M A G I S T E R A R B E I T

M A S T E R ' S T H E S I S

Implementation and Evaluation of Replication Protocols using the Neko Framework

Ausgeführt am

Institut für Informationssysteme, Abteilung für Verteilte Systeme,

Technische Universität Wien

unter der Anleitung von

Dr. Karl M. Göschka und

Dipl.Ing. Johannes Osrael

durch

Dipl.Ing. Matthias Gladt

Josefstädter Straße 16/2/36, A-1080 Wien

Wien, am 4. April 2006 _____

Abstract

In distributed computing systems users and resources are intended to be connected in a transparent, open and scalable way in order to interchange information. If they are set up in a reasonable way a major advantage of distributed systems is that compared to stand-alone systems they can increase availability by replicating data on different nodes. Consistency on the other hand might drop due to network or node failures.

A strict replication policy can ensure that any information published on any node of the system is consistent. A side effect would be, though, that availability suffers. Also the other extreme is possible, where availability is ensured at any time without answering for consistency.

The thesis at hand is focused on the implementation of the Primary-per-Partition replication protocol which provides a way to achieve a *configurable* trade-off between availability and consistency in distributed systems. In order to be able to simulate such a distributed system the implementation is integrated with the Neko framework, a Java framework that offers an easy to use platform for simulated and real distributed systems in equal measure. Neko is based on a protocol-stack like architecture which allows to easily assign different tasks to the individual components of a client application. The simulation of a distributed system is achieved by instantiating the implementation as often as there are nodes. Neko supports the execution of the same code optionally in a simulated or in a real environment. The whole setup information is stored in a configuration file which can equally be used for storing user defined parameters.

The implementation of the Primary-per-Partition protocol turned out to work fine. The Primary-per-Partition protocol maintains integrity by assuring that all user-defined constraints are met when the system is normal mode, that is, no node or link failures are present. In degraded mode, that is, node or link failures have occurred consistency can be traded for availability by accepting the unsuccessful evaluation of certain constraints. When the system re-enters normal mode constraint consistency has to be restored.

Zusammenfassung

In verteilten Systemen werden Anwender bzw. Hardwarekomponenten so miteinander verbunden, dass das Austauschen von Informationen in einem Maße möglich ist, dass eine dafür vorgesehenen Applikation problemlos funktioniert. Wenn verteilte Systeme richtig konfiguriert werden ist einer ihrer größten Vorteile im Vergleich zu einzelnen Komponenten, dass sie die Verfügbarkeit von Daten erhöhen können indem sie Kopien auf mehreren Knoten erzeugen. Die Konsistenz kann jedoch aufgrund von Netzwerkfehlern oder Fehlern einzelner Knoten abnehmen.

Genaue Replikationsvorgaben können gewährleisten, dass zu jedem Zeitpunkt Konsistenz garantiert werden kann. Ein Nebeneffekt davon wäre allerdings, dass die Verfügbarkeit leiden würde. Ebenso ist es möglich die Verfügbarkeit ohne Rücksicht auf Konsistenz zu garantieren.

Die vorliegende Arbeit hat die Implementierung des Primary-per-Partition Protokolls zum Gegenstand, das es ermöglicht, ein verteiltes System so zu konfigurieren, dass die Konsistenz zu Lasten der Verfügbarkeit erhöht wird oder umgekehrt. Um ein verteiltes System simulieren zu können, wird die Implementierung in das Neko Framework eingebettet, eine Java Plattform, die leicht konfiguriert werden kann und in einer simulierten Umgebung ebenso verwendet werden kann wie in einer echten. Neko Anwendungen implementieren einen Protokoll Stack, d. h. dass einzelnen Komponenten unterschiedliche Aufgaben zugewiesen werden können. Im simulierten Modus werden Knoten erzeugt indem mehrere Instanzen der einzelnen Klassen erzeugt werden. Es kann sowohl in einer simulierten als auch in einer echten Umgebung ein und derselbe Code ausgeführt werden.

Die Implementierung des Primary-per-Partition Protokolls stellt Integrität sicher indem bestimmte, im Wesentlichen frei wählbare, Randbedingungen erfüllt werden. Im so genannten „degraded“ Modus, in dem Knoten- oder Netzwerkfehler aufgetreten sind, müssen einige dieser Randbedingungen nicht jederzeit erfüllt werden um eine höhere Verfügbarkeit zu erreichen. Sobald das System wieder repariert ist muss die Integrität wieder hergestellt werden.

Acknowledgements

I would like to thank my advisor Johannes Osrael and my supervisor Karl M. Goeschka for supporting me as efficiently as they did during the whole work.

Table Of Contents

Abstract **1**

Zusammenfassung **1**

Acknowledgements **1**

1 Problem Description **1**

1.1 Introduction **1**

1.2 Motivation and Problem Definition **2**

1.3 Organization of this thesis **3**

2 Replication and Consistency **4**

2.1 Introduction **4**

2.2 Consistency Models **5**

2.2.1 Strict Consistency 5

2.2.2 Sequential Consistency 6

2.2.3 Linearizability 6

2.2.4 Other Models 7

2.3 Consistency Protocols **7**

2.3.1 Primary-Backup Replication 8

2.3.2 Active Replication 9

3 The Primary-per-Partition Protocol **11**

3.1 Introduction **11**

3.2 System Model **11**

3.2.1 Synchronous vs. Asynchronous Update Propagation 12

3.2.2 State Transfer vs. Operation Transfer 12

3.3 Overview of the Primary-per-Partition Protocol **13**

3.4 Normal Mode **14**

3.4.1 Read Operation 14

3.4.2 Write Operation 14
3.5 Degraded Mode 15
3.5.1 Read Operation 15
3.5.2 Write Operation 16
3.5.3 Constraint Evaluation 17
3.5.4 Reconciliation Phase 17
4 The Neko Framework 21
4.1 Introduction 21
4.2 Architecture 22
4.3 Important Members of the Neko Framework 22
4.3.1 Layers 22
4.3.2 NekoProcesses 23
4.3.3 NekoMessages 24
4.3.4 Configurations 24
4.4 Startup Process 24
5 Implementation of the Primary-per-Partition Protocol 27
5.1 Introduction 27
5.2 Architecture 28
5.3 Objects and Replicas 32
5.4 Replication Manager 33
5.5 Invocation Service 36
5.6 Constraint Consistency Manager 38
5.7 Request Handler 41
5.7.1 Callback Handler vs. Threads: 41
5.7.2 Request handling step by step: 44
5.7.3 Nested Invocations 48
5.8 Reconciliation Manager 49

5.9 Node Observer 52

5.10 The Client 52

5.11 Simulation Manager 53

5.12 Group Membership Service 54

5.12.1 Partition Split 54

5.12.2 Reconciliation 56

5.13 UML Chart 60

6 Conclusion 62

6.1 The Primary-per-Partition protocol 62

6.2 Differences between the Implementation and the Specification 63

6.3 Neko 64

7 Future Work 67

7.1 Group Communication 67

7.2 Integration with an Application 68

7.3 Further Simulations 69

8 References 71

8.1 Literature 71

8.2 Figures 72

8.3 Tables 73

1 Problem Description

1.1 Introduction

Distributed computing studies the coordinated use of physically distributed computers. First concepts of computer networks and research date back to the early 1960s when J.C.R Licklider wrote memos about his “Intergalatic Network” where everyone on the globe is interconnected and can access programs and data at any site from anywhere. The design of the most well known distributed system, the World Wide Web, has its roots in 1989 when Tim Berners-Lee addressed the issue of the constant change in the currency of information and the turn-over of people on projects. Instead of a hierarchical or keyword organization, Berners-Lee proposed a hypertext system that would run across the Internet on different operating systems. The World Wide Web was invented. From then on the enhancements of distributed computer technologies are well known by most people since it was only some years later that the World Wide Web was available to the public of the western hemisphere.

