef Establish* message adds a new incoming reference to Y, whereas an *InRef Remove* message deletes an incoming reference from Y. Both messages are acknowledged with the corresponding ACK message.

If N_Y deletes Y, it has to decrement the reference counter of all of Y's outgoing references. If the reference counter for one of these outgoing references, e. g. to object Z, drops to zero, N_Y sends an *InRef Remove* message to N_Z. When the acknowledgments for all sent *InRef Remove* messages have arrived, Y is deleted.

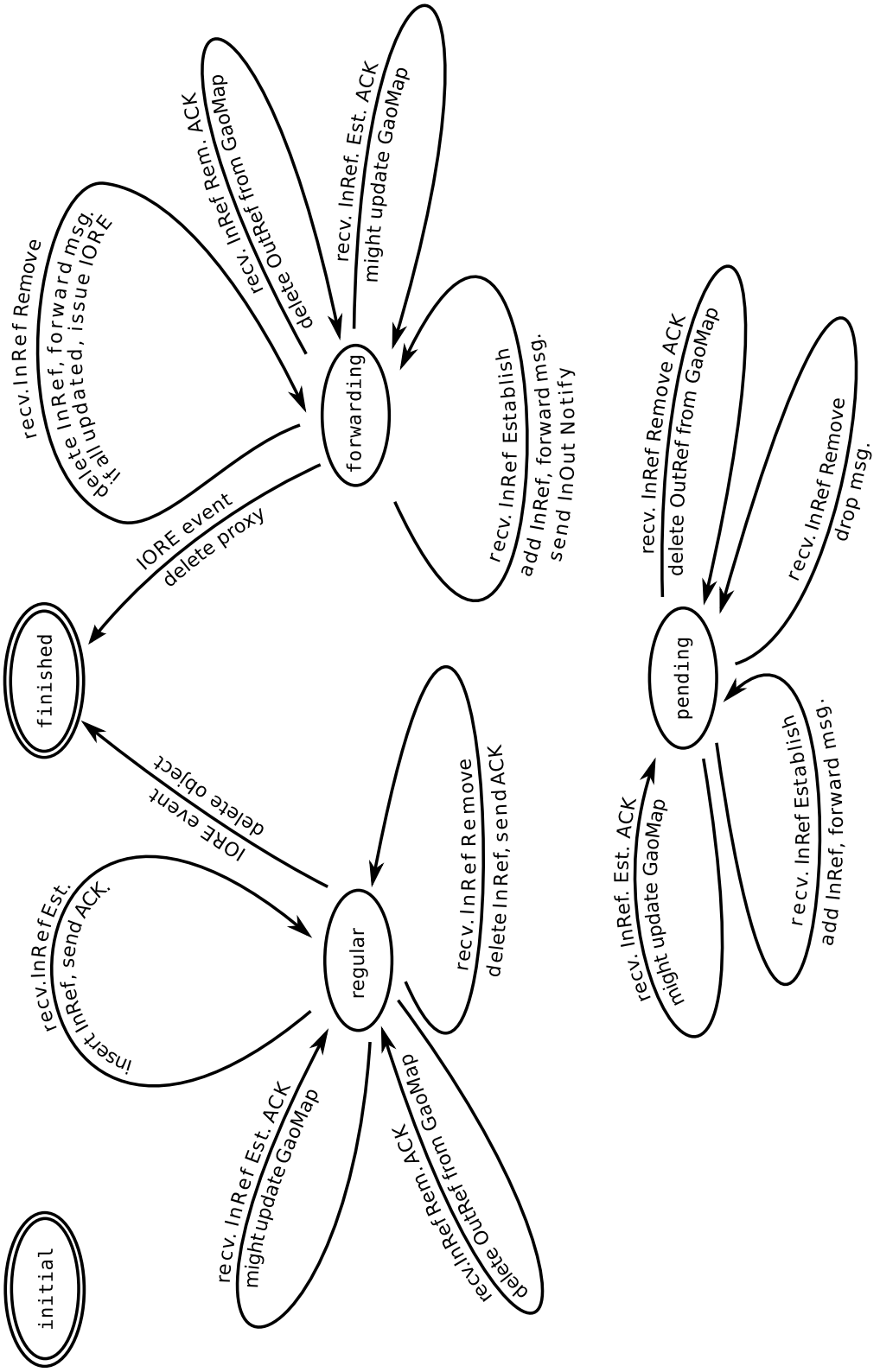


Figure 8.10: State diagram for the incoming reference management. The incoming reference management is tightly coupled to both, the regular operations and the migration process, so that some effects in the other state diagrams cause transitions in this state diagram.

8.6.2 Pending State

A node that holds a pending object Y^* drops all *InRef Remove* messages and forwards all *InRef Establish* messages to the assumed new home node N_Y to prevent an inconsistent object state. Furthermore, the reception of an *InRef Establish* message causes N_{Y^*} to add the new incoming reference to Y^*, without acknowledging the message or sending an *InOut Notify* message. This additional incoming reference is necessary to ensure the sending of an *InOut Notify* message when the migration has completed and the proxy Y' comes into existence.

8.6.3 Forwarding State

A node that holds a proxy Y' forwards all *InRef Establish/Remove* messages to N_Y. For an *InRef Establish* message, $N_{Y'}$ adds the new incoming reference to Y' and sends an *InOut Notify* message back to the sending node. For an *InRef Remove* message, $N_{Y'}$ marks the announced incoming reference in Y' as updated and forwards the message to N_Y. If this was the last not updated incoming reference in Y', and all outgoing references have been successfully updated as well, Y' is deleted.

9 Evaluation

This chapter describes the evaluation of the presented protocols and their optimizations.

The first part describes the evaluation and comparison of the three location update protocols from Chapter 7 and Chapter 8 in the OMNeT++ discrete event simulator framework. For this evaluation, I implemented the reactive location update protocol (RU) without access path optimization, and the proactive location update protocol with the regular triangular access (PU) and with the enhanced triangular access, i. e. the enhanced proactive update (EPU).

The second part describes the software simulation environment that does not use the OMNeT++ framework, but directly instruments the source code of two simple benchmark applications. This simulation environment is used to evaluate the access path optimization as described in Section 7.3. Furthermore, it examines the influence of object caching on the optimistic execution of transactions. This second approach was chosen to be able to examine larger simulation scenarios without the message and networking overhead introduced by OMNeT++.

9.1 OMNeT++ Simulation

This section evaluates and compares the performance of the RU, PU and EPU approaches in a distributed simulation environment that executes a micro benchmark application in the OMNeT++ [Var01; VH08] discrete event simulator framework, version 3.3.

The foundation of the micro benchmark is a Java implementation of a *Red-Black tree* (*RB tree*) data structure [Bay72]. This data structure is a fundamental building block, which is e. g. used by Ferri et al. [Fer+10] as an example benchmark for the memory management in embedded applications.

The chosen network environment of the simulator consists of a $10x10$ grid network of toroidally connected nodes. To communicate within the network, nodes exchange messages that trigger appropriate actions. Thus, the implemented OMNeT++ simulator emulates a network of nodes, where nodes are connected via links, and where messages are sent via these links, with the possibility to assign, among others, a link delay or a message loss probability to each connection.

In this environment, each of these nodes executes one thread that uniform randomly accesses an RB tree with x elements y times, where x and y are the chosen simulation parameters. The entry point into the RB tree is a Java class object. This static object holds the reference to the root object of the tree and must be known by all threads that access the tree. Therefore, each thread receives a reference to this static object during the initialization phase of the simulation.

During a single access, each thread draws two uniform random numbers in an interval that contains x elements. These numbers are used as keys r_1 and r_2 to identify objects that are stored in the RB tree. Each thread first inserts a new object with key r_1 if the tree does not yet contain

this element, and re-balances the tree, if necessary. Then, the thread searches for an object with key r_2, deletes it if it was present, and again re-balances the tree, if necessary.

As a result, the elements of the RB tree data structure are distributed throughout the network; Firstly, because a newly created object initially resides on the creating node, and secondly, because the simulator was designed to be compliant with the DecentSTM protocol, so that a thread must perform all write operations on *local* objects. Thus, a write operation on a remote object requires the implicit migration of the object to the local node, which leaves a proxy on the previous home node. As a result, it depends on the uniform random order of threads and their performed tree operations if an object is local or remote.

Because the implementation of a Java VM inside the OMNeT++ simulator introduces an unnecessary overhead, I separated the execution of the micro benchmark from the network simulation. In a first step, the source code of an RB tree implementation is instrumented to write out all object operations, namely, the creation and deletion of objects, as well as each access to a reference field. In a second step, this benchmark application is executed in a single-threaded Java VM. As a result, the tree access is strictly sequential and simulates for each thread the atomic access to the tree in a single transaction, as required by the DecentSTM protocol. When one thread finishes its tree access cycle described above, a uniform random generator chooses the next thread to continue with the next cycle. This process is repeated until each thread executed the tree access cycle y times.

The output of this simulation run is the control file for the OMNeT++ simulator. It contains the tree access operations of the 100 threads in the network. A customized OMNeT++ simulation scheduler reads this file and subsequently executes the given operations on the different nodes, such as the object creation and deletion and the read and write operations to local or remote objects. Here, each remote operation requires the interaction with remote nodes, namely, the sending and receiving of access request and response messages.

Note that the OMNeT++ simulator does not implement an explicit object cache per node. Thus, a node does not cache object copies or outdated object versions, and does not keep a copy of an object when this object migrates to another node. This approach was taken to evaluate the remote object access and the network protocols only.

The caching behavior, as well as the access path optimization, are part of the software simulation described in Section 9.2.

9.1.1 Simulation Runs

The simulation environment described in the previous section is used to measure the access latency to remote objects in number of hops (in the sense of the underlying routing protocol) that are needed to reach the remote objects. Additionally, the simulator counts the number of proxy forwards that are encountered on the way to the referenced object, and, with PU and EPU, the number of additional maintenance messages.

To measure the influence of explicit object migrations, an additional simulation parameter defines an explicit migration rate. This migration rate gives the probability that an object migrates after every 10 simulation events to another, uniform randomly chosen node and leaves a proxy on the previous home node (in one simulation event a message travels for example one hop).

Operation/Approach	50 objects RU/PU/EPU	100 objects RU/PU/EPU
NEW	4 950	9 992
DELETE	4 898	9 890
Total GET	984 766	3 993 158
Local GET	63 341	144 037
Remote GET	921 425	3 849 121
Total PUT	51 279	101 719
Local PUT	12 979	25 441
Remote PULL	38 300	76 278

Table 9.1: Total number of NEW, GET and PUT operations split into local and remote operations.

The OMNeT++ simulations were executed ten times with $x = y = 50$ and migration rates from 0 % to 50 % in 10 % steps. Afterwards the average over all simulation runs was computed. The simulation was repeated with $x = y = 100$ objects and the migration rates of 0 %, 10 % and 20 %.

The lower rates for the simulations with 100 objects have been chosen for a qualitative evaluation of the protocols and a comparison against the results of the runs with 50 object. I did not simulate multiple runs or higher migration rates because the OMNeT++ simulation was too CPU intensive.

However, these small network and tree sizes are sufficient to evaluate the network protocols for the remote object access. Furthermore, larger scenarios with up to 1 000 nodes and 10 000 objects are evaluated in Section 9.2.

9.1.2 Evaluation of Implicit Object Migrations

Table 9.1 shows the number of different operations (NEW, DEL, GET, PUT) for the simulation runs without explicit migrations. However, these simulations implicitly migrate objects when a remote PUT operation requires the migration of the remote object to the local node. In Table 9.1, the number of operations is the same for all three scenarios because the same application is executed. Furthermore, one remote GET or PUT operation requires the sending of two messages: one request and one response message.

The table shows that from 984 766 GET operations for the tree with 50 objects only 63 341 operations, or 6.43 %, were answered locally, while 93.57 % had to access an object on a remote node. Due to a higher depth of the tree with 100 objects, together with the increased re-balancing overhead, the number of GET operations increases by about four times to 3 993 158 in the simulations with 100 objects, of which 3.61 % were executed locally.

For both tree sizes, about 25 % of the 51 279, respectively 101 719, PUT operations were directly executed locally, while all other local PUT operations required a previous fetch (remote PULL migration) of the remote object, which corresponds to an implicit object migration.

	50 objects			100 objects		
Operation/Approach	RU	PU	EPU	RU	PU	EPU
Remote GET Proxy Frwds	298 332	0	0	1 036 525	0	0
Implicit PULL Proxy Frwds	286	0	0	525	0	0
Proxy Deletes	0	19 150	19 150	0	38 139	38 139
Proxy Remaining	345	0	0	535	0	0

Table 9.2: Number of proxy forwards and deletions for simulation runs with implicit migration only. One sees that only in the RU approach messages are forwarded along chains of proxies, and that only in the RU approach some proxies remain in the system after the simulation terminated.

In addition to Table 9.1, Table 9.2 shows the total number of proxy forwards for remote *GET Request* and implicit PULL migration request messages. It is seen that only the object access messages in the RU approach were forwarded along (chains of) proxies. These proxy forwards are counted in number of forwarding operations, and not in number of hops of the underlying routing protocol. In this table, a message that is forwarded multiple times adds also multiple counts to the total number of proxy forwards, i. e. a message that was forwarded twice adds two proxy forwards to the total sum. For this reason, the histogram in Figure 9.1(a) on page 120 shows the distribution of *GET Request* messages depending on the number of encountered proxy forwards. In numbers, 795 737 GET messages reached their destination directly, i. e. after 0 proxy forwards, 114 764 GET messages encountered one proxy, and 51 320 GET messages were forwarded along a chain of two proxies. Not seen in the figure are the 47 GET messages that traversed the longest encountered proxy chain with eight proxies.

This evaluation does neither plot nor break down the number of proxy forwards for the PULL migrations, because these are only a couple of hundred messages that traversed a chain of proxies, which is insignificant compared to the *GET Requests*.

This substantially smaller number results from the fact that a remote PULL migration requires in most cases a previous remote GET operation, which reads the needed reference and implicitly updates the locally cached location information of the accessed object. The only case that a remote PULL operation is forwarded is when the reference for the PUT operation was read from an object on a node where the corresponding locally cached object location was outdated.

Besides the proxy forwards, Table 9.2 also shows the number of deleted proxies and the number of proxies that stayed in the system after the application finished. Here, one sees that the PU and EPU approaches were able to delete all unnecessary proxies, which was not possible in the RU approach, where a couple of hundred proxies remained in the system. This number is reasonably smaller than the deleted proxies in the PU and EPU approach. The reason for this is the fact that a proxy is implicitly deleted if the object re-migrates back to the node where the proxy resides. These implicit proxy deletions were not counted in the simulator.

Another observation from Table 9.2 is that with the PU and EPU approach, the access messages reach the corresponding object directly, without proxy indirections. Nevertheless, this direct access comes with a high penalty: Table 9.3 shows for each approach the total number of object

Msg./Approach	50 objects			100 objects		
	RU	PU	EPU	RU	PU	EPU
Send Msg ⇄	1 919 450	3 705 582	4 957 482	7 850 798	15 381 362	20 978 260
Obj Acc ⇄	1 919 450	1 919 450	1 919 450	7 850 798	7 850 798	7 850 798
Maintain ⇄	0	1 824 432	3 076 332	0	7 530 564	13 127 462
InRef Est ⇄	0	24 852	1 276 750	0	50 344	5 647 242
InRef Rem ⇄	0	1 629 482	1 629 482	0	7 168 616	7 168 616
InOut Notify ⇄	0	118 458	118 458	0	208 500	208 500
OutIn Notify ⇄	0	51 642	51 642	0	103 104	103 104

Table 9.3: Access and reference maintenance messages of the different approaches for simulation runs with implicit migration only. The message numbers count the request and response messages (⇄).

access messages in contrast to the reference maintenance messages. Additionally, the table breaks the total number of reference maintenance messages down into the different types of reference maintenance messages.

The first row shows the number of total message that each approach sent. Here, it is seen that the total number of messages is far higher for the PU and the EPU approach than for the RU approach. Namely, the PU approach sends about two times more messages, while the EPU sends about three times more messages. With the separated numbers of reference maintenance messages, it is seen that the main portion of the message overhead comes from the *InRef Remove* messages, which are sent whenever a reference field that contains a reference (and not a NULL reference) is overwritten. Furthermore, the higher overhead of the EPU protocol is solely caused by the additional *InRef Establish* messages that are necessary for the enhanced triangular access.

A thorough look at the object access and *InRef Establish* messages in the EPU approach shows that there is a gap between these two numbers. This deviation is caused by the fact that no *InRef Establish* message is sent if the accessed object is co-located with the referenced object on the same node.

Compared to the EPU approach, the PU approach has a significantly smaller number of *InRef Establish* messages. These smaller number of messages result from a node that receives a new OutRef during an object migration, and thus has to inform the referenced object about the new incoming reference.

