T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments

Björn Saballus and Thomas Fuhrmann

TUM-I1025
Dezember 10

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I1025-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Technical Report: A Decentralized Object Location and Retrieval Algorithm for Distributed Runtime Environments

Björn Saballus Thomas Fuhrmann

December 2010

Contents

1 Introduction

Software development for fully decentralized distributed systems faces three challenges: Finding and retrieving remote data, synchronized concurrent access to that data, and assignment of data and threads to the system's resources.

Our envisioned system shall accomplish all three challenges and provide a *single system image* [1, 2], which allows applications to run transparently on clusters of heterogeneous multi-core machines. It distributes code, objects and threads onto the compute resources, which may be added or removed at run-time. This dynamic property leads to an ad-hoc network of processors and cores. In this network, a fully decentralized object location and retrieval algorithm has to guarantee the access to distributed shared objects.

On an single core or a symmetric multiprocessing (SMP) processor, a reference is represented as a local memory address. In a distributed system, a reference can either point into local memory or into remote memory on distant nodes. Such a remote reference can become invalid if a node fails or an object migrates without precautions to another node.

In this paper we examine different object location and retrieval algorithms under the assumption of object migration. Migrating objects is an important functionality of distributed systems. It allows or facilitates:

- Maintenance: Migrate the running threads and locally stored objects from a node A to another node B to exchange node A during runtime.
- Communication: Migrate objects that access each other to nodes close to each other 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 access latency of accessing nodes in the close neighborhood of a replica.
- Resources: Migrate objects and threads to nodes with free resources for an optimal load-balancing or to nodes with specialized resources such as floating-point units.

An object access requires that the referencing object holds a reference to the referenced object and that the referencing node can reach the node where the referenced object is stored. Whenever an object migrates from one node to another, the system must ensure that it is still accessible from referencing objects. To achieve this requirement, we separate the object identifier from the object locator. In this paper, we describe various ways to achieve this task:

- Broadcast
- Reactive Updates (On-the-Fly, upon access)
- Proactive Updates (On-the-Fly, upon migration; forward requests)
- Enhanced Proactive Updates (On-the-Fly, upon migration; fast responses)
- Distributed Registry, e. g. Distributed Hash Tables (DHT)
- Centralized Registry

We have chosen the Java programming language [3] as as starting point of our work. Java is a general purpose programming language, and as such has gained a rising interest as a programming language for scientific and engineering computing. Moreira et al. [4] for example describe the usage of Java for high performance numerical computing. Their approach is built around the use of a numerical Java library, especially for arrays and complex number. Additionally, they discuss compiler optimizations that their HPCJ compiler performs on the numerical library code, e. g. array bound check optimizations.

Taboada et al. [5] analyze current research projects (as of 2009) that use Java for High Performance computing. They conclude, that Java is well suited for hybrid shared memory (intra-node) and

distributed memory (inter-node) architectures, because Java threads support shared memory, and Java network support assists distributed memory communication. According to Taboada et al., most research focuses on scalable Java communication middleware for high-speed networks, such as InfiniBand or Myrinet. According to Taboada et al. the various projects can be classified into:

- shared memory programming with Java threads, OpenMP-like implementations or a PGAS Java dialect, e. g. Titanium [6] (see 6.3).
- usage of Java sockets.
- usage of Remote Method Invocation (RMI).
- usage of message passing, that can again use RMI, Java sockets or wrap methods from the Java Native Interface (JNI).

We did not follow these approaches, but decided to develop a distributed Java runtime environment for our envisioned system. Other projects, e. g. cJVM [7] or JESSICA2 [8] (described in more details in section 6) followed this approach as well.

In this paper, our Java runtime environment is simulated with a network simulator that is used to evaluate the reactive and the proactive update approaches. The results show that the overhead of the proactive update approach is not worthwhile for most application scenarios, while the proactive update approach has the best performance with respect to the remote object access latency.

1.1 Object Model

We use the general term *object* to name all kinds of high-level programming language data; objects and arrays but also execution contexts and program code. We distinguish objects between *Local Object Copies* (LOCs) and *Global Accessible Objects* (GAOs). Another distinction is between *dynamic* and *static* objects (c. f. figure 1).

Local Object A *local object (copy) (LOC)* is an object that resides in the local memory. It is only accessible from within the local node through a *local memory pointer*, e. g. a C-pointer. This pointer is only valid on the local node. If the local object becomes accessible from a remote node, e. g. due to object migration, it is transformed into a *global accessible object*.

Global Accessible Object A *global accessible object (GAO)* is addressed by its *global unique identifier (GUID)* or via a chain of references (c. f. sec. 2.2.7). It is accessible from each node in the network, given that the node holds the GUID of the object or a valid chain of references. With the GUID, the GAO is reachable regardless of its physical location.

Replicas of GAOs might exist on different nodes in the network. This paper does not address the task of replication management. It also does not deal with the access to a distinct replicate (unicast), any replicate (anycast) or all replicates (multicast). This is ongoing work in our group. A first study of replication mamagement was done in our group by [9].

Dynamic Object A *dynamic* object is allocated on the heap whenever the application instantiates a new object. The accessing entity must know a reference to the object to access it. An entity must not be able to make up a valid reference to a dynamic object on its own. An object gets hold of a new dynamic reference if this new reference is written to one of its reference fields. The execution context gets hold of a reference to a dynamic object e. g. if it allocates a new object A or reads reference field of an object B (for this, the execution context must already hold the reference to object B). Other possibilities, e. g. the reception of a reference as method parameter during method invocation, are only special issues of the two basic cases.

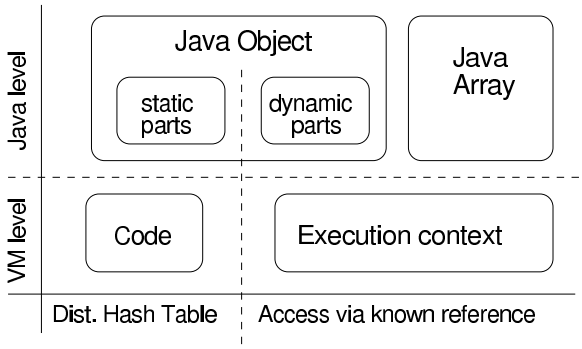


Figure 1: GAO Types using the example of Java.

Static Object (Parts) A *static* object (part) contains the class members. It is unique within a class-domain tuple (c. f. section 1.3). In Java, the static object parts of a class are created during class loading.

To access this static object part of a class, the accessing entity can compute a reference to it. Such a reference might be for example the hashed class definition e. g. the hash of the Java bytecode. All entities are able to compute and access this reference. To do this, the system is required to offer a distributed lookup service such as a *distributed hash table (DHT)*. Such a service is ongoing work in our group and not topic of this paper.

1.2 Reference Graph

The starting point for our proposed solution is the separation of object identification and object location, i. e. the GUID of a GAO does not encode the GAO's location in the network. Thus, we need a mechanism to locate the home node of the corresponding GAO to retrieve or access it on its remote home node.

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. Execution contexts hold references to the code they execute and the objects which belong to their parameters or their local variables. Figure 2 gives an example of references to code and objects.

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.

References in the common sense are unidirectional: they point from the referencing object to the referenced object. We call them the *outgoing references* of an object. Beside these references, our proactive update approach, described in section 3.4 uses bi-directional references. These bi-directional references introduce additional backward reference, which point back from the referenced object to the referencing one. According to outgoing references we call these backward references the *incoming references* of an object.

1.3 Domain Concept

Since a GAO exists only once for each application, we would not be able to execute more than one application in our system. To break this constraint, we introduce *domains*, c. f. [10], that identify the scope of an application. A *domain* is a virtual self-contained environment that executes the application(s) that are associated with it. Domains are identified by a secret key which is used to restrict access to the domain and, if required, to encrypt the messages sent between participants of the domain. Nodes that know the domains secret key are those who participate in running the associated application.

Inter-domain communication is done via static object parts. As stated above, static object parts are unique within a class-domain tuple. The thread that is responsible for the inter-domain communication

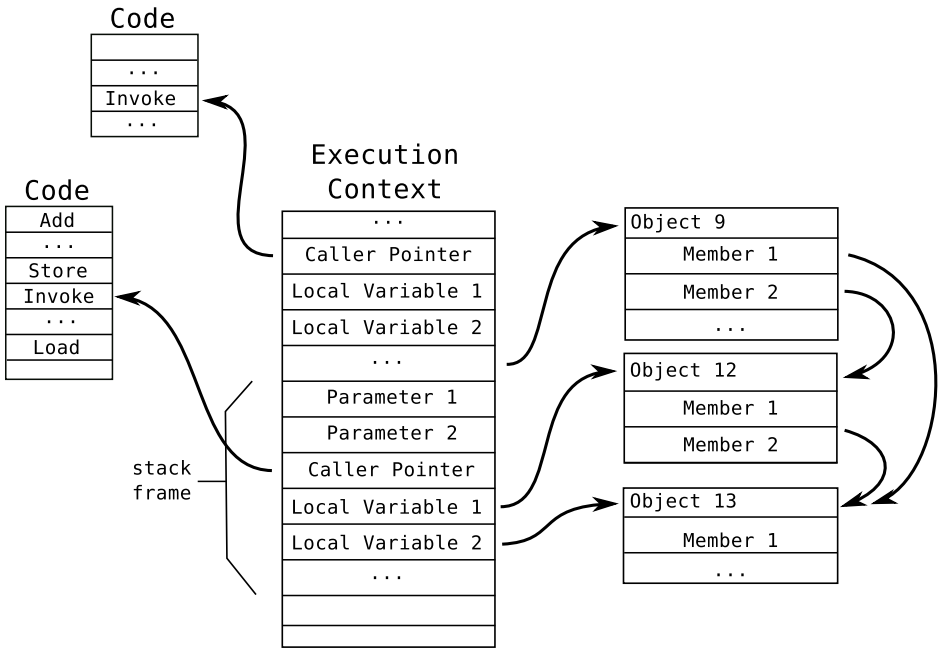


Figure 2: Execution context, Code and Objects.

accesses two different static object parts in two different domains. In this way, the reference graph of the thread is spread between two domains. The details of this concept are subject on ongoing work in our group and we will not detail this work in this paper.

1.4 Concurrent Object Access

To ensure a consistent access in face of concurrent threads, we support two different approaches: The common *locking approach*, which is used to synchronize the access, e. g. with a (binary) semaphore that keeps threads from entering the 'critical section' when another thread is already in that section. Another approach is the use of *Software Transactional Memory* (STM).

Software Transactional Memory is an alternative to lock-based synchronization. STM uses optimistic transactions, in which a thread performs a series of read and write operations on shared memory in an *atomic* block. These operations are performed without regards to any other thread that might operate on this memory. The changes are not visible to other threads until a transcation *commits* its changed data. If no other transaction modified the data that has been touched by the commiting transaction, the commit succeeds. If the data was modified in the meantime, the transaction fails and has to be rolled back and re-executed.

To execute a transaction without interfering with other transactions, each thread retrieves a *Local Object Copy* (LOC) of the read or written GAOs and operates locally on this copy. When a transaction finishes, the modified LOC is written back. If more than one thread modified the LOC, a distributed consensus protocol has to decide which thread is allowed to commit and which threads have to roll back. By this, the programmer gets the illusion that each thread manipulates the data in an atomic block without being disturbed by others. The development of decentralized STM protocols is ongoing work in our group. A first proposal is DecentSTM [11].

In the context of DecentSTM, a GAO is a mutable structure that consists of a list of immutable GAO versions, see [11]. Whenever a transaction successfully commits, all LOCs that have been written in the transaction become new versions of the corresponding GAOs. If theses new GAO head version are located on other nodes than the previous versions, it looks for the system as if the corresponding GAOs migrated. By this, our system implicitly migrates GAO on write accesses.

The rest of the paper is structured as follows: Section 2 describes the different components thar are involved in the reference maintenance. These are part of the entity of the runtime environment

that is responsible for the resources of a single node, e. g. one instance of a distributed Java Virtual Machine. Additionally, this section introduces an example setup that we use in section 3 to describe the various location update strategies for dynamic objects. In section 4 we give a short overview of location retrieval mechanisms for static (parts of) objects. Afterwards, we present the evaluation of our protocols in section 5. Finally, the paper concludes in section 6 with an extended overview of related work.

2 System Specification

Our target system consists of heterogeneous multi- or manycore processors that are interconnected to an network of computing nodes. To execute a parallel application, we place one instance of a collaborating runtime environment entity on each node in the network. This entity is responsible for the local memory of the node, e.g the local Java heap, and consists itself of various modules.

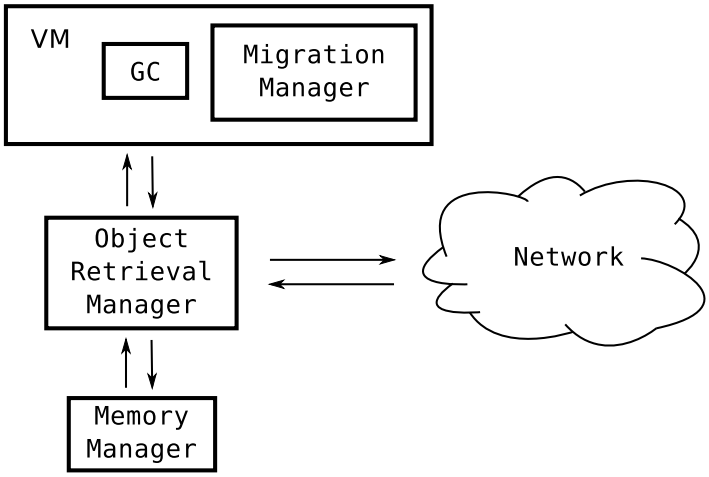


Figure 3: Conceptional System Design.

These modules are shown in figure 3. Each of them fulfills a different task to ensure the reachability of local and remote objects under object migration. Between each other, these modules interact via well defined interfaces.

The *memory manager (MemMgr)* is responsible for the local memory and the locally stored objects. The *object retrieval manager (ORMgr)* manages the local and remote object access. The ORMgr handles the translation between local and remote references and keeps track of the location of locally referenced remote objects. The runtime environment, in our case a *virtual machine (VM)*, executes the application. The VM executes the PEC and accesses the objects of the application. Each VM has a *local garbage collector (LGC)* and manages parts of the global, *distributed garbage collector (DGC)*. The VM also contains the *migration manager (MigMgr)*. This manager handles all local actions that are involved in the migration of local or remote objects.

2.1 Virtual Machine

The runtime environment executes one or more applications. Typically, the runtime environment will be a virtual machine (VM), e. g. a Java VM or a Hypervisor. Since our work was inspired by a Java VM, we use the corresponding terminology.

Each application consists of multiple threads which are concurrently executed. Internally, each of these threads consists of chunks of local memory. These hold the execution context (program counter, stack frame, etc.) and the corresponding bytecode. Both, execution context and bytecode, can be local object copies (LOCs) of globally accessible objects (GAOs). The usage of LOCs is preferable because the execution context is thread local data anyway and the bytecode is immutable and can

thus be replicated easily. Using LOCs also improves the performance as no messages have to be sent through the network for each single execution step.

Other application data may be stored and modified on separate nodes. Either updates are sent timely to the node that stores the GAO (write through), or the LOCs must be synchronized with the GAO at a given time (write back).

2.1.1 Scheduler

The VM executes the application by executing the bytecode of the threads. A scheduler considers all its threads and assigns each thread a fixed unit of execution time at the local processor. While the thread is executed it is in the *VM running* state.

After the assigned execution time, the scheduler puts the execution threads on hold and continues with the execution of another one. While the thread is on hold, it is in the *VM wait* state.

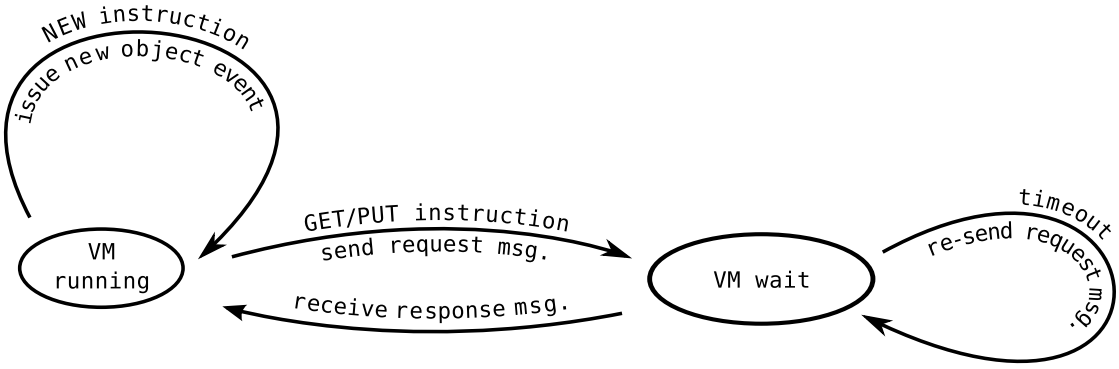


Figure 4: Virtual Machine State Diagram.

Figure 4 shows the state diagram with the different states and transitions of the VM. The state diagram only shows the transitions that are due to object access operations and not due to scheduler revocation.

During the program execution, the VM creates objects and executes PUT or GET instructions on their fields. These PUT and GET instructions might access local and remote objects by reading or writing the fields of these objects. We distinguish these fields into reference fields, that hold a reference to another object, or numerical fields, that hold a value. In this paper we are dealing with the reachability of objects, namely, the maintenance of references between objects. For this reason, we are only interested in operations that access reference fields.

Access instructions to local objects return immediately from the *VM wait* state. If the object is located on a remote node, the VM stalls the corresponding thread in the *VM wait* state. Upon receiving a response message that indicates the success of the remote operation, the local thread resumes.

The response for a remote GET operation contains the content of the field that has been read. Remote PUT operations are acknowledged with an acknowledgement (ACK) message that indicates the success of the operation.

If the node does not receive a response or ACK message within a given threshold, the request message is repeated.

2.1.2 Memory Manager

The *Memory Manager* (MemMgr) is responsible for the node-local memory of the VM. It allocates and frees local memory for GAOs that are stored and managed in the *local object store*. For these tasks, the MemMgr offers the following methods:

```
locMemPtr_t MemMgr::AllocateObject();
void MemMgr::FreeObject(locMemPtr_t);
```

Additionally, the MemMgr allocates and manages the memory for the internal data structures of the VM. Some of these data structures are described in the following, all others are not in the scope of this paper.

2.2 Object Retrieval Manager

The *Object Retrieval Manager* (ORMgr) is responsible for the access to the local object (copies) and for locating and retrieving remote objects. Conceptually, we separate object references not only from their location in the network, but also from their location in the local memory. This separates the local object view from the global object view.

To reflect this separation, the ORMgr uses a two staged dereferencing system (c. f. fig. 5). This system uses a *ReferenceMap*, a *GaoMap* and an optional *Incoming Reference Map (InRefList)*. The first two maps are necessary to access and retrieve local and remote objects. The latter map is used for the proactive location update approach described in section 3.4. The ORMgr dynamically allocates this *InRefList* on a per-object basis.

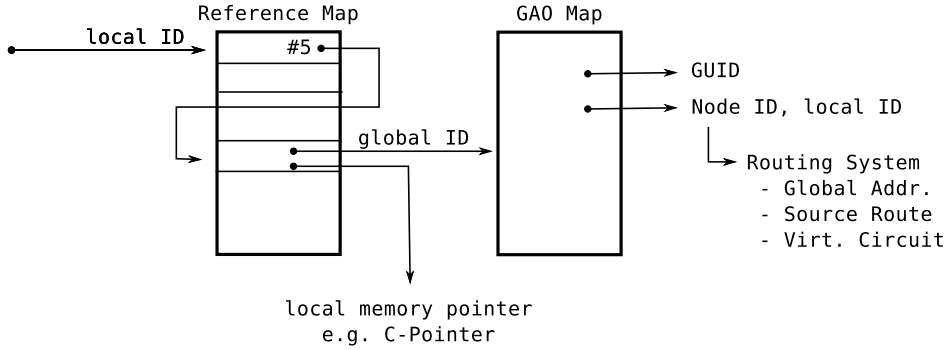


Figure 5: ReferenceMap and GaoMap.

Internally, the VM uses *localIDs* to reference objects. A localID points into the ReferenceMap, that is used to resolve this ID. This shields the VM from the knowledge about the actual object location and allows an easier VM implementation. E.g., the VM can use internal 16-bit localIDs, while the GUID of object might be e. g. 160-bit or more.

2.2.1 ReferenceMap

The *ReferenceMap* stores the mapping of *localID* to either a *local memory pointer* or a *globalID*. In some cases, the ReferenceMap can also build up chains of references (see sec.2.2.7).

An entry to a locally stored object maps the *localID* to a *local memory pointer*. A local reference counter indicates how many local objects reference this entry. Additionally, the map stores an external reference counter and a garbage collector epoch index for the garbage collector.

For the proactive location update approach, a pointer to the *InRefList* of the object is added as well. The number of entries in the *InRefList* corresponds to the value of the external reference counter.

An entry to a remote object maps the *localID* to a *globalID* i. e. a pointer into the *GaoMap*. Figure 7 shows two nodes: Node A holds three objects that all references another object on node B. The *ReferenceMap* on node A maps the *localID* to a GUID and a location information, leading to node B. The location information might additionally contain the *localID* of the object on node B (see below). On node B, the *ReferenceMap* maps a request for the object to the local memory pointer and the *InRefList* pointer. In this case, the *InRefList* stores node A as an incoming reference.

The local reference counter of this *ReferenceMap* entry indicates how many local objects reference the remote object. In accordance with the one entry in the local *Reference Map* and *GaoMap*, there is one *incoming reference* entry in the remote *InRefList* of the referenced object, c. f. figure 7.

Note that an entry in the *ReferenceMap* can point to both, a local memory location and an entry in the *GaoMap*. This is the case for objects that are local object copies of remote objects.

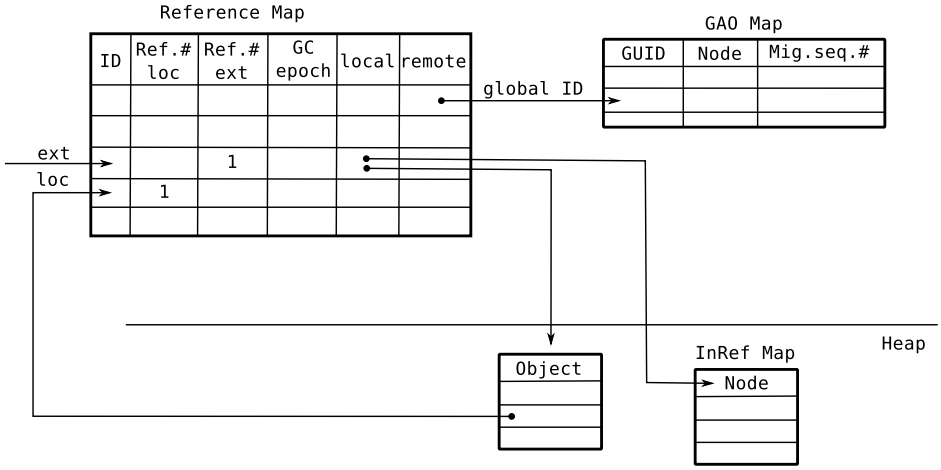


Figure 6: Detailed view on Reference, InRef and GaoMap.

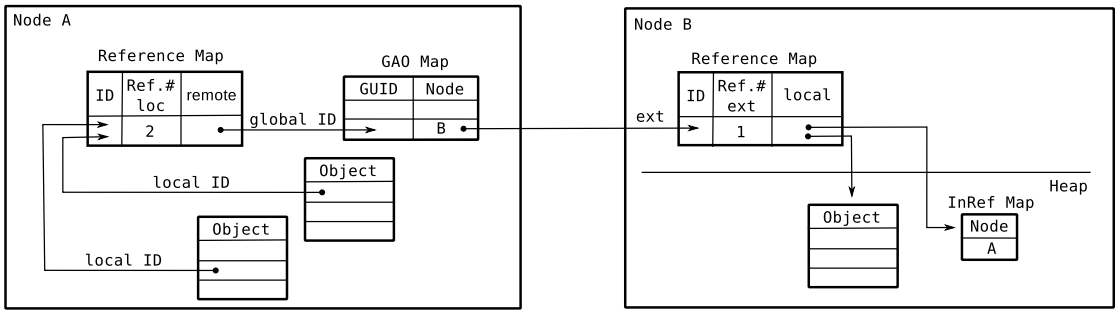


Figure 7: Three references on node A to a remote object on node B.

2.2.2 Allocation of Local Objects

Since the ORMgr maintains the *ReferenceMap*, it is also involved in the object create and delete operations. For these two tasks it offers the following two methods:

```
locId_t ORMgr::AllocateObject();
void ORMgr::FreeObject(locId_t);
```

To create a new object, the ORMgr chooses a new locally unique *localID*. Afterwards, the ORMgr invokes the MemMgr which allocates the object in the local memory and returns the *local memory pointer* to the object. Then, the ORMgr allocates a new slot in the *ReferenceMap* and inserts the *local memory pointer* together with the assigned *localID*. The reference fields of the object are handled by the object access methods described below.

If the *ReferenceMap* is full or if the MemMgr cannot provide enough heap space, the allocation fails with an *OutOfMemory exception*. Alternatively, the node can try to allocate the object on a remote node if there is not enough local memory to create the object but enough for an additional entry in the *ReferenceMap*.

2.2.3 Obtaining Access Information for Remote Objects

References to remote objects can

- be read from a static object part,
- be read from a remote dynamic object to which the reference is known,

- result from the migration of a local object,
- result from an object which migrates to the local node or
- result from a NEW operation which can not be fulfilled locally and must be executed on a remote node.

If a reference to a remote object is received or created, the ORMgr assigns a global and local ID. Then, the ORMgr inserts these mappings with the additional information such as the location information into the *ReferenceMap* and *GaoMap*.

2.2.4 Locating Remote Objects

A distributed system can address and locate remote objects using different types of location information. The most generic information is the GUID. With the GUID it is always possible to broadcast a request into the network, see section 3.1 for more information. Other location information are e. g. a tuple of (node ID, local ID) or any other form of routing information. This routing information depends on the underlying routing algorithm, which is responsible for the delivery of messages to a given location.

Our system uses *source routes* that reflect a *hop-by-hop path* from the referencing node to the referenced node. The system stores these source routes in the local *route cache*. As a result, our location information consists of the GUID, the home node of the object and an additional source route leading to the home node of the object. If different nodes in the network reference the same GAO, the GUID (part of the) reference and the location information on each node is the same but the associated source routes might be completely different.

2.2.5 Access to Remote Objects

To access a referenced object the node first translates the *localID* either to a *local memory pointer* or a *global ID*.

If the mapping resolves a *global ID*, the *GaoMap* translates this ID into the *location information* of the remote object. If the location information is not the GUID but a node address, the location information must also contains the *localID* of the object on its remote *home node*. Using remote *localIDs* simplifies the access to the requested object on its remote *home node*, as no additional ID translation step is required.

The ORMgr module offers these two methods to access objects:

```
void ORMgr::PutReferenceTo(locId_t Obj, Index, locId_t Ref);
locId_t ORMgr::GetReferenceFrom(locId_t Obj, Index);
```

The `PutReferenceTo(locId_t Obj, Index, locId_t Ref)` method writes (puts) a reference `Ref` to the reference field with index `Index` of object `Obj`. The `GetReferenceFrom(locId_t Obj, Index)` method reads (gets) a reference stored in the reference field with index `Index` of object `Obj`.

GET and PUT operations read or write local or remote references. If these operations read or write a reference that is already present on the accessed node, only the corresponding reference counter is incremented. If these operations create a new local reference, the ORMgr must allocate local memory for a new entry in the *ReferenceMap* and the *GaoMap*. The proactive update approach needs additional memory for the incoming reference that is stored in the *InRefList*. If there is no local memory left, the runtime system throws an OutOfMemory exception. If such an exception is thrown on the local or remote node, the operation fails. To prevent that failure, the node can invoke the garbage collector to free memory. If this fails as well, the accessed object might be a good candidate for migration or another object must be found which can be migrated. E.g. if an object is only referenced from remote nodes, this object is a good candidate for a migration.

2.2.6 Alternative Approaches

We identified three alternative approaches for the described *ReferenceMap*/*GaoMap* process. They aim at the reduction of the two stage resolution process by eliminating one or both indirections.

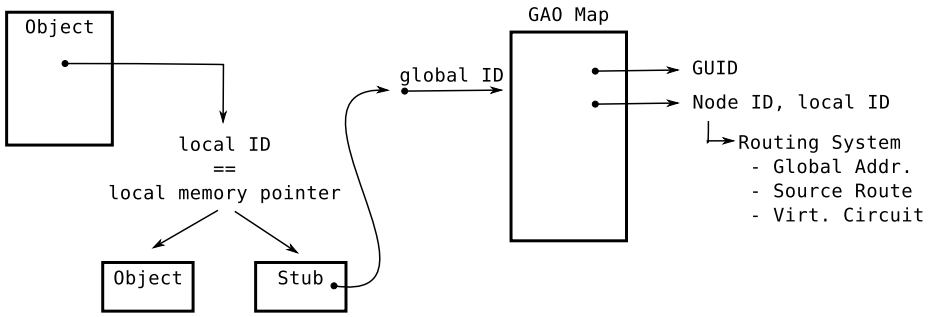


Figure 8: Local Memory Stub and GaoMap.