Even though many experts think that big parts of distributed computer technology are little mature because it were rather the fastest technical solutions than the best thought-out ones that took hold on the market the development until today cannot be repented.

An interesting question that arises, though, is what the reason for this precipitous development was. Since this is not a topic of the thesis at hand only a couple of thoughts to this question shall be put down here.

- There was the need for a new communication system. For example the E-mail was not only a quickly tolerated but apparently a for a long time desired technology. Apart from the phone it therefore has become the number one means of communication over wide distances, in private spheres as well as in job-related ones. Similar thoughts also apply to the World Wide Web and its breakthrough idea, the hyperlink.
- The technological advantages of distributed computer systems compared to stand alone computers were more advantageous than the problems caused by callow technologies. For a long time supercomputers have been the first choice for computing large amounts of data. For some very special employments they still are but in most cases the processing power of a distributed computing network is bringing more speed at lower price.

So on the one hand distributed computer systems, including the World Wide Web, make up a patchwork of technologies, which often leaves users and developers exasperated because of its immatureness, on the other hand there is plenty of room for computer scientists to achieve improvements or to advance technologies. Some of this room shall be used by the author of the thesis at hand, which is now the point to start going into detail and to outline what the following pages are all about.

1.2 Motivation and Problem Definition

The demand for dependable allocation of data from distributed systems nowadays is often not only limited to safety critical applications. Unreliable data allocation would usually complicate the development of an application based on a distributed system because it would be up to the application to deal with potential node or network failures. So a major goal of most distributed systems is dependability. One way to improve dependability is replication of data on several nodes of a distributed system. But as much as replication contributes to dependability as profound is the difficulty that emerges out of a potential lack of consistency due to varying copies of one and the same information. So the price of a highly available system is a potential lack of consistency. On the other hand, as soon as consistency is enforced availability can not be guaranteed at all times anymore. The two states pictured so far – perfect consistency tied down to insufficient availability and perfect availability without any guarantees of consistency – bring up the question if also a tradeoff between consistency and availability would be possible.

Consistency can be divided into three types in a distributed and replicated system [D1.1.1]:

- Replica consistency: the consistency with respect to replicated data. Two copies of the same logical object are classified as consistent if they are identical. Some programming languages offer support for a check of replication consistency. E.g. in Java replica consistency can be checked by invoking the equals-method of one object with another object as an argument.
- Concurrency consistency: defines the correct processing of concurrent transactions.
- Constraint consistency: defines the correctness of the system state with respect to a set of data integrity rules.

Various investigations of the tradeoff between consistency, performance and availability in distributed computer systems have been made and are well understood [Goe05], [SK03], [HV00], [BBG+05]. However, none of these research projects explicitly considers the tradeoff between constraint consistency and availability.

A research project investigating the tradeoff between consistency and availability that does consider all three types of the above described consistency is the DeDiSys[1] project. DeDiSys is the acronym for “Dependable Distributed Systems”. The DeDiSys project is approved by the European Union and its main objective is the development of a concept for optimizing availability in distributed software systems by partially or temporarily relaxing integrity. But DeDiSys does not only focus on the tradeoff between consistency and availability but also on well-defined metrics and evaluation methods for such a system, proven by prototype implementations.

As already mentioned above replication is a common means of providing fault-tolerance and improving availability in distributed systems. Thus the specification of replication protocols belongs to the core elements of every dependable distributed system; DeDiSys is no exception here. The implementation and the evaluation of some parts of these replication protocols will be the central topic of the thesis at hand.

1.3 Organization of this thesis

The next chapter of this thesis will be about replication protocols in general. It will provide some categories which protocols can be divided into and explain the differences. Section 4 will present Neko, a Java framework on which all the implementations and evaluations within the scope of this thesis were based. The following chapters will be about the design and the implementation of the replication protocols investigated.

[1] http://www.dedisys.org

2 Replication and Consistency

2.1 Introduction

As already shortly outlined in the previous section, replication is a means to provide dependability. This is a fact that can most easily be pointed out with a couple of examples.

Many people use replication as a means to increase dependability without even knowing that they do so. The most trivial example is a file system that has been replicated, e.g. for backup reasons. When someone has important data on the hard disk of his computer and he stores a copy of it on a CD, so that in case the hard disk of the computer crashes he has still access to the data, he uses replication as a means to increase dependability.

Another example that may appear a little less obvious is settled in the field of mobile computing [CDK01]. Considered somebody uses a notebook on a train. There may be a wireless network that is interrupted or drops completely out or may be they even have no such capability. So the access to a centrally managed server storing all the data needed by a user may be blocked. In order to be able to continue to work in these circumstances, heavily used data can be copied onto the notebook. This is where replication is most advantageous. Thanks to replication the user can continue working on the copy on the notebook.

However, this is also where consistency has to be cut back. Supposed among the data there are the contents of a shared diary and the user consults or updates the diary, he risks reading data that someone else has altered in the meantime. For example when he makes an appointment in a slot that someone else has already occupied with another appointment. So disconnected working is only feasible if the user or the application can cope with stale data and with potential conflicts that arise when the user regains access to the server.

A third and last example shall be given for a well known application that uses replication as a means to enhance performance – the World Wide Web. Web browsers or proxy servers often cache copies of Web resources to avoid the latency of fetching resources from the originating server. Or the data forming the resources are replicated transparently between several servers in order to reduce the individual workload of each server.

As long as the data remain unchanged replication remains trivial because there cannot be any consistency problems. But what happens e.g. if a Web Site that is replicated between the above servers is modified? The moment when the updated site is copied on the first of the servers and the other servers still host the old version of the site users may retrieve two different versions of the same site, only one of which is up to date. The same applies for a Web Site stored in the cache of a Web browser or on a proxy server. After an update a user may get the old site without even being aware of it. That is because a common requirement when data are replicated is replication transparency which means that users should not have to know that multiple physical copies of data exist.

The fact that it is still possible to retrieve an old version of a site produces the question if this is acceptable. If so what is the acceptable percentage of sites that are out of date? The answers to these questions can only be given in consideration of the defined degree of consistency – the second requirement for replicated data that can vary in strength between applications. This concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects. So the degree of consistency depends on the specification of correctness for a certain application.

2.2 Consistency Models[2]

A consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. So a consistency model does not make a statement about how easily a model can be realized but it just defines exactly what kind of behavior is acceptable in the presence of conflicting operations.

The following sections discuss some well known types of consistency models.