Another penalty that comes with the PU approach is the increased access latency that results from the regular triangular access. This penalty is seen in the histograms of the *GET Request* and *GET Response* messages of all three approaches, found in Figure 9.2(a), Figure 9.2(c) and Figure 9.2(e). The x-axis in these figures shows the number of hops (counted in hops of the underlying routing protocol, not proxy forward hops) that a message travels before it reaches the home node of the accessed object, while the y-axis shows the number of messages that traveled

the given number of hops. The comparison shows that the latency for a *GET Request* message for the PU approach is about twice as high as for the RU and EPU approach.

In numbers, the peak for the PU approach is around 9 hops with 82 253 messages. To the left, 89 319 messages traveled 8 hops and to the right, 87 805 messages traveled 10 hops.

For the EPU approach, the peak of the distribution is at 5 hops, with 166 446 messages, which is also the peak for the RU approach with 136 227 messages. Furthermore, Figure 9.2(a) shows that the *GET Request* message distribution for the RU approach is heavy-tailed. This tail is a result of the proxy forwards and is still visible in the figure with 691 messages that traveled 30 hops. The tail ends at 54 hops with 3 messages. These very high hop counts are a result of inefficient routes at the beginning of the simulation, cf. [Fuh05]. At this stage, the underlying routing protocol holds only few, potentially sub-optimal routes to only some nodes in the network. During the run time of the simulation, each node 'learns' more routes, until eventually it knows an optimal route to all other nodes in the network.

Mean Value and Variance For the further evaluation, I am interested in the mean value and the variance of these results. For this reason, I fitted the Gaussian normal distribution function to all *GET Request* distributions:

$$f(x) = a \cdot e^{\frac{(x-b)^2}{2 \cdot c^2}} \tag{9.1}$$

The results of this fit operation, for all simulation runs, are shown in Table 9.8 on page 124. For a migration rate of 0 % (i. e. implicit migrations only), the *GET Requests* of the RU approach have a mean value of 5.12 hops and a variance of 2.94 hops. The mean value for the PU approach is 8.10 hops with a variance of 4.48, while the EPU approach has a mean value of 4.92 hops and a variance of 2.50 hops.

These values show that the EPU approach has of all three approaches the best access latency, followed by the RU approach, where the latency is with 0.2 hops only slightly higher, but heavy-tailed, because not all request messages reached the accessed object on the shortest path.

The intermediate conclusion of the evaluation of implicit object migrations is that the RU approach is best suited for the envisioned system. The main reason is that the proxy indirections in this setting only slightly increase the access latency by 0.2 hops, but the approach comes with no reference maintenance message overhead.

A small drawback are the remaining proxies, which require some additional memory. However, the RU approach represents proxies by a single entry in the GaoMap, which will be deleted eventually by the garbage collector.

Altogether, it depends on the network and the application if it is preferable to send fewer messages and tolerate a longer access latency (RU), or if an object must be accessed as fast as possible, while a higher number of sent messages is tolerable (EPU). In any case, the PU approach with its triangular access messages is not applicable, because the access latency is about twice as high compared to the other two approaches. Furthermore, the PU and EPU approaches both come with a high reference maintenance management overhead to keep the location information of migrating objects up to date.

Events/Operations	50 objects					
	RU		PU		EPU	
Migration Rate	0 %	50 %	0 %	50 %	0 %	50 %
Total GET	984 766	984 766	984 766	984 766	984 766	984 766
Local GET	63 341	25 422	63 341	25 406	63 341	25 435
Remote GET	921 425	959 344	921 425	959 360	921 425	959 331
Total PUT	51 279	51 279	51 279	51 279	51 279	51 279
Remote PULL	38 300	61 686	38 300	61 602	38 300	61 550
PUSH Migration	0	13 623 574	0	13 124 891	0	12 673 024

Table 9.4: Number of local and remote operations/events for simulation runs with implicit migrations only, compared to simulations with an explicit migration rate of 50 %.

9.1.3 Evaluation of Implicit versus Explicit Migration

The following section investigates the influence of explicit object migrations on the remote object access. Some reasons for such explicit object migration are e. g. load balancing or system maintenance, but because this topic is not in the main scope of this thesis, I will not go into more detail here, and just assume that there is an entity in the system that triggers explicit migrations for its own reasons.

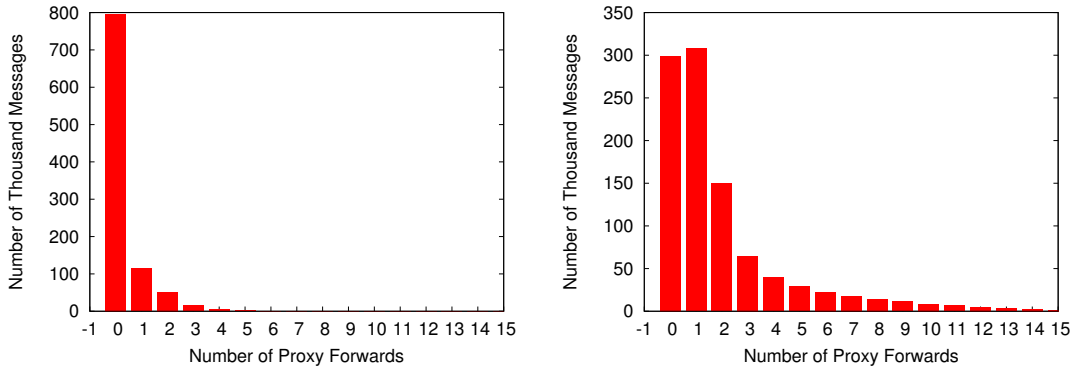
To analyze the influence of explicit object migrations, the migration rates are set to values ranging from 10 % to 50 %, in 10 % steps. However, to simplify the comparison, the following section only discusses the migration rate of 50 % and compares it to the case with a migration rate of 0 % (i. e. implicit migration only). The other simulation runs scale linearly between these two points, see Figure 9.3(a) on page 125.

In accordance to Table 9.2, Table 9.4 compares the influence of a migration rate of 50 % to the 0 % migration rate results.

A migration rate of 50 % means that half of all objects migrate every 10 simulation steps, with the result that over the complete simulation run, about 13 million explicit PUSH migrations take place. I do not expect to find systems with such a high migration rate. Nevertheless, I ran these simulations for two reasons: First, to see if the maintenance protocols can cope with that many object migrations and secondly, to further investigate and evaluate the overhead that is necessary for the reference maintenance. Moreover, the higher migration rates revealed some protocol flaws that had to be solved. The resulting protocol subtleties are described in more detail in [SF10].

As expected, with explicit migrations the number of local GET operations drops for all protocols from 6.5 % to 2.6 %. Furthermore, the number of necessary PULL migrations increased significantly and even exceeds the actual number of total PUT operations.

The reason for this deviation is specific for the simulation environment that does not cache object copies: The migration policy of the simulator migrates an arbitrary local object, regardless of whether it was recently pulled to the local memory to perform a local PUT operation. For this reason, it happens that a node *implicitly* migrated (pull) an object to the local memory, and



(a) Migration rate 0 %. (b) Migration rate 50 %.

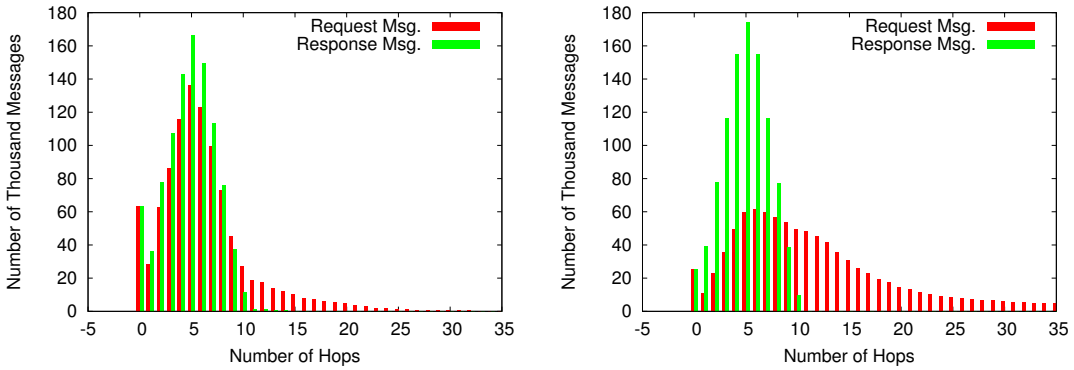
Figure 9.1: RU: Histogram of the number of proxy forwards, i. e. the proxy chain length, that a *GET Request* message had to traverse before the accessed object was reached for a tree size of 50 objects.

immediately afterwards *explicitly* migrated (push) it to another node, without executing the local PUT operation. When the node now tries to perform the local PUT operation it fails and has to pull (*re-pull*) the object a second time. Because this evaluation is not concerned with a high execution performance, but interested in the comparison of the performance of the different location update protocols, this behavior is not considered harmful.

Table 9.5 compares the number of proxy forwards for remote GET and PUT operations. While in the 0 % case the PU and EPU approach do note make active use of proxies, this changes if more objects migrate. However, compared to the proxy forwards in the RU approach, these numbers are still two orders of magnitude lower and indicate that the use of the EPU approach is advantageous over the PU approach due to the faster arrival of *GET Response* messages. Thus, fewer objects are able to migrate between two *GET request* messages. As a result, 10 071 messages in the EPU approach with a migration rate of 50 % have been forwarded only once, while 14 306 messages in the PU approach have been forwarded once and 2 584 messages have been forwarded twice.

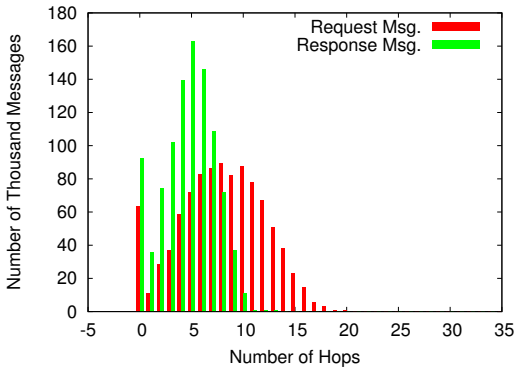
In contrast to the PU and EPU approaches, the majority of the 307 588 GET messages in the RU approach is forwarded at least by one proxy, see Figure 9.1(b). This number is even higher than the 299 026 messages that reach their destination directly (0 proxy forwards). As a result, the *GET Request* message distribution for the 50 % migration RU case in Figure 9.2(b) is widened and significantly more heavy-tailed, compared to the 0 % case in Figure 9.2(a). The tail passes the 20 hops with 14 842 messages, has 2 239 messages that traveled 50 hops, and 11 messages that traveled 125 hops. For the PU approach, the worst case is 12 messages that traveled 31 hops, and for the EPU approach 14 messages that traveled 19 hops.

Again, I fitted the Gaussian normal distribution to the results of the *GET Request* message distribution, cf. Table 9.8 on page 124. For the 50 % migration rate, the RU approach has a mean value of 8.57 hops and a variance of 6.48 hops, compared to the 0 % case, that has only a mean value of 5.12 hops and a variance of 2.94 hops. Thus, the length of the average number of hops per object access has increased by 3.45 hops.

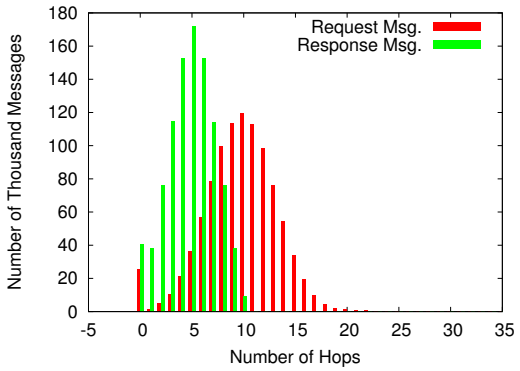


(a) RU: Migration rate 0 %.

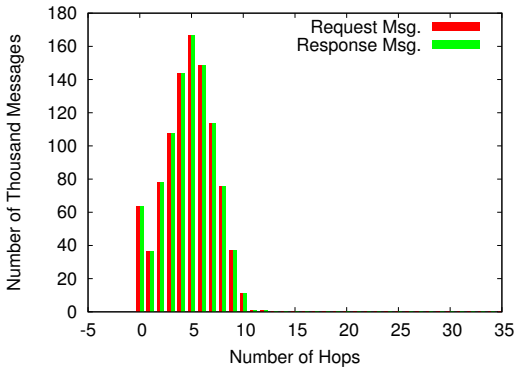
(b) RU: 50 Objects, Migration rate 50 %.



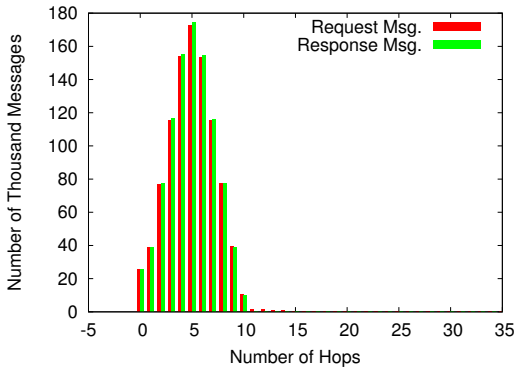
(c) PU: Migration rate 0 %.



(d) PU: Migration rate 50 %.



(e) EPU: Migration rate 0 %.



(f) EPU: Migration rate 50 %.

Figure 9.2: Histogram of *GET Request* and *GET Response* messages for the different approaches for simulation runs with implicit migrations only, compared to simulation runs with a migration rate of 50 % and a tree size of 50 objects.

Events/Operations	50 objects					
	RU		PU		EPU	
Migration Rate	0 %	50 %	0 %	50 %	0 %	50 %
Remote GET Proxy Frwds	298 332	1 941 086	0	19 474	0	10 071
Remote PUT Proxy Frwds	286	21 192	0	434	0	453
Proxy Deletes	0	0	19 150	6 323 436	19 150	6 110 454
Proxy Remaining	345	6 335	0	481	0	535

Table 9.5: Number of proxy forwards and deletions for a simulation with implicit migration only, compared to the simulation with an explicit migration rate of 50 %. Here, it is seen that this time not only the RU approach encountered proxy chains, but that in the PU and EPU approach *GET Requests* and *PUT Requests* were forwarded as well.

The comparison of the *GET Request* message distributions for the PU and EPU approaches, shown in Figure 9.2, do not show such significant differences. For the PU approach, the result is a mean value of 9.94 hops and a variance of 3.22 hops. Compared to the 0 % case, the access latency has increased by 1.84 hops.

For the EPU approach, the message distribution indicates that the *GET Request* messages traveled the same number of hops as the corresponding *GET Response* messages traveled back. Furthermore, the fitting results in a mean value of 4.98 hops and a variance of 2.34. Compared to the 0 % migration case with a mean value of 4.92 hops and a variance of 2.50 hops, this result shows that the performance of the EPU approach is not influenced by 50 % explicit object migrations at all.

Table 9.6 compares the various maintenance messages that are needed for the PU and EPU approach. Compared to the 0 % case, the PU and EPU approaches need about 82 times more maintenance messages than object access messages. Here, a comparison with a lower explicit migration rate of 10 % – which is not further investigated here – shows that the PU and EPU protocols already send about 15 times more maintenance messages than object access messages.

The two main differences compared to the 0 % case are: first, the difference between the number of *InRef Establish* and *InRef Remove* messages in the PU approach, and secondly, the difference between the numbers of *InRef Establish* messages in the PU and EPU approach.

These differences are smoothed in the 50 % case, as a result of the high migration rate: With about 13 times more migrations than *GET Request* messages, the additional *InRef Establish* messages for the enhanced triangular access vanish in the total number of all sent *InRef Establish* messages.

Table 9.6 additionally splits the number of *InOut Notify* and *OutIn Notify* messages into request and ACK messages each, to see the difference between them. The different numbers indicate that more request messages have been sent than ACKs have been received, which is an allowed protocol feature. It indicates that a message was dropped at the destination node and had to be resent. This happens for a migration rate of 50 % for 16.22 million *InOut Notify* messages in the PU approach, and for about 15.68 million *InOut Notify* messages in the EPU approach. The

Msg./Approach	50 objects			
	PU		EPU	
Migration Rate	0 %	50 %	0 %	50 %
Send Msg. $\rightleftarrows$	3 705 582	154 669 949	4 957 482	153 440 913
Object Acc $\rightleftarrows$	1 919 450	2 041 924	1 919 450	2 041 762
Maintain $\rightleftarrows$	1 824 432	152 839 570	3 076 332	151 389 327
InRef Establish $\rightleftarrows$	24 852	21 820 794	1 276 750	23 208 792
InRef Remove $\rightleftarrows$	1 629 482	23 523 286	1 629 482	23 072 856
InOut Notify	59 229	42 777 731	59 229	41 808 974
InOut Notify ACK	59 229	42 490 969	59 229	41 531 149
InOut Notify Resend	0	16 222 153	0	15 679 044
OutIn Notify	25 821	11 113 400	25 821	10 883 783
OutIn Notify ACK	25 821	11 113 390	25 821	10 883 772
OutIn Notify Resend	0	0	0	0

Table 9.6: Reference maintenance messages for implicit migrations only and a migration rate of 50 %.

reason for this high number is that a node that receives an *InOut Notify* message and that holds local objects in the pending state, has to drop the *InOut Notify* message to prevent an inconsistent objects location state.