The first alternative removes the *ReferenceMap* and makes direct use of the local memory pointer as *localID*. In this case, the local memory pointer always points into a valid memory region. Therefore, the approach has to distinguish a local from a remote object. To do this, we introduce a local *stub object*. At this location, the stub stores the corresponding index (*globalID*) into the GaoMap, c. f. figure 8. If a *localID* that points to a stub is resolved, the access is redirected to a GaoMap lookup that resolves the location information needed for the remote object access.

Instead of removing the *ReferenceMap*, the second alternative removes the *GaoMap*. It moves the location information from the *GaoMap* into the *ReferenceMap*, c. f. figure 9.

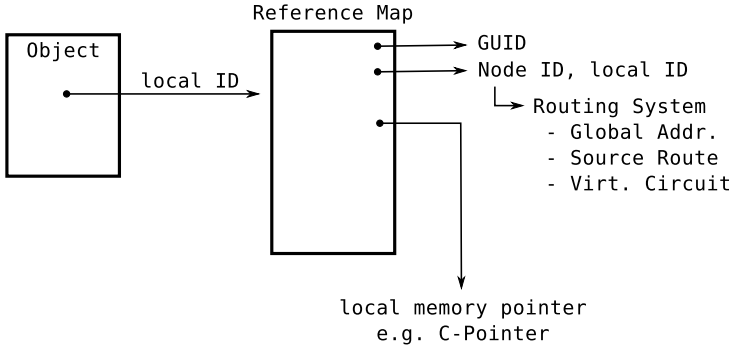


Figure 9: ReferenceMap without GaoMap.

Combining the two previous alternative approaches, the third removes both, the *ReferenceMap* and the *GaoMap*. It uses only regular local objects and *stub objects*. In this approach, the *stub object* stores the location information for the remote object directly. Namely, it stores the same information that was previously stored in the GaoMap, see figure 10.

This approach is comparable to a *tagged pointer*. A tagged pointer is a union of a *local memory pointer* and an additional tag. The tag is e. g. used to indicate the type of data to which the pointer points, or specific access conditions for the data. In our case, it is the location information of the remote object.

All the different approaches have advantages and disadvantages. The main advantage of the presence of the *ReferenceMap* is the additional information that is stored in this map, e. g. the reference counter, garbage collector epoch counter, etc. This allows a fast access to the information that e. g. the Java garbage collector needs.

The removal of the *ReferenceMap* and the usage of direct local memory pointers has the advantage of a direct access to local objects, without the indirection via the *ReferenceMap*. With the DecentSTM algorithm, all computation is done on local object copies only, see sec. 1.4. If a transaction reads or writes a remote object, the system first has to fetch this object and create a local object copy. As the object is remote, the *localID* points to the stub object that reveals, directly or indirectly via the *GaoMap*, the location of the object. The node uses this location information to send a request for a

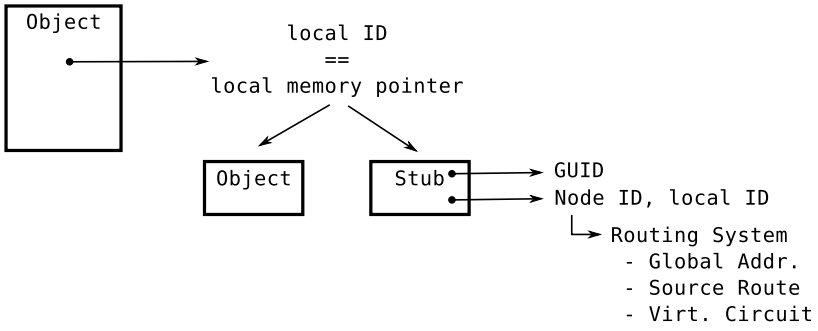


Figure 10: Local Memory Stub only, no ReferenceMap, no GaoMap.

copy of the remote object. When the remote object copy arrives at the local node, the node has to deal with two objects: the object copy and the corresponding local stub object. Note that the *localID* still points to the stub object.

As the *localID* is a pointer into the local memory, the system has three options to create a valid reference that points to the received local object copy:

- The stub object holds an additional local memory pointer that points to the LOC. This option has the disadvantage that it introduces an additional indirection.
- All *localIDs* that point to the stub object must be re-written to point to the LOC. This option requires that all *localIDs* on the stack and in all local objects must be check and eventually re-written. Depending on the number of objects the thread holds, this might require a lot of time and slow down the system. If the LOC is deleted later but at least one local reference is kept, the same procedure has to be performed to re-write the *localID* to point to the stub object again.
- The stub object is moved to another memory location and the LOC is placed at the previous location of the stub. The LOC must store the pointer to the new stub location. The replacement of the stub has two disadvantages: First, there might not be enough continuous memory behind the stub to hold the whole LOC. In this case, the LOC has to be stored at some other memory location and the stub has either to re-direct all access to the LOC or all *localIDs* to the stub object have to be re-written (see the two options above). Second, if the LOC replaced the sub and the LOC is deleted, the stub must be copied back, if there is at least one local reference to the corresponding object. To prevent memory fragmentation, the system would have to take additional measures.

The only advantage of the removal of both, the *ReferenceMap* and the *GaoMap* is that it removes the indirection that leads from the stub object to the *GaoMap*. All other problems described above apply in the same manner as they do for the solely removal of the *ReferenceMap*.

The removal of the *GaoMap* does not has the problems of the two other alternative approaches. Instead, it has the advantage to remove the indirection from the *ReferenceMap* to the *GaoMap*. This is possible because the information from the *GaoMap* is moved into the *ReferenceMap*. The disadvantage is, that each entry in the *ReferenceMap* has to provide enough space for the remote location information, even if the object is private and will always stay on the local node.

Under these considerations, it seems as if the original approach with both maps, the *ReferenceMap* and the *GaoMap*, is most suitable for a general system. If and when one of the alternative approaches is suitable depends on the concrete application. This evaluation is ongoing work in our group.

2.2.7 Reference Chains

In object oriented programming languages, it happens that a reference field in an object is read, only to read from that object another reference, and so on. This is e. g. the case in a red-black tree

implementation, where the color a of the right node of the parent of the parent of an object B is read:

```
color a = rightOf(parentOf(parentOf(Object_B))).getColor();
```

A common object access first reads and stores the reference to the parent of object B. Afterwards, this reference is used to access the parent object to read the reference to the parent-parent. This reference is again stored while the reference to the parent is deleted. Finally, the reference to the parent-parent is used to read the reference to the right node of this object.

Up to now, this procedure requires that a LOC of all intermediate, potentially remote, objects if iteratively fetched, even though the intermediate objects are not necessarily needed. To prevent this overhead, we introduce *reference chains*: The runtime environment builds up a chain of *localIDs* in its local *ReferenceMap*, as long a thread only accesses the reference fields of objects. The chain is recursively resolved only if a numeric value field is accessed, or the reference is needed, e. g. for a compare operation.

To resolve the chain, a table lookup is done until the first remote object is found. Then, a request is sent to the remote object, together with the information about the rest of the reference chain. The information about the reference chain must contain the GUIDs of the referenced objects, rather than the only locally valid *localIDs*. This process is continued on the remote node(s), until the last object is reached. The home node of this last object returns the LOC to the node that initiated the resolution of the chain. If the chain accesses a reference field that contains a *NULL reference*, a *NULL Reference Exception* is returned immediately.

Figure 5 shows a chain of two *localIDs*. In that figure, the first object references the fifth object in the local *ReferenceMap*.

2.3 Migration Manager

The *Migration Manager* (MigMgr) handles the migration of GAOs from one node to another. Discussing details and the reason for the migration is beyond the scope of this paper. Potential scenarios are e. g. the need to bring two objects closer together to prevent unnecessary message traffic, load balancing or the need to free local memory.

The MigMgr is invoked on the node that initiates the GAO migration. A migration is performed either as *push*, *pull* or *transfer* operation.

The methods or the MigMgr are:

```
void MigMgr::PushObjectTo(locID_t, Node);
void MigMgr::PullObjectFrom(locID_t, Node);
void MigMgr::TransferObjectFromTo(locID_t, NodeA, NodeB);
```

In a *push* migration, the old home node initiates the migration. It takes a local object and pushes it to a remote node. The remote node has to acknowledge the migration, before the pushing node is allowed to delete the migrated object.

A *pull* migration is initiated by the new home node of the object. The new home node *pulls* the GAO from the old home node and stores it into the local object store. When this is done, the new home node has to send an acknowledge message to the old home node. This allows the old home to delete its local object.

A *transfer* migration is initiated by a third party. It transfers an object from its old home node to another new home node without the involvement of the initiating node. When the transfer is finished, the new home node of the object has to send two acknowledgment messages. One to the old home node so that this node can delete its local object and one to the initiating node so that this node does not invoke the transfer again.

All these operations can cause an *OutOfMemory Exception* on the old and/or the new home node of the object, depending of the location maintenance approach. For example, the reactive and proactive approaches need at least a new entry in the *GaoMap* on the old home node for the *proxy entry*. If the local object holds references to these objects, an additional outgoing reference entry in the *GaoMap* must be created. The new home node must allocate memory for the object. In some cases it must create additional entries in the different maps and the source route cache.

The migration of an object requires an update of the *ReferenceMap* entries of its outgoing references. On the old home node, the reference counter is decremented. If this counter drops to zero, there are no other local objects which need this reference and its entries in the *ReferenceMap* and the *GaoMap* entries can be deleted.

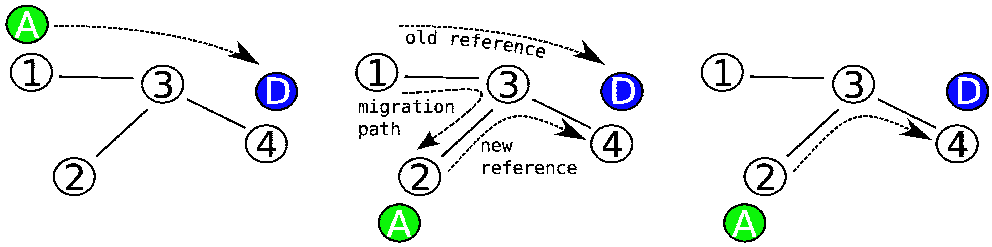
Migration Process An object migration process passes through the following steps:

- Retrieve the location information of the new home node (for a push/transfer operation) or old home node (for a pull operation) of the GAO.
- Send request to the remote MemMgr to allocate memory for the migrating object in the remote memory (push/transfer case) or to allocate local memory for the new GAO (pull case).
- Retrieve the GAO from the local MemMgr (push case) or retrieval of the GAO from the remote memory (pull/transfer case).
- Retrieve the location information of each outgoing reference (e. g. source route to reference home node) the migrating GAO holds from the GaoMap on the old home node (any case).
- If proactive update: Retrieve the location information for each incoming reference entry in the InRefList (e. g. source routes to InRef nodes).
- Include the migrating GAO, all outgoing references (together with their location information) and its InRefList (is present) in one message. Send this message to the new home node (any case).

When the migration message reaches the new home node the migrated object is stored in it's *local object store* and the ORMgr assigns a new *local ID*. For each outgoing reference either a new entry is created in the *ReferenceMap* and the *GaoMap* or the reference counter is incremented. Then, the shipped *InRefList* of the migrated object is stored in the local memory.

As we use source routes to reach a remote node, the new home node must compute the corresponding source routes for all InRefList entries and all outgoing references. To do this, the node concatenates the shipped reference source routes with the inverted migration route and adds them to the local *route cache*:

$$\text{NewReferenceRoute} = (-\text{MigrationRoute}) + \text{CurrentReferenceRoute} \tag{1}$$



(a) Before migration. (b) Migration. (c) After migration.

Figure 11: Outgoing Reference redirection after migration.

Fig. 11 gives an example. for which the source routes are concatenated as follows:

```
Old Ref. A -> D : (1-3-4)
      Mig. path : (1-3-2)
    Ref. update :
         A -> D : -(1-3-2) + (1-3-4)
         A -> D :  (2-3-1) + (1-3-4)
         A -> D :  (2-3-4)
```

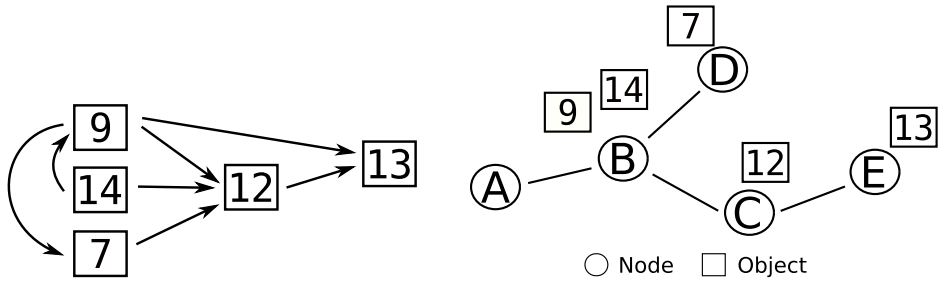
This update operation implicitly removes all potential loops in the new routes. Nevertheless, this may result in temporarily quite sub-optimal routes; but [12] has shown, the routing algorithm is able to quickly remove indirections and optimize the path.

2.4 Example Setup

The following example is used to illustrate a fully distributed setting in which objects are free to migrate. The example in figure 12 shows five objects with GUIDs 7, 9, 12, 13 and 14, which reference each other. We use small numbers for the GUIDs of the GAOs in this example. Typically, GUIDs are larger than the node local memory pointers.

We use the example to discuss:

- the maintenance overhead, i.e. number and types of messages that are needed to keep the references updated
- the state that must be stored and maintained on each node in the network, and
- the object and location information retrieval overhead.



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

Figure 12: Reference Graph and Physical Location.

Object 7	Object 9	Object 14	Object 12	Object 13
$\longrightarrow$ 12	$\longrightarrow$ 07	$\longrightarrow$ 09	$\longrightarrow$ 13	
	$\longrightarrow$ 12	$\longrightarrow$ 12		
	$\longrightarrow$ 13			

Table 1: Outgoing References of each GAO.

Node B	Node D	Node C	Node E
07 $\longrightarrow$ D : 1	07 $\longrightarrow$ D : 0	12 $\longrightarrow$ C : 0	13 $\longrightarrow$ E : 0
09 $\longrightarrow$ B : 1	12 $\longrightarrow$ C : 1	13 $\longrightarrow$ E : 1	
12 $\longrightarrow$ C : 2			
13 $\longrightarrow$ E : 1			
14 $\longrightarrow$ B : 0			

Table 2: Referenced GAO Location Information per Node.

Figure 12(a) shows the reference graph of the example. Figure 12(b) shows an initial location of the GAOs in the our example network. Table 1 shows the outgoing references of each objects. The right arrow ($\longrightarrow$) is read as this object "holds a reference to" GUID or this object "references" GUID, e.g., "object 7 references object 12". Table 2 shows the corresponding location information. Here, the right arrow ($\longrightarrow$) is read as this object "is stored on" node or this object "is located on" node. E.g. for node B, where the entry (07 $\longrightarrow$ D : 1) is read as "object 7 is stored on node D". This table

include the local *self-reference* to the locally stored objects, e.g. for the objects 9 and 14 on node B. Each location information entry is accompanied by a reference counter that indicates how many local objects reference the corresponding object. A value of zero indicates that a local object is not referenced from the local node. E.g. for node C, where the entry $(12 \longrightarrow C : 0)$ indicates that the local object 12 is not referenced locally.

3 Locating Dynamic Objects

Each read or write operation to a dynamic object requires a reference to this object. As described above, the referencing object must not be able to make up a valid reference. In a fully distributed system a reference must not only identify the object. It must also enable a location system to supply the information, where the object is located.

If a GAO migrates from one node to another, the stored GAO location information on all other nodes becomes invalid. For this reason an additional object location retrieval protocol is needed. This protocol either has to

- ensure that access messages always reach the corresponding GAO, even if the message was initially sent to an outdated location, or
- keep the location information of the migrated GAO updated on all other nodes.

The first requirement is met if a migrated object e.g. leaves a proxy behind that forwards the request to the new home node of the GAO. Another solution could be a central registry where an accessing node can ask for the new location if the first access attempt at an outdated location failed. Finally, the request can be sent as a broadcast message to all nodes in the network.

The second requirement can be met by sending update messages to all incoming and outgoing references of the migrating object.

In practice, it is hard 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 that eliminate proxies eventually. If this protocol fails for some reason, the fallback to broadcast messages is always possible, even though costly.

In the following, we describe different object retrieval and location information maintenance approaches that ensure that a node can access a dynamic object while it is allowed to migrate from one node to another.

3.1 Broadcast

The fallback approach that always works is to *broadcast* the access request to all nodes in the network. This approach requires only the GUID of the accessed object, no additional location information is needed.

The broadcast message floods the network and propagates from node to node until it reaches the current home node of the accessed GAO. To send the response, the accessed GAO can use the underlying routing mechanism.

This approach has to ensure that the broadcast message reaches each node. Additionally, the algorithm has to terminate so that the message does not flood the network indefinitely.

In this approach, objects are free to migrate between different nodes. There is no need to send any update messages or to keep any additional state on the old home nodes like proxies. The drawback of this approach is the potentially high communication cost and access latency. Nevertheless, it is a last resort in case that the other protocols fail.

Various broadcast protocols have been developed. Williams and Camp [13] describe twelve broadcast protocols for mobile ad-hoc networks. They classify these protocols in four categories: simple flooding, probability based methods, area based methods and neighbor knowledge methods. They evaluated one or two protocols of each category, in total five, to make assumptions about the applicability of all protocols in this category.

Dalal and Metcalfe [14] introduced another protocol, the reverse path forwarding protocol for broadcast packets.

Bolton and Love analyzed the reverse path forwarding protocol in [15].

Bellur and Ogier describe in [16] a broadcast protocol for dynamic networks that is based on reverse-path forwarding.

Träff et.al. [17, 18] describe another approach for an optimal broadcast algorithm for SMP cluster and fully connected processor-node networks.

As all broadcast protocols introduce a high message overhead, we decided against an evaluation of a broadcast protocol in the network simulator.

Nevertheless, broadcast will be the fallback solution in our runtime environment, in case a remote reference is irreparable lost. We are currently developing a prototype of our distributed runtime environment, where we will include a broadcast protocol as fallback. When this prototype is finished, we will re-evaluate our protocols and examine the broadcast approach as well.

3.2 Central Registry

A *central registry* is a common approach to locate or track mobile objects. It is used to store the location information for all objects present in the system, e. g. at a single node. As such, this node is a single point of failure. For this rason, the registry should be replicated redundantly on different nodes. This is also desirable for load balancing, e. g. to distribute a huge number of lookup requests and decrease the access latency in large networks.

Bhattacharya et. al. [19] describe a flexible, hierarchical location directory service for tiered sensor networks, called Multi-resolution Location Directory Service (MLDS). Their approach aims at locating and tracking physical "objects", such as tools or employees, e. g. doctors in a clinic. MLDS has a four tiered hierarchy with a central registry at the top. The second tier is formed by the base stations of the different sensor networks. These networks are again clustered into groups of sensors, with a distinguished sensor node as *clusterhead*. The clusterheads form the third tier, while the individual sensor nodes below form the fourth tier. Different queries can be send to their *location directory service* to e. g. locate a specific object. To reduce the communication overhead, MLDS does not propagate all requests and information to the top most hierarchy. With this approach, MLDS has similarities to the Domain Name Service (DNS) [20], see below, on the Internet.

The *Common Object Request Broker Architecture (CORBA)* [21, 22, 23] uses an object registry, the so called *Naming Service*. CORBA is a middleware specification to allow multiple applications to communicate and exchange data among each other. These applications might be written in different programming languages, which makes it necessary to use the *interface definition language (IDL)*, that defines the interface to objects. To retrieve a reference to an object, the client either has to query the *Naming Service* with a string identifier that identifies the requested object or the referenced is passed as a parameter of a method call.

Henning describes in [24] the problems of CORBA and what one can learn from the CORBA mistakes. The problems that Henning identifies are e. g. the need for the *Naming Service*, as a client can not create an object reference without the use of this external service. Also, the specification ignores multi-threading, so that threaded applications are non-portable.

The Java *Remote Method Invocation* (RMI), see e. g. Wollrath et.al. [25], describes an approach to invoke methods on remote objects. Before an object is accessible from remote nodes, the creating node must register the object at the central *RMI Registry*, a bootstrap name server. Afterwards, a remote client can access this new remote object by requesting the *remote reference* from the central *RMI Registry*. To do this, the client has to lookup the specific name of the object at the *RMI Registry*. This name must be known in advance. The return value of the lookup is the *remote reference*, a tuple of endpoint and object identifier.

Munoz et.al. [26] compare the distributed object model from CORBA and Java RMI. They use a simple client server application as *average response benchmark*, described by Orfali and Harkey [27]. With this benchmark, the performance and timing issues of both systems are measured and compared under different conditions. The paper identifies various sources for latency overhead, such as object location and parameter transformation. They find that CORBA and RMI introduce an overhead that is about twice as high as socket calls but that they are about three times faster than HTTP/CGI. According to their conclusion, both systems are well suitable for request/response applications over Ethernet networks.

The Domain Name System (DNS) [28, 29, 20] is another example of a distributed, hierarchical lookup service using centralized *DNS name server*. The local *DNS resolver* is responsible for the resolution of a given name to the corresponding IP address. The resolver does this by by sending a query to its pre-configured name server. The name server can handle DNS queries either recursively or iteratively. An iterative query is answered by the queried name server either with the requested *resource record*, or with the address of another name server. If the response contains another name server, the resolver re-sends its query to this other 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 first queried name server. In case that the server does not know the answer to the request, itself "asks" additional server until it gets an answer, that is send back.

Similar to broadcast, we do not test one or more centralized registry approaches in the network simulator. The reason is the limited scalability of this approach and our goal to design a fully-decentralized system.

An alternative to a centralized registry is e. g. a *distributed hash table (DHT)*. DHTs provide a scalable, distributed lookup service. They are used in peer-to-peer systems such as CAN [30], Chord [31] or Pastry [32]. As we need a lookup service, e. g. to locate static (parts of) objects, we will use a DHT, or a similar approach, in our prototype as well.

3.3 Reactive Updates with Migration Proxies

If an object migrates from one node to another, the location information for this object is invalid on all other nodes except for the two nodes that are involved in the migration. To still be able to access the migrated object, an alternative to broadcast and a central registry is the use of a *proxy*. The proxy is kept on the former home node of the object. It can then forward all requests to the new location of the object. Proxies are applied by some systems found in the literature, c. f. the related work section 6.

In our system, the proxy does not need an explicit representative in the object store. It is only represented by an entry in the *GaoMap*, that stores the new location of the object, e. g. its GUID, together with its new home node and an object migration sequence number (see below).

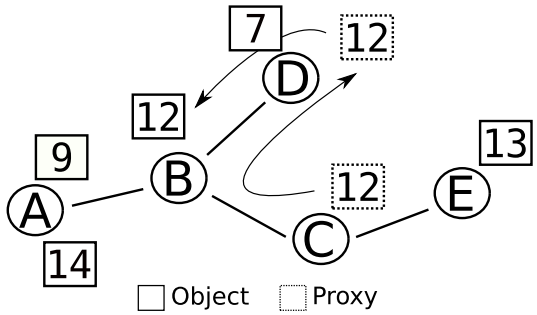


Figure 13: Proxy Chain of GAO 12.

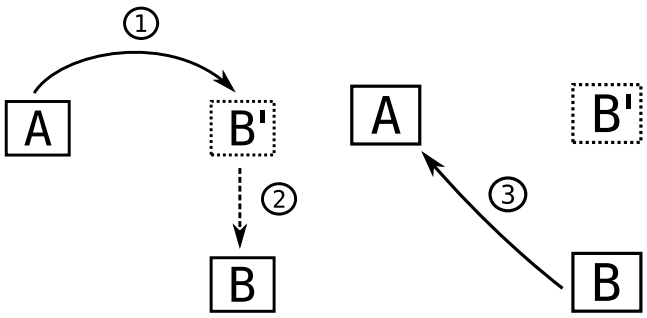
With each new migration the former home node of the object creates a new proxy. This leads to a *chain of proxies* that the migrating object creates while moving through the system. As an example of a *chain of proxies* see figure 13: GAO 12 migrated from node C to D and afterwards further to node B.

3.3.1 Object Access

The simple object access along (a chain of) proxies is shown in figure 14. Subfigure 14(a) shows the state where object B migrated to another node and left the proxy B' at its former home node.

At the beginning of the access process, object A holds an "outdated" reference to object B, c. f. fig. 14(a). Object A can be a Java object, from which a thread reads the reference to object B for the object access. Or, object A can represent the execution context itself, that holds the reference to object B, e. g. in a local variable.

To access object B, the home node of object A sends an access request (1) to the former home node, where the proxy B' resides. Proxy B' forwards the access request (2) to the now location of object

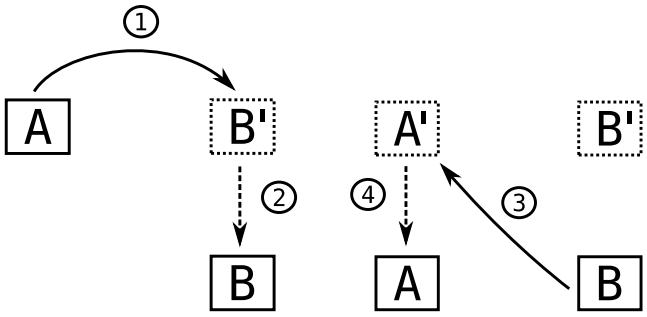


(a) Access request and forwarding.

(b) Access response.

Figure 14: Simple Object Access along Intermediate Proxies.

B. The new home node of object B answers the request and sends the response (3) back to object A. When the home node of object A receives this response, it updates it location information for object B. Afterwards, all access requests are send directly to object B, without the indirection via proxy B'.



(a) Access request and forwarding.

(b) Access response.

Figure 15: Simple Object Access along Intermediate Proxies.

Figure 15 shows the same process, this time with the additional migration of object A. In this case, object A represents the execution context that executes the access to object B. This is possible, because our system represents execution contexts as objects as well.

Again, the access of object B is initiated by the home node of object A (1), and the request is forwarded to the new home node of object B (2). Afterwards, object A migrates as well, leaving the proxy A', c. f. fig. 15(b). The new home node of object B answers the request and sends the response (3) to the old location of A, where now the proxy A' is located. Proxy A' forwards the response (4) to the new location of object A. When the home node of object A receives this response, it updates it location information for object B.

In this second example, object A cannot be a Java object, as the object access process finishes at the instance that initiated the access. This is never the object itself, but always the thread, that executes the code that accesses the object B.

Proxy Deletion Proxies remain on the former home node indefinitely, because a node cannot determine if there are references in the network that reference the proxy. For this reason the system deletes all proxies only during a distributed garbage collection run. The garbage collection (GC) starts at a root object and from there follows all valid references until an object with no further references is reached. The GC traverses all proxies which are still used and by this updates the outdated location information. Afterwards, all proxies are marked for deletion and the system can remove them.

The garbage collection makes use of the fact that each usage of outdated location information

results in the update of this information. The system can perform this update reactive during the object access either in a recursive or iterative manner.

Reactive and Recursive Updates In the recursive update approach only the home node of the accessed object sends a response message. All traversed proxies only recursively forward the object access *request message* to the next proxy in the proxy chain. This process continues until the message reaches the current home node of the accessed GAO. The home node answers the request with a *response message* that is sent back to the home node of the accessing GAO. This *response message* is sent directly to the requesting node, it does not traverse the chain of proxies.

When the requesting node receives the response, it can update its GaoMap (reactive update on the fly) and send all further requests directly to the GAO. If the requesting object itself migrated in the meantime, it left a proxy that is now used to forward the *response message* to the object.

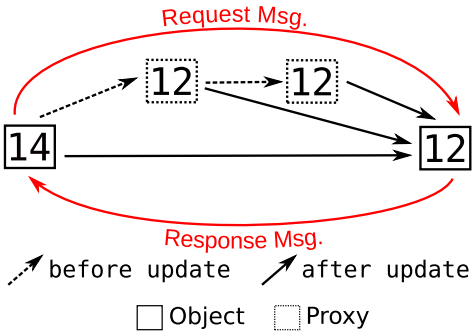


Figure 16: Reactive and Recursive Update of Proxy Chain.

After receiving the response, the requesting node might send an optional *update message* to the old, outdated location information, namely the first proxy. If this message is forwarded along the proxy chain, all proxies in this chain can update their outdated location information. On one hand, this shortens the access for all other GAOs that use one of these proxies. On the other hand, this increases the traffic overhead. Figure 16 shows the proxy chain before and after the access of object 14 to object 12 and the following proxy chain update. The two labeled arrows indicate the travel of the *Request Message* and the *Response Message*. The unlabeled, dashed arrows indicate the outdated location information of the corresponding references. The unlabeled solid arrows indicate the updated location information.