2.2.1 Strict Consistency

The strict consistency model is the most stringent model. It is defined by the following condition:

[2] Based on [TS02]

Any read on a data item x returns a value corresponding to the result of the most recent write on x.

In a system with a single process aside from the server this condition can quite easily be fulfilled. Race conditions between two or more processes concerning read or write method calls cannot occur.

The reason that strict consistency can not be achieved in system with more than one process aside from the server is that it relies on absolute global time. To reach strict consistency timestamps that correspond to actual global time would have to be assigned to operations in a distributed system.

The fact that for strict consistency perfectly synchronized clocks would be needed, though, brings about that the implementation of strict consistency is impossible in a real-world distributed system.

2.2.2 Sequential Consistency

Sequential consistency is a slightly weaker model than strict consistency. For sequential consistency the following condition has to be satisfied:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

This means that when a request for a remote method call is sent off first it does not need to be processed first. The position e.g. in a FIFO list of requests does not have to undergo any restrictions aside from application-specific ones. So any valid interleaving of read and write operations is acceptable behavior but all processes have to see the same interleaving of operations.

The most essential difference between strict consistency and sequential consistency is that no reference to actual global time is needed.

2.2.3 Linearizability

Linearizability is an enhanced form of sequential consistency. In addition to sequential consistency each operation is assigned a time-stamp by a globally available clock with only finite precision. In order to reach linearizability the following condition has to be fulfilled additionally to the condition for sequential consistency:

In addition, if $ts_{OP1}(x) < ts_{OP2}(y)$, then operation OP1(x) should precede OP2(y) in this sequence.

So the main difference between sequential consistency and linearizability is that ordering according to a set of synchronized clocks is taken into account. In any case every linearized model is also sequentially consistent.

2.2.4 Other Models

There are many more consistency models whose level of significance is not that high for the thesis at hand because the implementation that is discussed in the following sections is not based on them.

The table below gives a compact overview of some consistency models that do not need explicit synchronization operations in order to provide a consistent data store.

Consistency	Description
Strict	Absolute time ordering of all shared accesses.
Linearizability	All processes see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were issued. Writes from different processes may not always be seen in that order

Table 1: Consistency models not using synchronization operations [TS02]

2.3 Consistency Protocols

A consistency protocol describes the implementation of a specific consistency model. Since the characteristics of some consistency models have been outlined in the previous section a consistency protocol that is based on one of the models is now going to be

regarded in detail. The most important consistency protocols are the ones that are based on models where operations are globally serialized.

2.3.1 Primary-Backup Replication[3]

The primary-backup replication model is also called passive replication model. The system model contains a primary replica manager and one or more secondary replica managers – “backups” or “slaves”.

The primary replica manager is the component that is in charge of any communication with clients in order to provide the service that the system is designed for. It processes any operation and sends copies of the updated data to the backups. If the primary fails one of the backups is promoted to act as the new primary.

Primary-backup replication models can be divided into synchronous and asynchronous models. In a synchronous Primary-backup replication model the primary returns the result of the request it has processed only after all secondaries have processed the request, too. That is the primary has to wait for a confirmation from every secondary before it delivers the result to the client.

In an asynchronous Primary-backup replication model the primary can deliver the result to the client as soon as it has processed the request without having to wait for a confirmation from the secondaries.

The way a synchronous system reacts to a request sent by a client can be summarized as follows:

- The primary receives a request containing a unique identifier.
- If the primary has already executed the request it re-sends the response else it executes the request now and stores the response.
- If the request has caused an update the primary forwards the updated state, the response and the unique identifier to the backups. The response and the unique identifier are needed in case a secondary has to replace the primary if the primary fails. In return the backups send an acknowledgement.
- In the last step the primary responds to the client, which forwards the response to the client. In an asynchronous system the response would already be sent right after the primary has executed the request.

[3] Based on [CDK01]

Since the primary sequences all operations upon a shared object obviously linearizability is implemented.

If the primary fails a secondary is to take over the primary's job at exactly the point where the primary dropped out. Two conditions have to be fulfilled in order to achieve a consistent changeover seamless enough to hide it from the client:

- The primary has to be replaced with a unique backup.
- The surviving replica managers have to agree on which operations have been performed until the point that the primary was replaced and which have not.

A major advantage of passive replication is that linearizability can easily be achieved with a sound implementation.

A disadvantage is, however, that passive replication has a relatively large overhead due to several rounds of communication in case the primary fails.

2.3.2 Active Replication[4]

In the active replication model replicas play equivalent roles and are organized as a group. A client multicasts its request to the group of replicas and each of the replicas belonging to the group serves the request independently but identically. Thus, replicas need to be deterministic. A crash of a replica need not have an impact on the reliability of the service since the remaining replicas continue to respond in a normal way and the client can collect and compare the replies it receives.

The sequence of events when a client launches a request for an operation to be performed can be summarized as follows:

- The client generates a unique identifier and attaches it to the request which is then sent to the group of replicas using a totally ordered, reliable multicast primitive.
- A group communication system has to make sure that all requests are delivered in the same total order to all replicas.
- The replicas process the request and generate a response to which the client's unique identifier is attached.
- The response is sent to the client by each replica of the group. The client collects one or more responses depending upon the failure assumptions and on the

[4] Based on [CDK01]

multicast algorithm. If, for example, only crash failures are tolerated and uniform and ordering agreements are met, the client passes the first response to arrive back to the client and discards the rest.

The system achieves sequential consistency.

3 The Primary-per-Partition Protocol

3.1 Introduction

As already outlined in the previous sections the tradeoff between consistency and availability comes with major advantages for applications that might profit from relaxing consistency in order to gain availability. Safety critical systems for example where availability at all times is the prime requirement may benefit from this tradeoff [SG03].

The Primary-per-Partition-Protocol is a replication protocol that reveals how the tradeoff can be achieved in practice.

3.2 System Model

This section describes specifications of the system model that the Primary-per-Partition-Protocol is designed for [D1.2.1]. The system is a distributed object system. It consists of a set of server nodes and client nodes. Any object that is located at a server node contains the information it is designed for. This information forms the state of the object. In addition an object provides a set of methods which can be invoked by other objects or rather the client. These methods serve to on the one hand access the information an object contains and on the other hand alter this very information within the scope of the system design.

A method call can cause other method calls thus nested invocations are supported. A transaction is preceded by a sub-transaction regardless of whether it is caused by a nested invocation or not. So every transaction can be canceled e.g. in case of an unsuccessful evaluation of constraints.

A node which is part of the system may fail as well as a link between two or more nodes. As nodes which are still operating can not distinguish between node or link failures, every failure is treated as partitioning.

The system model is a relaxed passive replication model. In passive replication requests are only processed by the primary copy. As far as there are any updates, they are propagated to the secondary copies when the primary is ready. For the DeDiSys model passive replication is relaxed so that read-only operations can also be served by secondary copies.

3.2.1 Synchronous vs. Asynchronous Update Propagation

As already outlined in section 2.3.1 update propagations from the primary to the secondary copies can be synchronous or asynchronous. In the DeDiSys model it depends on the state of the system whether synchronous or asynchronous update propagation is favored. In a healthy system synchronous update propagation is applied whereas in a partitioned system asynchronous update propagation is applied.