In contrast to the 0 % migration case, in the 50 % migration case maintenance messages are forwarded along proxies. Thus, Table 9.7 breaks down the number of proxy forwards for the different maintenance messages for the PU and the EPU approach. It can be seen that about half of all *InRef Establish* and *OutIn Notify* messages and about 10 % of all *InRef Remove* and *InOut Notify* messages have been forwarded along at least one proxy.

Events/Operations	50 objects			
	PU		EPU	
Migration Rate	0 %	50 %	0 %	50 %
InRef Establish Proxy Frwds	0	5 378 263	0	5 275 699
InRef Remove Proxy Frwds	0	1 274 845	0	1 248 225
InOut Ref Notify Frwds	0	4 260 777	0	4 172 379
OutIn Ref Notify Frwds	0	5 481 425	0	5 367 398

Table 9.7: Number of proxy forwards for the different reference maintenance messages for simulations runs with implicit migration only, compared to simulations with a migration rate of 50 %.

Mig. Rate	50 objects					
	RU		PU		EPU	
	Mean Value	Variance	Mean Value	Variance	Mean Value	Variance
0 %	5.12976	2.94188	8.10285	4.47905	4.92078	2.50190
10 %	5.35024	2.95474	9.92066	3.21197	4.95215	2.36388
20 %	5.91910	3.77410	9.93012	3.21403	4.96162	2.35139
30 %	6.72076	4.90280	9.93568	3.21779	4.96956	2.34800
40 %	7.62386	5.81309	9.94078	3.22346	4.97480	2.34221
50 %	8.57008	6.47863	9.94252	3.22666	4.97979	2.34148

Table 9.8: Mean value and variance of the *GET Request* msg. distributions, migration rate 0 % to 50 %

9.1.4 Protocol Comparison for all Migration Rates

After the examination of the differences between implicit and explicit object migrations at the example of the 50 % explicit object migration case, the following section compares the message distributions for GET and PUT operations when applying various migration rates.

Comparison with 50 Objects

Table 9.8 lists the mean values and variances of all migration runs, while Figure 9.3(a) plots the results for the remote *GET Request* message distributions and Figure 9.3(b) plots the results for the *PULL Migration Request* messages, which are equal to implicit object migrations.

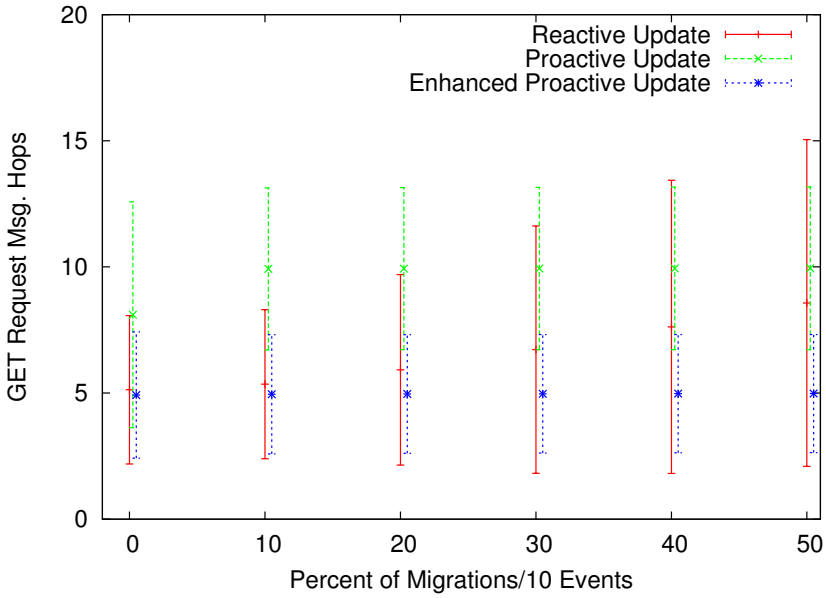
Figure 9.3(a) shows, that the mean values of the RU approach have a steady, linear slope. The PU approach shows a 1-hop step at the transition from implicit object migrations only to 10 % explicit object migrations, but then stays constant over all migration rates. This indicates that explicit object migrations, together with the increased latency of the Triangular-GET messages, result in objects that are further apart from each other. This step is not seen in the EPU approach, where the mean value and variance stay constant for all migration rates.

With a migration rate of 50 %, the RU approach reaches a mean value of 8.57 hops, where the PU approach is with a mean value of 9.94 still higher. The result of the EPU approach was not influenced by explicit object migration at all, and still has a mean value of 4.98 hops and a variance of 2.34.

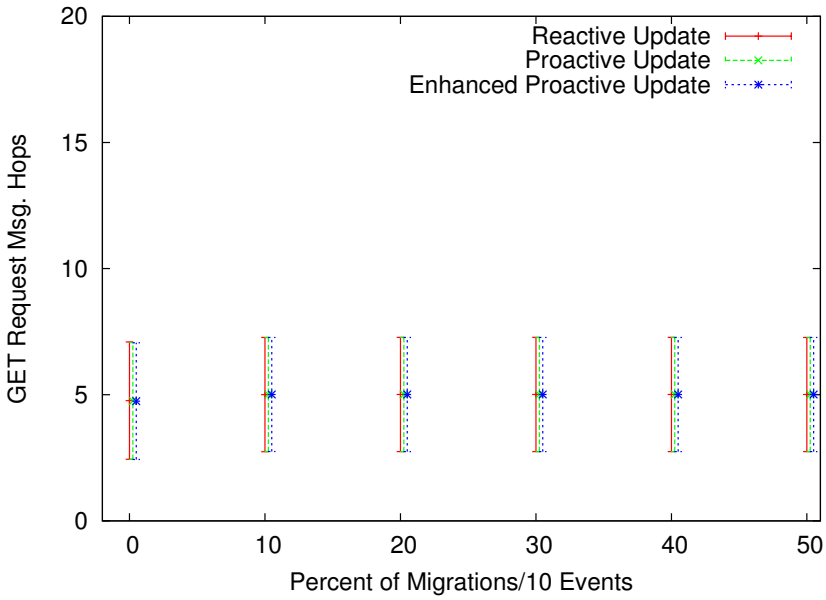
Due to the fact that PUT operations are preceded by *GET Requests*, which updated the object location information, there is almost no difference between the three approaches for the remote *PULL Migration* operations (implicit object migration), see Figure 9.3(b).

Comparison with 100 Objects

Besides the simulation runs with 50 objects, I ran a simulation with 100 objects. For this simulation, I only applied the migration rates from 0 %, 10 % and 20 %. I did not run higher migration rates because implicit migrations with a migration rate of 0 % are most interesting for



(a) GET Requests.



(b) Implicit PULL Migration Requests.

Figure 9.3: Comparison of average hop counts for the RU, PU, EPU approach for a tree size of 50 objects.

Mig. Rate	100 objects					
	RU		PU		EPU	
	Mean Value	Variance	Mean Value	Variance	Mean Value	Variance
0 %	5.28056	2.49902	8.67647	3.87287	5.03789	2.26971
10 %	5.56372	2.83152	9.98676	3.13437	5.00473	2.25944
20 %	6.35526	3.80372	9.99157	3.13956	5.01113	2.26670

Table 9.9: Mean value and variance of *GET Request* msg. distributions, migration rate 0 % to 20 %.

the envisioned scenario. Furthermore, I assume that the expected migration rate in a realistic scenario will not be much higher than 20 %.

The mean values and variances of these simulations are listed in Table 9.9. The results are similar to the simulations with 50 objects, with slightly higher values for the RU approach. Thus, I did not add a figure, because it looks similar to Figure 9.3.

9.2 Software Simulation for Access Path Optimization and Caching

Because the OMNeT++ event simulator framework emulates a complete network with nodes, separate memory address spaces per node, messages, connections and connection delays etc., it adds a significant management overhead that decreases the simulation performance. With this overhead, the time to run simulations with much more than 100 nodes and more than 100 objects was not feasible. However, the further evaluation of the developed protocols does neither require the accurate message delivery delay in milliseconds nor the distinction between the various message and event types.

For this reason, I implemented a second simulator to evaluate the access path optimization that is described in Section 7.3. Because the evaluation in the previous section has shown that the proactive location update approach is sub-optimal for the envisioned scenario, neither the PU nor the EPU approach were further investigated. Thus, the simulator only implements the reactive location update approach and evaluates it, together with the access path optimization and object caching, in larger networks and larger application scenarios.

This second simulator is also closer to the envisioned scenario: First, explicit object migrations are not considered anymore, and objects migrate only if a new head version comes into existence on a node other than the home node of the previous object version (implicit migration only). Secondly, each node is equipped with a cache of a given size. Each node uses this cache to store all objects on which the local threads currently work. Furthermore, each node in the simulator keeps a copy of an object that migrates to another node as outdated object version in its cache, until this copy is either updated, evicted by the *least-recently-used* policy, or deleted by the garbage collector. In this way, multiple copies of different object versions might be scattered across multiple nodes in the network. Thus, the DecentSTM algorithm on one of these nodes can continue its optimistic execution of transactions, potentially on outdated object versions. In this way, a thread does not need to stall until the head version of the accessed object is fetched

from a remote node. However, it is always possible, yet not implemented, to fetch the object in the background, e. g. by piggy-backing the object on some other necessary system or application messages to be prepared for a potential rollback.

Besides a C-implementation of an RB tree, the simulator additionally works on an AVL tree implementation, which is also written in C. The AVL tree [AL62], named after its inventors Adelson-Velsky and Landis, is a balanced binary search tree with the invariant that the height of any two branches in the tree differ by at most 1. To guarantee this invariant, each insert or delete operation may have to re-balance the whole tree, starting from the root.

In contrast to the AVL tree, an RB tree has weaker balancing constraints. Thus, the re-balancing operations start at that node in the tree where the element was inserted or deleted, and continue towards the root until the tree invariants are restored. As a result, the balancing operations of an RB tree modify less objects than those of the AVL tree.

The simulation environment simulates a given number of nodes, where each node executes one thread that operates on the given data structure. Each simulated thread uniform randomly inserts and deletes elements into/from the tree.

The simulator only simulates – rather than emulates – the distributed shared access to the data structures. It is not concerned with a particular network topology or protocol, but only considers proxy forwards. Hence, it can tackle larger data structures with more operations than the emulator in the previous section.

A single simulation run executes one million transactions that manipulate the given tree. Similar to the scenario from Section 9.1, each transaction draws two uniform random numbers, this time from an interval that contains 10 000 elements. Again, these numbers are used as keys r_1 and r_2 to identify the objects in the tree. Each transaction first inserts the object with key r_1 if the tree does not yet contain this element, and re-balances the tree if necessary. Then, it searches for the object with key r_2 in the tree, deletes it if it was present, and again re-balances the tree.

Each write operation on an object in the tree creates a new head version of the object. The previous head version of the object automatically becomes the latest proxy in the objects version history, i. e. in the proxy chain. Additionally, an update message is sent to the k previous versions to shorten the proxy chain.

9.2.1 Cache Characteristics

I ran simulations with 500 and 1 000 nodes and various cache sizes, ranging from 50 to 450 objects. The numbers shown are averages over 100 runs each. Each simulation run made in total 25 114 763 initial cache accesses for the AVL tree and 23 998 597 initial cache accesses for the RB tree. Note that the total number of initial cache accesses is equal for all cache sizes. Here, “initial access” denotes the first access to an object within a transaction. To prevent the transactions’ memory snapshot from becoming inconsistent, the DecentSTM system requires that subsequent accesses use the same cached object version. The total number of object accesses is 103 404 806 for the AVL tree and 37 242 620 for the RB tree.

Table 9.10 shows the basic parameters of the different simulation scenarios and the number of rotations needed to balance the tree. The rotation rate is the same for all AVL tree simulation runs,

	Nodes	Objects	Transactions	Rotations
AVL Tree	500	10 000	1 000 000	331 416
	1 000	10 000	1 000 000	331 416
RB Tree	500	10 000	1 000 000	395 962
	1 000	10 000	1 000 000	396 128

Table 9.10: Simulation parameters for the AVL and RB tree software simulation runs.

while they slightly differ for the RB tree runs. The reason are the different invariants of the tree algorithms: while the AVL tree is guaranteed to be balanced at all times, the RB tree structure is more flexible and allows for a slightly different structure, depending on the access characteristics.

For the rest of this section, I define the following three cache, and thus object, access cases:

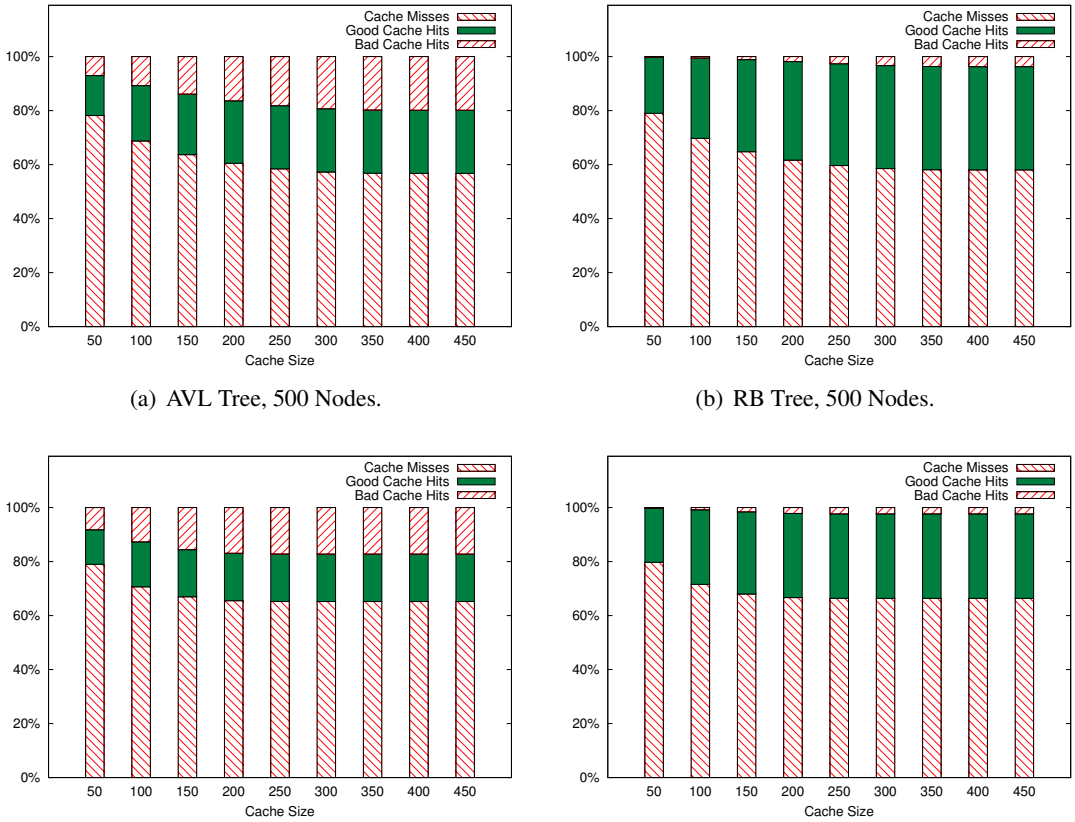
- *cache miss*: The accessed object is not cached, and the thread has to stall until the object's head version has been retrieved.
- *good cache hit*: The accessed object has been cached and the cached version is the current head version of the object. The thread continues its execution immediately (without knowing that it works with the head version). It has a good chance to succeed when committing.
- *bad cache hit*: The accessed object is cached, but the cached version is outdated. The thread continues its execution immediately (because it cannot know that the version is outdated). It is likely to fail when committing.

Figure 9.4 shows the percentage of *cache misses*, *good cache hits*, and *bad cache hits* for the different cache sizes for 500 and 1 000 threads.

As expected, the *good cache hit* rate compared to the total hit rate, is higher for the RB tree than for the AVL tree. For 1 000 threads (Figure 9.4(c) and Figure 9.4(d)), it is 12.8 % for the AVL tree and 20.0 % for the RB tree with a cache size of 50 objects (Table 9.11, column ⑦), and 17.6 % and 31.3 % for a cache with 450 objects. Furthermore, the *bad cache hit* rate, compared to the total accesses, is lower. It is 8.23 % for the AVL tree and 0.26 % for the RB tree with a cache size of 50 objects, and 17.23 % and 2.36 % for a cache with 450 objects (Table 9.11, column ⑧).