Reactive and Iterative Updates In the iterative update approach each proxy sends an *update message* directly back to the requesting node instead of forwarding the message along the chain of proxies. This update message contains the new location information for the next proxy in the proxy chain or the final location of the accessed GAO. The accessing node iteratively re-sends the request to this new location until the request reaches the home node of the accessed GAO. After the requesting node received the response, it might again send an optional *update message* to the proxy.

Figure 17 shows again the proxy chain. As before, the labeled arrows show the travel of the *Request*, *Update* and *Response Messages*.

Cyclic Routing Object migration and different GAO access patterns can lead to inconsistent object location information in the network. As an example see fig. 18 where an application with two threads access the same data. The application created a number of objects and migrated them through the network. The following time line shows an execution flow that results in an infinite proxy loop:

$t = 0$: Thread 1 on node A receives a reference to GAO 3 and stores it in GAO 1. At the same time, the location information "GAO 3 is located on node C" is stored. In the GaoMap, this is represented by the tuple $(3, C)$.

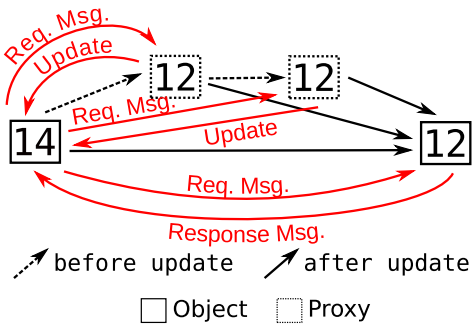


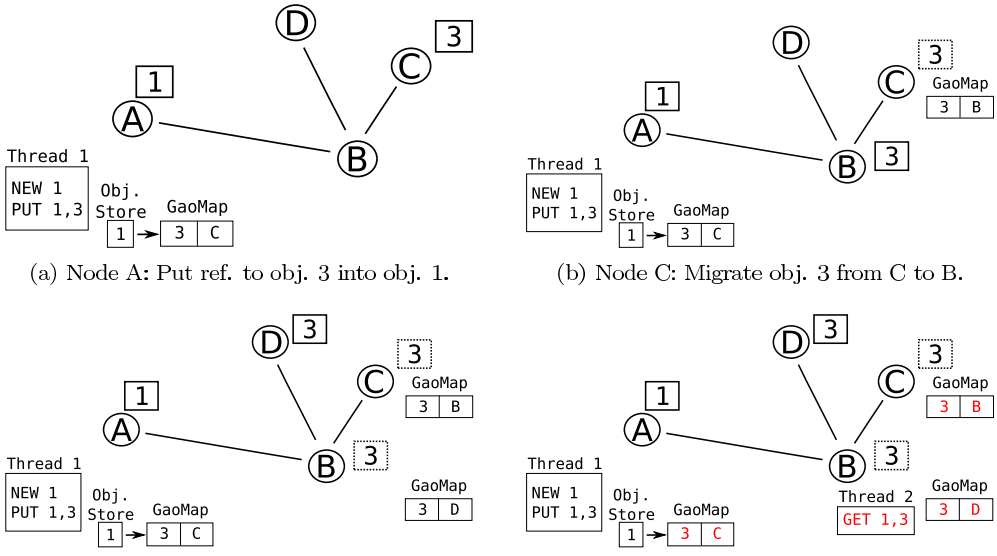
Figure 17: Reactive and Iterative Update of Proxy Chain.

$t = 1$: GAO 3 migrates from node C to node B and leaves a migration proxy. The reference of GAO 1 on node A is not updated.

$t = 2$: The runtime instance on node B migrates GAO 3 to node D and leaves the proxy $(3, D)$.

$t = 3$: Thread 2 on node B executes a GET operation for the reference field of GAO 1 which holds the reference to GAO 3.
The access message is send to node A. The following access response from node A to node B contain the old location information "GAO 3 is located on node C". But node B already stores the location information "GAO 3 is located on node D".

Now, the runtime instance on node B cannot determine which information, (3, C) or (3, D), is the correct one. If it stored the information (3, C) the result would be an infinite loop: $(B \to C \to B \to C \to ...)$. The situation would only be resolved, if GAO 3 migrated to one of the nodes that are part of the loop. Figure 18 shows the situation before node B decides which location information it should store.



(a) Node A: Put ref. to obj. 3 into obj. 1.

(b) Node C: Migrate obj. 3 from C to B.

(c) Node B: Migrate obj. 3 from B to D.

(d) Node B: Get ref. to obj. 3 from obj. 1.

Figure 18: Potential Reference Override by GET operation.

We identified three different approaches to solve the problem of cyclic routing: sending two access messages, using timestamps and using migration sequence numbers.

Approach 1: Two access messages In the first approach, node B sends two access messages along the two possible references. Only one of them will reach GAO 3 and result in a response message.

This response message contains the latest location information, that is used to update the location information on node B. As a side-effect, this access removes all proxy indirection. The other message results in a loop and cycles back to node B, where it is deleted.

Instead of sending the two access messages immediately, the node can hold them back until the runtime environment issues an access to the object.

Approach 2: Timestamps The second approach uses *timestamps*, as e. g. suggested by Moreau and Ribbens [33, 34]. As our system , they keep a record of the last known location of the object in a local lookup table. Additionally, the assign a timestamp to each location information in that table, that indicates at what time the corresponding entry was inserted into the table. In contrast to our migration sequence number approach below, this approach has to synchronize the clocks on all nodes in the network.

Approach 3: Migration Sequence Numbers In our system, a global timestamp, as described above, is unnecessary, as the migration information of a single object is unique to this object. For this reason, we do not use timestamps, but assign each object with a per-object migration sequence number. This number is stored with each remote reference entry in the GaoMap. It indicates for each entry when it was created.

Upon an object migration, the migration sequence number of the migrating object is incremented. Additionally, the sequence numbers of the GaoMap entries (for the migrating object) on the old and on the new home node are updated as well.

Of the three described approach, this one is the one that is best suited for our decentralized, distributed scenario, as it does not require additional messages or any time synchronization.

3.3.2 Object State Diagram

An object on a given node can assume five different states during its lifetime. It can be a local *object*, a *proxy* or an object in the *pending* state. Additionally, an *initial* state indicates the point at which an object comes into existence on a node, while the *finish* state indicates the end of an objects lifetime. Depending of the state of an object, the home node has to handle messages for this object differently.

Additionally to the object states, we have to consider some node and network conditions. First, we suppose that neither the underlying network nor the routing protocol is reliable and that for this reason messages might get lost. Therefore, we introduce a timeout period in which the system expects a response or acknowledgement message. If a message is not acknowledged within this time period, it is marked to be re-sent. If the original message is re-sent or if the message is modified depends on the message and the given node conditions. Consider an example, for that an access message was sent to object x on node A and got lost. At the time the message has to be resent, a new location information for object x might exist that states that object x now resides on node B. In this case, the access message is modified and sent to node B instead of node A.

Second, the update of the GaoMap might result in the update of the location information. In our scenario, this additional information is the source route that is needed to reach a remote home node. A new outgoing reference entry in the GaoMap, for example, adds a new entry to the *GaoMap* and the corresponding source route to the *RouteCache*.

Local Object State An object in the *local object (copy)* state is located in the local memory. An object is local either because of the invocation of a NEW operation that triggers a *NEW object event* or the reception of a *migration message* that indicates that an object migrates to this node.

The *NEW object event* invokes the local ORMgr and MemMgr to create the new object; and it adds the corresponding entries to the various location maintenance maps, c. f. section 2.2.2.

An *object migration message* triggers the check if there is a local proxy or an *outgoing reference (OutRef)* entry in the GaoMap. If there is an OutRef entry in the GaoMap, the node updates this entry to point to the local node. If there is a local proxy, this proxy is deleted. The node adds the location information for all OutRefs to the corresponding maps. If there is already an entry for a OutRef present on the local node, the reference counter is incremented. Additionally, the migration sequence of the current entry is checked. If a location information that came with the object migration

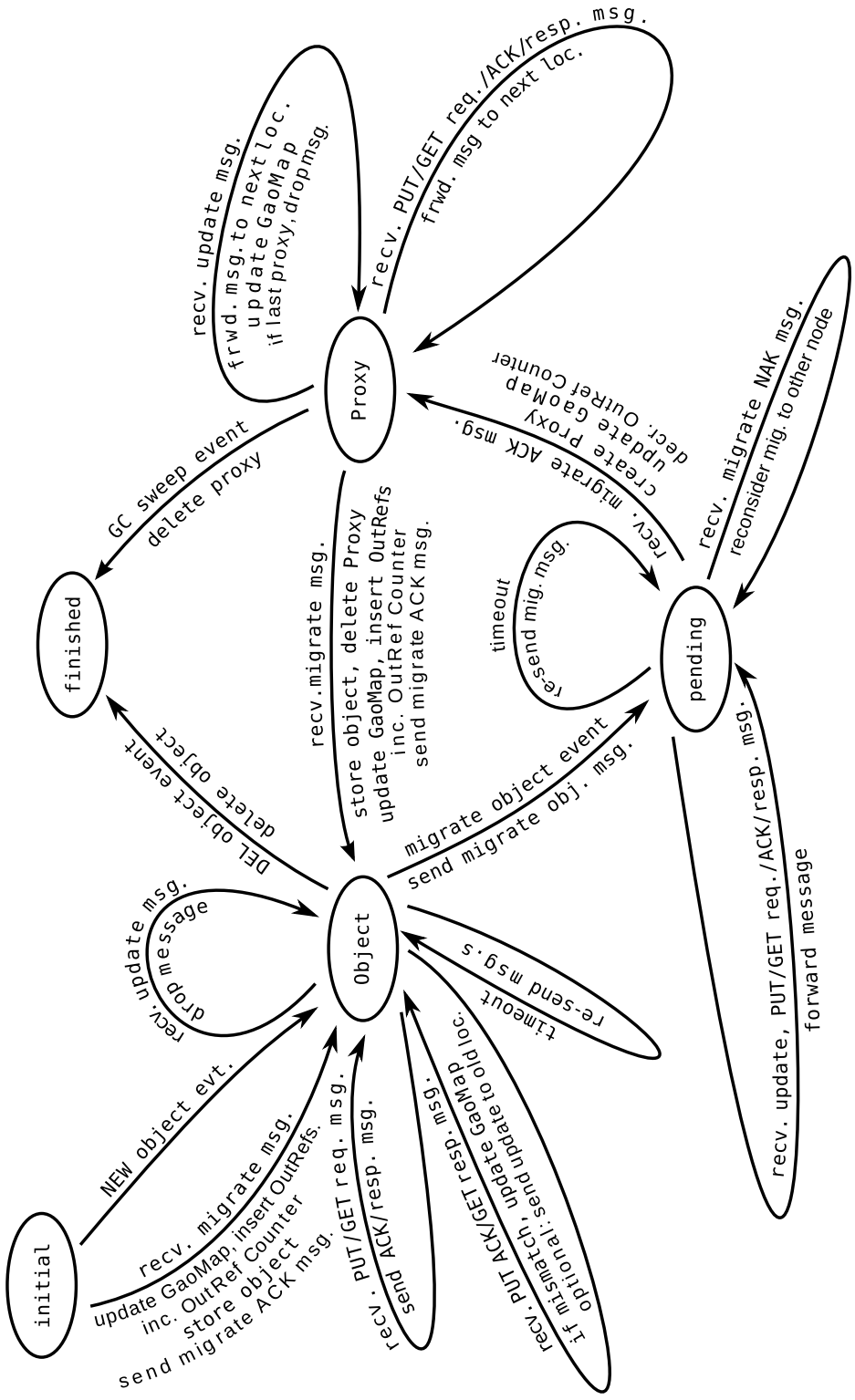


Figure 19: State Diagram: Reactive Update with Migration Proxies (Proxy Only).

is newer, the current entry is replaced, otherwise, it is kept and the information from the migration message is discarded. After this process finished successfully, the node sends a *migration ACK message* back to the sending node. It can happen the the node receives the same *migration message* twice, e. g. due to a timeout event on the sending node, c. f. *pending* state below. If this happens, this second message will be dropped. If e. g. an *OutOfMemory Exception* occurs, instead of an ACK a *NAK (not acknowledged) message* is sent back.

A PUT or GET *request message* for a local object is always answered directly with the corresponding *response message*. A PUT request writes a new reference to a reference field and is acknowledged with an ACK message. The home node answers a GET request with a response message that contains the read reference and the latest known location information of this reference.

The read reference that is delivered by a GET response is handled in the same way as an OutRef of a migrated object described above. If the current location information is outdated and replaced by the read reference, an optional *update message* can be sent to this outdated location information (the proxy). This *update message* travels along the whole proxy chain towards the current location of the object, see figure 16. When the message reaches the current home node of the object, the message is dropped. This should not happen regularly, because the last proxy in the chain should drop the message as well, c. f. the *proxy* state below.

There are two events, that cause an object to change from the *local object* state into either the *pending* or the *finished* state: the *object migration event* and the *delete object event*.

The *object migration event* initiates the transition into the *pending* state. This event invokes the MigMgr of the node, which then generates the migration message as described in section 2.3.

If there are local references to the migrated object, these references are changed from local to remote references, namely the home node of the object is set to the node where the object is migrated to. This is an optimistic approach that assumes that the migration will succeed, see description of the *pending* state below. Nevertheless, we suppose that a local access message that is sent directly after the migration message reaches the new home node when the migration was successful. If this assumption is wrong, the new home node will drop this access message which will be resend after the timeout period.

The *delete object event* results in the deletion of the object and the transition into the *finished* state. The local reference counter of each OutRef of the deleted object is decremented during the deletion process. If one of these counters drops to zero, the ORMgr deletes the corresponding entries in the location maintenance maps.

Pending State If an object resides in the *pending* state, its home node forwards all *request* and *response message* for this object. The node is not allowed to answer the request to prevent an inconsistent object state in the network.

If the migration message did not reach the destination node or the migration process was aborted due to an *OutOfMemory exception*, the node will drop the forwarded message. In both cases, the requesting node will eventually re-send the message.

Another solution is to drop all messages, wait for the re-send after the timeout and then forward the message. Our approach is optimistic that the migration succeeds. In this case, the node will be able to answer the forwarded request and no re-sending is necessary.

If the home node of the pending object receives a *migration NAK message*, the migration process was aborted, e. g. due to an *OutOfMemory exception*. In this case, the entity that triggered the migration has to reconsider the migration. Depending on the migration policy, the entity can either choose another node to migrate the object to or to invoke the garbage collector on the corresponding node to try to free memory for the migrating object. The actual policy is beyond the scope of this paper. Our simulation environment expects that there is always enough memory for the the migrating object. For this reason, no *migration NAK message* occurs in our simulation. Nevertheless, it can happen that a migration does not succeed before the above mentioned forwarded access message reaches the node. This is e. g. the case if the migration message was sent along a longer source route than the forwarded access message. This happens if the sending node received an new source route to the destination node inbetween these two messages.

The reception of the *migration ACK message* triggers the transition into the *proxy* state. At this time the actual proxy is created and the reference counter of the object's outgoing references is

decremented. This decrement must not happen before the migration succeeded. Thereby, we prevent the heady deletion of the entries in the location maintenance maps, which might be needed if the migration aborts. After the reference counter are decremented, the MemMgr deletes the object from the local object store. After that, only the proxy entry and a possible outgoing reference entry to the migrated object remain in the GaoMap.

If the home node of the pending object does not receive the *migration ACK message* during the timeout period, the *migration message* is re-sent.

Proxy State The node that holds an object in the *proxy* state forwards all messages for this object to the next known object location. This state does not handle the timeout period, because no messages are sent actively in the *proxy* state.

If the corresponding object re-migrates back to this node the proxy is deleted. If a remote OutRef to a former home node of the object exists locally, this OutRef is changed to a local reference. Additionally, the location information, namely the home node of the object is changed to the local node. Finally, all steps for a *migration message* are executed as done for the transition from the *initial* to the *local object* state.

If the system supports the optional *update message* mentioned above, the node checks the corresponding location information of the proxy. The node updates this location information only if the information in the *update message* is newer. If this is true, the node also forwards the message to the former location, which is the next proxy in the chain of proxies. If the location information is older or equally old, the message is dropped.

As mentioned before, the distributed garbage collection will update all outdated OutRefs that point to proxies during its mark phase. The sweep phase will then delete all remaining proxies which results in the transition from the *proxy* state into the *finished* state.

3.4 Proactive Update Messages with Incoming References

The reactive update approach described in section 3.3 has two potential drawbacks. Both result from a high migration but low garbage collection rate. First, each migration increases the length of the proxy chain, with the result of long chains of proxies. If an object is seldom accessed, a long proxy chain increases the object access latency. Second, if objects are accessed frequently and the location information are up to date, proxies that are not needed anymore, stay in the system and occupy memory.

To circumvent these drawbacks, we introduce the *proactive update* approach. Here, invalid location information are updated directly after the success of an object migration. This approach requires additional backward references, the so called *incoming references (InRefs)* of an object.

The main idea is that the home node of the remaining proxy sends update messages to all of the proxies incoming and outgoing references. This has two advantages: A all location information are up to date, access messages are send directly to the migrated object, with no (or if, only a few) proxies in between. Second, if the home node of a proxy can be sure that the proxy is not referenced anymore, the node can delete this proxy. The disadvantage is the increased management and message overhead that is needed to update the remote location information.

3.4.1 Incoming References

An *incoming reference (InRef)* is the *backward reference* of an outgoing reference (OutRef): it points from the *referenced* object back to the *referencing* object. For all OutRefs from object A to any other object B ($A \longrightarrow B$) there exist a corresponding InRef from object B back to object A ($A \longleftarrow B$). If object A references a number x of other objects on the same node Y, object A has x incoming references from node Y. This requires that object A has to store x InRefs. If update messages are send to all InRefs after object A migrated, the former home node of object A would have to sent x update messages: One to each referencing object on node Y, all containing the same new location information. To prevent this, the protocol does not store an InRef for each referencing object but consolidates all InRefs from the same node in only one InRefs for this *referencing node*. As a result, object A only stores one InRef node: the node Y.

This approach is the counterpart of the *GaoMap*. The *GaoMap* consolidates references as well: it combines all local OutRefs to the same remote object in only one *GaoMap* entry, and a reference counter that indicates the number of local objects holding this reference. As a result, for each *GaoMap* entry on node Y leading to a node Z, there exists an object on node Z that has an *incoming reference list (InRefList)* with an entry to node Y.

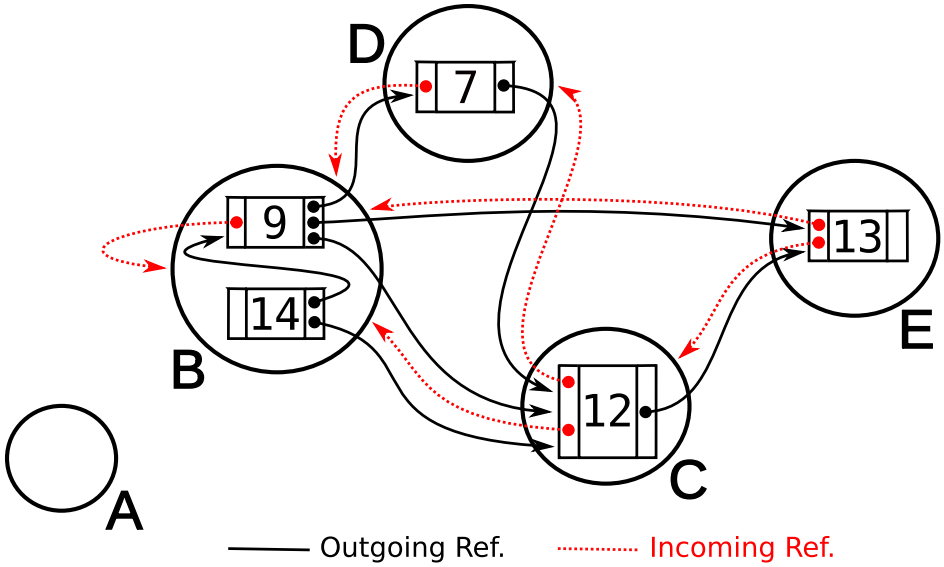


Figure 20: Reference Graph with InRefs and Physical Location.

Figure 20 shows the modified reference graph and object placement of the chosen example setup (c. f. fig. 12). On the left side of an object are the InRefs that lead to the referencing nodes, e. g. object 7 on node D has an InRef from node B that corresponds to the OutRef from object 9 to object 7. On the right side of an object are the OutRefs that lead to the reference object, e. g. object 9 that references object 7. Note that object 12 on node C has only one InRef from node B (seen object 12 in table 3), but two OutRefs, from object 9 and 14 on node B, lead to object 12. These two references are consolidated on node B in only on *GaoMap* entry for object 12 with a reference count of two: (12 $\longrightarrow$ C : 2) (see table 4).

Object 07
Out:
$\longrightarrow$ 12
In:
$\longleftarrow$ B

Object 09
Out:
$\longrightarrow$ 7
$\longrightarrow$ 12
$\longrightarrow$ 13
In:
$\longleftarrow$ B

Object 14
Out:
$\longrightarrow$ 9
$\longrightarrow$ 12
In:

Object 12
Out:
$\longrightarrow$ 13
In:
$\longleftarrow$ A
$\longleftarrow$ B

Object 13
Out:
In:
$\longleftarrow$ B
$\longleftarrow$ C

Table 3: Incoming and Outgoing References of each GAO.

Table 3 shows the *outgoing* and *incoming references* of each object. OutRefs are marked by a right arrow ($\longrightarrow$), which is read as this GUID "*is located on*" that node. InRefs are marked by a left arrow ($\longleftarrow$), which is read as this GUID "*is referenced by*" that node. Table 4 shows the GaoMaps of each node. The value that is separated by a colon shows the local reference counter.

3.4.2 Differences to Reactive Update Approach Proxies

The proactive update approach uses proxies in the same way as the reactive update approach, c. f. sec. 3.3: namely to forward all messages until all In- and OutRefs of the remaining proxy have

Node B
07 $\longrightarrow$ D : 1
09 $\longrightarrow$ B : 1
12 $\longrightarrow$ C : 2
13 $\longrightarrow$ E : 1
14 $\longrightarrow$ B : 0

Node C
12 $\longrightarrow$ C : 0
13 $\longrightarrow$ E : 1

Node D
07 $\longrightarrow$ D : 0
12 $\longrightarrow$ C : 1

Node E
13 $\longrightarrow$ E : 0

Table 4: GaoMap: Referenced GAO Location Information.

been updated. But additionally, it is required to store more state than in the reactive update approach to maintain the In- and OutRefs.

In the reactive update approach, a proxy was a single entry in the *GaoMap* that was used to forward all request messages. In the proactive update approach, the node that holds the proxy must keep the whole In- and OutRef information of the object. Additionally, it must keep track which references have been successfully updated and for which the acknowledgement of the update is pending. If all In- and OutRefs of the proxy are updated, the proxy is not needed any more and the home node of the proxy can delete it. This ensures the reachability of all objects before, during and after a migration.

3.4.3 In- and OutRef Maintenance

To enable the proactive update approach, we introduce two new classes of *reference maintenance* messages. The first class of messages is used to announce the establishment and removal of InRef. The second class is used to maintain the In- and OutRefs after an object migration succeeded.

The first class of *maintenance messages* contains *InRef Establish* and *InRef Remove* messages and their acknowledgement messages. Note that there is no *OutRef Establish* message because OutRefs are implicitly established during PUT and GET operations. The establishment of a new OutRef causes the sending of an *InRef Establish* message.

Whenever a PUT or GET operation adds an OutRef to a local object, the node also checks the local *GaoMap* for this OutRef. If there is an entry for this OutRef, the reference counter of this entry is incremented by one. If there is no entry yet, a new entry is created and added to the GaoMap. Additionally, the node announces this reception by sending an *InRef Establish* message to the referenced object. When receiving this message, the home node of the referenced object adds the sending node as new InRef node to this objects InRefList.

PUT operations also overwrite reference fields of objects. If a valid reference is overwritten, the reference counter of the *GaoMap* entry for this reference is decremented by one. If the reference counter drops to zero, the node announces the removal of the last local OutRef by sending an *InRef Remove* message to the corresponding object. When receiving this message, the home node of the referenced object removes the sending node from the InRefList of the object.

The sending node does not wait until the *InRef Remove ACK* message arrives, but has to delete the corresponding OutRef immediately from the *ReferenceMap* and the *GaoMap*. This is necessary, as the application might establish a new OutRef to the object in another context, while still waiting for the *InRef Remove ACK*. If the node afterwards receives the ACK message and then deletes the entry, it would remove this newly established and still needed entry.

Alternatively, the node could mark the entry with a *to-be-deleted* flag that is set when the *InRef Remove* message is send, and that causes the deletion of the entry, if the *InRef Remove ACK* message arrives. If afterwards a new OutRef entry is established, this flag is deleted again.

Which of these two alternatives is chosen depends on the protocol implementation. The first approach requires the node to store the corresponding *InRef Remove* message information, to enable the re-sending of the message, in case no *InRef Remove ACK* message is received within the given threshold. The second approach keeps this information in the *GaoMap*.

The second class of *maintenance messages* contains the *Incoming-to-Outgoing (InOut) Notify* and *Outgoing-to-Incoming (OutIn) Notify* messages and their acknowledgement messages. A node sends these messages after an object migration has finished successfully. By this, the node announces the new home node of the migrated object to all the In- and OutRefs of the remaining proxy object.

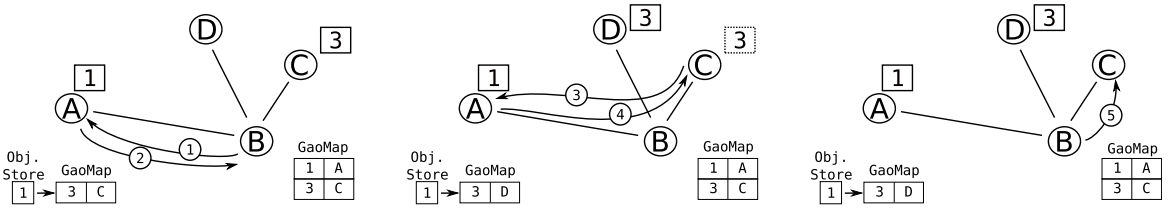
InOut Notify messages are send to the InRefs of the proxy, to update the remote OutRefs, resp. their *GaoMap* entries on the InRef nodes. *OutIn Notify* messages are send to the OutRefs of the proxy, to inform the corresponding InRefs of the referenced objects.

The nodes that receive these messages have to handle them according to the local state of their local objects. A detailed description is given below in section 3.4.6.

3.4.4 Object Access: Triangular Access Messages

The proactive update approach tries to delete proxies as soon as possible when all In- and OutRefs have been updated. Nevertheless, the update process or the proxy can overlap with pending PUT and GET operations. These operations might use the location information of the proxy object and create new remote references to the proxy.

E.g., a remote GET operation creates a new reference to a proxy. Now it can happen that the new InRef of the proxy is not established in time, before all outdated InRefs in the InRefList are updated and the proxy is deleted. See figure 21 for an example.



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

(b) Node C: InRef notification message.

(c) Node B: InRef establishment message.

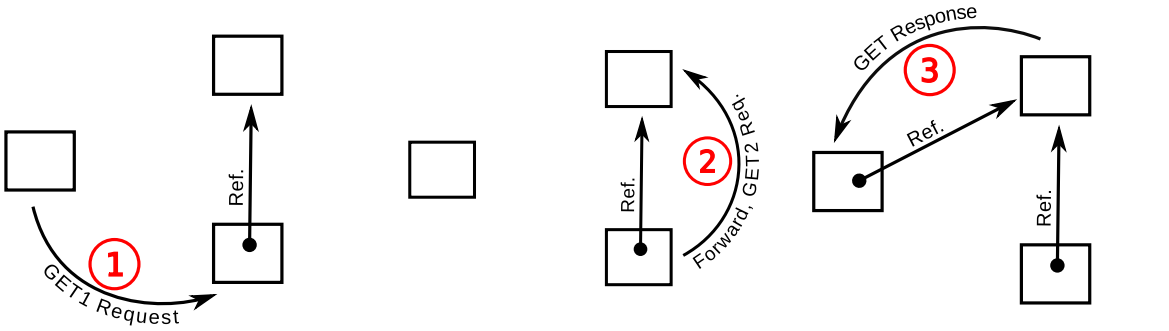
Figure 21: Reason for a failed InRef Establishment.

In this example, an object 1 on node A holds a reference to an object 3 on node C. In the first step, figure 21(a), node B reads the reference to object 3 from object 1 with a GET message (1). Node A responds with the reference (2) and the current location information *object 3 on node C* (3 $\longrightarrow$ C). Before node B sends the *InRef Establish* message to node C, node C migrates object 3 to node D. After the successful migration, node C sends an *InOut Notify message* (3) to node A, c. f. fig. 21(b). When the acknowledge message (4) from node A arrives at node C, node C deletes the proxy. If node B now sends the *InRef Establish* message (5), after the proxy was deleted, this establishment must fail, as the proxy and all information about object 3 was deleted on node C.