3.2.2 State Transfer vs. Operation Transfer

No precise definition of the terms can be found in literature. Nevertheless the distinction between these two ways of update propagation has to be specified at this point. In [D1.1.1] a specification is offered that appears most convenient for the upcoming sections.

Operation transfer is applied whenever the object that undergoes the update propagation has to read some information that it contains itself in order to be able to perform the update.

In contrast state transfer is applied if the message that causes the update contains all information needed to perform the update.

Some example operations are displayed in the figure below. The first one represents an operation transfer. In order to perform the update the old value of A has to be read. The message that triggers the update propagation contains only the difference between the old value and the new one. In the second and in the third example a state transfer is performed. Even if the new value of A has to be calculated in a separate operation like it is displayed in the third example state transfer is applied if the old value does not have to be read.

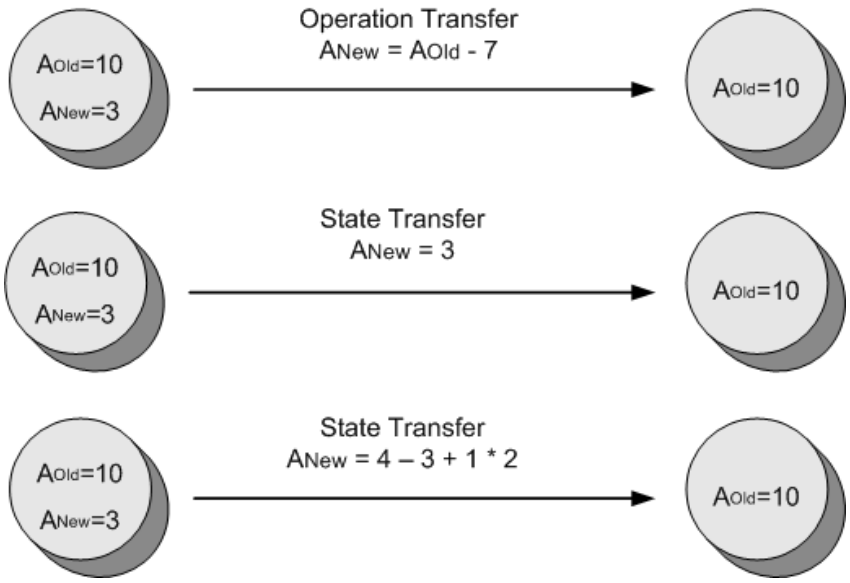

Figure 1: Operation transfer vs. state transfer

Since the Primary-per-Partition protocol does not contain any rule on whether to use state or operation transfer both ways of update propagation are allowed at the first glance. Problems with nested invocations may occur, though, which will be dealt with in the relevant section.

3.3 Overview of the Primary-per-Partition Protocol

One of the targets of the Primary-per-Partition protocol is to let all nodes continue to operate in case of a network failure. Since it is only the primary that can process write operations, the only way this can be achieved is to promote a secondary copy to the primary whenever the original primary is unreachable. Read operations, however, can still be processed by secondary copies, also in case of a network failure. So to promote a secondary copy to the primary is clearly a means to increase availability. But how can consistency be ensured?

Consistency in the Primary-per-Partition protocol is based on integrity constraints. Whenever an operation is going to be executed a constraint is evaluated before (*Pre-condition*). If the evaluation of the constraint does not return a positive result the operation that was going to be executed is canceled. Logically the same applies for constraints that are evaluated after the operation is executed (*Post-condition*). If they do not return a positive result the operation has to be undone.

This does hardly need any more considerations with a properly working system. But what happens if the system is divided into several partitions due to network failures and

the original primaries remain unreachable for some other nodes? Some constraints can only be evaluated on potentially stale replicas of a primary residing in a different partition. During reconciliation these constraints have to be evaluated again when missed updates are propagated and they may turn out to have been violated respectively.

This behavior may be acceptable for some operations but not for all. There can be operations in a system that must remain durable even in case of a partitioned system e.g. due to irreversible side effects. Therefore these operations have to undergo so called *critical constraints*. Any object that is involved in a critical constraint evaluation must be up-to-date. If not the constraint has to reject the operation that was going to follow.

Other than critical constraints *regular constraints* need not exclusively involve up-to-date objects but may also be performed on possibly stale replicas. At reconciliation time a regular constraint has to be re-evaluated if any object involved was updated during the reconciliation process. If a regular constraint is not met then a conflict resolution has to be carried out. This can either be done by the application or be part of the replication protocol.

3.4 Normal Mode

The normal mode denotes the mode when all nodes are online and operating. There are two operations that shall be regarded in particular, a read operation and a write operation.

3.4.1 Read Operation

A read operation consists of the following steps:

- Find a read copy. Since synchronous update propagation is applied other than for write operations any replica of an object can serve for a read operation. The choice of a read copy is not limited to the primary by the Primary-per-Partition protocol. Any other replica containing a copy of the desired object may serve as a read copy.
- Call the object operation.
- Return the result to the client.

3.4.2 Write Operation

A write operation in normal mode consists of the following steps:

- The client sends a request of a write operation to one of the nodes.
- Get a reference of the primary copy. The node that receives the request has to notify the primary of an incoming write operation.
- All pre-conditions have to be checked. If the check of any of these pre-conditions is not successful, the write operation has to be aborted.
- In the next step the actual write operation of the object is called. This operation may cause invocations to other objects.
- The write operation must be followed by the evaluation of all post-conditions. Again, if the evaluation of any of these post-conditions does not return a positive result, the write operation has to be aborted.
- After the constraints of the object that was called were evaluated successfully all objects have to be identified that may have been updated by a sub-transaction.
- All post-conditions have to be checked for each of the objects identified in the previous step.
- Start the commit protocol. If the commit protocol returns a positive result, the updates are propagated to the backup replicas.
- Since synchronous update propagation is applied a positive result is returned to the client only when the update propagation is successfully finished, if not the operation has to be rolled back, including all sub-transactions.

3.5 Degraded Mode

The degraded mode denotes the mode when a failure within the system has occurred. If a node appears not to respond anymore other nodes cannot determine whether this is because of a node or because of a link failure. Thus any failure is treated as network partitioning. A characteristic of the Primary-per-Partition protocol is that the system does not shut down in case of a failure. Instead in case of a node failure the remaining nodes continue to operate as if they were forming a new system. In case of a network failure the resulting partitions act as if each of them were forming a new system. What happens in detail will be explained in the following sections.

3.5.1 Read Operation

There is no difference between a read operation in degraded mode and a read operation in normal mode. It is important to keep in mind, though, that potentially stale objects

may serve the read request. Therefore the information gathered via a read operation may not be up-to-date.

3.5.2 Write Operation

Write operations in degraded mode differ clearly from write operations in normal mode.