One can see that for the AVL tree the *good cache hit* rates set in at a cache size of 100 objects, whereas for the RB tree the cache size should be 150 objects. Larger caches do not (significantly) increase the system's performance.

Table 9.11 shows the corresponding numbers for 500 and 1000 threads in detail. Next to the total number of cache accesses, *cache misses*, *good cache hits*, and *bad cache hits*, the table shows the ratio and percentage for *good cache hits* to *bad cache hits*. Here, the ratio is for all AVL cache sizes around 1 and at most just above 2 for 500 threads and a cache with 50 objects (column ⑤). The percentage of *good cache hits* to *total cache hits* ranges from 50.49 % with a cache size of 450 objects to up to 60.84 % for a cache with 50 objects, for the tree with 1 000



(a) AVL Tree, 500 Nodes.

(b) RB Tree, 500 Nodes.

(c) AVL Tree, 1 000 Nodes.

(d) RB Tree, 1 000 Nodes.

Figure 9.4: RB and AVL tree cache accesses.

objects (column ⑥). In other words, the possibility to optimistically continue the execution with the latest head version of an object is between 50 % and 60 %.

For the RB tree with 1 000 threads, the ratio of *good cache hits* to *bad cache hits* is always above 13.25 for the cache sizes of 300 to 450 objects, and rises to 77.19 with a cache size of 50 objects. In accordance to these figures, the percentage of *good cache hits* compared to *total cache hits* ranges for 1 000 threads between 92.98 % for a cache size between 300 and 450 objects, and 98.72 % for a cache with 50 objects (column ⑥). In other words, the possibility to optimistically continue the execution with the latest head version of an object is between 92.98 % and 98.72 %.

Another figure is the percentage of *good cache hits* compared to the *total cache accesses* (column ⑦). For the AVL tree with 1 000 threads, this number ranges between 12.78 % for a cache with 50 objects and 17.57 % for cache sizes bigger than 200 objects. For the RB tree with 1 000 threads, this percentage ranges between 19.99 % for a cache size of 50 objects and 31.25 % for caches with more than 250 objects. For a cache with 150 object, this means, that in 30 % of all object accesses, the node finds the head version of the object in its own cache, and the optimistic execution is likely to succeed.

For the runs with 500 threads, all these results are slightly better and support the overall trend.

Another significant observation that supports the previous finding that larger cache sizes do not increase the performance, is that cache sizes above a certain number of stored objects do not increase these figures. For example for the AVL tree, all cache sizes with 250 objects or more have the same average *good cache hits* to *bad cache hits* ratio, and the same average percentage of *good cache hits* to *bad cache hits* and *good cache hits* to *total cache accesses*. Similar for the RB tree, where caches with 300 objects or more have the same figures. Hence, the conclusion is that cache sizes above 250 objects (for the AVL tree), and 300 objects (for the RB tree), do not increase the application performance for the given scenario. However, a cache that is too small increases the number of object versions that must be fetched from a remote node, see the *cache miss* column ② in Table 9.11.

Figure 9.5 shows the probability that a cache hit is good as a function of the number of intermediate transactions, i. e. the number of transactions that the *accessing* node has processed since the cache entry has been used previously. The smaller the cache the more quickly the *good cache hit* rate drops. Just before the probability reaches zero, the number of total hits is so small that the probability becomes erratic. Therefore, I removed those data points from the plots for which the number of total hits is less than 0.3 hits on average.

Figure 9.5 confirms the findings from Figure 9.4 and Table 9.11, namely, that large caches are not worthwhile. Even though a large cache can increase the overall number of cache hits (column ③), it also decreases the percentage of *good cache hits* over time. (Since the number of total hits decreases with the number of intermediate transactions, this effect is neither seen in Figure 9.4 nor in Table 9.11).

Comparing the AVL tree with the RB tree, it is again seen that the RB tree has a much better performance. In an AVL tree scenario, the probability of a *good cache hit*, and thus the probability that an object was not modified after two intermediate transactions, is only about 60 %. In an RB tree scenario, the probability of a *good cache hit* is 97.7 % after two intermediate transactions with a cache size of 50 objects. Overall, the conclusion is that 100 objects is the optimal cache size for the AVL tree, whereas it is 150 objects for the RB tree scenario.

9.2.2 Access Optimizations

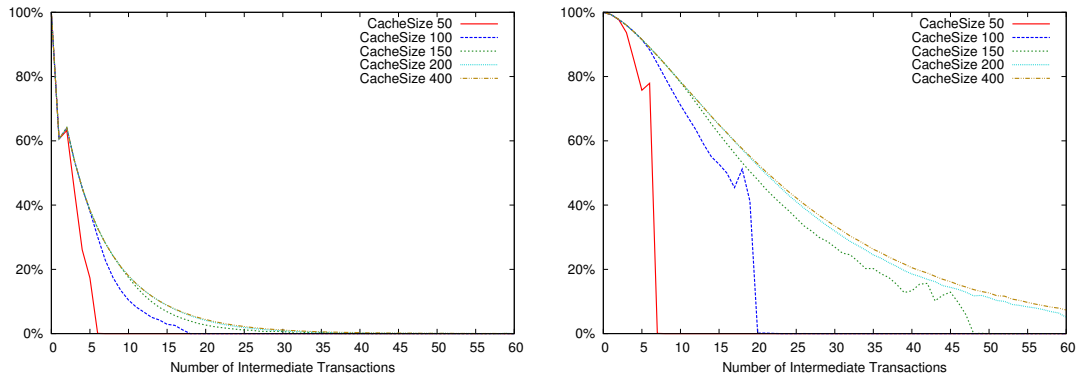
Upon a *cache miss* or *bad cache hit*, the thread must stall until it has retrieved the object's head version. However, the proxy chain that leads to the head version is the longer the more transactions have created new object versions of the object, and thus proxies. The longer the proxy chain that must be traversed, the longer a thread has to stall before the request completes.

The following two sections examine the proxy chains that are encountered after a *bad cache hit* and after a *cache miss*.

The examination starts with the *bad cache hit* evaluation, where the thread first optimistically executes the transaction on outdated data. During the commit phase, the DecentSTM consensus protocol has to traverse a chain of proxies to contact the head version. Thus, this is the time when the node that executed the transaction, finds out that the transaction operated on an outdated object version.

	Nodes	Cache Size	Access	Miss	Good Hit	Bad Hit	Good/ Bad Hit	Good/ Total Hit	Good/ Total Acc.	Bad/ Total Acc.
			①	②	③	④	⑤	⑥	⑦	⑧
AVL Tree	500	50	25114763	19633194	3716032	1765536	2.10	67.79 %	14.80 %	7.03 %
		100	25114763	17269595	5142555	2702612	1.90	65.55 %	20.48 %	10.76 %
		150	25114763	15992858	5632982	3488922	1.61	61.75 %	22.43 %	13.89 %
		200	25114763	15190120	5810493	4114149	1.41	58.55 %	23.14 %	16.38 %
		250	25114763	14677158	5866093	4571511	1.28	56.20 %	23.36 %	18.20 %
		300	25114763	14384369	5878643	4851749	1.21	54.78 %	23.41 %	19.32 %
		350	25114763	14269260	5880248	4965254	1.18	54.22 %	23.41 %	19.77 %
		400	25114763	14249188	5880340	4985234	1.18	54.12 %	23.41 %	19.85 %
		450	25114763	14248111	5880343	4986309	1.18	54.11 %	23.41 %	19.85 %
	1 000	50	25114763	19837063	3210783	2066916	1.55	60.84 %	12.78 %	8.23 %
		100	25114763	17744360	4176348	3194054	1.31	56.66 %	16.63 %	12.72 %
		150	25114763	16818469	4379877	3916416	1.12	52.79 %	17.44 %	15.59 %
		200	25114763	16453673	4410562	4250528	1.04	50.92 %	17.56 %	16.92 %
		250	25114763	16379680	4412625	4322457	1.02	50.52 %	17.57 %	17.21 %
		300	25114763	16375557	4412667	4326538	1.02	50.49 %	17.57 %	17.23 %
		350	25114763	16375510	4412667	4326585	1.02	50.49 %	17.57 %	17.23 %
		400	25114763	16375510	4412667	4326585	1.02	50.49 %	17.57 %	17.23 %
		450	25114763	16375510	4412667	4326585	1.02	50.49 %	17.57 %	17.23 %
RB Tree	500	50	23993641	18958495	4996398	38747	128.95	99.23 %	20.82 %	0.16 %
		100	23993641	16730906	7133862	128872	55.36	98.23 %	29.73 %	0.54 %
		150	23993641	15543203	8182926	267511	30.59	96.83 %	34.10 %	1.11 %
		200	23993641	14794025	8754118	445497	19.65	95.16 %	36.49 %	1.86 %
		250	23993641	14313782	9040058	639801	14.13	93.39 %	37.68 %	2.67 %
		300	23993641	14043194	9148805	801641	11.41	91.94 %	38.13 %	3.34 %
		350	23993641	13943905	9172264	877472	10.45	91.27 %	38.23 %	3.66 %
		400	23993641	13928991	9174306	890343	10.30	91.15 %	38.24 %	3.71 %
		450	23993641	13928326	9174365	890949	10.30	91.15 %	38.24 %	3.71 %
	1 000	50	23998597	19138733	4797712	62151	77.19	98.72 %	19.99 %	0.26 %
		100	23998597	17176035	6620965	201595	32.84	97.05 %	27.59 %	0.84 %
		150	23998597	16321650	7292023	384923	18.94	94.99 %	30.39 %	1.60 %
		200	23998597	15994873	7477777	525945	14.22	93.43 %	31.16 %	2.19 %
		250	23998597	15935044	7499519	564032	13.30	93.01 %	31.25 %	2.35 %
		300	23998597	15932246	7500189	566161	13.25	92.98 %	31.25 %	2.36 %
		350	23998597	15932218	7500194	566184	13.25	92.98 %	31.25 %	2.36 %
		400	23998597	15932218	7500194	566184	13.25	92.98 %	31.25 %	2.36 %
		450	23998597	15932218	7500194	566184	13.25	92.98 %	31.25 %	2.36 %

Table 9.11: Cache access characteristics for AVL and RB tree.



(a) AVL Tree: Good Hit/Total Hit Probability.

(b) RB Tree: Good Hit/Total Hit Probability.

Figure 9.5: Probability of good hit/total hit after x intermediate transactions, simulation with 1 000 nodes.

If the DecentSTM protocol cannot successfully commit because the object version was outdated, the transaction has to rollback and restart the transaction with the current head versions of the objects. Thus, the costs for a *bad cache hit* are not only the latency due to the object retrieval, but also the execution costs of the failed transaction.

The proxy chains that are encountered because of a *bad cache hit* are shorter than for a *cache miss*. The reason is that a cached object version means that the node read a former head version in the past. Here, it depends on the cache size and the object access characteristics how long the object stays in the cache before it is evicted by the LRU policy. On the contrary, an object access that results in a *cache miss* is an indication that the object was either evicted from the cache by the LRU policy, or that the reference to the object was never before resolved on the accessing node.

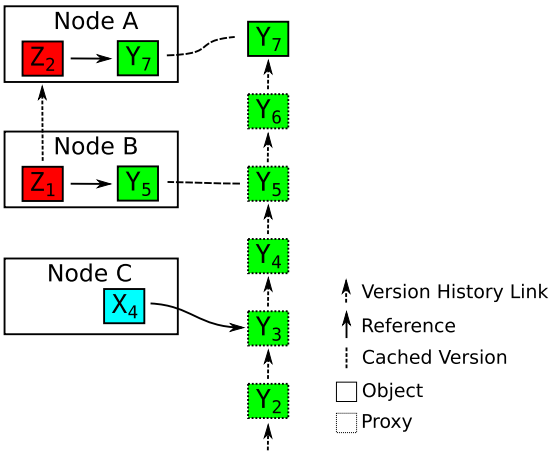


Figure 9.6: Object version history and cached object versions.

See, for example, Figure 9.6, which shows the version history of object Y and three nodes, A, B and C, that cached some object versions. Node C holds an object in version X_4 that references object Y in the version Y_3.

Because the reference to Y_3 was not yet resolved, or a previous copy of Y_3 was evicted from the cache, there resides no local copy of Y_3 in the cache of node C. Thus, the next access to Y_3 results in a *cache miss*. The caches on node A and node B both hold a copy of an object version of object Y. Cache A holds a copy of Y_7, which is the head version of object Y. Hence, an object access to the cached object is a *good cache hit*. On node B, the cached object version is outdated. Thus, the next access to the cached object version Y_5 is a *bad cache hit*.

Section 7.3 described the access path optimization approach where each object migration not only updates the latest proxy in the version history of an object, but also propagates this update further down the proxy chain. How deep this update should be propagated depends on the objects' access characteristics and a formula for the optimal propagation depth k_{opt} was given.

The next sections evaluate different experimental propagation depths k to support the analytic results, which are presented in the following sections as well.

9.2.3 Bad Cache Hits

Figure 9.7 shows the probability for the number of proxy forwards that are needed to retrieve the head version of an object after a *bad cache hit*. (Note that the figures are plotted with a double-logarithmic scale. They show the two cache sizes of 50 and 450 objects.)

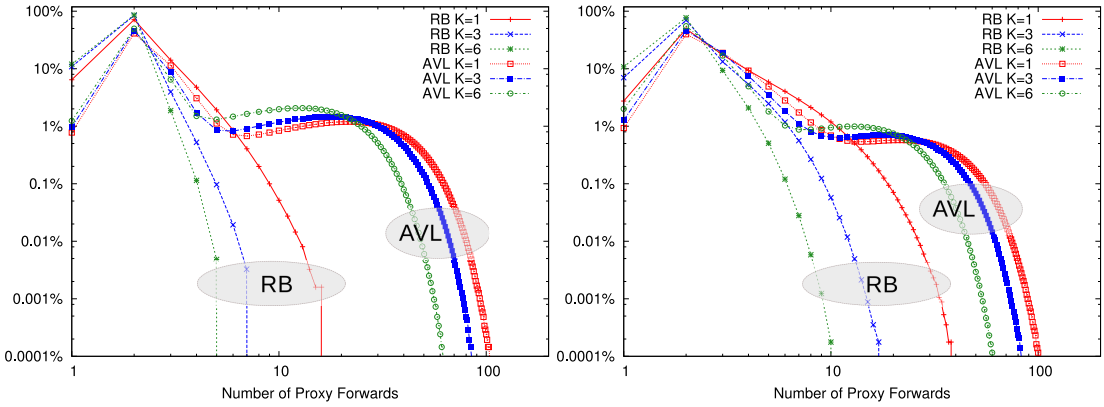
	Cache Size	k	0-hop	1-hop	2-hop	3-hop	5-hop
AVL	50	1	0	0.49	28.38	16.28	3.22
		3	0	0.96	45.99	8.73	0.85
		6	0	1.24	49.64	6.47	1.31
	450	1	0	0.53	26.22	19.76	7.42
		3	0	1.29	46.13	18.90	3.47
		6	0	2.00	54.86	16.91	1.79
RB	50	1	0	6.43	71.13	14.08	1.92
		3	0	10.83	84.56	3.95	0.09
		6	0	11.99	86.01	1.87	0.01
	450	1	0	2.74	49.18	17.40	5.88
		3	0	6.98	69.70	13.25	2.47
		6	0	10.79	77.15	9.31	0.50

Table 9.12: Probability for number of proxy forwards for AVL and RB tree after *bad cache hits*, simulation with 1 000 nodes.

To create the figure, the simulation ran with various update propagation depths k. (The figures only show $k = 1,3,6$ because the plots for $k > 6$ are not significantly different). In addition, Table 9.12 shows the length distribution of the traversed proxy chains, i.e. the probability that the object's head version was reached directly or after the first, second, third, and fifth proxy forward.

In contrast to the *cache miss* case (described later), one can see from the table that there are no access messages that reached the head version of the object directly. Furthermore, both, AVL tree and RB tree, have their maximum at a proxy chain length of 2 hops, but for the AVL tree, this maximum does not peak as highly as for the RB tree. That means that at least two, in most cases three new head versions came into existence between two subsequent accesses.

Figure 9.7 shows that especially the AVL tree has a non-negligible probability to produce very long proxy chains. For the RB tree, the maximal proxy chain length that was found, varies between 5 ($k = 6$) and 23 ($k = 1$) for a cache with 50 objects, and between 14 and 56 for a cache



(a) AVL and RB Tree, Cache Size: 50.

(b) AVL and RB Tree, Cache Size: 450.

Figure 9.7: Probability for number of proxy forwards for AVL and RB tree after *bad cache hits*, simulation with 1 000 nodes.

size of 450 objects. The increase in the proxy chain length is a result of the increased number of outdated objects in the cache: The larger the cache, the longer an object can remain in the cache before it becomes evicted by the LRU policy. Thus, the longer an object stays in the cache, the more new object versions are created on other nodes and the longer the proxy chain grows.