To guarantee the consistency of the InRefs of an object, we introduce *triangular access messages*, namely *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: the accessed object does not answer the *GET request* message but forwards the message to the referenced object, from where the response is send back to the requesting node.

First, the request message reaches the home node of the accessed object. The home node reads the reference from the accessed object field and forwards the request message to this new reference. When the forwarded request message reaches the home node of the referenced object, this home node adds a new InRef from the requesting node (the one that sent the initial request) to the objects *InRefList*. Afterwards, the request is answered with a *GET response* message back to the requesting node. Figure 22 shows the timing sequence and message flow of this access. The advantage of this process is that the requesting node not only receives the read reference (GUID) but also the latest location information of the corresponding object. The disadvantage is that the access latency is increased, because the *GET request* message has to travel to the accessed object and further on to the object to which the reference is read. With this approach, the referenced object adds the new InRef before the OutRef is established at the reading (accessing) node. Now, in case the object to which the reference was read migrates, its proxy will update this additional InRef as well.

The Triangular-PUT is similar: instead of sending the *PUT request* to the object where the reference is written to, the request is send to the referenced object, c. f. fig. 23. This message contains the GUID and location information of that object, to which the reference has to be written. The

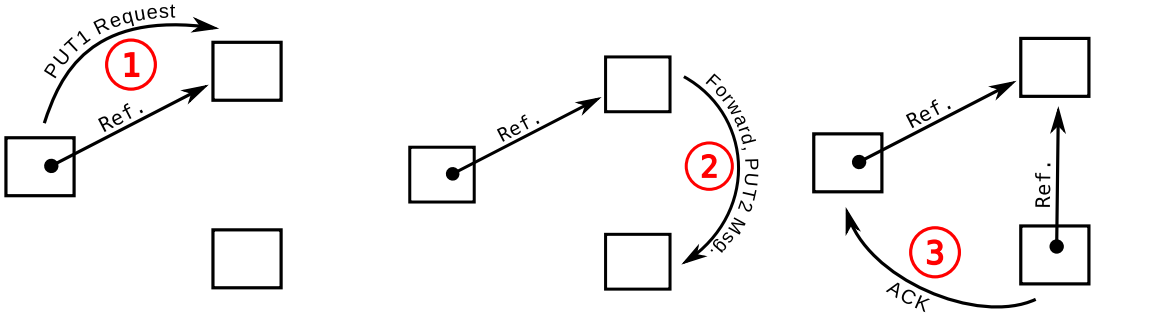


(a) Send GET Request.

(b) Forward GET Request to Referenced Object.

(c) Referenced Object sends GET Response to Reader.

Figure 22: Triangular-GET Operation.



(a) Send PUT Request to Referenced Object.

(b) Referenced Object forwards PUT Request.

(c) Written Object sends ACK Message back to Writer.

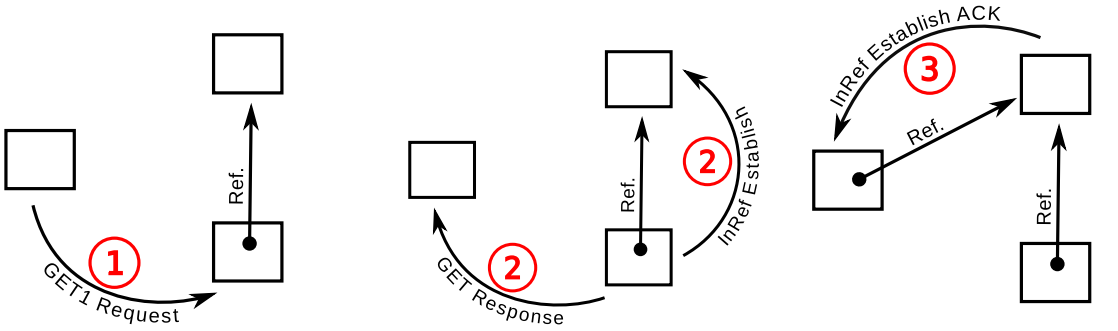
Figure 23: Triangular-PUT Operation.

receiving node adds the new InRef to the referenced objects and forwards the PUT request to the object where the reference is written to. The home node of the accessed object (where the ref. is written to) writes the new OutRef to the object and the latest location information of the referenced object is stored. Afterwards the node sends the ACK message back to the node that requested the PUT operation.

3.4.5 Enhanced Proactive Location Update

To circumvent the disadvantage of the increased access latency, we enhance the Triangular-GET approach, cf fig. 21 and fig. 24. In this approach, node A does not forwarding the request message to node C. Instead, the node sends two messages: the *GET response* message as in the original request-response protocol, back to node B, and an additional *InRef Establish* message to node C, where object 3 resides, to which the reference was read. This additional *InRef Establish* message announces the new InRef from node B to object 3 on node C. Node C acknowledges this message. But in accordance with the triangular approach of the protocol, the ACK is not send back to node A, but it is send to the requesting node A. By this, the message fulfills the same task as the *GET response* message in the 'normal' Triangular-GET approach: It acknowledges not only the new InRef of object 3, but also the success of the whole Enhanced Triangular-GET protocol. If node A would not receive the ACK within the defined threshold, it would re-send the initial *GET request* message and re-start the GET process. Note that the execution of the application might proceed after the *GET response* message is received, but that the GET operation has to be re-triggered if the ACK does not arrive. This is necessary to keep the In- and OutRefs consistent. Note that a re-trigger of the GET operation does not influence the result of the execution as the accessed object (the GAO version) itself is immutable.

The advantage of this approach is the decreased access latency, as the response is send immediately. The disadvantage is the one, additional *GET response* message.



(a) Send GET Request.

(b) Send GET Response and InRef Establish Msg..

(c) Send InRef Establish.

Figure 24: Enhanced Triangular-GET Operation.

Triangular-PUT operations can be enhanced in a similar way: It follows the original request-response approach with an additional *InRef Establish* message. But this time, the additional *InRef Establish* message is send by the node that initiated the PUT request operation.

3.4.6 Object State Diagram

The proactive update approach more complicated than the reactive update approach. The state diagram from fig. 19 stays valid, as the proactive approach uses proxies as well. Unlike the reactive approach, it does so only until all references are updated. As the proactive approach implicitly updates In- and OutRefs, the optional *update message* from the reactive approach is discarded.

The following sections presents the additional state diagrams for the proactive approach. These diagrams are split into three parts: the regular operations (fig. 25), object migration (fig. 26) and reference management (fig. 31). Note that the update of the object references includes the update of the corresponding location information, e. g. the corresponding source routes in the route cache, c. f. section 3.3.2.

The following sections deal with the case that messages have to be sent between remote nodes in the system. For access operations to referenced objects, which are located on the local, accessing node, no message is send but the request is processed immediately on the local node.

3.4.7 State Diagram: Runtime Operations

The state diagram for the runtime operations describes the handling of remote GET and PUT messages. Because we are only interested in the maintenance of object references, we only consider GET and PUT operations that access the reference fields in objects, not those that access numerical fields.

The different states of an object are the:

- *initial* state: The object is created.
- *Local Object*, or short, *Object* state: The object is a regular object that can be used in a regular way.
- *pending* state: The object is currently migrating, but the migration did not yet finish.
- *Proxy* state: The migration is finished, and the object on the former home node changes from *pending* to *Proxy*.
- *finished* state: The terminal state of an object or a proxy after it is deleted.

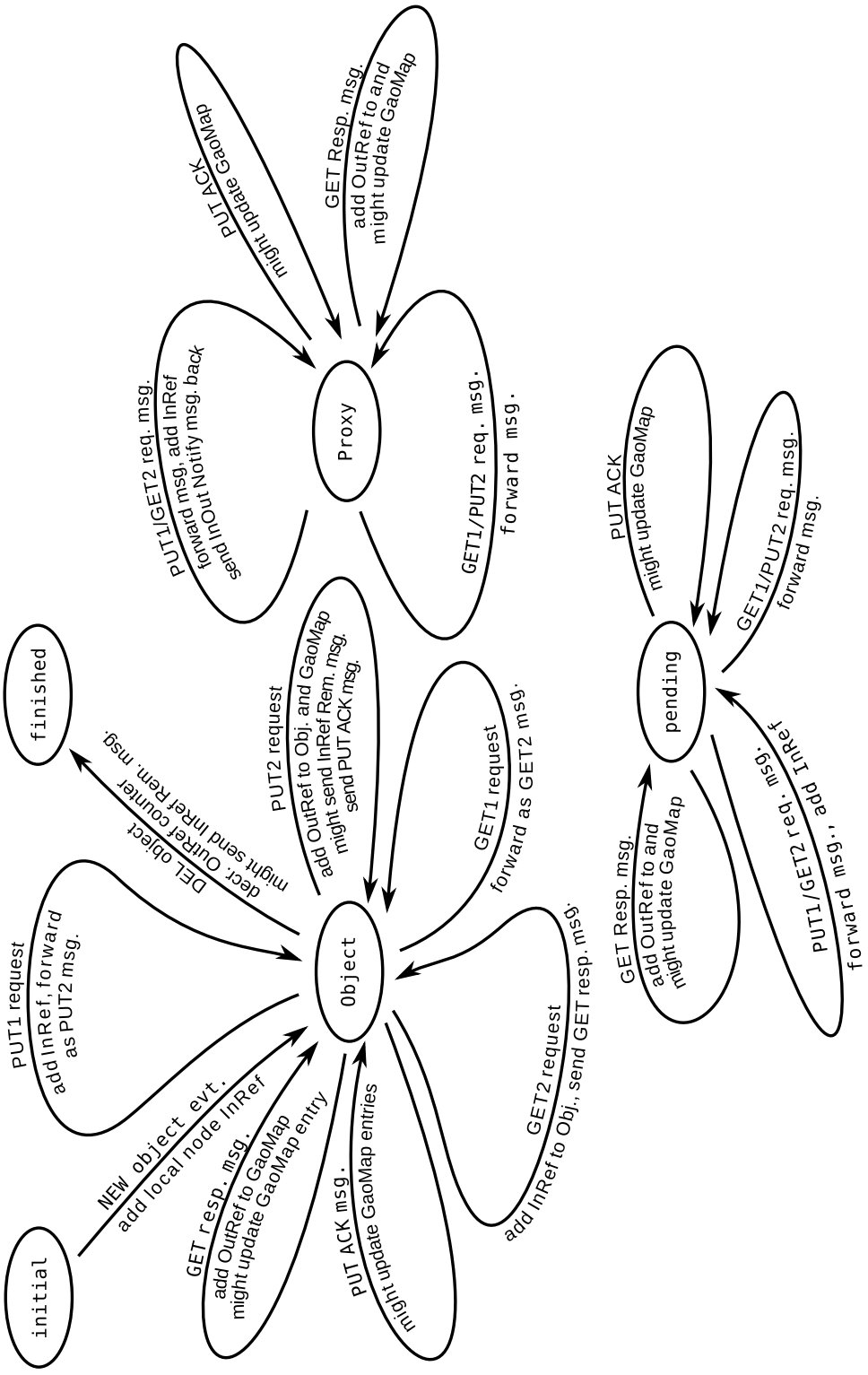


Figure 25: State Diagram: Runtime Operations, Iterative Update (Proxy and Location Update).

Local Object State This section describes the different state transitions of an object in the *Object* state. In figure 25, each transition is represented by an arrow.

An object comes into existence at the *initial* state if a thread creates a new object with a *NEW object event*. This causes the home node to add an initial *InRef* from the local node to the objects *InRefList*. This InRef indicates that the local execution context holds a reference to the object on its stack (in a local variable).

As described in section 3.4.4, each triangular access consists of two parts. We name the corresponding messages in the state diagram GET1 and GET2 request for GET requests and PUT1 and PUT2 request for PUT requests. A *PUT1 request* message is the initial message for the Triangular-PUT operation. The node that receives the *PUT1 request* adds the new InRef from the sending node to the *InRefList* of the referenced object. Afterwards the node forwards the *PUT1 request* as a *PUT2 request* message to the node where the new reference will be written.

When receiving a *PUT2 request* for a local object, the receiving node writes the new reference into the corresponding reference field of the object. Additionally, it updates the *GaoMap* by either only incrementing the reference counter of an already present OutRef entry, or by adding a new OutRef entry. As the establishment was done with the *PUT1 request* message, it is not necessary to send an *InRef Establish* message. Still, the node sends the *PUT ACK* message back to the requesting node.

A PUT operation might overwrite a reference field. If there was a valid reference stored in the overwritten field, the reference counter of the corresponding entry in the *GaoMap* is decremented. If the reference counter of this entry drops to zero, there are no more local OutRef to this object. In this case, the node sends an additional *InRef Remove* message.

A *GET1 request* message is the initial message for the Triangular-GET operation. The node that receives this message and holds the accessed object reads the corresponding reference from the accessed object field. Afterwards the request is forwarded as a *GET2 request* to the home node of corresponding object that belongs to the read reference.

The node that holds the reference object receives this *GET2 request* message. It then inserts a new InRef from the accessing node to the referenced objects *InRefList*. Afterwards, the node sends the *GET response* back to the accessing node. When the accessing node receives this *GET response* message, it handles the received reference according to the application logic and adds the OutRef to the GaoMap.

Nodes may drop messages if they can not answer a request at the time the message arrived. Additionally, packets might get lost in the system due to network failures. A timeout interval is used to deal with these two events. If a node that sent a request messages does not receive the response or ACK message within this timeout interval, the node re-transmits the lost request message. The node checks the conditions of the 'lost' *request* message before its re-transmission. This is necessary to adapt the request message, e. g. in the case that the destination of the first message changed in the meantime.

An object goes from the *Object* into the *finished* state if the runtime system executes a *DEL* operation for this object. This operation causes the node to delete the object from the local memory store. For each OutRef that the deleted object held, the node decrements the reference counter in the *GaoMap*. If the reference counter drops to zero, the node sends an *InRef Remove* message to the referenced object and waits for the acknowledgment.

Pending State An object is in the *pending* state if it has been local before and is currently migrating from the local node to another node in the network. An object in this state forwards all PUT and GET requests to the new home node of the object. If no packet reordering occurs, the new home node of the object can immediately handle the request messages. In the case of packet reordering or if the migration process fails, the new home node drops the messages. In this case, the requesting node has to re-send the dropped request message after the timeout period. In contrast to the *proxy* state, as explained below, no notification messages of any kind are sent. This is necessary because it is not sure that the migration will succeed.

For *PUT1* and *GET2* messages, the node adds a new InRef to the pending object. This InRef is necessary because there exists an additional object that holds a reference/location information to this pending object and that must thus be informed when the migration succeeds.

Because our current system does not support thread migration, *PUT ACK* and *GET response* messages are not forwarded. Response and acknowledgement messages always terminate on the same node/thread that initiated the request. For this reason, our system handles these message as described in the *Object* state: the thread stores the read reference in a local variable and the protocol adds the new OutRef to the *GaoMap* (GET) or continues the execution (PUT).

If thread migration was allowed and the initiating thread migrated, the node would have to update the *GaoMap* entry and forward the message to the requesting thread.

Proxy State A node that holds an object that is in the *proxy* state always forwards all access messages to the new home node of the object. Again, as in the *pending* state, it is not sufficient to only forward *PUT1* or *GET2 request* messages. Additionally, the node adds a new InRef to the proxy because there is still an object in the system that holds a reference/location information to this proxy. This entry is necessary to prevent the deletion of the proxy. To delete the proxy, the node has to update the new InRef. In contrast to the *pending* state, the node perform this update immediately. To do this, it sends an *InOut Notify* message back to the requesting node. After the requesting node acknowledged this message, the InRef is deleted.

Note that this procedure results the sending of more than one location update message: One or more location updates result from *InOut Notify* message sent by the proxies that are encountered on the way to the accessed object, and one update results from the final response message from the new home node of the object. This process increases the message overhead of the protocol, but it is necessary to keep the location information consistent and prevent the premature deletion of proxies.

PUT ACK and *GET response* messages are handled as in the *pending* state, they do not need to be forwarded as long as no thread migration is supported.

3.4.8 State Diagram: Migration

The state diagram for the migration process describes the different transitions of an object while it migrates from one node to another. It is the only diagram in which transitions between the three states, *Object*, *pending* and *Proxy*, occur. Additionally, this section describes in detail our solutions to a number of challenges that arise with the In- and OutRef update approach.

Local Object State The migration process starts when an object is scheduled for migration. Nevertheless, we start the description of the whole process at the *initial* state, i. e. when a migrating object reaches its new home node because this is the start state of the state diagram.

A node that accepts a migration stores the migrated object in its *local object store*. If a local OutRef entry to this object exists in the GaoMap, the entry is updated to point to the local node. If a proxy for this object resides on the local node, this proxy is deleted and replaced by the migrated object.

Afterwards, the node has to perform the necessary tasks for the reference maintenance. First, the node adds all OutRefs of the object to the local GaoMap. If the OutRef is already present, its reference counter is incremented and the migration time stamp is checked: if the location information of the OutRef from the object is newer, this information overwrites the old one. Additionally, the source route to the new OutRef, starting at the local node, is computed and added to the *route cache*. Otherwise, the locally present location information remains unchanged. If the OutRef is new on this node, the node sends an *InRef Establish* message to the home node of the referenced object.

For all In- and OutRefs of the object, the migration message contains the source route that leads to the corresponding home node. The new home node needs these routes if it has no route to the respective node yet. If no such route is present, the new route to the home node, starting at the local node, is computed and added to the route cache.

The reception of an *InOut Notify* messages results in the update of the corresponding OutRef in the GaoMap. This message does not target a specific local object itself but the corresponding OutRef in the GaoMap. If there exist only objects in the *Object* state that hold OutRefs to the updated reference, the *GaoMap* entry is updated and an *InOut Notify ACK* message is sent back. If there is at least one local object in the *pending* state, the *InOut Notify* message must be dropped because the state of the migrating object is unclear until the migration fails or finishes successfully. If the node

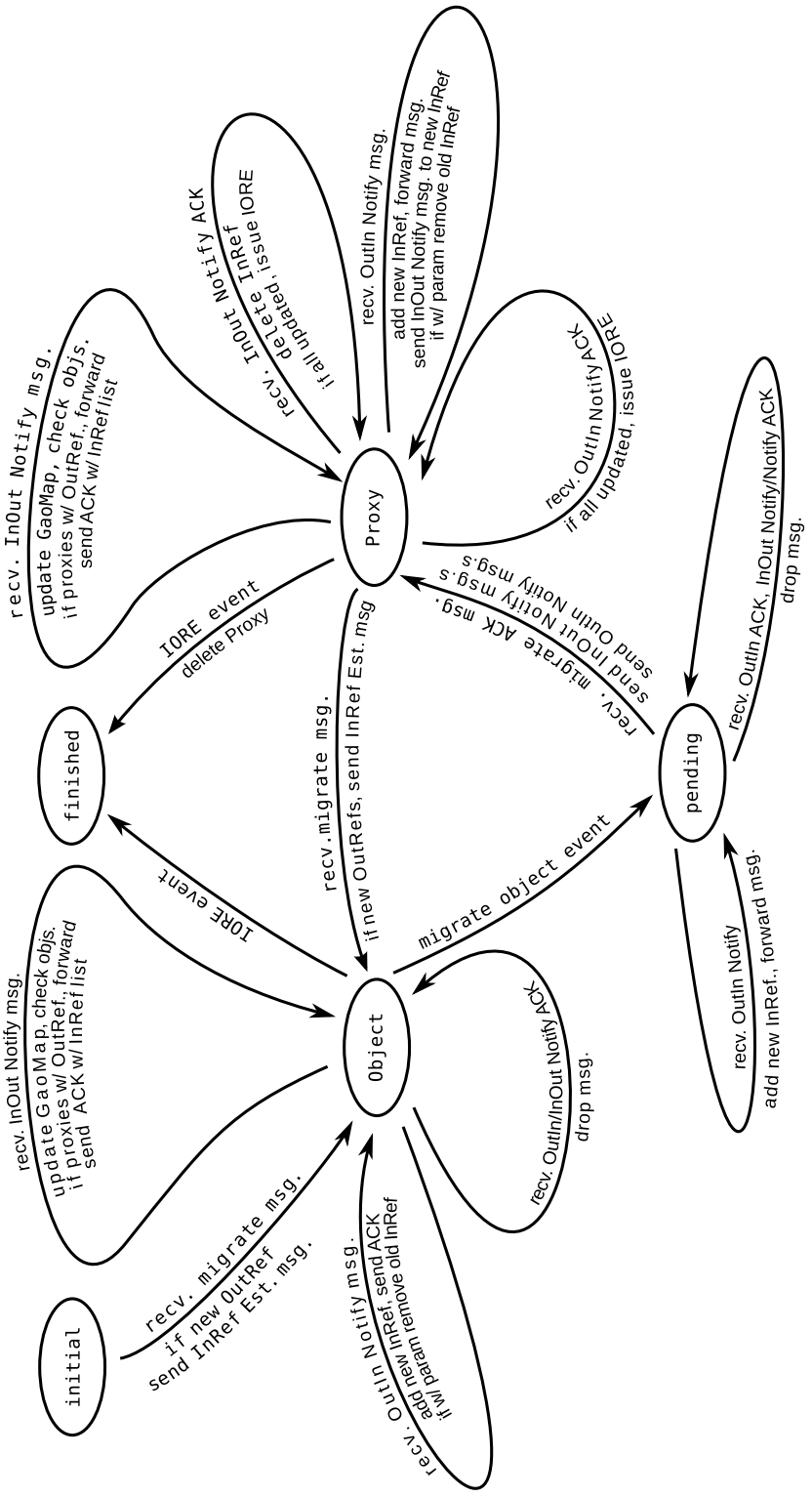


Figure 26: State Diagram: Migration Process, Iterative Update (Proxy and Location Update).

holds objects that are in the *Proxy* state, the node has to take additional measures as described below. These measures are necessary to prevent the loss of object location information. We describe this problem with the following example.

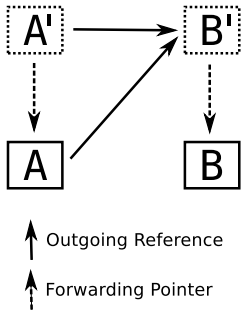


Figure 27: Simple Example of two Migrating Objects.

We begin with the description of the need for *OutIn Notify* messages and explain this with figure 27. There, the objects A and B migrate at the same time and object A is the only object that holds a reference to object B. Suppose there are only *InRef Establish* and Remove messages. If the migration of object A succeeds, the old home node of object A, the one that holds the proxy A', does not need the OutRef to object B any more. For this reason, the node could sent an *InRef Remove* message to object B that will terminate at proxy B'. During the reception process of the migrated object A, the new home node of object A will send an *InRef Establish* message. If the home node of proxy B' receives the *InRef Remove* message first, it will delete the corresponding InRef. Because this InRef was the only InRef of the proxy, the node will delete proxy B'. If the node afterwards receives the *InRef Establish* message for object B from the new home node of A, this node is not able to forward the message anymore. Thereby, object A loses the location information of object B.

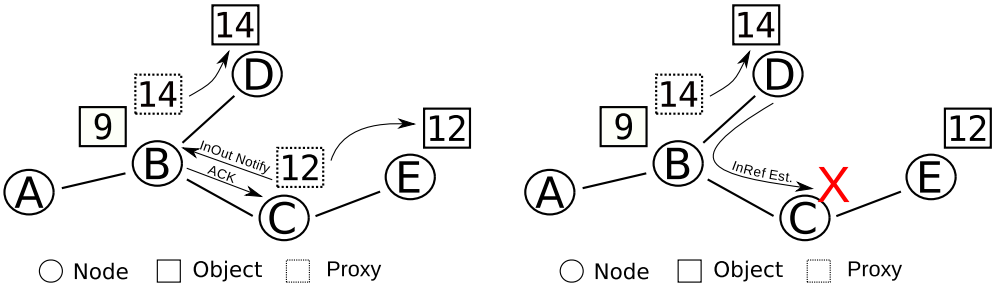
To prevent this loss, we have introduced the *OutIn Notify* message. It is sent by the old home node of a migrated object to announce the new home node of object A. It might additionally announce the deletion of the old InRef from the sending (old home) node. In the example, the node that holds the proxy A' sends the *OutIn Notify* message announcing first, the new home node of object A as a new InRef of proxy B' and second, the deletion of its own node ID from the InRefList from the proxy. The proxy B' replaces the old InRef with the new one and forwards the message to object B.

If an *OutIn Notify* message is received for a local object, the receiving node inserts the new InRef into its *InRefList*. If the *InRef Remove* parameter is set, the node deletes the respective InRef from the *InRefList*.

InOut Notify messages can cause the deletion of still needed proxies as well. As an example take figure 27 again: Object A migrates to another node, but the *InRef Establish* message did not reach B' yet. At the moment the migration of object B is finished, proxy B' sends the *InOut Notify* message back to the former home node of A, the node holding A'. When this node receives the *InOut Notify* message, it updates the location information of object B and sends an *InOut Notify ACK* message back to the node holding proxy B'. As this was the only InRef of proxy B', the *InOut Notify ACK* message causes the deletion the proxy B'. If afterwards the *InRef Establish* message of object A reaches the former home node of proxy B' the message is dropped and again, object A losses the location information of object B.

An additional problem occurs if two objects reference a third one. Figure 28 shows the setting of an example scenario: The two objects 9 and 14 on node B, denoted as (9,B) and (14,B), both reference object 12 on node C, denoted as (12,C). At the beginning, object 14 migrates to node D and object 12 migrates to node E. Before the *InRef Establish* message from node D is sent to the object (12, C), node B receives an *InOut Notify* message from node C that object 12 migrated to node E. Node B updates its local OutRef entry in the *GaoMap* and sends the *InOut Notify ACK* message back to node C. When node C receives this ACK, it deletes the last InRef of the proxy (12,C) and deletes the proxy. If now the *InRef Establish* message from node D to object 12 on node C reaches this node, it has no information about object 12 anymore. As a result, a node has to take further measures before sending

an *InOut Notify ACK* message.

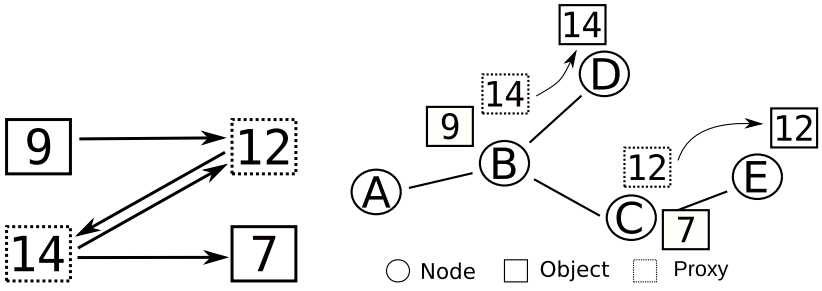


(a) Object Reference Graph. (b) Object Location in Distributed Network.

Figure 28: Reference Graph with InRefs and Physical Location.

One might assume that it is a solution to drop *InOut Notify* messages until all proxies are deleted. But this approach is not applicable, as a deadlock might occur: Suppose two proxies on two different nodes reference each other and both objects are referenced by another object on the remote node. See figure 29: proxy (12,C) is referenced by object (9,B) and proxy (14,B) while proxy (14,B) itself is referenced by object (7,C) and proxy (12,C). If the proxies 12 and 14 both change their state from *pending* to *Proxy*, both send *InOut Notify* messages to delete themselves after they have received the acknowledgment. But this ACK is never sent as these messages will be dropped on their destination nodes indefinitely. The reason is that there is always a proxy that holds an OutRef to the migrated object. Another problem is that neither node B nor node C will send an *OutIn Notify* message with a set *InRef Remove* parameter, as there are still the local objects (9,B) and (7,C).

Only when one of the objects (9,B) or (7,C) migrates as well this deadlock would happen to be broken. At this moment, the *OutIn Notify* message would be sent with a set *InRef Remove* parameter: If e. g. object (9,B) migrates to node D as well, node B sends the *OutIn Notify* message to proxy (12,C) with the *InRef Remove* parameter set to indicate the deletion of the InRef from node B. When node C receives this message this last InRef of proxy (12,C) from node B is removed and the proxy is deleted. Afterwards, node C will acknowledge the *InOut Notify* message from proxy (14,B) and this proxy is deleted as well.



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

Figure 29: Bi-Directional References of Proxies on different Nodes.

To prevent this deadlock situation, another approach must be taken: First, the node has to wait until all *pending* objects changed either into the *local* or into the *proxy* state. Until then, all these messages must be dropped. Afterwards, the node is allowed to handle the *InOut Notify* message. To do this, the node checks all local proxies if any of them holds a reference to the corresponding object in the *InOut Notify* message. If so, the *InOut Notify* message is forwarded along the chain of proxies to their new home nodes. Afterwards, the *InOut Notify ACK* message is sent back to the notifying node. This *InOut Notify ACK* message includes a *NewInRefList* that contains all new home nodes of the local proxies to which the node has forwarded the *InOut Notify* message. When the node that

sent the *InOut Notify* message receives the *InOut Notify ACK* message, it takes this list and adds the new InRefs to the *InRefList* of the corresponding local proxy. Because the *InOut Notify* message was already forwarded to these nodes, no further measures must be taken. The node simply waits for the ACK messages from these nodes. If these ACKs are not received within the timeout period, the node re-sents the *InOut Notify* messages again.