- As in normal mode first the client sends a write request to one of the nodes.
- The node that the write request was sent to has to find the primary of the object whose write operation is going to be called.
- If the actual primary cannot be found because it resides in another partition, a secondary has to be chosen as a temporary primary. The choice of the new primary can be made in any arbitrary way, but it is important that all nodes of the same partition choose the same secondary as a new temporary primary.
- If any possibly stale objects are involved in a critical constraint the write operation has to be aborted.
- All pre-conditions are evaluated. If any pre-condition cannot be evaluated successfully the write operation is cancelled. The evaluation process is outlined in detail in one of the following sections.
- The current version of the object is saved together with the addresses of the nodes residing in the same partition. If any regular constraints are involved also the object state is saved. The object version is incremented every time a write operation of an object is called.
- The write operation must be followed by the evaluation of all post-conditions. Again, if the check of any of these post-conditions does not return a positive result, the write operation was not successful.
- After the constraints involving the object whose method was called were evaluated successfully all objects have to be identified that may have been updated by a sub-transaction.
- All post-conditions have to be evaluated for each of the objects identified in the previous step.
- Start the commit protocol. If the commit protocol returns a positive result, the updates are propagated to the backup replicas.

- When also the update propagation is successfully finished a positive result is returned to the client, if not the operation has to be rolled back, including all sub-transactions.

3.5.3 Constraint Evaluation

Whereas in normal mode the constraint evaluation process is rather simple and straight forward in degraded mode some aspects have to be considered which are tied to the fact that some primary copies may not be reachable.

- If a constraint is critical all objects involved are checked for staleness before the actual evaluation process is started. If a critical constraint does involve stale copies it is not evaluated and a negative result is returned. That way it is made sure that no operations have to be returned that are tied to critical constraints.
- If a constraint is critical and does not involve any stale objects it is evaluated just like in normal mode. If the evaluation is successful a positive result is returned.
- If any intra-object constraint is evaluated and turns out to be violated a negative result is returned.
- It is important to keep in mind that evaluations of inter-object constraints including potentially stale objects are not significant because updates that have occurred in an unreachable partition cannot be incorporated. Hence if a regular constraint involves any stale objects it is marked threatened. This is because it has to be re-evaluated at reconciliation time.

3.5.4 Reconciliation Phase

When two or more partitions re-join the reconciliation phase is started. An option is to let the system block any write operations from outside then. There are two ways of processing the reconciliation of two or more partitions. One is to let the application resolve potential conflicts. The other is to let the reconciliation process become part of the replication protocol. This is on the one hand a passable way to deal automatically with all arising conflicts. On the other hand the protocol is in control which will possibly not be acceptable for some applications.

Conflicts can be divided into write-write conflicts and read-write conflicts. A read-write conflict emerges when the application reads a value that is altered afterwards in the reconciliation phase. Read-write conflict-handling is left to the application.

Write-write conflicts can be dealt with either automatically by the protocol or by the application. If it is up to the application to resolve conflicts a handler must be registered which is called when a conflict has to be resolved. The handler can be an application specific implementation of the conflict resolution or it can ask for user interaction. If the replication protocol is in charge of conflict resolution the reconciliation process can be divided into several steps each of which will be regarded in detail. The first part of the reconciliation process is to restore replica consistency and the second part is to recover constraint consistency. As already mentioned both parts can be controlled by either the application or the replication protocol.

- First all nodes must update their visible-nodes-list. Whenever a new node joins a partition all other nodes of the partition have to update their visible-nodes-list and the other way round. The set of nodes visible for another node is called view.
- Primary copies multicast their state to all new nodes in their view. The state includes the full version list that was created by write operations in degraded mode.
- In return all primary copies receive messages with the states of all new primaries in their view. Secondary copies discard these messages.
- One of the primary copies of an object is the original primary copy whereas the other primary copies are temporary ones. The original primary is in charge of the further reconciliation process.
- The original primary compares its own version list with the ones it was sent by the temporary primaries. It scans all version lists for the maximum number of nodes in view. The version list with the maximum number of nodes in view is the one that is installed. If all nodes host the same set of replicas all version lists being installed originate from the same partition.
- If the local version list is the one with the maximum number of nodes in view it is sent to all temporary primary copies for installation else the remote version list is installed locally.
- When the original primary has finished updating its version list either by installing a remote version list or by imposing the local version list to the temporary primaries it sends its version list to the secondaries in view.

This is the point when replica consistency is achieved. The next step is to recover constraint consistency. This can be done by either calling a callback function that has

been registered before by the application or by extending the replication protocol so that it includes the recovery of constraint consistency. The latter will now be explained in detail.

As already outlined in the “3.5.2 Write Operation” section the object state and the current view is saved whenever a stale object is involved in a constraint evaluation process in degraded mode. This is needed for the re-evaluation of constraints during the reconciliation phase. Since the version lists of different objects may originate from various partitions regular constraints have to be re-evaluated after replica consistency is ensured.

- Therefore a regular constraint that was evaluated on possibly stale objects in degraded mode has to be re-evaluated on the latest versions of all involved objects after all nodes have installed the version list of the previous step.
- If the constraint cannot be evaluated successfully an object has to be chosen whose version is reverted to the one before the latest.
- The constraint is now evaluated again on the new current version of the object that was chosen in the previous step.
- If the constraint still cannot be evaluated successfully the object that was chosen before is again reverted by one version.
- The last two steps are repeated until either the above constraint is met or the first stored version of the object is reached. This is also the last known version to have met all constraints.
- If the constraint is still not met although the object has been reverted to its first known version another object has to be chosen. This is the next object whose version is to be reverted step by step until a version is found that meets the constraint or the first stored version is reached.
- The last step has to be repeated until the constraint is met.
- In the worst case all objects have to be reverted to their first known versions. These are also the last versions to have been generated in normal mode.
- The steps outlined so far to re-establish constraint consistency have to be repeated for all regular constraints.

Thus the process of re-establishing constraint consistency that has now been described in detail may in the worst case force all objects to be reverted to the state they were in before partitioning.

Since operations covered by critical constraints must not be called if any stale objects might be involved a rollback of these operations can never be necessary.

Cascading rollback as described above has a major disadvantage, though: Depending on the evaluation of the integrity constraints the last known point where constraint consistency can be guaranteed is where the system was in normal mode for the last time. Under unfavorable circumstances it may be necessary to perform a rollback as far as this point. Thus the potential consequences of the above described process of re-establishing constraint consistency may be incalculable if the duration of the degraded mode cannot be forecast.

An alternative way of re-establishing constraint consistency by requiring user interaction may be more promising. The interaction may result in updating the states by taking an application-specific compensating actions.

4 The Neko Framework[5]

4.1 Introduction

As already denoted in the prior chapters of the thesis on hand, the development of distributed algorithms is a highly complex task for several reasons. It is not only the algorithm as such that contributes to this complexity but also the fact that it is usually not possible to run the algorithm in a single environment during the development phase, the test stage and the go-live phase. Nevertheless none of these three phases can be skipped since during the test stage and the development phase the involved more than ever have to pay tribute to nowadays' need for the best performing algorithm available.

Since the environment for the go-live phase is usually not available for developing and testing reasons another environment has to be set up. This environment must meet the same criteria that affect the development and the performance of an algorithm as the environment for the go-live phase.