For the AVL tree, the proxy chains in the simulations grow to 78 hops (cache size 50) and 177 hops (cache size 450). The reason for these high values are the frequent tree rotations that are necessary to balance the tree. Each such rotation creates new object versions and thereby increases the proxy chain length. These frequent rotations also explain the quickly decreasing *good cache hit* probability for the AVL tree in Figure 9.5, and the plateau at about 1 % in the AVL plot in Figure 9.7. Only the finite number of nodes in the simulated system keeps the proxy chains from growing even larger, because a finite number of nodes results in the probability that adding a proxy introduces a loop. These loops are automatically cut out by the algorithm, which overwrites the old forwarding pointer with the new one.

As described in Section 7.3, the optimal propagation depth k_{opt} for the *bad cache hits*, is determined by the average number of newly created head versions M and the number of reads R_{bad} that result in a *bad cache hit*. Both values can be determined at run time. Additionally, the simulations allow the measurement of the average proxy chain length ℓ_k that was traversed after a *bad cache hit*. Together with the simulation parameter k_{sim}, the average message cost c_{bad} can be computed as

$$c_{bad} = M \cdot k_{sim} + R \cdot \ell_k \tag{9.2}$$

Figure 9.8 shows the average message costs c_{bad} for the RB and the AVL tree as a function of k in millions of messages: The two linear curves at the bottom show the results obtained from the RB tree; the three U-shaped curves are from the AVL tree. With a cache size of 300 objects, the message costs are already close to the maximal message costs of both data structures. A larger cache does not increase the message costs anymore, because a larger cache only contains objects

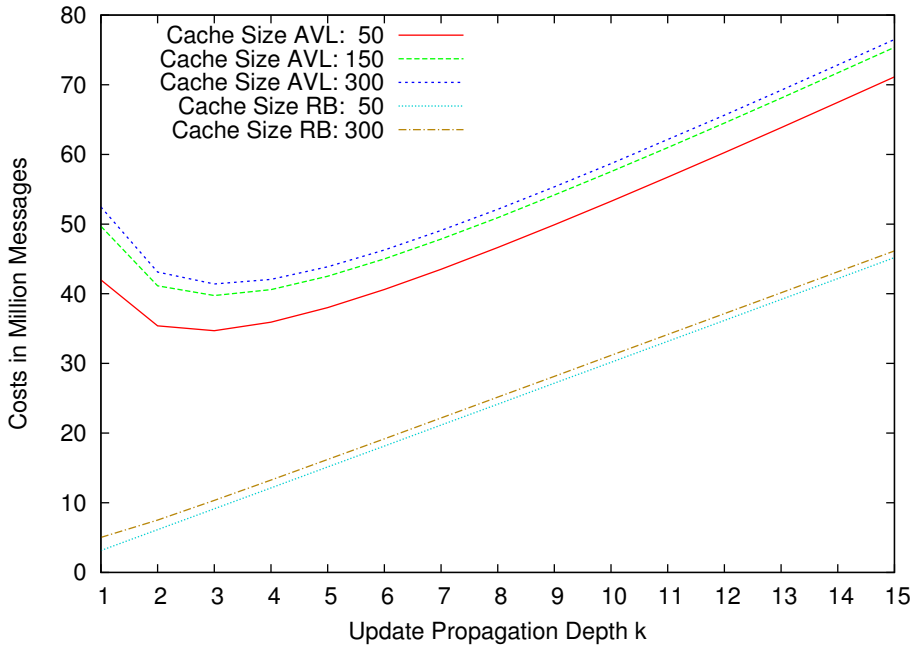


Figure 9.8: Message costs c_{bad} for AVL and RB tree, depending on k_{sim}, simulation with 1 000 nodes.

that will never be used again. For the AVL tree, a smaller cache slightly reduces the message cost, because it leads to shorter proxy chains. The reason for these shorter proxy chains is the need to fetch the latest head version after a *cache miss*. One sees that the minima of the message costs are at $k = 3$ for the AVL tree, and at $k = 1$ for the RB tree.

	Cache Size	M	R_{bad}	$l_{k=1}$	k_{opt}
AVL	50	3 955 611	2 066 916	18.40	3.10
	450	3 955 611	4 326 585	11.21	3.50
RB	50	3 004 185	62 151	2.31	0.22
	450	3 004 185	566 184	3.59	0.82

Table 9.13: Theoretical optimal propagation depth k_{opt} after *bad cache hits*, simulation with 1 000 nodes.

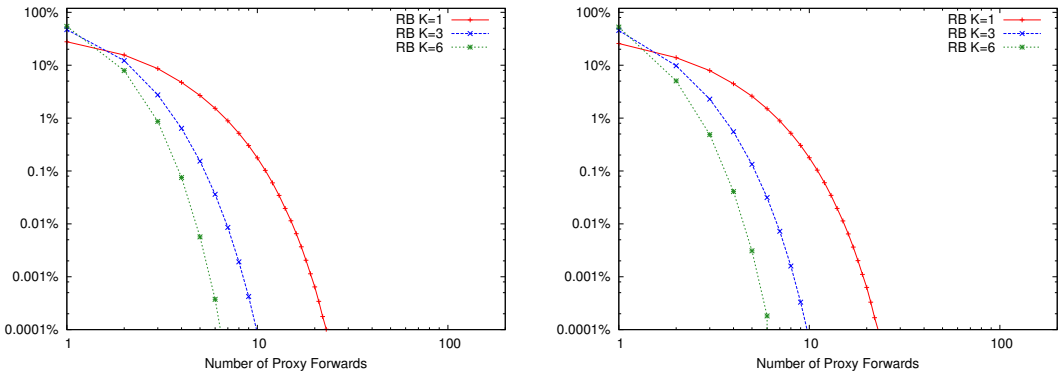
These experimental results shown in the graphs support the analytical findings from Table 9.13. This table shows the optimal propagation depth k_{opt}, which was computed with Equation (7.3) on page 90. Here, the numbers of total writes (implicit object migrations) M, total reads of outdated cached versions R_{bad} (cf. Table 9.11, column ④), together with the average proxy chain length $l_{k=1}$ have been taken from the simulation output.

One can see that it is sufficient for the RB tree to update only the latest proxy, and not to send additional update messages further down the proxy chain. For the AVL tree, the message costs are optimal if updates are sent to the three latest proxies in the proxy chain. However, the AVL

tree still needs about 25 million more messages than the RB tree. Thus, the AVL tree is not a well suited data structure for the envisioned scenario and the next section does not consider the AVL tree anymore.

9.2.4 Cache Misses

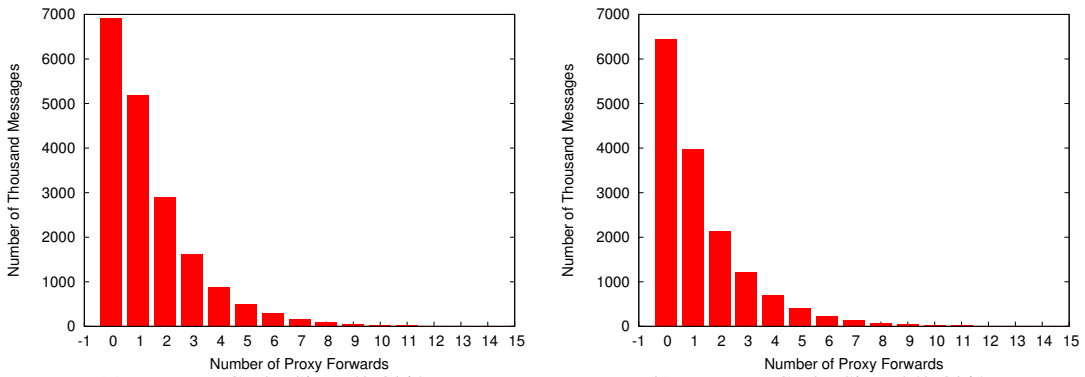
As stated above, the following section evaluates the access path optimization for *cache misses*, but only for the RB tree. Furthermore, it considers only the simulation runs with 1 000 threads, because the simulation runs with 500 threads do not deliver further insights.



(a) RB Tree, Cache Size: 50 Obj.'s. (b) RB Tree, Cache Size: 450 Obj.'s.

Figure 9.9: Probability for number of proxy forwards after *cache misses*.

Figure 9.9 shows the probability of numbers of proxy forwards that are needed to retrieve the head version of an object after a *cache miss*. (Again, the figures are plotted with a double logarithmic scale and show the two cache sizes of 50 and 450 objects.) The absolute values are shown as histograms in Figure 9.10.



(a) RB Tree, Cache Size: 50 Obj.'s. (b) RB Tree, Cache Size: 450 Obj.'s.

Figure 9.10: Histogram of proxy forwards after *cache misses*.

The conditions are the same as for the *bad cache hit* case: the simulation ran with various update propagation depths k but the figures only show $k = 1, 3, 6$. Table 9.14 shows the length distribution of the traversed proxy chain, i. e. the probability that the object's head version was reached directly or after the first, second, third, and fifth proxy forward.

Cache Size	k	0-hop	1-hop	2-hop	3-hop	5-hop
50	1	37.03	55.46	6.80	0.66	0.00
	3	37.03	39.99	14.97	5.13	0.67
	6	37.03	53.03	8.74	0.76	0.01
450	1	41.79	54.06	3.83	0.30	0.00
	3	41.79	38.24	12.77	4.54	0.62
	6	41.79	51.48	5.95	0.69	0.01

Table 9.14: Probability for number of proxy forwards after *cache misses*.

In contrast to the *bad cache hit* case, this time about 37.03 % of all messages for a cache size of 50 objects, and 41.79 % of all messages for a cache size of 450 objects, reached the head version directly. Figure 9.10 shows the histograms of the proxy forwards for $k = 1$, which correspond to the histogram of the RU approach in Figure 9.1(a) on page 120. It starts with 6 909 110 messages that reached the head version of the accessed object directly (without any proxy forwards), to 1 message that was forwarded 27 times.

Cache Size	M	R_{miss}	$l_{k=1}$	k_{opt}
50	3 004 185	19 138 733	1.44	3.02
450	3 004 185	15 932 218	1.35	2.68

Table 9.15: Theoretical optimal propagation depth k_{opt} after *cache misses*.

Again, the optimal propagation depth k_{opt} is determined by the average number of newly created head versions M, the number of reads R_{miss} that resulted in a *cache miss*, and the average proxy chain length $l_{k=1}$.

The result is shown in Table 9.15. Unlike for the *bad cache hit* case, the optimal propagation depth k_{opt} is not negligible. The reason is the larger number of read accesses R_{miss} that profit from shorter proxy chains. As seen in Table 9.15 the number of R_{miss} is 5.30 times higher than M for a cache size of 50 objects, and 6.37 times higher for a cache size of 450 objects. Thus, the analytic results for k_{opt} give an optimal propagation depth of 3.02 hops for a cache size of 50 objects, and 2.68 hops for a cache size with 450 objects.

To compare these findings to the simulation result, the average message cost c_{miss} is computed as:

$$c_{miss} = M \cdot k_{sim} + R_{miss} \cdot \ell_k \tag{9.3}$$

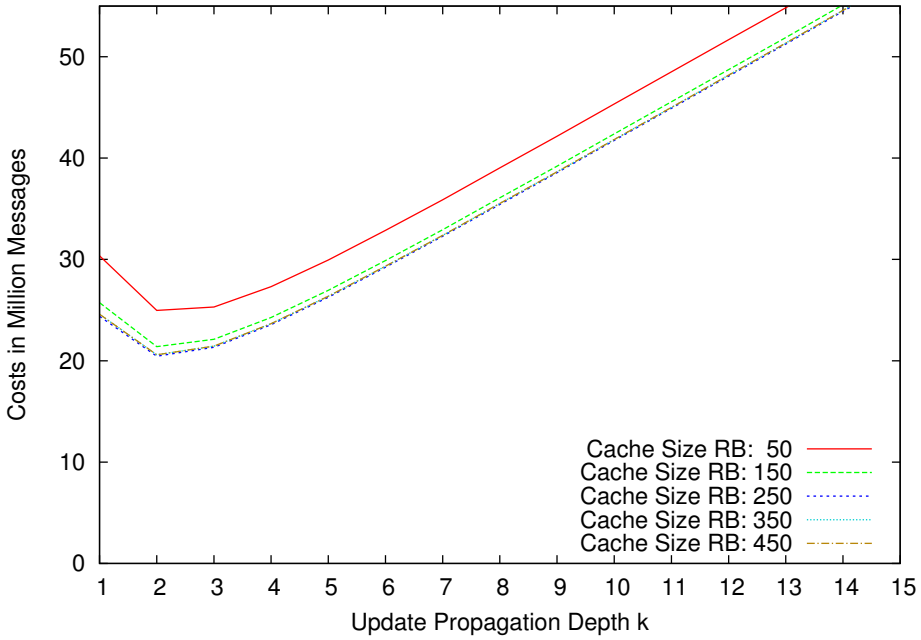


Figure 9.11: Message costs c_{miss} for RB tree, depending on k_{sim} after *cache misses*.

The result is shown in Figure 9.11. In contrast to Figure 9.8, where the message cost were the lowest for a cache size of 50 objects, they are the highest in this figure. The reason is the number of reads R_{bad}, which increases with the cache size for *bad cache hits*, cf. Table 9.11 on page 131, column ④, and R_{miss}, which decreases with the cache size for the *cache misses*, cf. Table 9.11, column ②, and thus influences the computation of c_{miss} in the same way. Furthermore, one can see again that the simulation results support the analytically predicted optimal propagation depth k_{opt}.

10 Conclusion

I conducted the research of this thesis in the context of two projects I worked in. The first is the *AmbiComp* project, which aimed at distributed, embedded systems. The second is the *J-Cell* project, which aims at high-performance compute systems. Both projects envision scenarios in which a distributed runtime environment offers applications a single system image, which allows the transparent access to all resources of the distributed system in which nodes might join and leave at any time.

The creation of this single system image requires a scalable and fully decentralized object location and retrieval algorithm. The development and evaluation of such an object location and retrieval algorithm is the main topic of my thesis.

In this thesis, I described and evaluated different object location approaches that are commonly used in the literature, for example, broadcast, central or distributed registries, static home nodes, or the proxy forwarding approach. All but the proxy forwarding approach either use centralized components, which contradict the desired scalability and decentralization of the envisioned system, or spread the management overhead throughout the system, even if only a small subset of nodes has an actual interest in the corresponding object. Thus, the conclusion is that only the proxy forwarding approach is applicable for the envisioned scenario. In this approach, proxies can form chains, which is especially helpful in the DecentSTM context, where a mutable object is represented by a chain of immutable object versions, with the head version in this chain being the most recently written object version. In the context of proxy forwarding, this head version can be seen as the actual object, while the outdated object versions are the proxies that forward all access requests to the head version.

I developed different location update protocols that use proxies to forward all object access requests to the current location of the object. The main application of this work is the use of one of these protocols together with DecentSTM. However, these protocols do not especially aim at systems that make use of the DecentSTM algorithm. Instead, they are applicable for systems that generally support object migration, for example for load balancing or the consolidation of an application onto fewer nodes to save energy.

The first protocol I developed is the reactive location update protocol, which updates outdated location information upon object access. I discussed the design decisions and obstacles, together with their solutions.

I evaluated this reactive location update protocol using an OMNeT++ network emulator. As expected, the results show that the protocol leaves long chains of proxies, which increase the object's access latency. Moreover, there is no way to delete the remaining proxies other than by a distributed garbage collection run.

One straight forward approach to reduce the access latency was to inform all referencing objects about the migration of the referenced object. Therefore, I proposed two versions of a

proactive location update protocol, which use additional incoming references to proactively send location update information upon object migration. These incoming references are backward pointers that lead to all referencing objects. They allow the propagation of the new object location of a migrated object to all nodes that hold referencing objects. This ensures that the location information at the home nodes of all referencing objects is almost always up-to-date. Additionally, it allows a proxy to detect when it is not referenced anymore, and can thus be deleted.

I described the protocol design with multiple state diagrams. Together with the protocol description, they give a first indication that the proactive location update protocol design is far more complex than the simple reactive location update. One of the main reasons for this complexity is the overhead that is required to keep the location information consistent across multiple nodes. Especially, it is not sufficient to keep all referencing objects up-to-date, but it is additionally necessary to keep all referenced objects updated as well. Moreover, one has to keep in mind that all objects in the system potentially migrate and also send location update messages.

This observation that the proactive location update protocol design is complex is supported by the results from the evaluation of this protocol in the OMNeT++ network emulator. These results showed a severe reference maintenance message overhead, which is more than twice as high as the actual object access traffic. Thus, the first main result of this thesis is that the proactive location update approach, even though it looked promising, is not worthwhile for the use in the envisioned system. It might be useful, however, for objects that require the lowest possible object access latency.