If a local object receives *OutIn/InOut Notify ACK* messages, this is an indication that the object has been a proxy before and at that time sent the corresponding *OutIn/OutIn Notify* messages. Receiving these messages means that the object has migrated back to this node in the meantime. Therefore, these messages can be dropped.

A *migrate object event* is issued implicitly by a remote PUT operation or by an explicit migration request. In both cases, the node migrates the local object to a remote node. The node that migrates the object sends the migrating object together with the location information for all its In- and OutRefs to the node that is specified in the *migrate object event*. The required location information of each of these references is retrieved from the *GaoMap* and the *route cache*.

In contrast to the reactive update approach, the OutRefs are handled during the transition from *pending* to *proxy* state. This is necessary to prevent the deletion of the *GaoMap* entries that are needed to send the *OutIn Notify* messages.

Pending State As long as an object is in the *pending* state, the only messages that are handled are *OutIn Notify* messages. These messages cause the node to add the new InRef to the pending object and forwards the message to the new home node of the migrating object. If the message reaches the new home node before the migrating object they are dropped. If this happens, the sending node will resend the message after the timeout. A node that received this *OutIn Notify* message does not send an *InOut Notify* message as the *proxy* state does because it is still unclear if the migration will succeed. The message also does not cause the receiving node to delete an InRef. This will be done during the transition from *pending* into the *proxy* state.

An object in the *pending* should not receive *OutIn/InOut Notify ACK* messages because the migration did not finish yet. If such messages are received, these could be ACK messages that result from earlier migrations between this node and another one. E.g. the object migrated from node A to node B, became a proxy on node A and sent the notification messages. Before the ACK messages arrive, the object migrated back from node B to node A and then again, migrated to another node where it is currently in the *pending* state. For this reason, all received *InOut Notify* and *OutIn/InOut Notify ACK* messages are dropped.

Proxy State After the migration process has finished, the object changes from the *pending* into the *proxy* state. During this transition, the old home node of the object sends the *InOut Notify* messages along the InRefs of the object. This message informs all nodes that hold a reference to the migrated object about its new location. When a node receives this message, it updates its *GaoMap* entry and sends all future requests directly to the actual object without the indirection along the proxy. Again, the node that receives the *InOut Notify* message has to check if there are additional local proxies, to where the message must be forwarded.

Because *InOut Notify* messages update *GaoMap* entries, it might happen that this message gets caught in a message loop. See figure 30 for an example. Such a loop is created, if two proxies on different nodes hold a reference to the announced updated OutRef and their corresponding objects resides on an other proxy node. In this example, object 14 informs the objects 7 and 12 about its migration. Node B and C both find an additional proxy that has to be informed. If the receiving proxy nodes do not check if the OutRef has already been updated, the *InOut Notify* message is ping-ponged between node B and C until the protocol deletes the proxies (7, B) and (12, C) eventually. To avoid this, each node checks if the corresponding OutRef, to object 14 in the example, was already updated and if so, it drops the message.

During the transition from *pending* to *proxy*, the node decrements the reference counter of all OutRefs of the migrated object in the GaoMap. If the reference counter of one OutRef drops to zero, the *InRef Remove* parameter in the *OutIn Notify* message is set. Note that the OutRef entry must not be deleted from the *GaoMap* yet as it might be needed for the re-sending of messages. Note also

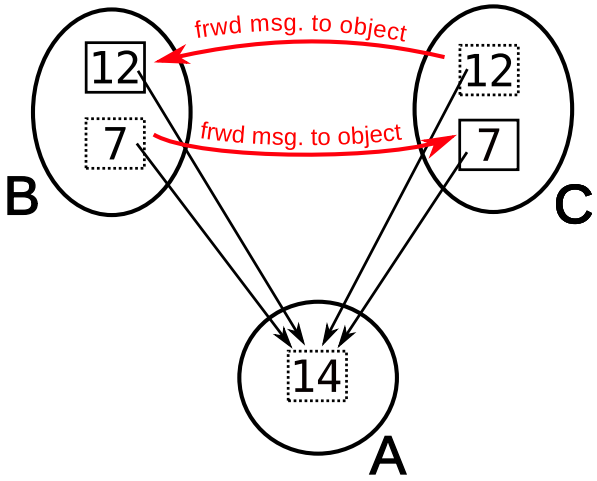


Figure 30: Simple Example of two Migrating Objects.

that the In- and OutRefs of a proxy might have been updated before the re-send timeout occurred. As a result, the re-sending routine must check if the message parameters are still valid, e. g. if the remote object location changed or if the formerly referenced object was deleted already. If e. g. the *GaoMap* entry for the notified object was updated, the *OutIn Notify* message is not re-sent because an updated OutRef entry indicates that the referenced object has been migrated as well. During this process, the referenced object has already been informed about the migration of the referencing object that corresponds to the proxy that sent the *OutIn Notify* message.

A *proxy* object that receives an *OutIn Notify* message adds the new InRef to the proxy. To delete this InRef again, the proxy sends an *InOut Notify* message back to the newly added InRef node. If the *InRef Remove* parameter is set in the *OutIn Notify* message, the corresponding node is deleted from the proxy's *InRefList*.

If a node that holds a proxy receives a *OutIn Notify ACK* message, the corresponding OutRef in the proxy object (not the one in the GaoMap) is marked as updated. The reception of an *InOut Notify ACK* message causes the deletion of the corresponding InRef from the proxy's *InRefList*.

When all InRefs are deleted and all OutRefs stored in the proxy object are updated, the node issues an *In- and OutRef Empty* event and the proxy is deleted.

3.4.9 State Diagram: InRef Management

The incoming reference management is tightly coupled to both, the regular operations and the migration process. Nevertheless, this state diagram describes the InRef establishment and removal only.

Local Object State The handling of InRef management messages is straight forward for a local object in the *object* state: An *InRef Establish* message adds a new InRef to the object whereas an *InRef Remove* message deletes an InRef from the object. Both messages are acknowledged with the corresponding ACK message.

The reception of the *InRef Establish ACK* message updates the *GaoMap* entry of the corresponding OutRef, if the referenced object has migrated in the meantime.

An *InRef Remove ACK* message is not associated with a specific object on the receiving node. The node that receives this message has to check if the local reference counter of the corresponding entry in the *GaoMap* is still zero because this was the reason why the node has sent the *InRef Remove* message in the first place. If so, there are still no local objects that hold a corresponding OutRef and the entry in the *GaoMap* can be deleted. If the reference counter is greater than zero, a new OutRef has been established in the meantime and the entry must not be deleted. In this case, the ACK message is dropped.

The local thread that created/used the object added the local node to the InRefList of the object.

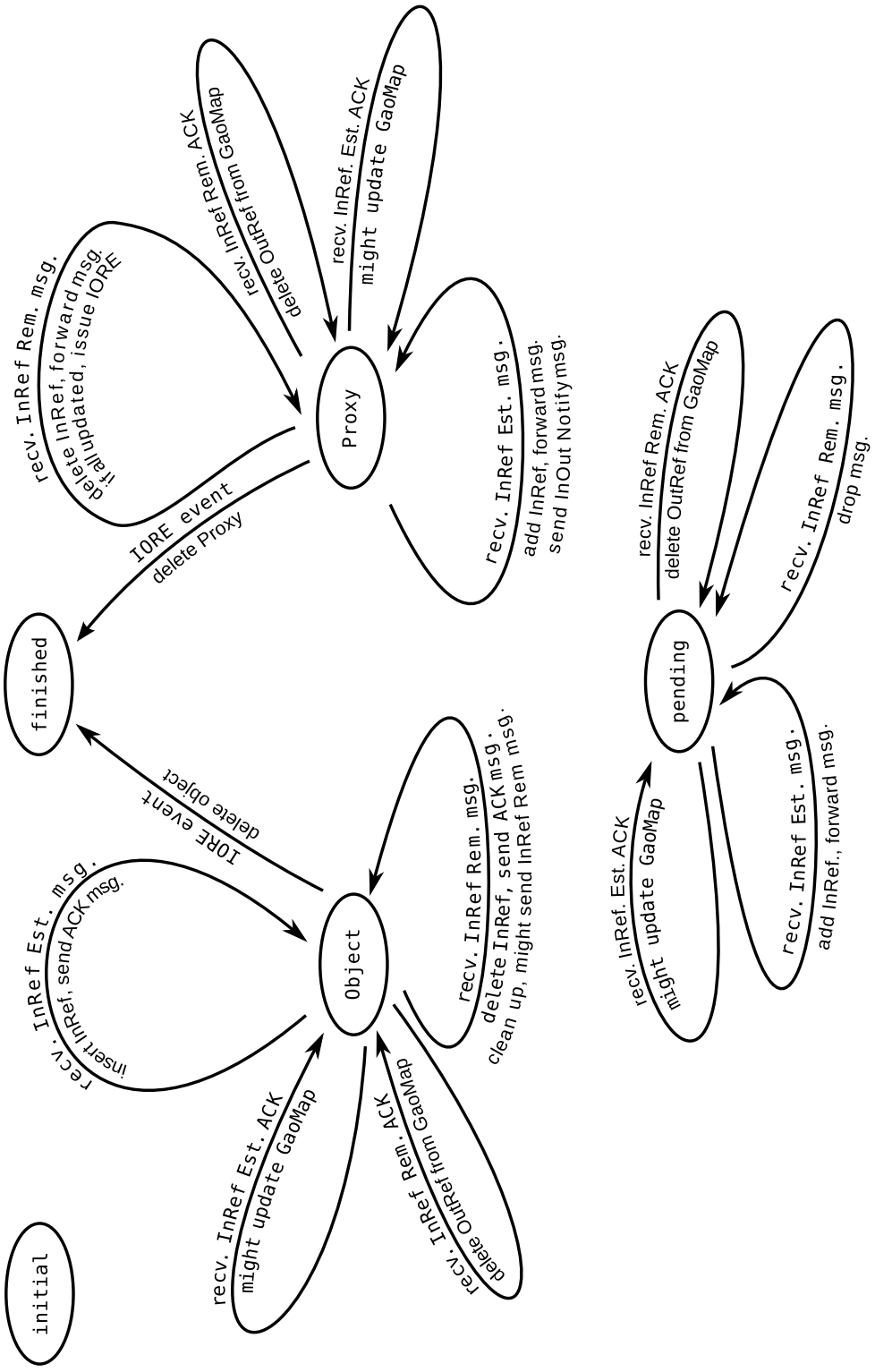


Figure 31: State Diagram: InRef Management, Iterative Update (Proxy and Location Update).

So, if the last InRef of an object is deleted, there are no local or remote objects or threads that hold a reference to that object. In this case, the object can be deleted before the garbage collector is invoked.

If a node deletes an object, it has to decrement the reference counter of all OutRefs of that object. If the reference counter for one of these references drops to zero, the node sends an *InRef Remove* message to the referenced object. Afterwards, an *InOutRefEmpty (IORE)* event is issued and the object is deleted.

Pending State An object in the *pending* state drops all *InRef Remove* messages to prevent an inconsistent object state during and after the migration process. The node that has sent the *InRef Remove* message is responsible to re-send the message after its timeout period.

In contrast to *InRef Remove* messages, *InRef Establish* messages must be forwarded to prevent an inconsistent object state. The new InRef is added to the pending object but no acknowledgement or *InOut Notify* message is sent. This ensures that the *InRef Establish* message will either successfully reach the migrated object or that the message is re-sent. The latter might happen if packet reordering occurs and the *InRef Establish* message reaches the new home node of the object before the migrating object itself. In any case, the additional InRef that was added to the *pending* object ensures that an *InOut Notify* message is sent after the migration succeeds.

The forwarding of *InRef Establish* messages is also necessary in case a high migration rate results in a chain of proxies. If one proxy in this chain would not receive this additional InRef, the proxy might be deleted erroneously and the chain breaks.

In the same way, the new InRef that is contained in an *OutIn Notify* message has to be added to the pending object. Nevertheless, if the message contains an *InRef Remove* request, this InRef must not be deleted to prevent the deletion of the proxy. Otherwise, the node that sent the *OutIn Notify* message and that happens to not receive the ACK message would delete the proxy. In that case, the sending node would try to re-send the *OutIn Notify* message indefinitely.

If a node receives an *InRef Establish ACK* message it updates the corresponding OutRef entry in the GaoMap. It can do so because the *InRef Establish* message is linked to the *GaoMap* and not to a specific object.

An *InRef Remove ACK* message is handled in the same way as described above for the *local* state.

Proxy State A node that holds an object in the *proxy* state forwards all *InRef Establish/Remove* message, as it does in the *pending* state.

A node that receives an *InRef Establish* message not only forwards the message but adds the new InRef to the proxy object and sends an *InOut Notify* message back to the sending node.

If a node receives an *InRef Establish ACK* message, it can update its corresponding OutRef entry in the GaoMap. If only proxies exist that hold the corresponding OutRefs, this step is not necessary.

If the node receives an *InRef Remove* message, the announced InRef is deleted from the *InRefList* of the proxy and the message is forwarded to the new home node of the object. If this was the last InRef of the proxy and all OutRefs have been successfully updated (see migration process above), the node issues an *InOutRefEmpty (IORE)* event and the proxy is deleted.

If a node receives an *InRef Remove ACK* message, it has to check if it is allowed to delete the corresponding entry from the *GaoMap* (see the description of the *local* state above for details).

4 Locating Static Objects

In Java, the *static fields* of an object are class variables, i. e. they exist only once per class. According to the Java specification, the first thread that accesses a class must execute the code that initializes the class' static fields. Afterwards, each Java virtual machine that accesses the class must be able to access the so initialized static fields. To do so, it does not require any explicit reference; the identifier of the class suffices. Hence, the virtual machine requires a resolver protocol that retrieves the location of the static fields for each class.

Since all the static fields of a class are initialized by the same thread, we combine them into a so-called static object. I. e. , we do not support separate locations for the static field of the same class.

A common Java virtual machine (JVM) initializes the static fields during the class loading. 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 field of that class. In a distributed environment, each instance of the distributed JVM has to know if the static fields have already been initialized when it first accesses a class. If it happens to be the first JVM instance to access a class, it instantiates the static object 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.

One possible solution to ensure the uniqueness of each static object and allow all nodes to retrieve a reference to that object is the use of a *Distributed Hash Table (DHT)*, e. g. CAN [30], Chord [31] or Pastry [32]. A DHT provides an abstract key space. All n nodes that participate in the DHT form an *overlay*, e. g. a *virtual ring*, on top of the underlying physical network. Depending on the chosen metric, the complete key space is partitioned into n parts, where each node in the overlay is responsible for one of these sub-parts.

When a node – say node A – wants to store a new *value* v in the DHT, it uses a hash function to generate a key k from a given application key. This key is an address in the abstract key space. Each key lies within the responsibility of one of the n nodes in the network, say node B. Now, depending on the DHT implementation, the publishing node, i. e. node A, either sends the tuple (k, v) to the node responsible for the key, i. e. node B, or it sends only the reference information *"the value that is associated with key k is located on node A"* to that node. The DHT's routing functionality resolves the associating between the key and the responsible node while routing the message. A node that wants to retrieve the value generates the same key k and sends a request towards that key. When node B revives that request, it replies with the stored value v (or the stored reference to the node that stores v).

In our case, the *key* k is the hash value of the class identifier, e. g. the string *java.lang.Object*. The stored *value* is either the *static object* or a reference to the node that (currently) stores that object.

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, the node uses this information to access the static object as it does with dynamic objects.

If the accessing node needs to migrate the static object, this change of location can be published in the DHT. This corresponds to the use of incoming references for proactive location updates.

5 Evaluation

In this section, we describe the simulation environment we used to evaluate and compare the three update approaches: the reactive, proactive and enhanced proactive update. We begin with a description our simulation environment and the Java micro benchmark application, that simulates the concurrent access of a number of threads to a red-black tree data structure. This application generated the input for our simulation environment, that is based on the OMNeT++ [35, 36] discrete event simulator framework, version 3.3. This description is followed by the evaluation of our simulation results, first, without explicit object migration and then with explicit migrations with different migration rates.

5.1 Simulation Environment

The simulation environment consists of a 10x10 grid network of toroidal connected nodes. Each of these nodes executes one thread of our benchmark. The execution environment of each node in the network is derived from the OMNeT++ *cSimpleModule* class. To communicate within the network, nodes exchange messages that trigger an appropriate action. To do this, the nodes are programmed in the *event-processing* programing model: whenever a message is delivered to a node, a special *handle method* is called with the received message as parameter. The other programing model that is offered by OMNeT++ is the *coroutine-based* programing model. Here, the coroutine runs in an infinite loop and continuously sends and receives messages. This mode is not suitable for our scenario because each node has to send and receive a variable number of different messages. Additionally, the *coroutine-based*

model requires each node to have its own CPU stack, which would require more memory for a single node than the event-base model.

Our micro benchmark is the Java implementation of a red-black tree data structure. This structure was first described by Guibas and Sedgewick [37]. A red-black tree is a self-balancing binary search tree in which search, delete and insert operations are performed in O(log n) time. If the tree becomes imbalanced by an insert or delete operation, balancing operations rotate the tree or sub-trees until it is reasonably re-balanced, i. e. until the longest path from the root to any leaf in not longer than twice as long as the shortest path from the root to any leaf.

Our application consists of a tree with x elements. In our network, 100 independent threads, one on each node, access the tree x times and search for a random element. If the chosen element is found, the thread deletes it from the tree. Then, a new, random element is created and inserted into the tree. Afterwards, this thread yields the execution to the next thread which performs the same operations. If the element is not found, the thread yields without further operations. This ensures, that there are always exactly x elements stored in the tree. We executed the application for our simulation two times: one time with $x = 50$ and one time with $x = 100$ objects.

We did not implement a full JVM instance on each node of the OMNeT++ simulator. Instead, we instrumented the Java source code of the red-black tree implementation to write out all object operations, such as the object creation or the accesses to reference fields. We do not log accesses to numeric value fields, as we are only interested in the maintenance of references. We use the instrumented source code to simulate the random access of 100 threads to the tree data structure. The application is executed in a single JVM instance on a single proceossor core. As a result, the application generates a sequential output file, which we use as input trace for the OMNeT++ simulator. In this file, each line represents one operation of one of the 100 application threads.

The simulator reads the sequential simulation trace file line-by-line, and associates each thread with one node in the simulation network. With each simulation event, the next instruction (GET, PUT, NEW, or DEL) from the simulation trace file is scheduled and executed on the corresponding node. If a GET or PUT operation accesses a remote object, the execution is stopped until the return message arrives.

One line in the output file, e. g. for a GET and a PUT operation, looks like this:

```
GET : 2 : 42 : 23 :
PUT : 2 : 23 : 17 : 64
```

In the first line we see that the second thread performs a GET operation on the object with GUID 42, and reads a reference to the object with GUID 23. In the second line, the same thread performs a PUT operation on the object with GUID 23. The thread writes a reference to object 17 to the object 23. By doing this, the thread overwrites a reference to object 64.

Since the trace was generated with a sequential generator, we do not have a parallel access to the red-black tree, but a strictly sequential one. Only the migrations run in parallel to the red-black tree operations, and can thus cause conflicts with respect to the object location. This is intended, as we are interested in the performance of our protocols under implicit and explicit object migration. Concurrent access control is not in the scope of this paper, but as described above, we envisage the DecentSTM protocol to target this task.

The entry point into the red-black tree is a Java class object, also called a *static* object. This static object is used to hold the reference to the root object of the tree. Each thread holds a reference to this static object to access the tree. But unlike the application that generated the simulation input file, the nodes in the simulator network do not share the same memory. Therefore, the location of the static object must be published before the simulation can start. For this, the simulator publishes the location and GUID of the static object to all threads during the initialization phase. Afterwards, the static object is allowed to migrate as well.

We use the simulation environment to measure the access latency to remote objects. This access latency is measured in number of hops, needed to reach the remote object. For each simulation run, we applied one of our reference maintenance strategies: Reactive Location Update (RU), sec. 3.3, Proactive Location Update (PU), sec. 3.4, and Enhanced Proactive Location Update (EPU), sec. 3.4.5.

Operation/Approach	50 objects RU/PU/EPU	100 objects RU/PU/EPU
NEW	4 950	9992
DELETE	4 898	9890
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 5: Number of Operations, Migration Rate 0%.

5.2 Simulation Runs

To measure the influence of proxies and update messages, we ran the simulation with different migration rates. The migration rate gives the probability with which each object is migrated every 10 simulation events. In other words, each object migrates explicitly every 10 simulation events, with the given probability to another, arbitrary node in the network. The destination of each migration is chosen by a random lookup in the local route cache. The chosen source route is the migration path of the object, along which the object travels to reach its new home.

For the simulation runs with 50 objects, it is set from 0% to 50% in 10% steps. For the simulation runs with 100 objects, we set the migration rate to 0%, 10% and 20%. The rates of 10% and 20% have been chosen to compare these runs against the 50 objects run for a qualitative evaluation of the protocols. We did not simulate with the other migration rates, as these are CPU intensive and not relevant, because we suggest that our prototype implementation will have implicit object migration, represented by the 0% migration rate, only. This implicit object migration results from successful local PUT operation executed in an STM transaction.

Our simulator executes all PUT operation on the local node that holds the executing thread. We choose this policy in accordance with the DecentSTM protocol [11]. In the DecentSTM context, this means that a node that executes a PUT operation on a remote object first creates a LOC, on which the operation is performed. This requires that the node fetches a copy from the latest remote GAO version with a PULL migration to the local node. When the transaction that executed the local PUT commits its changes and wins the distributed consensus protocol, the LOC becomes the new head version of the modified GAO. Therefore, this process is an implicit object migration due to a remote PUT operation, that results in a number migrations, even if no explicit migration takes place, and the migration rate is set to zero.

5.3 Evaluation of Implicit Migration Only

Table 5 shows the number of different operations (NEW, DEL, GET, PUT) for the simulation runs without explicit object migration (migration rate 0%). The number of operations is the same for all scenarios, as the same application is executed. All scenarios have also the same number of remote GET and object PULL migration operations, because the thread and object layout is the same as well.

For the tree with 50 objects, the table shows that from 984 766 GET operations only 63 241 operations, i. e. about 6.5%, could be answered locally. All other operations, about 93.5%, had to access an object on a remote node.

A tree with 100 objects has an higher depth than the tree with 50 objects. This leads to longer access paths from the root to any leave in the tree. Additionally, the application accesses the tree multiple times: first, to find an element in the tree, second, if the element is found, to delete the element from the tree, and third, to insert a new element. Both, the deletion and the insertion might result in the re-balancing of the tree. As a result, the tree with 100 objects has with 3 993 158 GET operations about four times more GET operations than the tree with 50 objects. Of these 3 993 158 GET operations, only 144 037 operations, about 3.6%, could be answered locally.

Operation/Approach	50 objects			100 objects		
	RU	PU	EPU	RU	PU	EPU
Remote GET Proxy Frwds	298 332	0	0	1 036 525	0	0
Remote PUT 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 6: Number of Proxy operations, Migration Rate 0%.

The tree with 100 objects has about twice as much PUT operations than the tree with 50 objects. PUT operations are performed whenever a new element is inserted into the tree, or if the tree is rebalanced. For both trees, about 25% of the PUT operations have been answered directly on the local node, all others operations had to fetch the remote object first, before the node could perform the operation locally.

In addition to table 5, table 6 shows the total number of proxy forwards for remote GET and remote PUT operations. Note that a remote PUT operation in this case means the message, that fetches the remote object, and not a remote PUT message, which performs the PUT on the remote node. Compared to no forwards in the PU and EPU approaches, the RU approach had to forward the messages for remote GET and remote PUT operations along (chains of) proxies.

Table 6 shows only the total number of GET request messages that have been forwarded along a (chain of) proxies. This number is broken down in the histogram of figure 32(b). It shows, that 795 736 messages reached their destination without a proxy forward, 114 764 messages have been forwarded along one proxy and 51 320 messages along a chain of two proxies. The number of proxy forwards continues to decrease, and ends with 47 messages that have been forwarded along a chain of eight proxies; which is not visible in the figure.

About 75% of all PUT operations required a previous PULL migration. Because GET operations reactively updated the object locations of the read object, a later PULL migration can use this updated location. Still, there are 286 PULL migration messages for the 50 object tree, and 525 for the 100 object tree that have been forwarded along a proxy. This can happen if e. g. a thread re-balances the tree. During this operation, the thread might PUT a new reference to an object that was not accessed for some time by that thread.

Reading a reference field from a remote object, that itself holds an outdated location information is another reason for an outdated object location. This might happen, if the home node of the read object did not itself access the read reference recently. In this case, an old reference location (leading to a proxy) is read during the GET request, which is then used for the following PULL operation.

In our benchmark application, each thread starts its access to the tree from the static root object. For this reason, each thread updates a number of remote object and reference locations frequently, depending on the current object value that has to be searched, deleted or inserted into the tree.

The PU and EPU approaches both deleted 19 150 proxies for the 50 object tree, and 38 139 proxies for the 100 object tree. After these simulation terminated, no proxies remained in the system.

In contrast to the PU and EPU approaches, the RU approach did not delete any proxies and after the simulation terminated, 345 proxies remained in the system. This number of proxies is small, compared to the number of deleted proxies in the PU and EPU approaches. This small number is the result of objects that migrate back to a node, where a proxy for this object resides. In this case, the node implicitly deletes the proxy and replaces it with the actual object.

As table 5 shows, access messages in the PU and EPU approaches reach the accessed object without proxy indirections. Nevertheless, this advantage is payed with a high penalty: First, these two approaches send a factor of about 2 (PU), and a factor of about 2.7 (EPU), more messages than the RU approach (see table 7). Second, the access latency in the PU approach is about twice as high to reach the accessed object, due to the Triangular-GET messages, c. f. fig. 32(a) and fig. 33(a). We will describe these drawbacks in more details below.

As said above, the PU and EPU approaches send a factor of about 2 (PU), and a factor of about 2.7 (EPU), more messages than the RU approach. Table 7 shows the sources of these additional messages,

Messages/Approach	50 objects			100 objects		
	RU	PU	EPU	RU	PU	EPU
InRef Establish + ACK	0	24 852	1 276 750	0	50 344	5 647 242
InRef Remove + ACK	0	1 629 482	1 629 482	0	7 168 616	7 168 616
InOut Notify + ACK	0	118 458	118 458	0	208 500	208 500
OutIn Notify + ACK	0	51 642	51 642	0	103 104	103 104
Total Maintenance + ACK	0	1 824 432	3 076 332	0	7 530 564	13 127 462
Total Object Acc. + Resp.	1 842 850	1 842 850	1 842 850	7 698 242	7 698 242	7 698 242
Total Send Msg.	1 842 850	3 667 282	4 919 182	7 698 242	15 228 806	20 825 704

Table 7: Access and Reference Maintenance Messages, Migration Rate 0%.

that are needed for reference maintenance. The numbers for each message type add up the number of request and acknowledgement messages.

The table shows, that the EPU approach sends about 1.7 times more messages than the PU approach. This additional amount of messages in the EPU approach results from the split of the PU approach's triangular GET/PUT messages. The triangular messages are split into a GET/PUT *Response* message and an additional *InRef Establish* message, which must be acknowledged as well. Compared to the triangular message approach, this adds one additional message, the *InRef Establish ACK* per access to the protocol, c. f. e. g. fig. 22 and fig. 24.

The additional *InRef Establish* messages and their acknowledgements are reflected in the 1 276 750 *InRef Establish + ACK* messages (for a tree with 50 objects) in the EPU approach, compared to only 24 852 *InRef Establish + ACK* messages in the PU approach. Adding the total number of *GET Request + Response* messages, that result in an additional *InRef Establish* messages, and the total number of *InRef Establish + ACK* messages of the PU approach, that are still needed, these are less *InRef Establish + ACK* messages, than expected. The reason is that not all *GET Request* operations must send an *InRef Establish* message, e. g. in case that the accessed object and the corresponding referenced object are located on the same node, and no *InRef Establish* message is send.

As said above, triangular messages in the PU approach need about twice as long to reach the accessed object as the bilateral request/response message pairs, c.f. fig. 32(a), fig. 33(a) and fig. 34(a) for the tree with 50 objects. The x-axis of these figures shows the number of hops, a message traveled before it reached its destination. The y-axis shows the number of messages that traveled the given number of hops before they reached their destination. The figures visualize the distribution of the *GET Request* and *GET Response* messages. The distributions follow a Gaussian normal distribution, c. f. e. g. equation 2.

The histograms show that the *GET Response* messages have about the same distribution for all scenarios. We will consider only the 50 object tree simulation for the rest of this section, unless stated otherwise, e. g. because the 100 object tree simulation offers further, notable insights.

Figure 34(a) shows the message distribution for the EPU approach. The figure shows the ideal message distribution for the simulation scenario. We consider this case ideal because all *GET Request* and *GET Response* messages reached their destination without being forwarded along a proxy. This observation is supported by table 6, that shows that no *GET Request* message was forwarded along (a chain of) proxies.