There are two ways of achieving an environment that is applicable for the development of a certain algorithm:

- One is to let constraints that cannot be provided by the environment used for developing reasons become part of the algorithm. This might cause major problems because the development of an environment can become more time consuming than the development of the algorithm itself.
- The other is to go for the environment without adapting the algorithm or any of its direct constraints. The setup process of the environment can be tiresome, though, because the composition of a complex distributed system can be even more complex than the development of the algorithm itself.

This is where Neko is at its most impressive. Neko is a Java framework for constructing distributed algorithms whose aim is among others to provide a uniform and extensible environment for the various phases of algorithm design and performance evaluation. It is a simple communication platform that is designed to allow both simulating a distributed algorithm and executing it on a real network, using the same implementation of the algorithm.

[5] Based on [Urb01]

4.2 Architecture

Neko applications are constructed by building protocol stacks. (See also Figure 2.) They usually consist of a collection of processes each of which has an application part as well as an associated `NekoProcess`. The `NekoProcess` is placed underneath the application part on top of the `Network`. The application part is made up of several `Layers` each of which has its own purpose. Processes communicate with each other via a `Network` and represent the actual implementation of an algorithm.

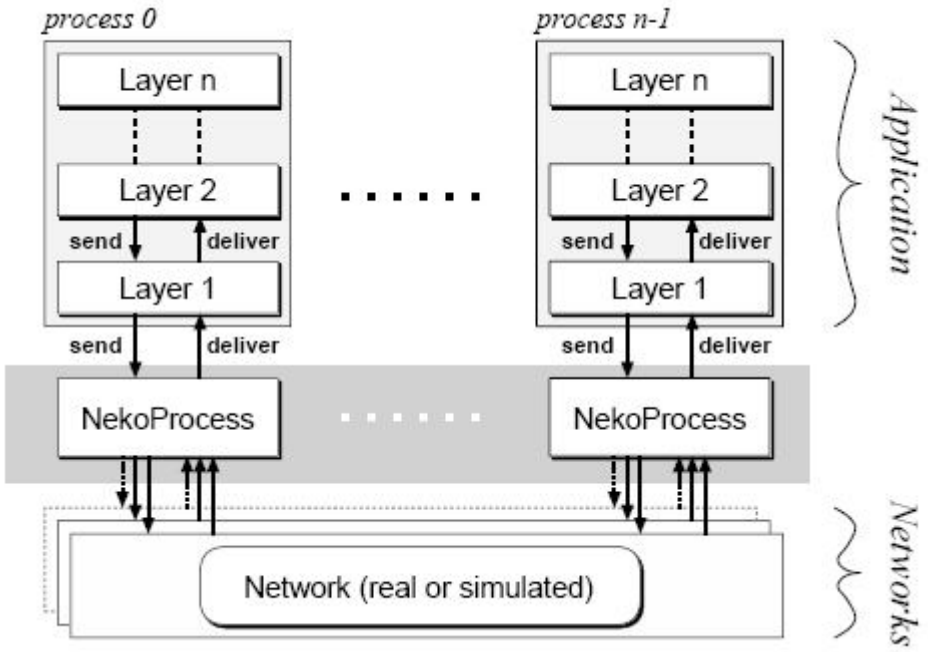

Figure 2: Basics of the Neko Architecture [Urb01]

4.3 Important Members of the Neko Framework

4.3.1 Layers

`Layers` are the most essential part of a Neko application. A `Layer` is a Java class implementing the `LayerInterface`. The `LayerInterface` contains among others the `send` and the `deliver` method. Messages to be sent are passed down the `Layer` hierarchy using the `send` method of the specific `Layer`, and messages delivered are passed up the hierarchy using the `deliver` method.

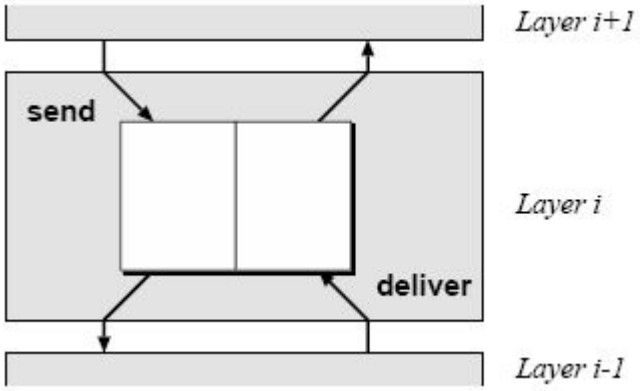

Figure 3: Passive Layers implement the `LayerInterface` [Urb01]

By overriding the `send` or the `deliver` method of the `LayerInterface` the course of an application can be controlled. The Neko framework comes with a class called `ActiveLayer`, which does not only implement the `LayerInterface` but also, has its own thread of control by sub classing `NekoThread`. This can be useful e. g. for starting up the whole application. `ActiveLayers` actively pull messages from the layer below, using *receive*.

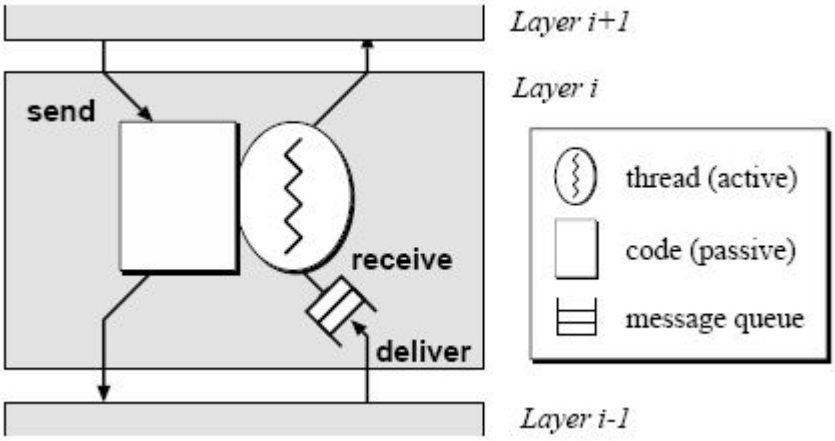

Figure 4: Active Layers have their own thread of control [Urb01]

`Layers` are not limited to overriding the methods defined in `LayerInterface`. They can also communicate by calling arbitrary methods on each other.

4.3.2 NekoProcesses

Each `NekoProcess` is associated with a real process. It holds the system wide address of the real process that belongs to it. This address is represented by an integer in Neko.

Not only does a `NekoProcess` catch all the messages that are addressed to its real process, it also takes care of all the messages sent from its real process to other processes and dispatches them to the appropriate network. The infrastructure of a Neko application can be assembled of any user-defined Network class, sub classing the (Neko) `Network` class. Furthermore the `NekoProcess` class implements some logging services.

4.3.3 NekoMessages

`NekoMessage` is the class instances of which are transmitted via any communication primitive. (`send`, `deliver` and `receive`) A major advantage of the `NekoMessage` is that its content part wraps any user-defined Java object. An instance of the `NekoMessage` class also contains the address of its sender and the addresses of its recipients and the type it is assigned to. A message type is a user defined integer which shall help to organize the communication pattern.