Hence, I developed an access path optimization for the reactive location update approach, which prevents long chains of proxies. Instead of updating all objects that reference a migrated object, this approach only propagates update messages down the proxy chain. To enable this propagation, the optimization approach uses 1-hop backward pointers, which are not necessary for the generic reactive location update approach, but which are required by the DecentSTM protocol to ensure a consistent object version history.

I developed an analytic formula to answer the question how deep these updates should be propagated down the proxy chain. This formula shows that the optimal propagation depth depends on the access characteristics for the object, namely, on the read-to-write ratio, and the length of the established proxy chain, which is caused by a number of implicit object migrations. Provided these numbers are given, the formula allows the computation of an optimal propagation depth that optimizes the total message costs for update propagation and object access. In conclusion, it is worthwhile to send location updates further down the proxy chain, if the object is more often read than it is written on remote nodes (implicitly migrated). Conversely, it is not worthwhile to propagate the location update deep down the proxy chain if the object is often written but seldom read.

I tested these analytic results with a software simulation that allowed larger networks and more objects than the OMNeT++ emulator. I used this software simulation for two purposes: First, I confirmed the results from the OMNeT++ emulation for the reactive location update protocol. Secondly, I observed the message costs for different location update propagation depths. The experimental results that measured these message costs matched the analytic results for an optimal update propagation depth and supported the correctness of the formula. Hence, the second main result of this thesis is a formula that allows the computation of the optimal update

propagation depth. This optimal update propagation depth minimizes the total message costs for object accesses and object migrations, if the object access characteristic is known.

In addition to the optimal propagation depth, the software simulation examined the caching characteristics of the two benchmark applications. Namely, it examined the likelihood that an optimistically executed transaction succeeds when it operates on cached, and thus potentially outdated, object versions. Here, the results from the software simulation have shown that it depends largely on the used data structure and its access pattern how long a cached object version is valid. With these results, I was able to classify the used benchmark applications and data structures with respect to their applicability for the envisioned scenario.

In conclusion, this work provides recommendations for the design of the remote object access for upcoming distributed compute clusters or embedded distributed systems.

List of Tables

List of Figures

Bibliography

References for Chapter 1, Introduction

[Adv10] Advanced Micro Devices, Inc. *Press Release: AMD's 16-Core "Interlagos" Server Processor Named a "Top New Product to Watch" for High Performance Computing*. `http://www.amd.com/us/press-releases/Pages/16-core-interlagos-2010nov16.aspx`. Nov. 16, 2010.

[BCJ01] Rajkumar Buyya, Toni Cortes, and Hai Jin. "Single System Image". In: *International Journal of High Performance Computing Applications* 15.2 (2001), pp. 124–135.

[Bro06] Manfred Broy. "Challenges in Automotive Software Engineering". In: *Proc. of the 28th Int'l Conf. on Software Engineering (ICSE'06)*. Shanghai, China, May 20–28, 2006.

[BZ07] Victor R. Basili and Marvin V. Zelkowitz. "Empirical Studies to Build a Science of Computer Science". In: *Communications of the ACM* 50 (11 Nov. 2007), pp. 33–37.

[FM11] Samuel H. Fuller and Lynette I. Millett. "Computing Performance: Game Over or Next Level?" In: *Computer* 44.1 (Jan. 2011), pp. 31–38.

[Hev+04] Alan R. Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), pp. 74–105.

[Int11] Intel Corporation. *The SCC Platform Overview, Revision 0.7*. Version 0.7. 2011.

[Kul+09] Milind Kulkarni et al. "How Much Parallelism is There in Irregular Applications?" In: *ACM SIGPLAN Notices - PPoPP'09* 44 (4 Feb. 2009), pp. 3–14.

[LH02] Gabriel Leen and Donal Heffernan. "Expanding Automotive Electronic Systems". In: *IEEE Computer* (Jan. 2002), pp. 88–93.

[Pfi98] Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-899709-8.

[Pha+05] Diep Pham et al. "The Design and Implementation of a First-Generation Cell Processor". In: *Proc. of the IEEE Int'l Solid-State Circuits Conf. (ISSCC'05) Digest of Technical Paper*. San Francisco, California, USA, Feb. 10, 2005, pp. 184–592.

[SG10] Bianca Schroeder and Garth A. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (Oct.–Dec. 2010), pp. 337–351.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. “Scaling Parallel Programs for Multiprocessors: Methodology and Examples”. In: *Computer* 26 (7 July 1993), pp. 42–50.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. “DRAM Errors in the Wild: A Large-Scale Field Study”. In: *Proc. of the 11th Int’l Joint Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS’09)*. Seattle, WA, USA, June 15–19, 2009.

[TOP11] TOP500.org. *TOP 500 Supercomputer Sites*. `http://top500.org`. 2011.

References for Chapter 2, Background

[All10] OSGi Alliance. *OSGi Service Platform, Core Specification - Release 4, Version 4.3 - Early Draft 2*. OSGi Alliance, Aug. 31, 2010.

[Amd67] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the Spring Joint Computer Conf. (AFIPS’67)*. Atlantic City, New Jersey, USA, Apr. 18–20, 1967, pp. 483–485.

[Arg11] Argonne National Laboratory. *MPICH2 Website*. `http://www.mcs.anl.gov/research/projects/mpich2/`. 2011.

[Arn+99] Ken Arnold et al. *Jini Specification*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201616343.

[Bal+09] Pavan Balaji et al. “Toward Message Passing for a Million Processes: Characterizing MPI, on a Massive Scale Blue Gene/P”. In: *Computer Science - Research and Development* 24 (1 2009), pp. 11–19.

[Bas+08] Victor R. Basili et al. “Understanding the High-Performance-Computing Community: A Software Engineer’s Perspective”. In: *IEEE Software* 25.4 (July–Aug. 2008), pp. 29–36.

[BCA07] Christopher Barton, Călin Cascaval, and Nelson Amaral José. “A Characterization of Shared Data Access Patterns in UPC Programs”. In: *Proc. of the 19th Int’l Conf. on Languages and Compilers for Parallel Computing (LCPC’06)*. New Orleans, Louisiana, USA, 2007, pp. 111–125.

[BCJ01] Rajkumar Buyya, Toni Cortes, and Hai Jin. “Single System Image”. In: *International Journal of High Performance Computing Applications* 15.2 (2001), pp. 124–135.

[BEF10] Annette Bieniusa, Johannes Eickhold, and Thomas Fuhrmann. “The Architecture of the DecentVM – Towards a Decentralized Virtual Machine for Many-Core Computing”. In: *Proc. of the 4th Workshop on Virtual Machines and Intermediate Languages (VMIL’10)*. Reno, Nevada, USA, Oct. 17, 2010.

[Ber+04] Konstantin Berlin et al. “Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures”. In: Springer Berlin / Heidelberg, 2004. Chap. Languages and Compilers for Parallel Computing, pp. 194–208.

[BF10] Annette Bieniusa and Thomas Fuhrmann. “Consistency in Hindsight, A Fully Decentralized STM Algorithm”. In: *Proc. of the IEEE Int’l Symp. on Parallel Distributed Processing (IPDPS’10)*. Atlanta, Georgia, USA, Apr. 2010, pp. 1–12.

[Bon+06] Dan Bonachea et al. *Titanium Language Reference Manual*. Tech. rep. UCB/EECS-2005-15.1. University of California, Berkeley, Aug. 2006.

[Buy00] Rajkumar Buyya. “PARMON: A Portable and Scalable Monitoring System for Clusters”. In: *Software: Practice and Experience* 30.7 (2000), pp. 723–739.

[CCZ04] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. “The Cascade High Productivity Language”. In: *Proc. of the 9th Int’l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS’04)*. Santa Fe, New Mexico, USA, Apr. 26, 2004, pp. 52–60.

[CDC99] William W. Carlson, Jesse M. Draper, and David E. Culler. *Introduction to UPC and Language Specification*. Tech. rep. CCS-TR-99-157. George Washington University, 1999.

[CE00] Franck Cappello and Daniel Etiemble. “MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks”. In: *Proc. of the ACM/IEEE Conf. on Supercomputing (SC’00)*. Dallas, Texas, USA, Nov. 4–10, 2000.

[Cha+05] Philippe Charles et al. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In: *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’05)*. San Diego, California, USA, Oct. 16–20, 2005, pp. 519–538.

[Cra11] Cray Inc. *Chapel Language Specification, Version 0.8*. `http://chapel.cray.com/spec/spec-0.8.pdf`. Apr. 11, 2011.

[FGK03] Ian Foster, William Gropp, and Carl Kesselman. “Sourcebook of Parallel Computing”. In: San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. Chap. Message Passing and Threads, pp. 313–329.

[Fuh05] Thomas Fuhrmann. “Scalable Routing for Networked Sensors and Actuators”. In: *Proc. of the 2nd Annual IEEE Communications Society Conf. on Sensor and Ad Hoc Communications and Networks (SECON’05)*. Santa Clara, California, USA, Sept. 26–29, 2005, pp. 240–251.

[Gei+90] George Al Geist et al. *A User’s Guide to PICL a Portable Instrumented Communication Library*. Tech. rep. ORNL/TM-11616. Oak Ridge National Lab, Oct. 1990.

[Gho+98] Douglas P. Ghormley et al. “GLUnix: A Global Layer Unix for a Network of Workstations”. In: *Software: Practice and Experience* 28.9 (1998), pp. 929–961.

[Goe+06] Brian Goetz et al. *Java Concurrency in Practice*. 1. Addison-Wesley Professional, 2006.

[HM93] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proc. of the 20th Annual Int'l Symposium on Computer Architecture (ISCA'93)*. San Diego, California, USA, May 16–19, 1993, pp. 289–300.

[IEE04] IEEE Standard Organization. *IEEE Std 1003.1,2004 Edition*. `http://www.unix.org/version3/ieee_std.html`. 2004.

[KB09] Hahn Kim and Robert Bond. "Multicore Software Technologies". In: *IEEE Signal Processing Magazine* 26.6 (Nov. 2009), pp. 80–89.

[Kus+94] Jeffrey Kuskin et al. "The Stanford FLASH Multiprocessor". In: *ACM SIGARCH Computer Architecture News* 22 (2 Apr. 1994), pp. 302–313.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. 2. Addison-Wesley Longman, 1999.

[Lee06] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (2006), pp. 33–42.

[Liu+05] Hongzhou Liu et al. "Design and Implementation of a Single System Image Operating System for Ad Hoc Networks". In: *Proc. of the 3rd Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys'05)*. Seattle, Washington, USA, 2005, pp. 149–162.

[Luf09] Meredydd Luff. "Empirically Investigating Parallel Programming Paradigms: A Null Result". In: *Proc. of the Evaluation and Usability of Programming Languages and Tools (PLATEAU'09)*. Orlando, Florida, USA, Oct. 25–29, 2009.

[Mal+09] Damián A. Mallón et al. "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures". In: *Proc. of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland, Sept. 7–10, 2009, pp. 174–184.

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. "Subtleties of Transactional Memory Atomicity Semantics". In: *IEEE Computer Architecture Letters* 5 (2 July 2006).

[Neu45] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. Philadelphia, Pennsylvania, USA: University of Pennsylvania, Moore School of Electrical Engineering, June 30, 1945.

[OB87] Ross A. Overbeek and James Boyle. *Portable Programs for Parallel Processors*. Philadelphia, PA, USA: Saunders College Publishing, 1987. ISBN: 0030144043.

[Obj04] Object Management Group, Inc. *Common Object Request Broker: Core Specification, OMA: formal/04-03-12*. `http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf`. Version 3.0.3. 2004.

[Ope11] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.1*. `http://openmp.org/wp/openmp-specifications/`. July 2011.

[PAO09] Victor Pankratius, Ali-Reza Adl-Tabatabai, and Frank Otto. *Does Transactional Memory Keep Its Promises? Results from an Empirical Study*. Tech. rep. TR 2009-12. University of Karlsruhe, Germany: Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, Sept. 2, 2009.

[Pfi98] Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-899709-8.

[PG08] Imran Patel and John R. Gilbert. "An Empirical Study of the Performance and Productivity of Two Parallel Programming Models". In: *Proc. of the IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS'08)*. Miami, Florida, USA, Apr. 14–18, 2008, pp. 1–7.

[PGL94] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. "Parallel Visualization Algorithms: Performance and Architectural Implications". In: *Computer* 27.7 (July 1994), pp. 45–55.

[Pos10] Stephan-Alexander Posselt. "Design of a Reliable, Fully Decentralized Software Transactional Memory Protocol". Diploma thesis. Munich, Germany: Technische Universität München, Aug. 2010.

[Rec+07] Renato J. Recio et al. *A Remote Direct Memory Access Protocol Specification*. RFC 5040 (Proposed Standard). Internet Engineering Task Force, Oct. 2007.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *Proc. of the 17th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing (PDP'09)*. Weimar, Germany, Feb. 18–20, 2009, pp. 427–436.

[RHW10] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. "Is Transactional Programming Actually Easier?" In: *ACM SIGPLAN Notices - PPoPP '10* 45 (5 Jan. 2010), pp. 47–56.

[Sar+11] Vijay Saraswat et al. *X10 Language Specification, Version 2.2*. `www.x10-lang.org`. May 31, 2011.

[SB01] Lorna A. Smith and J. Mark Bull. "Development of Mixed Mode MPI/OpenMP Applications". In: *Scientific Programming* 9.2,3 (Aug. 2001), pp. 83–98. ISSN: 1058-9244.

[Sha+01] Hongzhang Shan et al. "Message Passing Vs. Shared Address Space on a Clusters of SMPs". In: *Proc. of the 15th Int'l Parallel & Distributed Processing Symp. (IPDPS '01)*. 2001, p. 63.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. "A Component Architecture for LAM/MPI". In: *Proc. of the 10th European PVM/MPI Users' Group Meeting (Euro PVM/MPI'03)*. Venice, Italy, Sept. 29–Oct. 2, 2003, pp. 379–387.

[SL05] Herb Sutter and James Larus. "Software and the Concurrency Revolution". In: *ACM Queue* 3.7 (Sept. 2005), pp. 54–62.

[Sni+98] Marc Snir et al. *MPI - The Complete Reference: The MPI Core*. 2nd. Vol. 1. The MIT Press, 1998.

[SPF11] Björn Saballus, Stephan-Alexander Posselt, and Thomas Fuhrmann. "A Scalable and Robust Runtime Environment for SCC Clusters". In: *Proc. of the 3rd MARC Symposium*. Ettlingen, Germany, July 5–6, 2011.

[SS10] Caitlin Sadowski and Andrew Shewmaker. "The Last Mile: Parallel Programming and Usability". In: *Proc. of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10)*. Santa Fe, New Mexico, USA, Nov. 7–8, 2010, pp. 309–314.

[ST95] Nir Shavit and Dan Touitou. "Software Transactional Memory". In: *Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*. Ottowa, Ontario, Canada, Aug. 20–23, 1995, pp. 204–213.

[Sun90] Vaidy S. Sunderam. "PVM: A Framework for Parallel Distributed Computing". In: *Concurrency: Practice and Experience* 2.4 (1990), pp. 315–339.

[The05] The UPC Consortium. *UPC Language Specification, Version 1.2*. url-http://upc.gwu.edu/documentation.html. May 31, 2005.

[Wal92] David W. Walker. *Standards for Message-Passing in a Distributed Memory Environment*. Tech. rep. ORNL/TM-12147. Oak Ridge National Laboratory, Aug. 1992.

[Yel+98] Kathy Yelick et al. "Titanium: A High-Performance Java Dialect". In: *Concurrency: Practice and Experience* 10.11-13 (1998), pp. 825–836.

References for Chapter 3, Related Work

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. "Linda and Friends". In: *Computer* 19.8 (1986), pp. 26–34.

[Adl+96] Ali-Reza Adl-Tabatabai et al. "Efficient and Language-Independent Mobile Programs". In: *SIGPLAN Notices* 31 (5 May 1996), pp. 127–136.

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. "cJVM: A Single System Image of a JVM on a Cluster". In: *Proc. of the Int'l Conf. on Parallel Processing (ICPP'99)*. Wakamatsu, Japan, Sept. 21–24, 1999, pp. 4–11.

[AHN02] Sara Alouf, Fabrice Huet, and Philippe Nain. "Forwarders vs. Centralized Server: An Evaluation of Two Approaches for Locating Mobile Agents". In: *Performance Evaluation* 49.1-4 (2002), pp. 299–319.

[Aky+02] Ian F. Akyildiz et al. "Wireless Sensor Networks: A Survey". In: *Computer Networks* 38.4 (2002), pp. 393–422.

[Alp+05] Bowen Alpern et al. "The Jikes Research Virtual Machine Project: Building an Open-Source Research Community". In: *IBM Systems Journal* 44.2 (2005), pp. 399–417.