As table 6 indicates, no *GET Request* message was forwarded along proxies is the PU approach, as well. Nevertheless, figure 33(a) shows, that the Triangular-GET messages increased the access latency for *GET Request* messages by about a factor of two, compared to the EPU approach. The peak for the PU approach is around 9 hops, with 82 253 messages, but with 89 319 messages, that traveled 8 hops and 87 805 messages that traveled 10 hops. The distribution has a long tail, with e. g. 648 messages, that traveled 20 hops, and that ends with 17 messages, that traveled 32 hops, before they reached their destination.

For the EPU approach, the peak of the distribution is at 5 hops, with 166 446 messages. Their is a short tail, with e. g. 2 messages, that traveled 20 hops. The end of the tail is with 2 messages at 27 hops.

The RU approach has its peak at 5 hops as well, with 136 227 messages. But compared to the PU approach, figure 32(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 has e. g. 4 780 messages, that traveled 20 hops. It is still visible in the figure, that 691 messages traveled 30 hops, before they reached their destination.

The end of the tail of the RU distribution is at 54 hops with 3 messages. These very high hop counts are a result from inefficient source routes at the beginning of the simulation, c. f. [12]. At this stage, the route caches hold only few, potentially sub-optimal source routes, to only some nodes in the network. During the runtime of the simulation, each node 'learns' more source routes, until eventually it knows an optimal source route to all other nodes in the network.

We are interested in the mean value and the variance to further evaluate our results. For this reason, we fitted the Gaussian normal distribution function to all *GET Request* distributions:

$$f(x) = a \cdot e^{\frac{(x-b)^2}{2 \cdot c^2}} \tag{2}$$

The fitting results for all simulation runs are shown in table 13. For the RU approach, the *GET Requests* have a mean value (parameter b in equation 2) of 5.30 hops and a variance (parameter c^2 in equation 2) of 2.56 hops. The mean value of the PU approach is 8.37 hops with a variance of 4.05, while the EPU approach has a mean value of 5.04 hops and a variance of 2.28 hops.

These values show, again, that the EPU approach has the best access latency of all three approaches, followed by the RU approach. The latency of the RU approach is slightly lower, but heav-tailed[1] The reason is that some access requests did not reach the accessed object on the shortest path. Additionally, the RU approach does not delete any proxies, and therefore requires more memory. Nevertheless, it needs only half the number of messages as compared to the EPU approach, which has to send additional messages to keep the location information up to date. For this reason, it depends on the network and the application if it is preferable to send less 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).

The PU approach, with its triangular messages, is not applicable, as the access latency is higher, with a bigger variance, compared to the other two approaches. To make matters worse, it also requires additional messages to keep the location information up to date.

We will come back to these results in the next section, when we allow explicit object migration as well.

5.4 Evaluation of Implicit and Explicit Migration

Up to now, we have only examined implicit object migration, which is due to PUT operations. In this, we investigate the influence of explicit object migration with various migration rates. Possible reasons for explicit object migration are e. g. load balancing or system maintenance. We will not go into more details for these reasons, as they are beyond the scope of this paper.

We chose different migration rates, to analyze the influence of explicit object migration. We measure the object migration rate in "percent of all objects" and chose rates from 0% to 50%, in 10% steps. A migration rate of 10%, for example, means, that 10% of all objects migrate every 10 simulation steps. (Simulation steps are the granularity, with which the simulator runs. In one simulation step, a message travels for example one hop.)

With a migration rate of 50%, half of all objects migrate every 10 simulation steps. We do not expect to find systems with such a high migration rate. Nevertheless, we ran these simulations first, to see if our maintenance protocols can cope with that many object migrations and second, to further investigate and evaluate the overhead of our maintenance protocols.

To simplify the comparison, we only discuss the migration rates of 0% and 50%. The other simulation runs scale linearly between these two points, see e. g. fig. 35.

In accordance to table 6, table 8 compares the two simulation runs with 0% and 50% explicit object migration. Because more objects migrate within the system, the number of GET operations,

[1] This fact would indicate that the distribution follows a Poisson distribution. Nevertheless, we did not find any indications for this asumption.

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	25 307	63 341	25 321	63 341	25 365
Remote GET	921 425	959 459	921 425	959 445	921 425	959 401
Total PUT	51 279	51 279	51 279	51 279	51 279	51 279
Remote PULL	38 300	61 628	38 300	61 642	38 300	61 512
Remote Re-PULL	0	10 349	0	10 363	0	10 233
PUSH Migration	0	13 057 592	0	13 224 047	0	12 895 221

Table 8: Number of Operations/Events, Migration Rate 0% and 50%.

Events/Operations	50 objects					
	RU		PU		EPU	
Migration Rate	0%	50%	0%	50%	0%	50%
Remote GET Proxy Frwds	298 332	1 939 310	0	19 606	0	9 977
Remote PUT Proxy Frwds	286	21 293	0	419	0	438
Proxy Deletes	0	0	19 150	6 302 913	19 150	6 212 334
Proxy Remaining	345	6 025	0	584	0	535

Table 9: Number of Proxy Operations/Events, Migration Rate 0% and 50%.

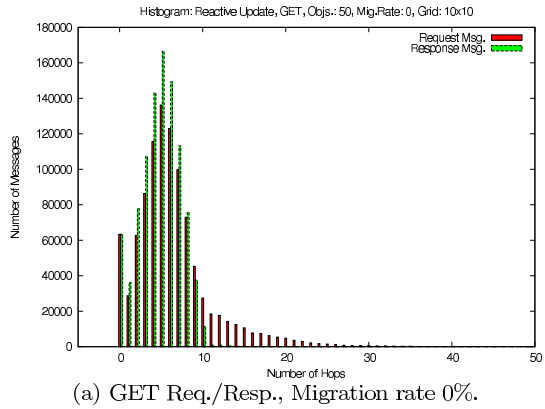
that can be performed locally, drops for all approaches from 63 341 operations (6.5%), to about 25 000 operations (2.6%). On one hand, this is not a big decrease, compared to about 13 million explicit object migrations, listed in the last row of the table. But on the other hand, even without explicit migrations, 93.5% of all GET operations access a remote object. With explicit migration, this number increases to about 96.4%.

Table 8 shows for PUT operations, that more objects have been pulled than actual PUT operations were performed. For this reason, an additional row for "Remote Re-PULL" operations was added. These "Re-PULLs" are specific to the simulation environment. The migration policy of the simulator migrates an arbitrary local object, regardless if it was recently pulled to the local object store to perform a local PUT operation. For this reason, it happens that a node *implicitly* migrated (pulled) an object to the local object store, and *explicitly* migrated it to another node, immediately afterwards, without executing the local PUT operation. In this case, the node tries to perform the local PUT afterwards, and has to *re-pull* the object again. A simple solution would be an object tag, that indicates if an object was pulled to perform a local PUT operation. We did not implement this tag, because we are not interested in a high execution performance, but in a performance comparison of the location maintenance approaches.

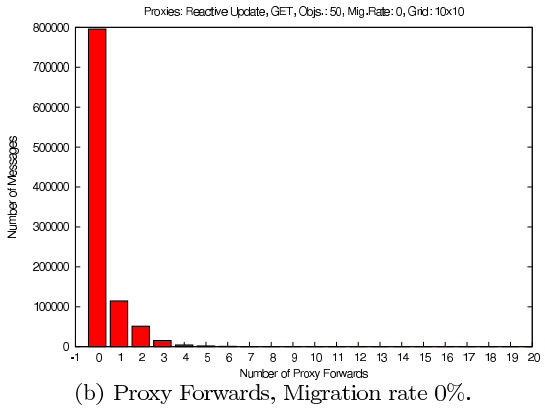
The additional row at the end of the table shows, that with 50% explicit object migrations, about 13 million migrations take place. With only 984 766 GET and 51 279 PUT operations, this means that, on average, each object migrates 13 times, before it is accessed. As stated above, we do not expect such a high migration rate in a real system, but use it to evaluate the performance of our location update protocols.

Table 9 compares the proxy operations for the two explicit object migration rates 0% and 50%. While in the 0% case the PU and EPU approach do note make active use of proxies, e. g. to forward GET messages, this changes if more objects migrate, i. e. in the 50% case. Still, compared to the 1 939 310 proxy forwards in the RU approach, these numbers are a factor 100 lower.

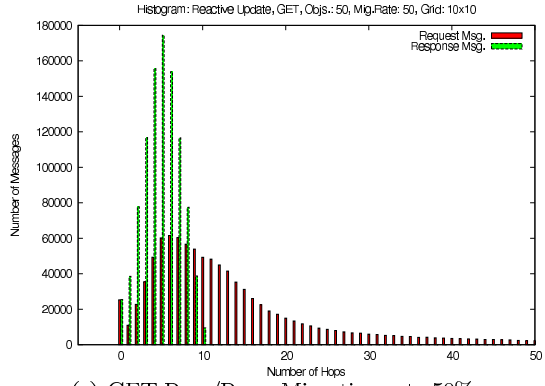
The number of proxy forwards in the EPU approach two times lower than in the PU approach. This is caused by the faster arrival of *GET Response* messages and the sending of the additional *InRef Establish* message. First, if the response is received faster, less objects are able to migrate between two GET Requests. Second, if the additional *InRef Establish* message leads to a proxy, an update



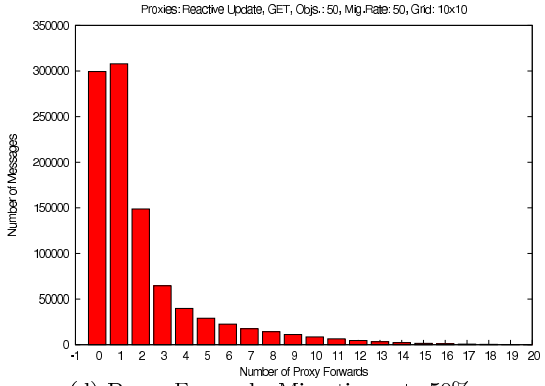
(a) GET Req./Resp., Migration rate 0%.



(b) Proxy Forwards, Migration rate 0%.

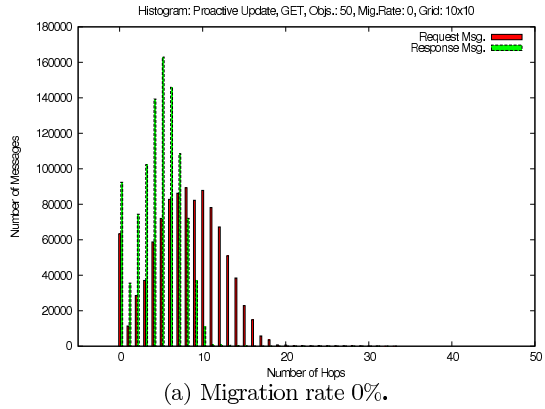


(c) GET Req./Resp.,Migration rate 50%.

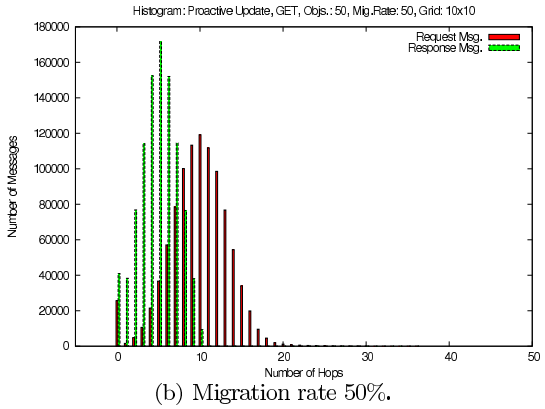


(d) Proxy Forwards, Migration rate 50%.

Figure 32: Reactive Update: Histogram of GET request and GET response messages and proxy forwards.



(a) Migration rate 0%.



(b) Migration rate 50%.

Figure 33: Proactive Update: Histogram of GET request and GET response messages.

message is send to the requesting node that is used to update the location information of the object, and by this, reduces the length of the proxy chain.

As a result, all 9 977 messages in the EPU approach have been forwarded only once. On the opposite, 14 473 messages in the PU approach have been forwarded once, while 2 565 messages have been forwarded twice, and one message has been forwarded three times. Note, that, because the table lists all proxy forwards, the number of messages, that have been forwarded twice, count twice (two forwards) in the table.

In contrast to the PU and EPU approaches, the majority of GET messages in the RU approach is forwarded along a proxy, at least once, see fig. 32(d). With 307 891 messages that are forwarded by one proxy, this number is even higher than the 299 374 messages that reach their destination directly. These numbers are followed by 148 787 messages that have been forwarded twice. As a result, the *GET Request* message distribution for the 50% migration case in fig. 32(c) is widened and significant more heavy-tailed, compared to the 0% case in fig. 32(a). The tail passes the 20 hops with 14 972 messages and has 2 233 messages that traveled 50 hops. The tail goes on with 103 messages, that traveled 96 hops and ends with one message that traveled 181 hops. For the PU approach, the worst case is one message, that traveled 35 hops, and for the EPU approach 11 messages that traveled 19 hops.

The result of fitting the Gaussian normal distribution function, c. f. equation 2, to the 50% migrations, *GET Request* message distribution, is a mean value of 8.85 hops and a variance of 5.90 hops, compared to the 0% case, that has a mean value of 5.30 hops and a variance of 2.56 hops.

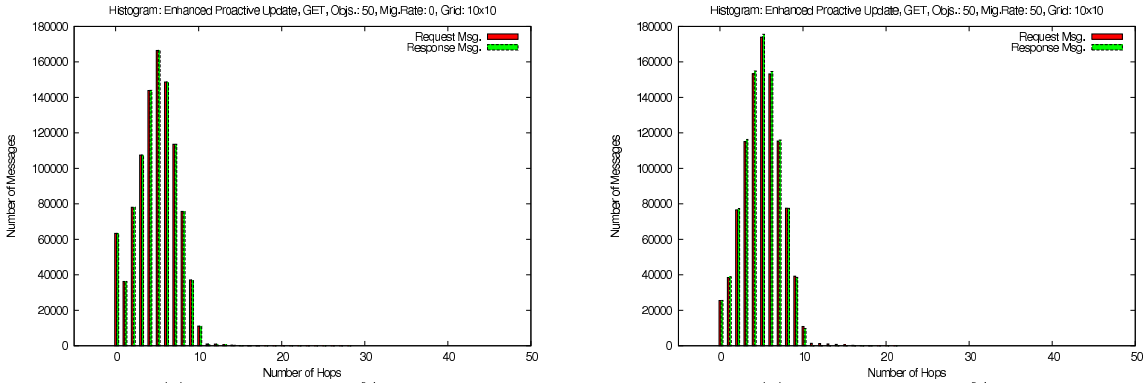
The comparison of the *GET Request* message distributions for the PU and EPU approaches, shown in fig. 33 and fig. 34, do not show such significant differences. The only difference between the 0% and the 50% case is the higher number of local (0-hop) *GET Request* and *GET Response* messages in the 0% case.

The EPU approach is closest to an optimal object access characteristic. The message distribution indicates that the *GET Request* messages traveled the same number of hops, as the corresponding *GET Response* messages traveled back.

Again, we fit the Gaussian normal distribution function, c. f. equation 2, to the *GET Request* message distribution. For the PU approach, the result is a mean value of 9.94 hops and a variance of 3.21 hops. The comparison to the 0% migration case, with a mean value of 8.37 hops and a variance of 4.05, shows, that the PU approach, with 50% explicit migrations, has a higher latency of 1.57 hops.

The fit to the message distribution of the EPU approach results in a mean value of 5.02 hops and a variance of 2.27. If compared to the 0% migration case, with a mean value of 5.04 hops and a variance of 2.28 hops, this result shows, that the performance of the EPU approach is not influenced by 50% explicit object migrations at all.

Table 10 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. We do not evaluate the 10% migration rate in more detail, but for that case, the PU and EPU approaches send a total of 27 204 874 (PU) and 28 395 538 (EPU)



(a) Migration rate 0%. (b) Migration rate 50%.

Figure 34: Enhanced Proactive Update: Histogram of GET request and GET response messages.

	50 objects			
Approach	PU		EPU	
Migration Rate	0%	50%	0%	50%
InRef Establish	12 426	10 914 200	638 375	11 604 406
InRef Establish ACK	12 426	10 914 200	638 375	11 604 406
InRef Remove	814 741	11 766 125	814 741	11 536 838
InRef Remove ACK	814 741	11 766 125	814 741	11 536 838
InOut Notify	59 229	42 788 684	59 229	41 919 139
InOut Notify ACK	59 229	42 444 550	59 229	41 631 944
InOut Notify Resend	0	19 634 431	0	16 501 939
OutIn Notify	25 821	11 116 611	25 821	10 882 250
OutIn Notify ACK	25 821	11 116 604	25 821	10 882 242
OutIn Notify Resend	0	0	0	0
Total Maintenance + ACK	1 824 432	152 827 099	3 076 332	151 598 063
Total Object Acc. + Resp.	1 842 850	1 842 850	1 842 850	1 842 850
Total Send Msg.	3 667 282	154 669 949	4 919 182	153 440 913

Table 10: Reference Maintenance Messages, Migration Rate 0% and 50%.

Events/Operations	50 objects			
	PU		EPU	
Migration Rate	0%	50%	0%	50%
InRef Establish Proxy Frwds	0	5 379 060	0	5 273 317
InRef Remove Proxy Frwds	0	1 275 558	0	1 248 860
InOut Ref Notify Frwds	0	4 260 122	0	4 167 820
OutIn Ref Notify Frwds	0	5 481 712	0	5 363 639

Table 11: Proactive Location Update Proxy Forwards, Migration Rate 0% and 50%.

maintenance messages, about 15 times more maintenance messages than object access messages.

The number of the different maintenance messages are nearly equal for the 50% migration case. In contrast to this, the 0% case shows two contrasts: First, the contrast between the numbers of *InRef Establish* and *InRef Remove* messages in the PU approach, and second, the contrast 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 (c. f. tab. 8), the additional *InRef Establish* messages for the EPU approach vanish in the total number of all sent *InRef Establish* messages.

Table 10 splits up the number of messages into Request and ACK messages. The reason for this split is the difference between Request and ACK messages. The 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 sent, but dropped at the destination node. If the migration process resolved the state that was responsible for the sending of the *InOut Notify* or *OutIn Notify* message, the re-sending is canceled after the timeout expired. If the state is not resolved, the message will be re-sent after the timeout. This happens 19 634 431 times for the *InOut Notify* messages in the PU approach, and for 16 501 939 *InOut Notify* messages in the EPU approach. The reason for this high number is again the location update protocol: if the node, that receives an *InOut Notify* message, holds local objects in the pending state, the *InOut Notify* message must be dropped, c. f. sec. 3.4.8.

In contrast to the 0% migration case, some maintenance messages in the 50% migration case are forwarded along proxies. The number of proxy forwards for the maintenance messages are listed in table 11.

The number of proxy forwards are about equal for both, the PU and EPU approach. They show that about half of all *InRef Establish*, half of all *OutIn Notify* messages, about 10% of all *InRef Remove* messages, and about 10% of all *InOut Notify* messages have been forwarded along at least one proxy.

5.5 Protocol Comparison for all Migration Rates

Up to now, we compared and evaluated the implicit object migration case and the 50% explicit object migration case. In this section, we compare the *GET request* message distributions when applying various migration rates. Again, we fit the Gaussian normal distribution function, c. f. equation 2, to the *GET Request* message distributions and plot the different mean values and corresponding variances.

5.5.1 Comparison for RB Tree with 50 Objects

The first comparison is for the red-black tree that stores 50 Objects. We ran the simulations for each location update approach with migration rates from 0% to 50%.

The mean values and variances of these runs are listed in table 12. Figure 35 plots the results for the *GET Request* message distributions, while fig. 36 plots the results for the *PULL Migration Request* messages.

Fig. 35 shows, that the mean values of the RU approach have a steady, linear slope. It starts with a mean value of 5.31 hops and a variance of 2.56 hops, if no explicit object migrations take place.

Without explicit object migration, the PU approach starts with a mean value of 8.37 hops and a variance of 4.05. The EPU approach has the best result, with a mean value of 5.04 hops and a variance of 2.29.

Mig. Rate	50 objects					
	RU		PU		EPU	
	Mean Value	Variance	Mean Value	Variance	Mean Value	Variance
0%	5.30428	2.56249	8.37611	4.05236	5.04427	2.28973
10%	5.45114	2.70836	9.92555	3.18504	5.00982	2.26701
20%	6.00729	3.35265	9.94657	3.19554	5.00388	2.26918
30%	6.92401	4.38721	9.94391	3.19286	5.00834	2.26472
40%	7.89621	5.28051	9.94879	3.20425	5.01632	2.27437
50%	8.84582	5.90493	9.94259	3.21257	5.01731	2.27315

Table 12: Mean values and variances of GET Request distributions, Migration Rate 0% to 50%, 50 objects.

The PU approach shows a 1-hop step at the transition from no explicit object migration to 10% object migrations. This step is not seen in the EPU approach. It 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, again, results in more hops a message needs to reach an object. With higher migration rates, the PU approach stays constant over all migration rates.

With a migration rate of 50%, the RU approach reaches a mean value of 8.85 hops and a variance of 5.90. The PU approach is still slightly above the RU results, with a mean value of 9.94, but a smaller variance of 3.21. The result of the EPU approach was not influenced by explicit object migration at all, and still has a mean value of 5.02 hops and a variance of 2.27.

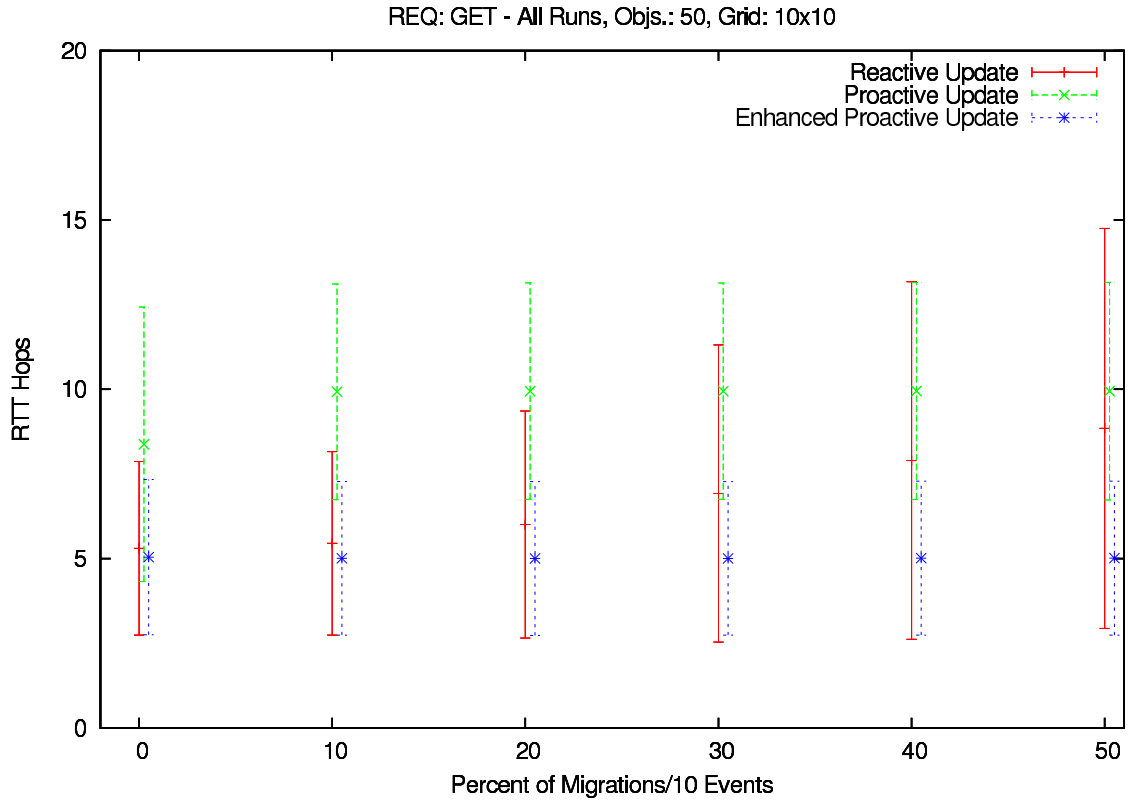


Figure 35: Comparison of hop counts for all three approaches, GET Requests, 50 Objects.

Due to the fact that *PUT Migration* message are preceded by *GET Requests*, which updated the object location information, there is almost no difference between the three approaches, see fig. 36.

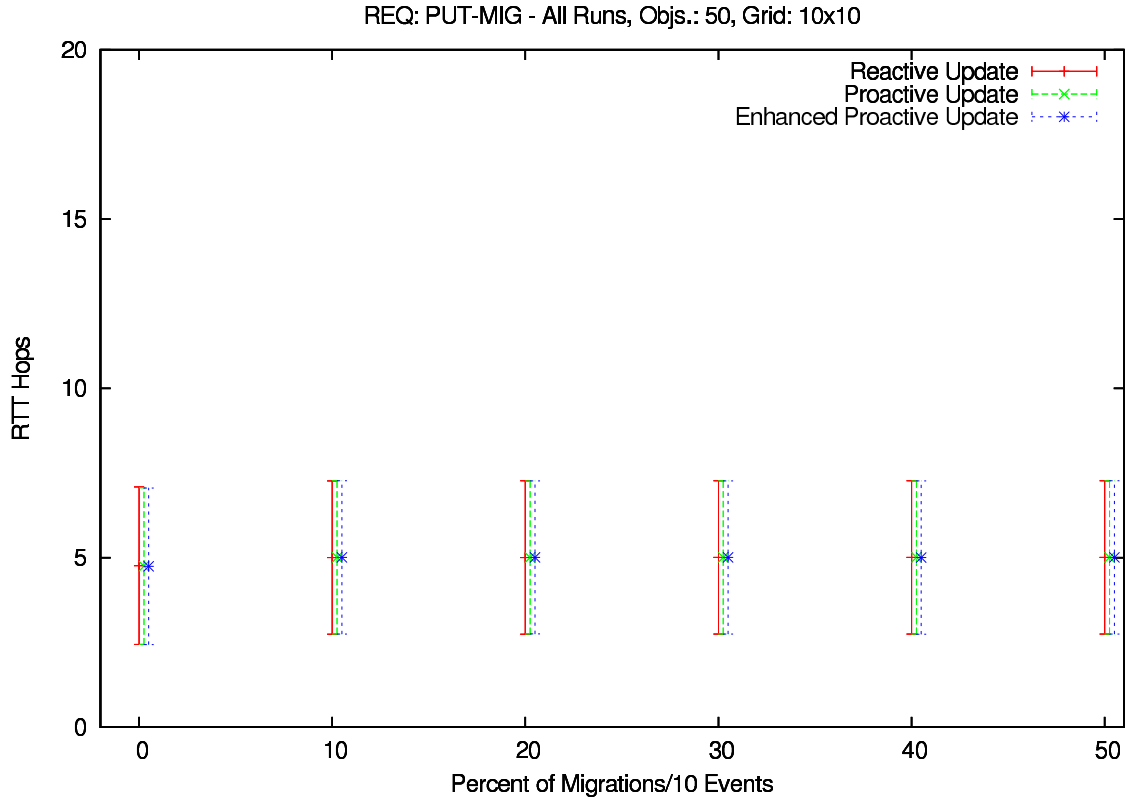


Figure 36: Comparison of hop counts for all three approaches, PULL Migration Requests, 50 Objects.

5.5.2 Comparison for RB Tree with 100 Objects

Additionally to the red-black tree with 50 objects, we ran simulations with a red-black tree that stored 100 objects. For these simulations, we applied the migration rates from 0%, 10% and 20%. We did not run higher migration rates as we are most interested in the 0% migration rate. Also, we do not expect to find systems with higher explicit migration rates so that we can spare the simulation effort.

The mean values and variances of these simulations are listed in table 13. The results are similar to the red-black tree with 50 objects, with slightly higher values for the RU approach.

Figure 37 plots these values. Again, the RU approach has a steady, linear slope. The PU approach has again the 1-hop step from no explicit migration to 10% explicit migration, while the EPU approach stays constant over all migration rates.

Again, there is almost no difference between the three approaches, when comparing the *PULL Migration Request* messages, see figure 38.

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 13: Mean values and variances of GET Request distributions, Migration Rate 0% to 20%, 100 objects.