4.3.4 Configurations

The configuration process can be especially tiresome in distributed systems. It may have to be done many times, once for each node at the extreme case. A centralized configuration process as it is offered by Neko is a considerable facility. Whereas the configuration of simulations on a single computer is trivial, the effort for distributed applications can be time-consuming. In Neko a single configuration file is read and analyzed by the *Master* process, which passes the parameters on to all the other processes, the so called *Slaves*. It is also the *Master* process that starts up the whole application by bootstrapping the *Slave* processes. In addition to arbitrary user-defined or Neko specific parameters the configuration file contains some basic information like the addresses of the slaves the initialization of the application is based on.

The configuration process in Neko is based on the `Configurations` class which is part of the org.apache.java.util.Configurations package.

4.4 Startup Process

Whereas the previous section contains information about the functionality of the most important components of the Neko framework there has not been any explanation about how these components are actually integrated with the startup process yet. Since the

startup process in a Neko application is a little different from the one of other Java applications it needs some extra discussion.

None of the user-defined Java classes being integrated with the Neko framework contains the `main`-method that is first of all executed by the Java Virtual Machine. Thus the class containing the `main`-method is included in the Neko framework, namely the `Main`-class in the lse.neko-package of the Neko framework. It has to be called with a single argument, that is, the path to the configuration file. So the command to start the execution after all Java-classes have been compiled looks like the following one:

```
java lse.neko.Main “C:\Path\To\Config File\configFile.config”
```

Or in case a Linux machine is used:

```
java lse.neko.Main
“/home/userxy/pathToConfigFile/configFile.config”
```

The configuration file is then read by the `NekoInitializer`-class of the Neko framework. The configuration file must contain any information needed to start up the Neko framework, that is

- a flag that indicates whether this is a simulated environment or a real one,
- the class that is to be used to simulate the network,
- the number of processes that are to be launched and, most importantly,
- the name of a user-defined class that has to implement the `NekoProcessInitializer` interface of the lse.neko-package.

The init-method of the above interface decides which user-defined classes are integrated with the current execution of the Neko framework and in which order they are instantiated and put on the protocol stack. Any of these classes must be derived from the `Layer` or the `ActiveLayer`-class of the Neko framework or implement the `LayerInterface` interface.

The `init`-method can also call any other method needed to set up the user-defined part of the application and instantiate whichever user-defined class is necessary.

Aside from the compulsory Neko-specific parameters the configuration file may contain any other arbitrary parameter, e.g. the type of object hosted by a certain node or

simulation-specific parameters for performing constraint consistency checks. How this is used in practice will be explained in the following section.

5 Implementation of the Primary-per-Partition Protocol

5.1 Introduction

The current chapter is about the most challenging part of this master's thesis. On the one hand there are the specifications and restrictions of the Primary-per-Partition Protocol as outlined in chapter 3 that have to be met, and on the other hand there is the Neko framework that the implementation of the protocol has to be adapted to.

Neko offers a powerful and easy to use platform that produces its most advantageous qualities in a simulated environment.

An arbitrary number of nodes can easily be simulated and every node can be setup as client or as server at will.

The whole setup information is stored in a configuration file which is read at runtime. This is how Neko applications can be kept extremely flexible because whenever any parameters are to be varied, they can be written to the configuration file and the actual source code need not even be re-compiled.

However, the compact design of Neko does not only have advantages. The implementation of both, clients and servers has to be based on a single class which is derived from Layer. (See also section 4.3.1.) This class sends, receives and distributes messages to other classes. If there are several types of clients and servers a distinction of cases has also to be made in this class which might therefore get a little crowded.

If an application has to be integrated with the Neko framework for simulation reasons like in the case at hand it is important to keep in mind, that every class is instantiated at least once per node. Even though the nodes of a network are just virtual ones, an object cannot access another object at a different node. Hence any information that has to be available at two or more nodes must be distributed by the means provided by the Neko framework, i.e. wrapped into NekoMessages. It is not possible, e.g. to get a direct node spanning access to a static field.

These aspects will all be explained in detail together with the individual parts of the implementation.

5.2 Architecture

In [OFG+05] a system architecture for the implementation of the Primary-per-Partition protocol is proposed on which the architecture of the implementation at hand shall be based on. Some adaptations will have to be made though due to the requirements of the integration with the Neko framework. The aspects of these adaptations will be regarded in detail in the following paragraphs.

The implementation of the Primary-per-Partition protocol needs several components each of which has to perform a particular task. For implementation and for scaling reasons every Java class represents a component of the system.

A good component to start with would be a rather simple part of the implementation. At first glance the objects that give access to the information needed by the user or other objects and that are replicated on different nodes of a distributed system appear to be among the simplest parts of the system. An object is hosted by a node and provides methods which can be called by a user or another object. There are two aspects that complicate things a little, though. The first is that nested invocations shall be possible. Thus an object which is not actually part of the replication protocol has to know the gateway provided by the system in order to get access to another object.

The second is that a major task was to keep the objects deliberately generic. A key idea of the implementation at hand was to make as few restrictions as possible to the implementation of the above mentioned objects. The protocol differentiates between read and write operations. Thus there has to be a way to tell the difference between read and write operations putting into effect the idea of generic objects at the same time.

One possibility would be to tell the difference by the name and the signature of a method. This may not be a reliable way, though, because the name and the signature of a method do not necessarily give information about whether the state of an object is changed or not.

A safer way would be to let the protocol know whether a read or a write operation is invoked by adding a flag that indicates the type of method. That way the implementation of a method could also be altered easily without having to consider possible consequences of a change of its name or signature. So the client would have to specify the type of the method that it is going to call and would therefore have to know whether it calls a read or a write method.

In this respect also the implementation of an object may be affected by the classification of methods if it calls another object’s method, that is, a nested invocation is made.

The component that holds references to the individual objects located at a certain node and that provides a service for the invocation of the available methods is the invocation service (`InvocationService`). Therefore the method of an object is never called directly but always by calling the relevant method of the invocation service. The invocation service contains some additional functionality compared to the proposal in the [OFG+05] paper: If the system is in degraded mode i.e. not all nodes are reachable, the invocation service adds an entry for every method call to an object's version list in order to allow a rollback if the re-joining of two partitions produces a conflict.

The second important job of the invocation service is to provide support for remote method calls that are forwarded from the request handler and to return the results accordingly.

Another component is the replication manager. (`ReplicationMngr`) The purpose of the replication manager is to keep track of where an object replica can be found. Thus it stores the addresses of primary and secondary replicas and provides a set of methods to get access to the information it comprises. The replication manager is available to practically all the other components of the system.

There are two classes which actually provide methods to send and to deliver messages and which are therefore derived from the Layer class of the Neko framework. One is the request handler (`RequestHandler`) and the other is the `GMS`, the group membership service.

The request handler is the core element for sending and receiving remote method calls. It provides a method to post a request containing the name of an object, a String representing the method that is going to be called, an array of parameters that is needed to call the method and a flag that indicates whether this is a read or a write method. When it receives a reply to one of the above requests it tells the one that posted the request to pick up the return value. The proposal that has been published in [OFG+05] does not contain an explicit `RequestHandler` component. However, within the context of the [OFG+05] paper the `RequestHandler` represents a part of the *Replication Protocol*.