[Amz+96] Cristiana Amza et al. “TreadMarks: Shared Memory Computing on Networks of Workstations”. In: *Computer* 29 (1996), pp. 18–28.

[And+01] Johan Andersson et al. “Kaffemik –A distributed JVM on a Single Address Space Architecture”. In: *Proc. of the 4th Int’l Conf. on SCI-based Technology and Research (SCI-Europe’01)*. Dublin, Ireland, Oct. 1–3, 2001.

[Ant+01] Gabriel Antoniu et al. “The Hyperion System: Compiling Multithreaded Java Bytecode for Distributed Execution”. In: *Parallel Computing* 27.10 (2001), pp. 1279–1297.

[APF10] Muhammad Aleem, Radu Prodan, and Thomas Fahringer. “JavaSymphony: A Programming and Execution Environment for Parallel and Distributed Many-Core Architectures”. In: *Proc. of the European on Parallel Processing (Euro-Par’10)*. Naples, Italy, Aug. 31–Sept. 3, 2010.

[Ari+00] Yariv Aridor et al. “A High Performance Cluster JVM Presenting a Pure Single System Image”. In: *Proc. of the ACM Conf. on Java Grande (JAVA’00)*. San Francisco, California, USA, 2000, pp. 168–177.

[Arm+09] Michael Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. Electrical Engineering and Computer Sciences, University of California at Berkeley, Feb. 9, 2009.

[Azu11] Azul Systems, Inc. *Zing Java Virtual Machine*. `http://www.azulsystems.com/`. 2011.

[Bad+06] Laurent Baduel et al. “Programming, Composing, Deploying for the Grid”. In: *Grid Computing: Software Environments and Tools*. Springer London, 2006, pp. 205–229.

[Bal+98] Henri E. Bal et al. “Performance Evaluation of the Orca Shared-Object System”. In: *ACM Transactions on Computer Systems (TOCS)* 16 (1 Feb. 1998), pp. 1–40.

[Bau+00] Françoise Baude et al. “Communicating Mobile Active Objects in Java”. In: *Proc. of the 8th Int’l Conf. on High-Performance Computing and Networking (HPCN-Europe’00)*. 2000, pp. 633–643.

[BK07] Jonas Bonér and Eugene Kuleshov. “Clustering the Java Virtual Machine Using Aspect-Oriented Programming”. In: *Proc. of the 6th Int’l Conf. on Aspect-Oriented Software Development (AOSD’07)*. Vancouver, British Columbia, Canada, Mar. 12–16, 2007.

[BMT03] Mario Bisignano, Giuseppe Di Modica, and Orazio Tomarchio. “Mobile Agent Location Management: A Comparison Between CORBA and P2P Based Systems”. In: *IEEE Symp. on Computers and Communications* (2003), p. 1029.

[Buy99] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems, Vol. 1*. Prentice Hall, 1999. ISBN: 978-0130137845.

[Cao+02] Jiannong Cao et al. “Mailbox-Based Scheme for Mobile Agent Communications”. In: *Computer* 35.9 (Sept. 2002), pp. 54–60.

[Cao+03] Jiannong Cao et al. “Path Compression in Forwarding-Based Reliable Mobile Agent Communications”. In: *Proc. of the 32nd Int’l Conf. on Parallel Processing (ICPP’03)*. Kaohsiung, Taiwan, Oct. 6–9, 2003, pp. 313–320.

[CGL07] Min Chen, Sergio Gonzalez, and Victor C. M. Leung. “Applications and Design Issues for Mobile Agents in Wireless Sensor Networks”. In: *IEEE Wireless Communications* 14.6 (Dec. 2007), pp. 20–26.

[Chu+97] P. Emerald Chung et al. *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*. `http://research.microsoft.com/en-us/um/people/ymwang/papers/html/dcomncorba/s.html`. Sept. 3, 1997.

[CWH99] Benny Wang-Leung Cheung, Cho-Li Wang, and Kai Hwang. “A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations”. In: *Proc. of the Int’l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA’99)*. Las Vegas, Nevada, USA, 1999.

[Day+93] Mark Day et al. “References to Remote Mobile Objects in Thor”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (1993), pp. 115–126.

[DH98] Michael J. Demmer and Maurice Herlihy. “The Arrow Distributed Directory Protocol”. In: *Proc. of the 12th Int’l Symp. on Distributed Computing (DISC ’98)*. London, UK: Springer-Verlag, 1998, pp. 119–133. ISBN: 3-540-65066-0.

[Dow98] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. 1st. Foster City, California, USA: IDG Books Worldwide, Inc., 1998.

[DS98] Kevin Dowd and Charles Severance. *High Performance Computing*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1998. ISBN: 1-56592-032-5.

[EE98] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[Fah00] Thomas Fahringer. “JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications”. In: *Proc. of the IEEE Int’l Conf. on Cluster Computing (CLUSTER’00)*. Chemnitz, Germany, Nov. 28–Dec. 1, 2000, pp. 145–152.

[FJ05] Thomas Fahringer and Alexandru Jugravu. “JavaSymphony: A New Programming Paradigm to Control and Synchronize Locality, Parallelism and Load Balancing for Parallel and Distributed Computing”. In: *Concurrency and Computation: Practice and Experience* 17.7-8 (2005), pp. 1005–1025.

[Fos02] Ian Foster. “What is the Grid? A Three Point Checklist”. In: *GRIDtoday* 6 (July 2002).

[Fos+03] Ian Foster et al. “The Physiology of the Grid”. In: *Grid Computing*. John Wiley & Sons, Ltd, 2003. Chap. 8, pp. 217–249. ISBN: 9780470867167.

[FSS03] Michael Factor, Assaf Schuster, and Konstantin Shagin. “JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Commodity Workstations”. In: *Proc. of the 5th IEEE Int’l Conf. on Cluster Computing (CLUSTER’03)*. Hong Kong, China, Dec. 1–4, 2003, pp. 110–117.

[FSS04] Michael Factor, Assaf Schuster, and Konstantin Shagin. "A Distributed Runtime for Java: Yesterday and Today". In: *Proc. of the 18th Int'l Parallel and Distributed Processing Symp. (IPDPS'04)*. Santa Fe, New Mexico, USA, Apr. 26–30, 2004.

[FWL02] Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. "Efficient Global Object Space Support for Distributed JVM on Cluster". In: *Proc. of the Int'l Conf. on Parallel Processing (ICPP '02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 371–378. ISBN: 0-7695-1677-7.

[FWL03] Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. "On the Design of Global Object Space for Efficient Multi-Threading Java Computing on Clusters". In: *Parallel Computing* 29.11-12 (2003), pp. 1563–1587.

[Gel85] David Hillel Gelernter. "Generative Communication in Linda". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7 (1 Jan. 1985), pp. 80–112.

[Göc+04] Ralph Göckelmann et al. "Plurix, A Distributed Operating System Extending the Single System Image Concept". In: *Proc. of the Canadian Conf. on Electrical and Computer Engineering (CCECE'04)*. Vol. 4. Niagara Falls, Ontario, Canada, May 2–5, 2004, 1985 –1988 Vol.4.

[Gos95] James Gosling. "Java Intermediate Bytecodes". In: *Proc. of the 22nd ACM Symp. on Principles of Programming Languages Papers (POPL'95)*. San Francisco, California, USA, Jan. 23–25, 1995, pp. 111–118.

[Hay08] Brian Hayes. "Cloud Computing". In: *Communications of the ACM* 51 (7 July 2008), pp. 9–11.

[Hen08] Michi Henning. "The Rise and Fall of CORBA". In: *Communications of the ACM* 51.8 (Aug. 2008), pp. 52–57.

[Hen98] Michi Henning. "Binding, Migration, and Scalability in CORBA". In: *Communications of the ACM* 41.10 (1998), pp. 62–71.

[Her99] Maurice Herlihy. "The Aleph Toolkit: Support for Scalable Distributed Shared Objects". In: *Network-Based Parallel Computing: Communication, Architecture, and Applications*. 1999, pp. 137–149.

[HW99] Maurice Herlihy and Michael P. Warres. "A Tale of Two Directories: Implementing Distributed Shared Objects in Java". In: *Proc. of the ACM Conf. on Java Grande (JAVA'99)*. 1999, pp. 99–108.

[IEE93] IEEE Standard Organization. *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface*. Aug. 1993.

[JES11] JESSICA3 Project. *An Advanced Distributed Java Virtual Machine on Commodity Clusters for High-Performance Memory-Intensive Computing*. `http://i.cs.hku.hk/~clwang/projects/JESSICA2/jessica3.htm`. 2011.

[JES11] JESSICA4 Project. *JESSICA4 Project*. `http://i.cs.hku.hk/~clwang/projects/JESSICA4.htm`. 2011.

[Jul+88] Eric Jul et al. “Fine-Grained Mobility in the Emerald System”. In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 109–133.

[Kaf11] Kaffe.org Website. *Kaffe VM*. `http://www.kaffe.org/`. 2011.

[Kel95] Peter Keleher. “Lazy Release Consistency for Distributed Shared Memory”. PhD thesis. Rice University, Houston, Texas, 1995.

[Kna97] Frederick Knabe. “An Overview of Mobile Agent Programming”. In: *Analysis and Verification of Multiple-Agent Languages*. Vol. 1192. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, pp. 100–115.

[Kus+94] Jeffrey Kuskin et al. “The Stanford FLASH Multiprocessor”. In: *ACM SIGARCH Computer Architecture News* 22 (2 Apr. 1994), pp. 302–313.

[LDS93] Barbara Liskov, Mark Day, and Liuba Shrira. “Distributed Object Management in Thor”. In: *Distributed Object Management* (1993), pp. 79–91.

[LH89] Kai Li and Paul Hudak. “Memory Coherence in Shared Virtual Memory Systems”. In: *ACM Transactions on Computer Systems (TOCS)* 7.4 (1989), pp. 321–359.

[Liu+05] Hongzhou Liu et al. “Design and Implementation of a Single System Image Operating System for Ad Hoc Networks”. In: *Proc. of the 3rd Int’l Conf. on Mobile Systems, Applications, and Services (MobiSys’05)*. Seattle, Washington, USA, 2005, pp. 149–162.

[LO99] Danny B. Lange and Mitsuru Oshima. “Seven Good Reasons for Mobile Agents”. In: *Communications of the ACM* 42 (3 Mar. 1999), pp. 88–89.

[Maa+01] Jason Maassen et al. “Efficient Java RMI for Parallel Programming”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23 (6 Nov. 2001), pp. 747–775.

[Mic11] Microsoft Corporation. *Distributed Component Object Model (DCOM) Remote Protocol Specification*. `http://download.microsoft.com/download/a/e/6/ae6e4142-aa58-45c6-8dcf-a657e5900cd3/%5BMS-DCOM%5D.pdf`. June 17, 2011.

[Mil+00] Dejan S. Milojicic et al. “Process Migration”. In: *ACM Computing Surveys (CSUR)* 32 (3 Sept. 2000), pp. 241–299.

[MMH98] Mark W. MacBeth, Keith A. McGuigan, and Philip J. Hatcher. “Executing Java Threads in Parallel in a Distributed-Memory Environment”. In: *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON’98)*. Toronto, Ontario, Canada: IBM Press, 1998, pp. 40–54.

[Moc87] Paul V. Mockapetris. *Domain Names - Implementation and Specification*. RFC 1035 (Standard). Internet Engineering Task Force, Nov. 1987.

[MPP87] Barton P. Miller, David L. Presotto, and Michael L. Powell. “DEMOS/MP: The Development of a Distributed Operating System”. In: *Software - Practice & Experience* 17 (4 Apr. 1987), pp. 277–290.

[MR02] Luc Moreau and Daniel Ribbens. “Mobile Objects in Java”. In: *Scientific Programming* 10.1 (2002), pp. 91–100. ISSN: 1058-9244.

[MRS08] Anolan Milanés, Noemi de La Rocque Rodriguez, and Bruno Schulze. “State of the Art in Heterogeneous Strong Migration of Computations”. In: *Concurrency and Computation: Practice and Experience* 20.13 (2008), pp. 1485–1508.

[MS10] Ross McIlroy and Joe Sventek. “Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures”. In: *Proc. of the 25th ACM Int’l Conf. on Object Oriented Programming Systems Languages, and Applications (OOPSLA ’10)*. 2010, pp. 205–222.

[MWL00a] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. “JESSICA: Java-Enabled Single-System-Image Computing Architecture”. In: *Journal of Parallel and Distributed Computing* 60.10 (2000), pp. 1194–1222.

[MWL00b] Matchy Ma, Cho-Li Wang, and Francis Lau. “Delta Execution: A Preemptive Java Thread Migration Mechanism”. In: *Cluster Computing* 3.2 (2000), pp. 83–94.

[MZ01] Cesar Munoz and Janusz Zalewski. “Architecture and Performance of Java-Based Distributed Object Models: CORBA vs RMI”. In: *Real-Time Systems* 21.1/2 (2001), pp. 43–75.

[NGF08] Albert Noll, Andreas Gal, and Michael Franz. “CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor”. In: *Proc. of the Workshop on Cell Systems and Applications*. Beijing, China, June 2008.

[Obj02] Object Management Group, Inc. *Life Cycle Service Specification, Version 1.2*. `http://www.omg.org/spec/LFCYC/1.2/PDF`. Version 1.2. Sept. 2002.

[Obj04] Object Management Group, Inc. *Naming Service Specification, Version 1.3*. `http://www.omg.org/spec/NAM/1.3/PDF`. Version 1.3. 2004.

[Obj08a] Object Management Group, Inc. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1*. `http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF/`. Version 3.1. 2008.

[Obj08b] Object Management Group, Inc. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2*. `http://www.omg.org/spec/CORBA/3.1/Interoperability/PDF`. Version 3.1. 2008.

[OTG02] Scott Oaks, Bernard Traversat, and Li Gong. *JXTA in a Nutshell*. O’Reilly Media, 2002.

[Pfi98] Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-899709-8.

[PHN00] Michael Philippsen, Bernhard Haumacher, and Christian Nester. “More Efficient Serialization and RMI for Java”. In: *Concurrency: Practice and Experience* 12.7 (2000), pp. 495–518.

[PK98] Vu Anh Pham and Ahmed Karmouch. “Mobile Software Agents: An Overview”. In: *Communications Magazine, IEEE* 36.7 (July 1998), pp. 26–37.

[PM83] Michael L. Powell and Barton P. Miller. “Process Migration in DEMOS/MP”. In: *Proc. of the 9th ACM Symp. on Operating Systems Principles (SOSP’83)*. Bretton Woods, New Hampshire, United States, Oct. 10–13, 1983, pp. 110–119.

[PR99] Charles E. Perkins and Elisabeth M. Royer. “Ad hoc On-Demand Distance Vector Routing”. In: *Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA’99)*. New Orleans, LA, USA, Feb. 25–26, 1999, pp. 90–100.

[PS01] Evaggelia Pitoura and George Samaras. “Locating Objects in Mobile Computing”. In: *IEEE Transactions on Knowledge and Data Engineering* 13.4 (2001), pp. 571–592.

[PS98] Frantisek Plasil and Michael Stal. “An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM”. In: *Software - Concepts & Tools* 19 (1 1998), pp. 14–28.

[PZ97] Michael Philippsen and Matthias Zenger. “JavaParty – Transparent Remote Objects in Java”. In: *Concurrency: Practice and Experience* 9.11 (Nov. 1997), pp. 1225–1242.

[SJ95] Brjarne Steensgaard and Eric Jul. “Object and Native Code Thread Mobility Among Heterogeneous Computers”. In: *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP’95)*. Copper Mountain, Colorado, USA, Dec. 3–6, 1995, pp. 68–77.

[Sri95] Raj Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 1831 (Proposed Standard). Obsoleted by RFC 5531. Internet Engineering Task Force, Aug. 1995. URL: `http://www.ietf.org/rfc/rfc1831.txt`.

[Tan+91] Andrew S. Tanenbaum et al. “The Amoeba Distributed Operating System – A Status Report”. In: *Computer Communications* 14.6 (1991), pp. 324–335.

[Ter11a] Terracotta Inc. *A Technical Introduction to Terracotta*. `www.uwyn.com/download/intro_terracotta.pdf`. 2011.

[Ter11b] Terracotta Inc. *The Definitive Guide to Terracotta*. New York, USA: Apress Media LLC, Mar. 2011.

[YMG08] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. “Wireless Sensor Network Survey”. In: *Computer Networks* 52.12 (2008), pp. 2292–2330.

[Zhu+04] Wenzhang Zhu et al. “A New Transparent Java Thread Migration System Using Just-In-Time Recompilation”. In: *Proc. of the 16th IASTED Int’l Conf. on Parallel and Distributed Computing and Systems (PDCS’04)*. MIT Cambridge, USA. Nov. 2004, pp. 766 –771.