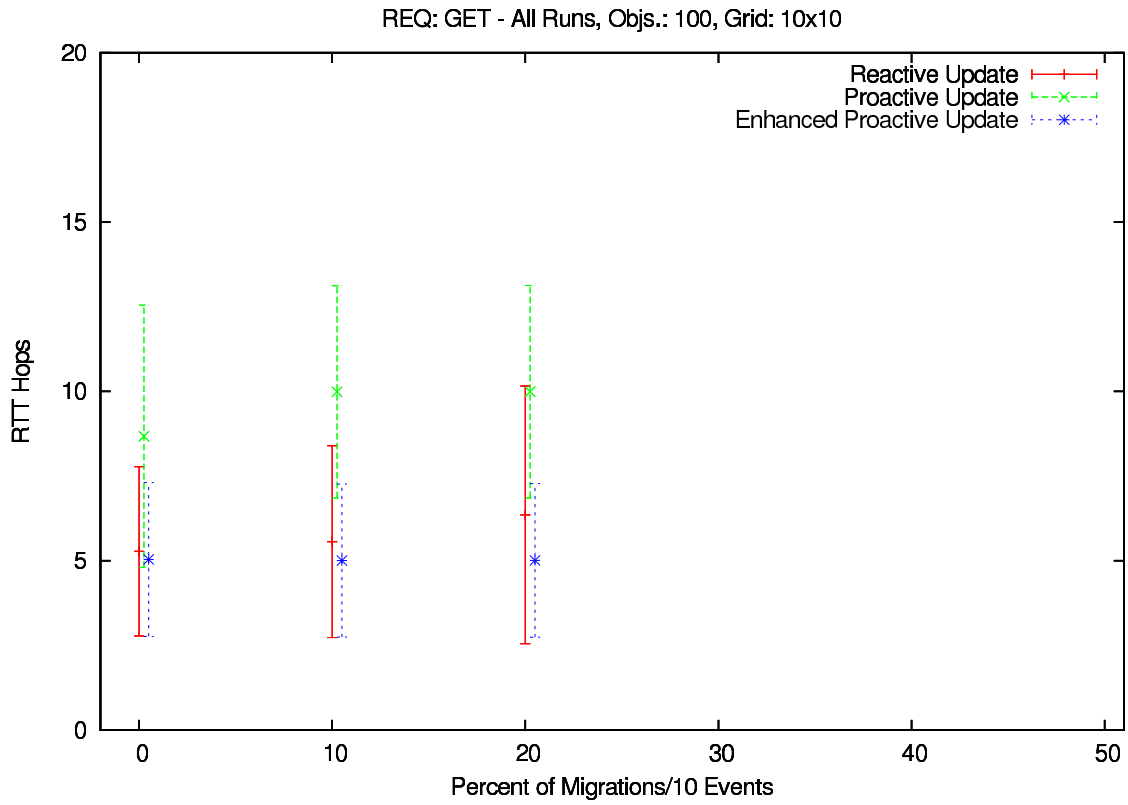


Figure 37: Comparison of hop counts for all three approaches, GET Requests, 100 Objects.

6 Related Work

Different application domains deal with locating and retrieving *mobile objects*. Depending on the domain, a *mobile object* can be

- 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 worker and tools together.
- a piece of mobile code and data, called software 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, as defined in object oriented programming languages, e. g. a Java object. The object is passively moved through the network towards the code that requires the object's data for its execution.

The movement of these objects can be either *active* or *passive*. Active movement is governed by the object itself; e. g. by a worker who decides to go to another location, or a mobile software agent that has finished its task and moves to the next node. Passive movement is managed by a separate entity that is different from the object. This entity can either be the programmer, who explicitly embeds the object migration in the program, or the distributed runtime environment that implicitly migrates objects. Examples are e. g. a distributed garbage collector that decides to migrate objects to resolve cyclic dependencies or a virtual machine that decides to migrate a thread or a process to balance the load in the system.

In the following sections, we describe the literature that we belief to be most related to our work. We start with an introduction into mobile computing and distributed objects. Afterwards, we give an overview over different programming models,such as the distributed shared memory programming

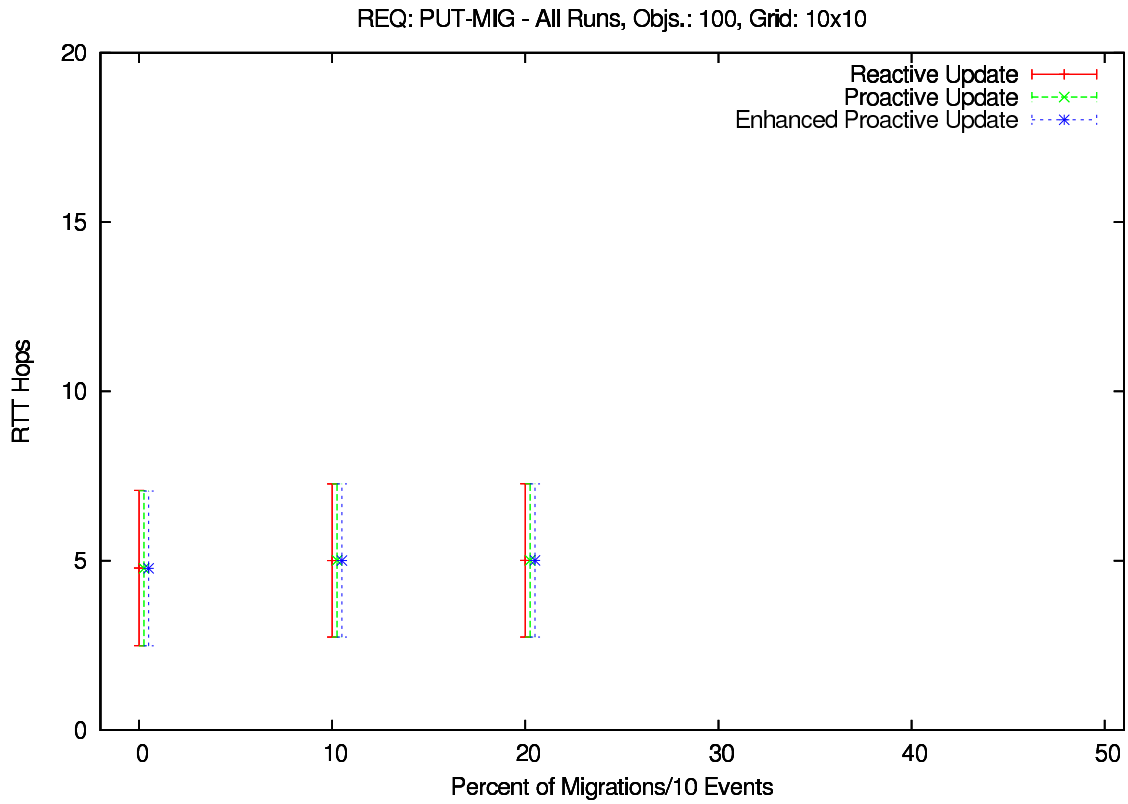


Figure 38: Comparison of hop counts for all three approaches, PULL Migration Requests, 100 Objects.

model and the *Parallel Global Object Space (PGAS)* paradigm. Finally, we discuss various projects that have developed distributed runtime environments that are similar to our approach.

6.1 Mobile Code, Mobile Objects and Mobile Computing

In distributed computing, nodes in a computer network communicate and interact with each other to achieve a common goal. Therefore, the application task is split into parts that are executed on the different nodes. There, these tasks are processed, either without further interaction between the tasks i. e. "embarrassingly parallel", or with further communication and interaction, e. g. as remote method invocations (RMI) or remote procedure calls (RPC) or via the message passing interface (MPI).

Instead of transporting the data to the code, an alternative approach is to transport *mobile code* to the data. Different classes of mobile code exist. They have e. g. been characterized by Thorn [38]. For our discussion, only two of Thorn's classes are interesting:

- a program that can be executed on different, heterogeneous processors without further adaptions.

  Examples are: Java [3, 39] that was designed with mobility in mind, Omniware [40], which uses a RISC virtual machine, the OmniVM, or the Orca [41] or Emerald [42, 43] programming languages. A sub-type of this class is the shipment of only parts of the program, e. g. threads or processes, which are distributed onto the different nodes.

- *mobile agents* [44]: an object that contains code and that is given a list of destinations and a number of operations it should perform at these destinations.

The use of mobile code can significantly reduce the amount of communication between two remote nodes. Instead of encapsulating each method call in a remote procedure call (RPC), the whole code is shipped to the remote site where the invocation is local.

Orca [41] is a programming language that aims at the easy development of parallel programs for distributed systems. It supports mobile code as a fundamental programming construct, e. g. by forking a process that is started on a remote node, and it allows a program to migrate objects between nodes. A reliable broadcast protocol handles the communication between nodes. To keep shared objects consistent, each node caches all shared objects. If one node modifies a shared object, it broadcasts these changes and all other nodes update their data.

Emerald [42, 43] is another programming language that uses a distributed runtime system and specialized compiler. It aims as homogeneous [42] our heterogeneous [43] processor clusters of up to 100 nodes. This runtime environment allows threads and objects to migrate between nodes. In the Emerald model, threads follow objects when objects are moved. Thereby, Emerald keeps the threads co-located with the objects that they access.

To reference an object regardless of its location, Emerald uses globally unique and location independent object identifiers (OIDs). Each node holds an additional object descriptor for each global object that is referenced on that node. If the object is local, the object descriptor contains the local memory pointer to the object. Otherwise, it contains information about the objects' location.

If this object location is outdated, the system uses the concept of *forwarding addresses* [45], which is the same concept as our proxy approach. If the latest known object location node is unavailable, the protocol falls back to a broadcast protocol.

As the Emerald system compiles a program to machine code, and not an intermediate language, the program must be compiled for each platform in the heterogeneous network separately. How to use the same OID for the same code object on heterogeneous platforms is described in [43], even though the authors did not implement this feature in their prototype. Their solution is to use a centralized program database that stores the different code objects for the different platforms. The first platform that compiles a code object assigns the OID and stores it, together with the code, in the database. Afterwards, all other platforms first retrieves this OID and assign it to the code object that they generate for their own architectures.

Both, Orca and Emerald introduce a new programming language that the programmer has to learn. Additionally, both systems aim at small to medium cluster sizes with up to 100 nodes, only.

Mobile objects in Java [34] is a middleware library to support the development of *mobile agent* systems. It associates objects, which communicate with each other, with a client-side stub at the sending object and a server-side stub at the receiving object.

Both, client and server object are allowed to migrate. Similarly to Emerald and our approach, each node maintains a table of all known mobile objects, together with their last known location and a timestamp. While an object migrates through the system, it leaves a trail of forwarding pointers, similar to our chain of proxies. Similarly to our approach, the location information of a migrated object is updated whenever a node accesses this object. To be able to remove proxies if they are not longer needed, the authors suggest the use of a *distributed reference counting algorithm*.

Moreau and Ribbens [34] investigate two routing strategies to forward an object access request: *call forwarding*, which is similar to our recursive proxy forwarding, and so called *referrals*, which are similar to our iterative proxy approach.

Call forwarding forwards the object access from proxy to proxy until the object is reached. Unlike our approach, the answer to an access request travels back along the chain of proxies as well. It is unclear, why this solution is chosen, and why the answer is not sent back directly.

With the *referral* approach, each proxy returns the location of the next proxy. Afterwards, the caller re-tries the access by sending the request to the next proxy, until the object is reached. The referral approach is the same as the recursive version of our reactive update approach (c. f. sec. 3.3.1).

Additionally, Moreau and Ribbens describe two proxy update mechanisms: *Eager Acknowledgements* which sends the new object location to all previous locations, namely to all existing proxies and *One Acknowledgement* which only updates the directly previous location. In contrast to our incoming reference approach, these two mechanisms only update the chain of proxies, but not the referencing objects. If Eager Acknowledgements are used, object access requests are forwarded at most once, while the One Acknowledgement mechanism only shortens the proxy chain by at most one forwarder.

The benchmark results suggest that the call forwarding approach is slower than the referral approach, while eager acknowledgments increase the object access latency. According to [34], this is

due to the fact that the call forwarding approach has to traverse the chain of proxies two times, one time on the way towards the object, and the second time, when the answer is sent back. In our opinion, this second traversal is unnecessary, as the answer should be sent back directly without the indirection via the chain of proxies (c. f. sec. 3.3.1). Unfortunately, the paper does not make any measurements how many messages are needed for the different approaches or how many proxy forwards have been needed.

Alouf et al. [46] compare the two approaches to locate a mobile agent that are implemented in the *ProActive* [47] Java library. The first approach uses *chains of forwarders*. It uses the same mechanisms as our chains of proxies approach. The second approach is based on a *centralized server*. There, a node first tries to access the object on the formerly known location. If this access fails, the node sends a location request to the centralized server.

It might happen, that the server replies with a wrong location, if it was not informed about the new location of an object yet. In this case, the requesting node will again fail to reach the object and has to issue a second request to the server. The paper makes no assumptions about frequently migrating objects, for which this approach might increase the access latency significantly. Additionally, a central server is a single point of failure.

Alouf et al. compare these two approaches with simulations and in an experimental environment. In contrast to our approach, they examined only the communication between one source and one agent. They also assume that an agent does not return to a previously visited node where a forwarder still exist. Additionally, no communication between source and agent can take place while the agent migrates.

According to [46], the centralized server works best on "high-speed" network, such as a 100 MBit/s local area network (LAN). There, it outperforms the forwarder approach. In networks with an increased latency, e.g in a 7 MBit/s metropolitan area network (MAN), which have a higher object migration time, the performance of the forwarder approach surpasses the centralized approach. One of the reasons for this result is that no communication can take place during the migration of the object.

Day et al. [48] discuss references to remote mobile objects in the distributed object-oriented database system *Thor*. This database is stored on highly-available servers, so called object repositories (ORs). These ORs, together with their objects, are replicated at a number of servers.

Thor attempts to cluster objects which reference each other, both on the same OR and close by in the local memory. With this attempt, the authors assume that references across OR boundaries are rare. They use distributed garbage collector to free objects. Object accesses are atomic transactions.

Objects are referenced by *names*. Two types of references are discussed: *location independent names*, where the reference does not change when the object migrates and *location dependent names*, where the reference changes when the object migrates.

In the case of location independent names, a locator is responsible for each object. If an object migrates, the local reference on the new node changes and the new node informs the locator about the address change. To determine the current location of an object, each access must first ask the locator where the object currently resides.

In the case of location dependent names, a proxy is left at the old home node of the object. Additionally, the old node updates all referencing objects about the new address and location of the object. To enable this update, each node holds an incoming reference list of all other nodes which reference a local object. This approach is similar to our incoming reference approach, discussed in section 3.4.

In contrast to our approach, the object reference itself is not decoupled from the object location. This means that the migration of an object requires each referencing node to touch all local objects that hold a reference to the migrating object in order to update their references.

Srinivas et al. [49] describes optimizations of the shared object space *Virat* that allows it to scale in large networks like the Internet. Objects in the Virat system are located via lookup servers that maintain the current location of the *object meta-data repository* (OMRs) in their associated cluster. These OMRs form a *pastry ring* [32], using the freepastry implementation of pastry. The routing

between the OMRs use a DHT-like lookup mechanism to retrieve the responsible OMR for a given object. For each object, k replicas are created throughout the network to deal with OMR failures.

We decided against the use of a DHT to locate dynamic objects to prevent nodes from maintaining location information for objects that they do not use themselves. Nevertheless, we use a DHT to locate static objects, which must be available to all nodes in the network.

6.2 Distributed Shared Memory (DSM)

One approach to program applications for a network of distributed computers is the use of the message passing. This approach follows a 'share-nothing' paradigm that requires all nodes in the network to explicitly send and receive messages e. g. to exchange data.

Another approach uses shared memory, where the whole memory is accessible from all nodes and processes in the network. A process can have references to all memory addresses within the global address space. Most shared memory systems require hardware support, as e. g. offered by the FLASH multiprocessor [50].

A combination of message passing and a shared address space are *Distributed Shared Memory (DSM)* systems. They provides the abstraction of shared memory in a distributed memory environment. To do this, DSM systems hide the underlying communication between the different nodes.

DSM systems can be distinguished into hardware and software DSMs. Hardware DSMs support the shared memory abstraction on the hardware level, while software DSMs provide this abstraction within the runtime environment. Another dimension to classify DSMs is fine-grain vs. coarse-grain memory or object-based vs. page-based memory. In general, each node in a DSM system can access its local memory via a coherent address space, while the network of all nodes form a non-coherent address space. In the following, we only investigate fine-grain and coarse-grain software DSMs.

6.2.1 IVY

Li and Hudak [51] implemented IVY, the first page-based DSM. IVY allows the migration of memory pages in a page-based distributed shared memory virtual address space but it allows only a single writer. The ownership of a page is transfered to the node that is currently granted write access. After the write succeeded, invalidation messages are broadcast to invalidate copies of the memory page.

In [51], Li and Hudak examine various centralized and distributed page location approaches for IVY.

The centralized manager approach maintains a table that has one entry for each memory page in the shared memory system. Pages do not have a fix owner; only the manager knows who owns the page at a given time. The manager serializes the access to the page by locking the page for all processors but the one that was granted access.

An improved centralized manager approach moves the page access synchronization from the central manager to the individual page owner. Still, a processor that wants to access a page has to first access the central manager to retrieve the owner of a page.

A fixed distributed manager approach partitions the shared address space into fixed chunks, which are distributed among all nodes. Li and Hudak state that this approach is superior to the centralized managers. Nevertheless, they argue that it is difficult to find a good fixed partition scheme that suits all applications well.

Another distributed protocol uses broadcasts to find a page whenever a page is not yet available at the requesting processor.

Finally, Li and Hudak describe a proxy approach with forwarding addresses. These proxies are updated whenever an invalidation message propagates the true owner of the page.

The prototypic implementations of these concepts have been evaluated on *Apollo* workstations that have been connected by a proprietary token-ring network, the *Apollo token ring*, introduced by the company Apollo Computer in 1981. In this network, a token travels around the ring and grants that node permission to transmit data that currently holds the token. The authors state, that the dynamic distributed manager approaches, using broadcast or proxies, perform better than the centralized approaches, if only a small number of processors share the same page for a short period of time. Nevertheless, the experiments have only be done on a network of eight processors.

6.2.2 Linda

Linda [52] is a programming tool to develop parallel programs. A program in Linda is not described as a process graph, but described as a "spatially and temporally unordered bag of processes" ([52]). Instead of partitioning the parallel program into n logical pieces that are modeled by n coupled processes, 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, un-coupled workers, which ignore each other, search for tasks to execute in the program's data space.

Linda's memory is *tuple*-granular, where tuples are stored in the *tuple space*. A tuple consists of a *logical name* and an ordered set of immutable values. A tuple is accessed via its *logical name*, rather than its physical address in memory.

The Linda implementation for the S/Net multicomputer broadcasts all messages to all nodes in the network, and stores a copy of the *tuple space* on all nodes, as well. The implementation for the Intel iPSC hypercube used a DHT in which a tuple was hashed and stored on the node that was responsible for this hash.

Compared to the 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 in the *tuple space*, the tuple has to be removed, changed and reinserted.

LIME [53] extends the Linda model to mobile environments. It was designed to allow the development of mobile agents over both, wired and ad hoc networks, where mobile agents may reside on mobile hosts.

6.2.3 JavaSymphony

JavaSymphony [54, 55, 56] is built on top of Java RMI. It offers a broader range of possibilities to control object and code locality than Linda and its derivatives JavaSpaces [57, 58], Javelin [59] and Jada [60]. The main feature of JavaSymphony is the ability of the programmer to explicitly control the object locality, e. g. the location where an object is created or the migration of an object to a remote node. It is designed as a Java library, and Fahringer states [55] that it is mostly suitable for medium- to coarse-grained parallelism, but not for fine-grained.

JavaSymphony offers synchronous, asynchronous and one-sided remote method invocations, where the latter does not wait for a result or for a method to finish. Additionally, it does not replicate all code to all nodes, but allows selective class loading on only those nodes that need them.

The user might select the nodes and resources which should be used to execute a JavaSymphony application. These resources form the *virtual distributed architecture* on which the *JavaSymphony runtime system (JRS)* is executed. Each application must register itself with the JRS, and should un-register when it is finished.

Even though JavaSymphony supports automatic mapping, load balancing and object migration, Fahringer states that automatic systems have a poor performance because of a lack of information about the application and an insufficient ability for static and dynamic analysis.

In [56], the authors describe an extension of JavaSymphony to shared memory systems such as multi-core and many-core architectures.

6.2.4 Aleph

The *Aleph* toolkit [61] offers a collection of Java packages that use remote threads to extend thread parallelism and to help to construct distributed shared objects. The toolkit supports push and pull communication, as well as object migration and remote method invocations.

[62] implemented and evaluate three different distributed directory services for the *Aleph* toolkit These directory services are used to keep track of moving objects and their cached copies. One directory implements a home based protocol; another one uses the arrow directory protocol [63]. The third protocol is a hybrid protocol of the other two.

In the home based protocol, each object is associated with a *home node* that is responsible for this object. This home node keeps track of the location of the object and all cached copies of the object. The home node manages all accesses to the object; it is not possible to reach the object other than by invoking its home node. In this scheme, only one read/write copy or multiple read-only copies are allowed. To acquire exclusive access to an object, four messages are required, two of which are blocking.

The arrow protocol creates a binary tree of all nodes so that the location of each node in the tree can be calculated from its index. The protocol uses one-hop pointers, the *arrows*, that point to the direct neighbor in the tree in whose direction the object currently resides. For each object, all nodes have to have an *arrow* to one of their direct neighbors in the tree. This is necessary even if the node never accesses this object during its lifetime.

In this protocol, each remote object access follows the arrows through the network. During this travel, each node that forwards the request changes the direction of the arrow into the direction of the requesting node, respectively 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.

It is unclear how or if this protocol can manage multiple read-only copies of the same object. Additionally, the paper does not say anything about the creation of new objects, about the initialization of the network, or about the mechanism that is used to obtain a reference.

The authors use a single shared object to evaluate the two protocols [61]. The main thread initializes the object and starts a thread on all remote nodes, which themselves access the shared object.

The authors assume that a 'multi-hop' message has the same cost as a 'single-hop' message. The experiments for the paper included at most 16 nodes, which have been connected via one Ethernet local area network. The arrow protocol that uses a binary tree as a node directory did not reflect this topology but assumed a binary tree structure.

For this reason, the authors conclude that the home base protocol only takes two messages to retrieve and request an object. For the request, one message to the home and one from the home to the current node that holds the object copy. And conversely for the retrieval: one message from the node holding the object copy to the home node, and one message from there to the requesting node.

For the arrow protocol, the authors state that a request needs d messages for a d hop path between the requesting node and the node that currently holds the object. In their metric, the way back to the requesting node takes only one message.

The hybrid protocol uses a home node for each object and each home tracks the last node that requested the object. Hence, a request message is always forwarded to the last known node. Additionally, the source of the request is stored as the new 'last requesting' node. The retrieve message is sent directly from the node that holds the object back to the requesting node, without a detour over the home node. Applying the authors *message count* metric this results in two request and only one retrieve message. It thus outperforms both previous approaches. This approach is comparable to our proxy approach, where each object access message has to travel along one proxy hop.

6.3 Partitioned Global Address Space (PGAS)

The *Partitioned Global Address Space (PGAS) model* is a rather new programming model. It is similar to the DSM model, but its main focus is on distributed array access. According to Cantonnet et al. [64], PGAS model is just an alternative name for the DSM programming model.

In PGAS, all physically addressable memory on all nodes is part of a global address space with non-uniform access time. Similar to DSM systems, access to the local partitions of the global address space has a low-latency, while access to the remote partitions has a longer, and potentially non-uniform access latency. PGAS supports the shared memory model as well as the message passing model. Thereby, it combines the ease of programming shared memory systems with the efficiency of message passing.

One of the key features of the PGAS model is to give the programmer explicit control over the location of both, data and code execution. I. e. , the programmer decides which data is shared and how it is distributed among all nodes.

6.3.1 X10

X10 is an object-oriented programming language that is part of the PGAS language family. It was proposed by IBM and aims at the programming of heterogeneous, non-uniform clusters. [65] describes the version 0.41 of the language. As of today, the latest version is 2.1.1, which was released in January 2011.

X10 is derived from the Java programming language, while the authors of [65] state that the major drawback of Java is the tight coupling to a single, uniform heap.

X10 uses the notion of *places* to allow the programmer to control the location of code and objects. A *place* is a virtual concept of non-migratable mutable data, and the corresponding *asynchronous activities* that operate on this data.

Asynchronous activities are lightweight "threads", which replace regular threads and explicit message passing. They are created locally or remotely. They support thread-based parallelism as well as asynchronous data transfer. Chales et al. [65] describe remote activities as a generalization of *active messages* [66].

In X10 only places can migrate. It is not possible to migrate data and objects between different places. Nevertheless, an activity in one place can spawn activities in other places. This is especially necessary to read/write remote data. An activity can only read/write data that is local in its own place. There, the read and write memory access happens in one atomic, *sequentially consistent* [67] step. To read/write remote data, the activity has to spawn an activity at the remote place.

Additionally, X10 supports a sub-language for sparse, dense, distributed, and multi-dimensional arrays. In this context, a global array can be distributed throughout the PGAS space. This distribution cannot be changed during the execution of the application. The array is accessed via *fat pointers*, which consist of the globally unique VM ID of the creating virtual machine (VM) and an ID that is unique within this VM. Each place provides a mapping of the fat pointers to local memory locations. As global arrays can never change their place, fat pointers are always valid.

In contrast to other PGAS languages, X10 makes the location of data directly visible in the code. According to [65], the reason is that a transparent location of objects can be a performance bottleneck because it is not visible if a data access generates remote communication.

6.3.2 Unified Parallel C

Unified Parallel C (UPC) [68] is a parallel extension of the C programming language. The main goal of UPC is to minimize the communication overhead between two cooperating threads. Similar to X10, UPC allows the programmer to decide where to place data in the system.

Barton et al. [69] describe a UPC compiler and runtime system for the BlueGene/L supercomputer [70]. The compiler simplifies the code that is generated for parallel loops and eliminates indirections for the access to shared arrays. A distributed *Shared Variable Directory* (SVD) is used to access shared data. The entries in this directory are entirely managed by the compiler during compile-time.

The runtime system uses an algorithm to determine if an access to a shared object can be performed locally or remotely. If the data is local, the reference is transformed from a *fat pointer* to a local C-pointer like reference. In the beginning, the base address of an array is determined once and always requires to follow the various indirections of the SVD. Afterwards, the subsequent array accesses are potentially translated to direct memory accesses.

Another optimization describes the update of read-modify-update operations, used e. g. in the *RandomAccess benchmark* [71]. Instead of sending three messages (read, update, acknowledgement), the authors suggest to use only one asynchronous message for the operation.

The authors have evaluated two applications from the HPC Challenge benchmark suite [71] and one application from the *NASA Advanced Supercomputing (NAS)* benchmark [72, 73, 74]. When all described optimizations are applied, the authors found a speedup of 7 for the *RandomAccess* and a speedup of 240 for the *STREAM benchmark*. They conclude that the main performance gain was achieved because the compiler transformed most of the fat pointers into local references (C-Pointers).

The data access characteristics in UPC are investigated in Barton et al. [75]. One of their observations is that the performance is significantly improved if the programmer is able to decide how the shared data is distributed among the executing threads. If the programmer provides this

knowledge, the compiler can optimize the data access at compile time. During their analysis, the authors found that a majority of shared data accesses is to local data. To do this, the compiler analyzes which shared data is accessed locally and privatizes this data for local access, e. g. via a private pointer. Thereby, the compiler avoids the translation overhead through the SVD. To analyze different access patterns, the authors use a subset of the NAS benchmark [72, 74]. During their evaluations, Barton et al. found evidence that a number of algorithms in scientific computing exhibit regular access pattern to remote data.

6.3.3 Others

Various other PGAS languages have been proposed, e. g. the Java dialect Titanium [6], Ct [76], Co-Array Fortran [77] or Chapel [78]. We are not going into more details of these programming languages here and leave the further study to the interested reader.

6.4 Distributed (Java) Virtual Machines (DJVM)

The main feature of a distributed virtual machine (DVM), or, more specific, a distributed Java virtual machine (DJVM), is to provide a *Single System Image (SSI)* [1, 2]. An SSI hides the heterogeneous and distributed nature of e. g. a cluster of PCs, and 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. In [1, 2] an SSI is defined as one of the key features that are needed to use a cluster-based system.

The SSI can be implemented on different levels:

- At the hardware level as e. g. in the FLASH multiprocessor [50], which offers a hardware distributed shared memory (DSM).
- At the operating system level as e. g. with MangetOS [79], an operating system offering an SSI over ad hoc networks, or GLUnix [80], a global layer unix for cluster.
- As application as e. g. PARMON [81], which offers the user a single application window that offers access to all system resources or services.
- As a runtime environment, e. g. as a distributed Java virtual machine as described in the following.

The hardware level SSI offers the highest level of transparency, but it is inflexible if the system needs to be extended or enhanced. The OS level SSI is more flexible but it must be modified for each new hardware technology, which makes it expensive to develop and maintain. The application level SSI is limited to a single application, for which the developer is not supported by the hardware or the OS. The runtime environment level SSI 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 the OS. If e. g. a runtime environment is developed on top of a Linux OS, it is afterwards executable on all hardware platforms for which a Linux OS exists. A programmer who programs for this runtime environment has full, transparent access to the underlying system.

6.4.1 cJVM

cJVM [7] is a distributed Java virtual machine for homogeneous cluster of computers. It implements a *distributed heap* that uses a *master-proxy* approach to access remote objects. The master node of an object is the node where the object was created; the proxy resides on other remote nodes, which use the proxy to access the object on the master node. Locally, objects are referenced by regular Java references. The first time, a reference to a local object is passed to another node, e. g. as an argument to a remote operation, it is assigned with a unique global identifier, the global address of the object (GAO). Together with the global address of the class (GAC), remote nodes use the tuple of (GAO, GAC) to create the local proxy and to access the object on it's master node. A local DHT translates between the memory local proxy address and the global address of the master object.