[Zhu05] Wenzhang Zhu. “Distributed Java Virtual Machine with Thread Migration”. PhD thesis. Mar. 31, 2005.

[ZWL02] Wenzhang Zhu, Cho-Li Wang, and Francis Chi-Moon Lau. “JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support”. In: *Proc. of the IEEE Int’l Conf. on Cluster Computing (CLUSTER’02)*. Chicago, Illinois, USA, Sept. 23–26, 2002, pp. 381–388.

References for Chapter 4, Environment

[Arn86] James Q. Arnold. “Shared Libraries on UNIX System V”. In: *Proc. of the USENIX Summer Conf.* Altanta, Georgia, USA, June 9–13, 1986, pp. 395–404.

[Asa+06] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 18, 2006.

[Bar10] Max Baron. “The Single-chip Cloud Computer – Intel Networks 48 Pentiums on a Chip”. In: *Microprocessor Report* (2010).

[Dal08] DalvikVM.com. *Dalvik Virtual Machine*. `http://www.dalvikvm.com/`. 2008.

[Eic+08] Johannes Eickhold et al. “AmbiComp: A Platform for Distributed Execution of Java Programs on Embedded Systems by Offering a Single System Image”. In: *Proc. of the AmI-Blocks Workshop at the European Conf. on Ambient Intelligence (AmI-Blocks’08)*. Nuremberg, Germany, Nov. 19, 2008.

[Fla+05] Brian Flachs et al. “A Streaming Processing Unit for a Cell Processor”. In: *Proc. of the IEEE Int’l Solid-State Circuits Conf. (ISSCC’05)*. San Francisco, California, USA, Feb. 10, 2005, pp. 134–135.

[Göd+07] Dominik Göddeke et al. “Exploring Weak Scalability for FEM Calculations on a GPU-Enhanced Cluster”. In: *Parallel Computing* 33.10-11 (2007), pp. 685 –699.

[Gos95] James Gosling. “Java Intermediate Bytecodes”. In: *Proc. of the 22nd ACM Symp. on Principles of Programming Languages Papers (POPL’95)*. San Francisco, California, USA, Jan. 23–25, 1995, pp. 111–118.

[Int11a] Intel Corporation. *The SCC Platform Overview, Revision 0.7*. Version 0.7. 2011.

[Int11b] Intel Corporation. *The SCC Programmer’s Guide, Revision 0.61*. Version 0.61. 2011.

[Kah+05] James A. Kahle et al. “Introduction to the Cell Multiprocessor”. In: *IBM Journal of Research and Development* 49.4.5 (July 2005), pp. 589–604.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. “Cell Multiprocessor Communication Network: Built for Speed”. In: *IEEE Micro* 26.3 (May–June 2006), pp. 10–23.

[LH07] David Luebke and Greg Humphreys. “How GPUs Work”. In: *Computer* 40.2 (Feb. 2007), pp. 96–100.

[Mic11] SUN Microsystems. *Java ME Technical Documentation*. `http://download.oracle.com/javame/`. 2011.

[MW11] Tim Mattson and Rob F. van der Wijngaart. *RCCE: A Small Library for Many-Core Communication, Version 0.7*. Version 0.7. 2011.

[Owe+07] John D. Owens et al. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum* 26.1 (2007), pp. 80–113.

[Pep10] Christian Peper. *Towards an Object Distribution Management for Heterogeneous Ad-hoc Systems*. Tech. rep. IESE-Report No. 088.10/E. Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany: Fraunhofer IESE, Dec. 10, 2010.

[Pha+05] Diep Pham et al. "The Design and Implementation of a First-Generation Cell Processor". In: *Proc. of the IEEE Int'l Solid-State Circuits Conf. (ISSCC'05) Digest of Technical Paper*. San Francisco, California, USA, Feb. 10, 2005, pp. 184–592.

[Raj+10] Ragunathan Rajkumar et al. "Cyber-Physical Systems: The Next Computing Revolution". In: *Proc. of the 47th ACM/IEEE Design Automation Conf. (DAC'10)*. Anaheim, California, USA, June 13–18, 2010, pp. 731–736.

[Sha+08] Lui Sha et al. "Cyber-Physical Systems: A New Frontier". In: *Proc. of the IEEE Int'l Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'08)*. Taichung, Taiwan, June 11–13, 2008, pp. 1–9.

[SSB03] Nik Shaylor, Douglas N. Simon, and William R. Bush. "A Java Virtual Machine Architecture for Very Small Devices". In: *Proc. of the ACM SIGPLAN Conf. on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*. San Diego, California, USA, 2003, pp. 34–41.

[The11] The Eclipse Foundation. *Eclipse Platform*. `http://www.eclipse.org`. 2011.

References for Chapter 5, System Specification

[BEF10] Annette Bieniusa, Johannes Eickhold, and Thomas Fuhrmann. "The Architecture of the DecentVM – Towards a Decentralized Virtual Machine for Many-Core Computing". In: *Proc. of the 4th Workshop on Virtual Machines and Intermediate Languages (VMIL'10)*. Reno, Nevada, USA, Oct. 17, 2010.

[BF10] Annette Bieniusa and Thomas Fuhrmann. "Consistency in Hindsight, A Fully Decentralized STM Algorithm". In: *Proc. of the IEEE Int'l Symp. on Parallel Distributed Processing (IPDPS'10)*. Atlanta, Georgia, USA, Apr. 2010, pp. 1–12.

[JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN: 0-471-94148-4.

[PS95] David Plainfossé and Marc Shapiro. "A Survey of Distributed Garbage Collection Techniques". In: *Proc. of the Int'l Workshop on Memory Management (IWMM'95)*. Kinross, Scotland, UK, Sept. 27–29, 1995, pp. 211–249.

References for Chapter 6, Locating Objects

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. “cJVM: A Single System Image of a JVM on a Cluster”. In: *Proc. of the Int’l Conf. on Parallel Processing (ICPP’99)*. Wakamatsu, Japan, Sept. 21–24, 1999, pp. 4–11.

[Bal+98] Henri E. Bal et al. “Performance Evaluation of the Orca Shared-Object System”. In: *ACM Transactions on Computer Systems (TOCS)* 16 (1 Feb. 1998), pp. 1–40.

[Bha+07] Sangeeta Bhattacharya et al. “Design and Implementation of a Flexible Location Directory Service for Tiered Sensor Networks”. In: *Proc. of the 3rd IEEE Int’l Conf. on Distributed Computing in Sensor Systems (DCOSS’07)*. Santa Fe, New Mexico, USA, 2007, pp. 158–173.

[BMT03] Mario Bisignano, Giuseppe Di Modica, and Orazio Tomarchio. “Mobile Agent Location Management: A Comparison Between CORBA and P2P Based Systems”. In: *IEEE Symp. on Computers and Communications* (2003), p. 1029.

[BO99] Bhargav Bellur and Richard G. Ogier. “A Reliable, Efficient Topology Broadcast Protocol for Dynamic Networks”. In: *Proc. of the 18th IEEE Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM’99)*. Vol. 1. Mar. 1999, pp. 178–186.

[CWH99] Benny Wang-Leung Cheung, Cho-Li Wang, and Kai Hwang. “A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations”. In: *Proc. of the Int’l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA’99)*. Las Vegas, Nevada, USA, 1999.

[Cza00] Grzegorz Czajkowski. “Application Isolation in the Java Virtual Machine”. In: *Proc. of the 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’00)*. Minneapolis, Minnesota, USA, Oct. 15–19, 2000, pp. 354–366.

[DM78] Yogen K. Dalal and Robert M. Metcalfe. “Reverse Path Forwarding of Broadcast Packets”. In: *Communications of the ACM* 21.12 (1978), pp. 1040–1048.

[Fow86] Robert Joseph Fowler. “The Complexity of Using Forwarding Addresses for Decentralized Object Finding”. In: *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC’86)*. Calgary, Alberta, Canada, Aug. 11–13, 1986, pp. 108–120.

[FSS06] Michael Factor, Assaf Schuster, and Konstantin Shagin. “A Platform-Independent Distributed Runtime for Standard Multithreaded Java”. In: *International Journal of Parallel Programming* 34 (2 Apr. 2006), pp. 113–142.

[FWL03] Weijian Fang, Cho-Li Wang, and Francis C. M. Lau. “On the Design of Global Object Space for Efficient Multi-Threading Java Computing on Clusters”. In: *Parallel Computing* 29.11-12 (2003), pp. 1563–1587.

[Göc+04] Ralph Göckelmann et al. “Plurix, A Distributed Operating System Extending the Single System Image Concept”. In: *Proc. of the Canadian Conf. on Electrical and Computer Engineering (CCECE’04)*. Vol. 4. Niagara Falls, Ontario, Canada, May 2–5, 2004, 1985 –1988 Vol.4.

[Gos+00] James Gosling et al. *Java Language Specification, Second Edition: The Java Series*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201310082.

[Her99] Maurice Herlihy. “The Aleph Toolkit: Support for Scalable Distributed Shared Objects”. In: *Network-Based Parallel Computing: Communication, Architecture, and Applications*. 1999, pp. 137–149.

[HST01] Weiwu Hu, Weisong Shi, and Zhimin Tang. “Optimizing Home-Based Software DSM Protocols”. In: *Cluster Computing* 4 (3 July 2001), pp. 235–242. ISSN: 1386-7857.

[LB98] Sheng Liang and Gilad Bracha. “Dynamic Class Loading in the Java Virtual Machine”. In: *Proc. of the 13th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’98)*. Vancouver, British Columbia, Canada, Oct. 18–22, 1998, pp. 36–44.

[LH89] Kai Li and Paul Hudak. “Memory Coherence in Shared Virtual Memory Systems”. In: *ACM Transactions on Computer Systems (TOCS)* 7.4 (1989), pp. 321–359.

[Liu+05] Hongzhou Liu et al. “Design and Implementation of a Single System Image Operating System for Ad Hoc Networks”. In: *Proc. of the 3rd Int’l Conf. on Mobile Systems, Applications, and Services (MobiSys’05)*. Seattle, Washington, USA, 2005, pp. 149–162.

[MM02] Petar Maymounkov and David Maziéres. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *Peer-to-Peer Systems*. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 53–65.

[Moc87a] Paul V. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034 (Standard). Internet Engineering Task Force, Nov. 1987.

[Moc87b] Paul V. Mockapetris. *Domain Names - Implementation and Specification*. RFC 1035 (Standard). Internet Engineering Task Force, Nov. 1987.

[Mor01] Luc Moreau. “Distributed Directory Service and Message Routing for Mobile Agents”. In: *Science of Computer Programming* 39.2-3 (2001), pp. 249 –272.

[PWB07] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. “Failure Trends in a Large Disk Drive Population”. In: *Proc. of the 5th USENIX Conf. on File and Storage Technologies (FAST’07)*. San Jose, CA, USA, Feb. 13–16, 2007.

[PZ97] Michael Philippsen and Matthias Zenger. “JavaParty – Transparent Remote Objects in Java”. In: *Concurrency: Practice and Experience* 9.11 (Nov. 1997), pp. 1225–1242.

[Rat+01] Sylvia Ratnasamy et al. “A Scalable Content-Addressable Network”. In: *Proc. of the SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM’01)*. San Diego, California, USA, 2001, pp. 161–172.

[RD01] Antony I. T. Rowstron and Peter Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Proceedings of the IFIP/ACM International Conf. on Distributed Systems Platforms Heidelberg*. Middleware ’01. London, UK: Springer-Verlag, 2001, pp. 329–350. ISBN: 3-540-42800-3.

[SA83] Adrian Segall and Baruch Awerbuch. “A Reliable Broadcast Protocol”. In: *IEEE Transactions on Communications* 31.7 (July 1983), pp. 896–901.

[SJ95] Brjarne Steensgaard and Eric Jul. “Object and Native Code Thread Mobility Among Heterogeneous Computers”. In: *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP’95)*. Copper Mountain, Colorado, USA, Dec. 3–6, 1995, pp. 68–77.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. “DRAM Errors in the Wild: A Large-Scale Field Study”. In: *Proc. of the 11th Int’l Joint Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS’09)*. Seattle, WA, USA, June 15–19, 2009.

[Ste+98] Maarten van Steen et al. “Locating Objects in Wide-Area Systems”. In: *IEEE Communications Magazine* 36.1 (Jan. 1998), pp. 104–109.

[Sto+01] Ion Stoica et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM’01)*. San Diego, California, USA, Aug. 27–30, 2001, pp. 149–160.

[SUN06] SUN Microsystems. *JSR-121 - Application Isolation API Specification*. `http://www.jcp.org/en/jsr/detail?id=121`. Java Specification Requests. June 2006.

[Tan+91] Andrew S. Tanenbaum et al. “The Amoeba Distributed Operating System – A Status Report”. In: *Computer Communications* 14.6 (1991), pp. 324–335.

[WC02] Brad Williams and Tracy Camp. “Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks”. In: *Proc. of the 3rd ACM Int’l Symp. on Mobile Ad Hoc Networking & Computing (MobiHoc’02)*. Lausanne, Switzerland, 2002, pp. 194–205.

[WPD88] David Waitzman, Craig Partridge, and Stephen Edward Deering. *Distance Vector Multicast Routing Protocol*. RFC 1075 (Experimental). Internet Engineering Task Force, Nov. 1988. URL: `http://www.ietf.org/rfc/rfc1075.txt`.

[ZIL96] Yuanyuan Zhou, Liviu Iftode, and Kai Li. “Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems”. In: *SIGOPS Operating Systems Review* 30 (Oct. 1996), pp. 75–88.

[ZWL02] Wenzhang Zhu, Cho-Li Wang, and Francis Chi-Moon Lau. "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support". In: *Proc. of the IEEE Int'l Conf. on Cluster Computing (CLUSTER'02)*. Chicago, Illinois, USA, Sept. 23–26, 2002, pp. 381–388.

References for Chapter 7, Reactive Location Updates with Migration Proxies Protocol

[Fow86] Robert Joseph Fowler. "The Complexity of Using Forwarding Addresses for Decentralized Object Finding". In: *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*. Calgary, Alberta, Canada, Aug. 11–13, 1986, pp. 108–120.

[Moc87a] Paul V. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034 (Standard). Internet Engineering Task Force, Nov. 1987.

[Moc87b] Paul V. Mockapetris. *Domain Names - Implementation and Specification*. RFC 1035 (Standard). Internet Engineering Task Force, Nov. 1987.

[MR02] Luc Moreau and Daniel Ribbens. "Mobile Objects in Java". In: *Scientific Programming* 10.1 (2002), pp. 91–100. ISSN: 1058-9244.

[PS95] David Plainfossé and Marc Shapiro. "A Survey of Distributed Garbage Collection Techniques". In: *Proc. of the Int'l Workshop on Memory Management (IWMM'95)*. Kinross, Scotland, UK, Sept. 27–29, 1995, pp. 211–249.

[SF10] Björn Saballus and Thomas Fuhrmann. *A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments*. Tech. rep. TUM-I1025. Munich, Germany: Technische Universität München, Dec. 2010.

[Tan02] Andrew S. Tanenbaum. *Computer Networks*. 4th. Prentice Hall Professional Technical Reference, 2002.

References for Chapter 8, Proactive Location Update with Incoming References Protocol

[Day+93] Mark Day et al. "References to Remote Mobile Objects in Thor". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (1993), pp. 115–126.

References for Chapter 9, Evaluation

[AL62] Georgy Adelson-Velsky and Evgenii M. Landis. "An Algorithm for the Organization of Information". In: *Proc. of the USSR Academy of Sciences 146:263–266 (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263*. 1962, pp. 263–266.

[Bay72] Rudolf Bayer. “Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms”. In: *Acta Informatica* 1 (4 1972), pp. 290–306.

[Fer+10] Cesare Ferri et al. “Embedded-TM: Energy and Complexity-Effective Hardware Transactional Memory for Embedded Multicore Systems”. In: *Journal of Parallel and Distributed Computing* 70.10 (2010), pp. 1042–1052.

[Fuh05] Thomas Fuhrmann. “Scalable Routing for Networked Sensors and Actuators”. In: *Proc. of the 2nd Annual IEEE Communications Society Conf. on Sensor and Ad Hoc Communications and Networks (SECON'05)*. Santa Clara, California, USA, Sept. 26–29, 2005, pp. 240–251.

[SF10] Björn Saballus and Thomas Fuhrmann. *A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments*. Tech. rep. TUM-I1025. Munich, Germany: Technische Universität München, Dec. 2010.

[Var01] András Varga. “The OMNeT++ Discrete Event Simulation System”. In: *Proc. of the European Simulation Multi Conf. (ESM'2001)*. 2001, pp. 319–324.

[VH08] András Varga and Rudolf Hornig. “An Overview of the OMNeT++ Simulation Environment”. In: *Proc. of the 1st Int'l Conf. on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2008, pp. 1–10.

Index

ISBN 3-937201-31-9
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)
DOI: 10.2313/NET-2012-07-2