Instead of thread migration or object migration, cJVM supports method shipping. With method shipping, the thread itself does not migrate, but its stack can be distributed over different nodes. Nevertheless, synchronized methods and native methods must always be executed on the master node.

When a remote object is accessed, cJVM ships the method to the remote node. This process is supported with caching and replication policies. E. g. a remote access to a static field of a class is logged at the master node of that class, and a replica is cached on the accessing node. If the static field is updated, all its replicas are invalidated and require a new remote access to the field on the master node of that class. At object level, read-only objects are cached as well.

Various optimizations for the cJVM have been proposed [82]. For example, a simple object migration mechanism that handles those cases where one thread creates and initializes an object, and exactly one other, remote thread uses the object, without any overlap between these two threads. If an object contains application-defined native methods, the object is not allowed to migrate, because native local state can not be migrated.

Aridor et al. have developed and executed a prototype on the Windows NT operating system, which can execute unmodified, multi-threaded Java applications.

6.4.2 JESSICA

The *JESSICA* system is an ongoing research project at the University of Hong Kong. It was started in 1996 [2] and first presented in [83, 84]. JESSICA 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* [85] page-based DSM, which was later replaced by JUMP [86]. JUMP is another page-based DSM, which allows the home of a memory page to migrate to another node. This 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; the home migrated. This approach is similar to our migrate-on-modify approach.

To prevent a node from reading an outdated page from a former home node, JUMP sends *migration notice* messages to each node in the system at the synchronization points. If the new home node modifies several pages, all *migration notice* messages can be consolidated into one.

Fang et al. [87, 88] developed a fine-grain, object-based *global object space* for JESSICA that includes a simple object home migration method. The GOS supports object pushing to allow prefetching. The reason for this development was the poor performance of the page-based DSM systems, e. g. due to the false sharing problem.

The GOS distinguishes between *node-local* objects and *distributed-shared* objects (DSO). To detect a DSO, the GOS uses a DSO detection mechanism. This mechanism examines the communication between nodes to detect object references that cross the node boundary. If a remote DSO is requested, the GOS applies a prefetching strategy, i. e. it pushes additional objects that the requested object references to the requesting node until the maximal message length is reached.

As in cJVM, each object has a dedicated home node, which holds the master object. This home node can change, corresponding to an object migration ("object home migration"). Similarly to cJVM, the GOS takes a conservative approach and only allows the migration of objects that have a single writer access. In this case, the remote node first caches the master object and after it modified the object, it announces itself to the former home node as the new home node of the object.

With this approach, the GOS avoids the need for remote reference maintenance, as we apply it in our system. If a third thread tries to access the migrated object on its former home node, a *home redirection message* is sent back to the requesting thread. Afterwards the requesting thread updates its information about the object's home node and sends its request to the correct new home node. In

[2] http://i.cs.hku.hk/ clwang/projects/JESSICA4.htm, last visited 2011-01-19

addition to the *home redirection message*, a *forwarding pointer* is left on the original home node. This is similar to our proxy approach.

JESSICA2 [89, 8, 90] 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 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.

Luo et al. [91] study the connectivity of objects and the traversal behavior over the access paths among objects. This work was done for the JESSICA3 project in order to find a suitable prefetching policy.

They propose a profiling strategy that classifies classes and fields into different types according to their access patterns. Based on this classification, they described and evaluated different prefetching approaches, such as depth-first (fetch objects recursively which are reachable along one reference to the next layer) or breadth-first (fetch all objects at the next object layer).

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

6.4.3 Java/DSM

Java/DSM [92] is another modified JVM. Similar to JESSICA, its heap is implemented on top of the *TreadMarks* [85] DSM. Java/DSM does not allow object migration and does not implement thread migration. Additionally, the thread location is not transparent.

The Java/DSM garbage collector uses two lists: the export list, which contains remote references to local objects, and the import list, which contains references to remote objects. These two lists are comparable to our *InRef List* and *GaoMap*. Nevertheless, the remote references in the export list are only used for garbage collection and not for object migration.

6.4.4 Hyperion

Hyperion [93], in contrast to JESSICA and Java/DSM, is a JVM that is built on top of an object-base DSM. Hyperion uses a JIT approach that first compiles Java to C code. Just before its execution, this C code is compiled to machine code.

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

The object table contains tuples that consist of a local object pointer and a remote object pointer. A nodes' own portion of this object table is used to create local objects, whereas the remote portions are used to cache remote objects.

If a thread accesses an index in the object table and finds that the local object pointer is invalid, it retrieves a cached copy from the remote node that is associated with the accessed portion of the object table. Afterwards, the node replaces the entry for the remote object pointer in the nodes object table with the pointer to the locally cached copy.

Hyperion does not allow object migrations. Instead, the node that created an object holds the master object all the time. If a remote node accesses this master object it is shipped as a copy of the object. If the node modifies this object copy, the copy is written back to the master node at Java synchronization points.

6.4.5 Jackal

Jackal [94] is a compiler-driven, fine-grained DSM system for Java. Jackal compiles Java code to native machine code. During this compilation, Jackal performs the object-graph aggregation, similar to an object pushing in the JESSICA GOS [87]. Similar to Hyperion, the global address space is divided into chunks, that are owned by different nodes in the system.

Ungar et al. [95] describe a distributed Smalltalk virtual machine. This VM is based on the Squeak Smalltalk system [96, 97] and was enhanced to be a distributed VM that runs on the TILE64 many core processor from Tilera.

The VM distributes its object heap among the individual caches of 56 of the processor cores, such that each core has its own (part of the) heap. A shared address space allows any core to access any object in the heap. A global, centralized *object table* deals with the problem of changing memory addresses of objects, whenever they move from one cache to another. To evaluate their design, they executed a single thread which moved from node to node, but they did not measure parallel and/or multi-threaded workloads.

6.4.6 Barrelfish

Barrelfish [98] is a first implementation of the new multikernel model for operating systems which support multicore processor systems. For inter-core communication, Barrelfish uses user-level Remote Procedure Calls (RPCs), which read/write into dedicated shared memory locations in the reader's/writer's caches (cache lines).

Instead of communicating via caches and shared memory, Barrelfish uses message passing which the authors claim to be a much better paradigm for multi-core systems. Therefore, their approach is a shared nothing model. They show experimental results where the costs of updating shared memory on a multi-core machine is much more expensive than updating the local cache of a server node via message passing.

6.4.7 CellVM

The *CellVM* [99] is a DJVM that extends the JamVM. It is specially designed to run on the IBM Cell processor. The CellVM supports two modes of operation: a pure interpretative approach and a dynamic Java bytecode to native code compiler, which translates Java bytecode to native vector code on-the-fly. The interpreter uses the direct-threaded interpreter approach (instead of the switched interpreter approach), because the SPEs do not offer hardware branch prediction.

The VM comes in two flavors: the ShellVM, which is executed on the Power PC core (PPE) and the CoreVM, which is run on the SPEs. The ShellVM maintains the global system resources, while the CoreVM operates on its own local storage. The Java heap that is shared by all VMs is located in the main memory. 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. 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. E. g. for complex memory operations, the execution is transfered 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.

6.4.8 JavaSplit

JavaSplit [100, 101] is a distributed Java runtime system 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 system that was inspired by the HLRC protocol [102]. Objects can be *local* or *shared*, but only the latter ones are managed by JavaSplit. 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 as a shared object.

Like HLRC, JavaSplit is *home-based* in the way that each object has a home node that manages the master copy of the object. All threads that access the object create local copies of the object. At some synchronization point, the changes the thread has made are written back to the master copy.

Nodes that want to join JavaSplit simply use a Java-enabled browser to access a website that contains the Java applet that executes the code of a worker node.

JavaSplit takes a given parallel application written in standard Java. The bytecode of the application is automatically rewritten to incorporate all the needed distributed runtime logic. The resulting rewritten classes of the application are transparently distributed onto the participating worker nodes. These nodes then execute those threads of the application that were assigned to them on their standard JVMs.

The bytecode rewriter intercepts the bytecodes that start the execution of a new thread. It is replaced by a handler that transfers the thread to one of the nodes in the system. Synchronization

mechanisms such as monitors or synchronized methods are substituted by a appropriate synchronization handler.

To preserve memory consistency, each load and store operations first has to perform an access check. If this check fails, a newer version of the object is fetched from another node.

6.4.9 JavaParty

JavaParty [103] introduces transparent remote objects for Java. JavaParty is built on top of RMI. If modifies the Java language by introducing a new class modifier *remote*. A pre-processing phase transforms JavaParty code into regular Java code with RMI hooks. This code is then compiled with the RMI compiler.

The JavaParty runtime system is built around a central *RuntimeManager*. Each node runs a *LocalJP* JavaParty instance that must register itself at the manager. The manager knows all LocalJPs and the location of all class objects, e. g. the host that initialized the static parts of a class. This information is replicated to all LocalJPs to reduce the management overhead.

JavaParty allows object migration,; it uses proxies that are left behind after a migration. If a method call arrives at a proxy, the new location is sent back to the caller. This is similar to our reactive and iterative update approach (c. f. sec. 3.3.1).

Haumacher et al. [104] describe how Java's remote method invocation (RMI) can be used to transparently create and control distributed threads in their JavaParty system.

6.4.10 Commercial Solutions

The *Terracotta system* [105, 106] allows a Java application to run on multiple distributed Java virtual machines (JVMs). The JVMs run on multiple machines that are connected via a network-attached Java heap. This Java heap consists of an underlying server array, to which all JVMs connect. Hot standby servers provide fault tolerance. They take over when an active server fails.

In this system, the user needs to identify all objects that should be reachable while they reside in the network-attached heap. The bytecode of the classes that become shared objects is instrumented to allow object maintenance on the global heap. Terracotta does not allow data and threads to migrate between the machines in the cluster, but objects may be moved to the server, if the local Java heap has no memory left.

Azul Systems developed the *Zing* Java virtual machine. 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)*, which runs on an X86 hypervisor. Azul states that the ZVA is a better execution stack to execute the Java application. The Zing Resource Controller (ZRC) is a centralized management component that dynamically growth or shrinks the memory footprint of a Java application on demand.

Azul did not publish any scientific papers, but information about their VM can be found on their website [107].

References

[1] G. F. Pfister, *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing (2nd ed.)*. Prentice-Hall, Inc., 1998.

[2] R. Buyya, T. Cortes, and H. Jin, "Single System Image," *International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001.

[3] J. Gosling, "Java Intermediate Bytecodes," in *Proc. of the 22nd ACM Symposium on Principles of Programming Languages Papers (POPL'95)*, (San Francisco, California, USA), pp. 111–118, Jan. 23 – 25, 1995.

[4] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, "Java programming for high-performance numerical computing," *IBM Systems Journal*, vol. 39, no. 1, pp. 21–56, 2000.

[5] G. L. Taboada, J. Touri no, and R. Doallo, "Java for high performance computing: assessment of current research and practice," in *Proc. of the 7th Int. Conf. on Principles and Practice of Programming in Java (PPPJ'09)*, (Calgary, Alberta, Canada), pp. 30–39, Aug. 27 – 28, 2009.

[6] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: a high-performance Java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[7] Y. Aridor, M. Factor, and A. Teperman, "cJVM: A Single System Image of a JVM on a Cluster," in *Proc. of the Int. Conf. on Parallel Processing (ICPP'99)*, (Wakamatsu, Japan), pp. 4–11, Sept. 21 – 14, 1999.

[8] W. Zhu, W. Fang, C. Wang, and F. Lau, "A new transparent java thread migration system using just-in-time recompilation," in *Proc. of the 16th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'04)*, (Cambridge, Massachusetts, USA), pp. 766–771, Nov. 09 – 11, 2004.

[9] S.-A. Posselt, "Design of a reliable, fully decentralized software transactional memory protocol," diplomarbeit, Technische Universität München, 2010.

[10] B. Saballus, J. Eickhold, and T. Fuhrmann, "Global accessible objects (gaos) in the ambicomp distributed java virtual machine," in *Proc. of the 2nd Int. Conf. on Sensor Technologies and Applications (SENSORCOMM'08)*, (Cap Esterel, France), Aug. 25 – 31, 2008.

[11] A. Bieniusa and T. Fuhrmann, "Consistency in Hindsight, A Fully Decentralized STM Algorithm," in *Proc. of the IEEE Int. Symposium on Parallel Distributed Processing (IPDPS'10)*, (Atlanta, Georgia, USA), pp. 1–12, Apr. 19 – 23, 2010.

[12] T. Fuhrmann, "Scalable routing for networked sensors and actuators," in *Proc. of the 2nd Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'05)*, (Santa Clara, California, USA), pp. 240–251, Sept. 26 – 29, 2005.

[13] B. Williams and T. Camp, "Comparison of broadcasting techniques for mobile ad hoc networks," in *Proc. of the 3rd ACM Int. Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'02)*, (Lausanne, Switzerland), pp. 194–205, June 09 – 11, 2002.

[14] Y. K. Dalal and R. M. Metcalfe, "Reverse path forwarding of broadcast packets," *Communications of the ACM*, vol. 21, no. 12, pp. 1040–1048, 1978.

[15] C. Bolton and G. Lowe, "Analyses of the reverse path forwarding routing algorithm," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'04)*, (Florence, Italy), pp. 485–494, June 28 – July 1, 2004.

[16] B. Bellur and R. Ogier, "A reliable, efficient topology broadcast protocol for dynamic networks," in *Proc. of the 18th IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)*, vol. 1, (New York, New York, USA), pp. 178–186, Mar. 21 – 25, 1999.

[17] J. L. Träff and A. Ripke, "An Optimal Broadcast Algorithm Adapted to SMP Clusters," in *Proc. of the 12th European PVM/MPI Users' Group Meeting (EuroMPI'10)*, (Sorrento, Italy), pp. 48–56, Sept. 18 – 21, 2005.

[18] J. L. Träff and A. Ripke, "Optimal broadcast for fully connected processor-node networks," *Journal of Parallel and Distributed Computing*, vol. 68, no. 7, pp. 887–901, 2008.

[19] S. Bhattacharya, C.-L. Fok, C. Lu, and G.-C. Roman, "Design and Implementation of a Flexible Location Directory Service for Tiered Sensor Networks," in *Proc. of the 3rd IEEE Int. Conf. Distributed Computing in Sensor Systems (DCOSS'07)*, (Santa Fe, New Mexico, USA), pp. 158–173, June 18 – 20, 2007.

[20] P. Mockapetris and K. J. Dunlap, "Development of the Domain Name System," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 123–133, 1988.

[21] Object Management Group, "The Common Object Request Broker: Architecture and Specification, OMA: formal/99-10-07," Object Management Group, 1999.

[22] Object Management Group, "A Discussion of the Object Management Architecture," Object Management Group, 1997.

[23] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM*, vol. 41, no. 10, pp. 62–71, 1998.

[24] M. Henning, "The rise and fall of CORBA," *Communications of the ACM*, vol. 51, no. 8, pp. 52–57, 2008.

[25] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," in *Proc. of the 2nd USENIX Conf. on Object-Oriented Technologies (COOTS'96)*, (Toronto, Ontario, Canada), pp. 219–232, June 17 – 21, 1996.

[26] C. Munoz and J. Zalewski, "Archictecture and Performance of Java-Based DistributedObject Models: CORBA vs RMI," *Real-Time Systems*, vol. 21, no. 1-2, pp. 43–75, 2001.

[27] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*. New York: Wiley, 2. ed., 1998.

[28] P. V. Mockapetris, "RFC 1034 - Domain names - Concepts and Facilities," Nov. 1987.

[29] P. V. Mockapetris, "RFC 1035 - Domain names - Implementation and Specification," Nov. 1987.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, (San Diego, California, USA), pp. 161–172, Aug. 27 – 31, 2001.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "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), pp. 149–160, Aug. 27 – 31, 2001.

[32] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware'01)*, (Heidelberg, Germany), pp. 329–350, Nov. 12 – 16, 2001.

[33] L. Moreau, V. Tan, and N. Gibbins, "Transparent Migration Of Mobile Agents," *IEE Seminar Digests*, vol. 150, no. 2, 2001.

[34] L. Moreau and D. Ribbens, "Mobile objects in Java," *Scientific Programming*, vol. 10, no. 1, pp. 91–100, 2002.

[35] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proc. of the 15th European Simulation Multiconference: Modelling and Simulation (ESM'01)*, (Prague, Czech Republic), pp. 319–324, June 06 – 09, 2001.

[36] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. of the 1st Int. Conf. on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (SIMUTools'08)*, (Marseille, France), pp. 1–10, Mar. 03 – 07, 2008.

[37] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. of the 19th Annual Symposium on Foundations of Computer Science (SFCS'78)*, vol. 0, (Washington, District of Columbia, USA), pp. 8–21, Oct. 16 – 18, 1978.

[38] T. Thorn, "Programming Languages for Mobile Code," *ACM Computing Surveys*, vol. 29, no. 3, pp. 213–239, 1997.

[39] J. Gosling and H. McGilton, "The Java Language Environment: A White Paper," *Sun Microsystems*, 1996.

[40] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and Language-Independent Mobile Programs," in *Proc. of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*, (Philadelphia, Pennsylvania, USA), pp. 127–136, May 21 – 24, 1996.

[41] H. Bal, M. Kaashoek, and A. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190–205, 1992.

[42] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," in *Proc. of the 11th ACM Symposium on Operating System Principles (SOSP'87)*, (Austin, Texas, USA), pp. 109–133, Nov. 08 – 11, 1987.

[43] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers," in *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, (Copper Mountain, Colorado, USA), pp. 68–77, Dec. 03 – 06, 1995.

[44] F. Knabe, "An Overview of Mobile Agent Programming," in *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages (LOMAPS'96)*, (Stockholm, Sweden), pp. 100–115, June 24 – 26, 1996.

[45] R. J. Fowler, "The complexity of using forwarding addresses for decentralized object finding," in *Proc. of the 5th Annual ACM Symposium on Principles of distributed computing (PODC'86)*, (Calgary, Alberta, Canada), pp. 108–120, Aug. 11 – 13, 1986.

[46] S. Alouf, F. Huet, and P. Nain, "Forwarders vs. centralized server: an evaluation of two approaches for locating mobile agents," *Performance Evaluation*, vol. 49, no. 1-4, pp. 299–319, 2002.

[47] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, "Programming, Composing, Deploying for the Grid," in *Grid Computing: Software Environments and Tools*, pp. 205–229, 2006.

[48] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers, "References to Remote Mobile Objects in Thor," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 115–126, 1993.

[49] A. Vijay Srinivas and D. Janakiram, "Scaling a Shared Object Space to the Internet: Case Study of Virat," *Journal of Object Technology*, vol. 5, pp. 75–95, 2006.

[50] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," in *Proc. of the 21st Annual Int. Symposium on Computer Architecture (ISCA'94)*, (Chicago, Illinois, USA), pp. 302–313, Apr. 1994.

[51] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[52] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer*, vol. 19, no. 8, pp. 26–34, 1986.

[53] G. P. Picco, A. L. Murphy, and G.-C. Roman, "LIME: Linda meets mobility," in *Proc. of the 21st Int. Conf. on Software Engineering (ICSE'99)*, (Los Angeles, CA, USA), pp. 368–377, May 16 – 22, 1999.

[54] T. Fahringer, "JavaSymphony: a System for Development of LocalityOriented Distributed and Parallel Java Applications," in *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER'00)*, (Chemnitz, Germany), pp. 145–152, Nov. 28 – Dec. 01, 2000.

[55] T. Fahringer and A. Jugravu, "JavaSymphony: a new programming paradigm to control and synchronize locality, parallelism and load balancing for parallel and distributed computing," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 7-8, pp. 1005–1025, 2005.

[56] M. Aleem, R. Prodan, and T. Fahringer, "JavaSymphony: A Programming and Execution Environment for Parallel and Distributed Many-Core Architectures," in *Proc. of the 16th Int. European Conf. on Parallel and Distributed Computing (Euro-Par'10)*, (Ischia, Napels, Italy), pp. 139–150, Aug. 21 – Sept. 03, 2010.

[57] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[58] F. B. Engelhardtsen and T. Gagnes, "Using JavaSpaces to create adaptive distributed systems," in *Proc. of IFIP WG6.7 Workshop and 8th EUNICE Summer School on Adaptable Networks and Teleservices (EUNICE'02)*, (Trondheim, Norway), pp. 125–130, Sept. 02 – 04, 2002.

[59] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, "Javelin: Internet-based parallel computing using Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1139–1160, 1997.

[60] P. Ciancarini and D. Rossi, "Jada: Coordination and communication for Java agents," in *Selected Presentations and Invited Papers 2nd Int. Workshop on Mobile Object Systems (MOS'96) - Towards the Programmable Internet*, pp. 213–226, July 08 – 09, 1997.

[61] M. Herlihy, "The Aleph Toolkit: Support for Scalable Distributed Shared Objects," in *Proc. of the 3rd Int. Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC'99)*, pp. 137–149, Jan. 8, 1999.

[62] M. Herlihy and M. P. Warres, "A Tale of Two Directories: Implementing Distributed Shared Objects in Java," in *Proc. of the ACM Conference on Java Grande (JAVA'99)*, (San Francisco, California, USA), pp. 99–108, June 12 – 14, 1999.

[63] M. J. Demmer and M. Herlihy, "The Arrow Distributed Directory Protocol," in *Proc. of the 12th Int. Symposium on Distributed Computing (DISC'98)*, (Andros, Greece), pp. 119–133, Sept. 24 – 26, 1998.

[64] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast Address Translation Techniques for Distributed Shared Memory Compilers," in *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'05)*, (Denver, Colorado, USA), Apr. 03 – 08, 2005.

[65] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing," in *Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, (San Diego, California, USA), pp. 519–538, Oct. 16 – 20, 2005.

[66] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," Tech. Rep. UCB/CSD-92-675, University of California, Berkeley, Berkeley, California, USA, Mar. 1992.

[67] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[68] W. W. Carlson, J. M. Draper, and D. E. Culler, "Introduction to UPC and Language Specification," Tech. Rep. CCS-TR-99-157, George Washington University, 1993.

[69] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared Memory Programming for Large Scale Machines," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06)*, (Ottawa, Canada), pp. 108–117, June 10 – 16, 2006.

[70] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, *et al.*, "An overview of the bluegene/l supercomputer," in *Proc. of the ACM/IEEE Conf. on Supercomputing (SC'02)*, (Baltimore, Maryland, USA), pp. 60–82, Nov. 16 – 22, 2002.

[71] J. Dongarra and P. Luszczek, "Introduction to the HPC Challenge Benchmark Suite," Tech. Rep. ICL Technical Report, ICL-UT-05-01 (also appears as CS Dept. of Technical Report UT-CS-05-544), Innovative Computing Laboratory, University of Tennessee, 2005.

[72] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, *et al.*, "The NAS Parallel Benchmarks," *Int. Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 66–73, 1991.

[73] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks - summary and preliminary results," in *Proc. of the ACM/IEEE Conf. on Supercomputing (SC'91)*, (Albuquerque, New Mexico, USA), pp. 158–165, Nov. 18 – 22, 1991.

[74] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Tech. Rep. NAS-95-020, NASA Ames Research Center, Moffett Field, California, USA, 1995.

[75] C. Barton, C. Cascaval, and J. N. Amaral, "A Characterization of Shared Data Access Patterns in UPC Programs," in *Proc. of the 19th Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, (New Orleans, Louisiana, USA), pp. 111–125, Nov. 02 – 04, 2006.

[76] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, X. Zhou, *et al.*, "Ct: A Flexible Parallel Programming Model for Tera-scale Architectures," Tech. Rep. http://techresearch.intel.com/spaw2/uploads/files/Whitepaper-Ct.pdf, Intel Research, 2007.

[77] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[78] D. Callahan, B. Chamberlain, and H. Zima, "The cascade high productivity language," in *Proc. of the 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, (Santa Fe, New Mexico, USA), pp. 52–60, Apr. 26, 2004.

[79] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer, "Design and implementation of a single system image operating system for ad hoc networks," in *Proc. of the 3rd Int. Conf. on Mobile Systems, Applications, and Services (MobiSys'05)*, (Seattle, Washington, USA), pp. 149–162, June 06 – 08, 2005.

[80] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, "GLUnix: a global layer unix for a network of workstations," *Software: Practice and Experience*, vol. 28, no. 9, pp. 929–961, 1998.

[81] R. Buyya, "PARMON: a portable and scalable monitoring system for clusters," *Software: Practice and Experience*, vol. 30, no. 7, pp. 723–739, 2000.

[82] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster, "A high performance cluster JVM presenting a pure single system image," in *Proc. of the ACM Java Grande Conference (JAVA'00)*, (San Francisco, California, USA), pp. 168–177, June 03 – 05, 2000.

[83] M. Ma, C.-L. Wang, and F. Lau, "Delta Execution: A preemptive Java Thread Migration Mechanism," in *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, (Las Vegas, Nevada, USA), pp. 518–524, June 28 – July 01, 1999.

[84] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, "JESSICA: Java-Enabled Single-System-Image Computing Architecture," *Journal of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1194–1222, 2000.

[85] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[86] B. W.-L. Cheung, C.-L. Wang, and K. Hwang, "A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations," in *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, (Las Vegas, Nevada, USA), pp. 821–827, June 28 – July 01, 1999.

[87] W. Fang, C.-L. Wang, and F. C. M. Lau, "Efficient Global Object Space Support for Distributed JVM on Cluster," in *Proc. of the 31st Int. Conf. on Parallel Processing (ICPP'02)*, (Vancouver, British Columbia, Canada), pp. 371–378, Aug. 20 – 23, 2002.

[88] W. Fang, C.-L. Wang, and F. C. M. Lau, "On the design of global object space for efficient multi-threading Java computing on clusters," *Parallel Computing - Special issue: Parallel and distributed scientific and engineering computing*, vol. 29, no. 11-12, pp. 1563–1587, 2003.

[89] W. Zhu, C.-L. Wang, and F. Lau, "JESSICA2: a distributed Java Virtual Machine with transparent thread migration support," in *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER'02)*, (Chicago, Illinois, USA), pp. 381–388, Sept. 23 – 26, 2002.

[90] W. Zhu, "Distributed Java Virtual Machine with Thread Migration," *Ph.D. Thesis*, March, 31 2005.

[91] Y. Luo, K. T. Lam, and C.-L. Wang, "Path-Analytic Distributed Object Prefetching," in *Proc. of the 10th Int. Symposium on Pervasive Systems, Algorithms and Networks (ISPAN'09)*, (Kaohsiung, Taiwan), Dec. 14 – 16, 2009.

[92] W. Yu and A. Cox, "Java/DSM: A Platform for Heterogeneous Computing," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1213–1224, 1997.

[93] M. W. MacBeth, K. A. McGuigan, and P. 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), pp. 40–54, Oct. 16 – 20, 1998.

[94] R. Veldema, R. F. H. Hofman, R. Bhoedjang, C. J. H. Jacobs, and H. E. Bal, "Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems," in *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, (Snowbird, Utah, USA), pp. 83–92, June 18 – 20, 2001.

[95] D. Ungar and S. S. Adams, "Hosting an object heap on manycore hardware: an exploration," in *Proc. of the 5th Symposium on Dynamic Languages (DLS'09)*, (Orlando, Florida, USA), pp. 99–110, Oct. 26, 2009.

[96] Squeak.org, "http://squeak.org/," 2010.

[97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: the story of Squeak, a practical Smalltalk written in itself," in *Proc. of the 12th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, (Atlanta, Georgia, USA), pp. 318–326, Oct. 05 – 09, 1997.

[98] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The Multikernel: A new OS Architecture for Scalable Multicore Systems," in *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, (Big Sky, Montana, USA), Oct. 11 – 14, 2009.

[99] A. Noll, A. Gal, and M. Franz, "CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor," in *Workshop on Cell Systems and Applications (WCSA'08)*, (Beijing, China), June 21 – 22, 2008.

[100] M. Factor, A. Schuster, and K. Shagin, "JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations," in *Proc. of the IEEE Int. Conf. on Cluster Computing (CLUSTER'03)*, (Kowloon, Hong Kong, China), pp. 110–117, Dec. 01 – 04, 2003.

[101] M. Factor, A. Schuster, and K. Shagin, "A distributed runtime for java: Yesterday and today," in *Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, (Santa Fe, New Mexico, USA), pp. 159–165, Apr. 26 – 30, 2004.

[102] Y. Zhou, L. Iftode, and K. Li, "Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems," in *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, (Seattle, Washington, USA), pp. 75–88, Oct. 28 – 31, 1996.

[103] M. Philippsen and M. Zenger, "JavaParty transparent remote objects in Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, 1997.

[104] B. Haumacher, T. Moschny, J. Reuter, and W. Tichy, "Transparent distributed threads for Java," in *Proc. of the 17th Int. Parallel and Distributed Processing Symposium (IPDPS'03)*, (Nice, France), Apr. 22 – 26, 2003.

[105] Terracotta Inc., "A Technical Introduction to Terracotta," , www.terracottatech.org, Jun. 2008.

[106] Terracotta Inc., *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, 2008.

[107] Azul Systems, Inc., "Zing java virtual machine," , www.azulsystems.com, Jan. 2011.