The group membership service (`GMS`) has been extended compared to the [OFG+05] proposal. It does not only provide a membership monitor and the appropriate methods to add a node to or to remove a node from the current view with respect to voluntary group changes as well as to group changes due to node or link failures but it also takes over some tasks that are performed by the *Replication Protocol*. Therefore it has become the most extensive component of all. This is also because of the diversity of

tasks it has to perform. Basically it is designed to keep track of which nodes are reachable. Technically seen it generates consistent failure notifications out of the inconsistent notifications delivered by failure detectors. Group membership monitoring based on these notifications is so essential because every message is associated with a list of processes to which it should be delivered. Since the implementation at hand is integrated with a simulated environment the group membership service has to perform two other important tasks: First it has to remove the relevant set of nodes from the current view if a partition split is triggered; and second it has to take control of the reconciliation process if another partition wants to rejoin. In keeping track of which nodes are reachable and which are not, the `GMS` is supported by a failure detector, the `NodeObserver`. The `NodeObserver` tries to contact all nodes in view in periodical intervals. If a node cannot be reached anymore it calls a method provided by the `GMS` in order to remove this node from the current view.

The [OFG+05] proposal contains also a *Group Communication* component which provides multicast support for groups of replicas. The implementation at hand does not have a component according to the [OFG+05] *Group Communication* component because no unwanted node or network failures have to be expected in a simulated environment.

The reconciliation manager (`ReconciliationMngr`) is designed to offer support during the reconciliation phase of the *Replication Protocol*. It is the component whose services the group membership service uses for merging version lists originating from different partitions. When the new version list is ready the reconciliation manager has to let it be checked by the constraint consistency manager.

Although the constraint consistency manager (`CCManager`) has been adapted for simulation reasons its purpose is very similar to the one of the constraint-consistency-manager-component described in [OFG+05]. Both components have to make sure that data integrity is maintained to the required degree. Since the environment for implementing the Primary-per-Partition protocol is a simulated one the constraint consistency manager does not perform any real constraint consistency check but it provides a method that returns a random value of whether a constraint is met or not. The probability of a positive return value can be set in the configuration file. In addition there are some parameters that can be used for configuring different probabilities for inter-constraint and intra-constraint checking. Details of the configuration and of the functionality of the constraint consistency manager are discussed in section 5.6.

The client is a component whose prime task is to perform some test method calls and to wait for the return values. It is also used for triggering a partition split for testing reasons and to start the reconciliation phase.

The simulation manager is independent of the rest of the system and has nothing to do with the actual implementation of the protocol. Its purpose is to provide the other components with the initial node addresses of the primary and the secondary replicas of each object and with the total number of nodes in the system. Additionally the simulation manager contains methods to simulate a partition split, that is, it specifies the addresses of the nodes that are kept in the individual partitions.

The initializer class is instantiated by the Neko framework and starts on its part the instantiation of the above classes. It is possible to restrict the instantiation of individual classes to certain node addresses.

The following chart represents a rough overview of the components of the implementation. The names refer to the Java classes that will be explained in detail in the following sections. GMS stands for group membership service and CCManager stands for constraint consistency manager.

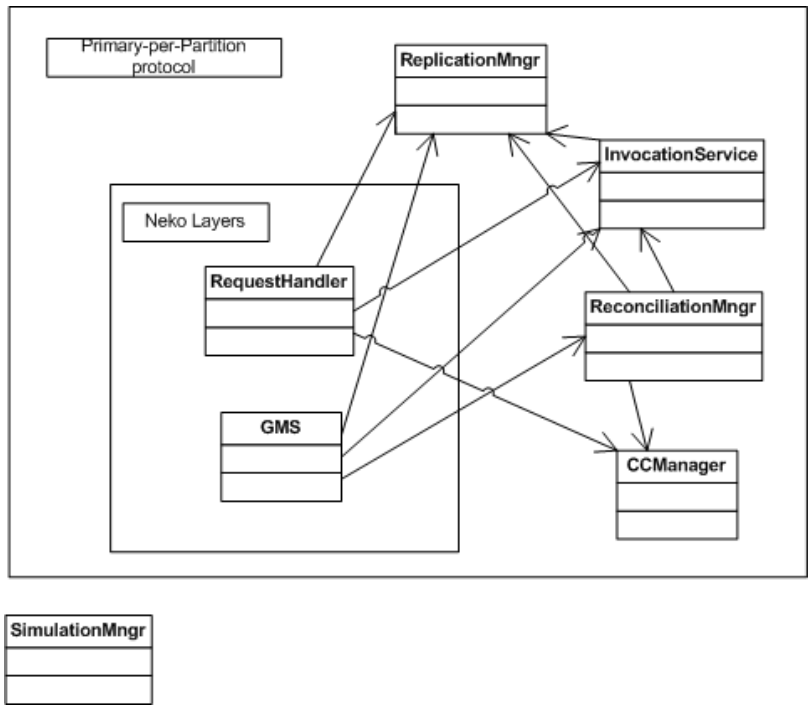

Figure 5: Overview of components to implement the Primary-per-Partition protocol

The [OFG+05] proposal contains also a *Group Communication* component that is to provide multicast support for groups and replicas. A feature of the Neko framework is

to provide multicast support without any further assistance of the application. Whereas a *Group Communication* component would still be needed in a real environment in order to ensure virtual synchrony and ordered multicast, in a simulated environment no unwanted node or network failures have to be expected. Hence the implementation of this component can be skipped in the simulated environment at hand.

5.3 Objects and Replicas

The objects that are replicated and whose methods are called by a remote client are in the center of interest for the client application. The client application has to be aware of what information is stored at which object and how it can be called up. Thus a set of public methods that is provided by the objects must be known in order to launch a remote method call.

As already mentioned in the previous section a major task was to keep the objects that are hosted by the nodes of the environment used for implementing the Primary-per-Partition protocol as generic as possible. This could possibly offer the opportunity to the client application to implement the objects itself and to just upload them to the individual nodes.

Some restrictions, however, could not be totally suspended. All objects have to implement the Java `Cloneable` interface from the java.lang-package. This is because a copy of the object has to be stored in the version list whenever a method is invoked on it in degraded mode. That way a complete rollback can be driven during the recovery phase of the system if revocable constraints are not met.

Another restriction concerns the demand of the protocol to support nested invocations. Obviously a method call from one object to another is somewhat different in a distributed system than in a local system. The calling object has to register as a client just like the client that it was originally called by. To make sure an object that needs to call another object disposes of the method needed to hand over a return value of a remote method call it has to implement the `IClient` interface.

```
public interface IClient {
       public void pickItUp(Object o);
       }
```

The process of launching a remote method call and picking up the return value is explained in detail in section 5.7.2.

5.4 Replication Manager

The purpose of the replication manager is to provide the addresses of the primary and the secondary replicas of every object. The core elements of the `ReplicationMngr` are four Hashtables. In the Neko framework node addresses are represented by integer numbers that is a node number is the equivalent of a unique address.

- The first one holds the numbers of primary replicas node-wise, that is, key-value-pairs where the keys represent the numbers of the nodes hosting primary replicas. The values associated with the particular keys are formed by HashSets containing the numbers of the objects whose primary replicas are located at this node.
- The second Hashtable holds the node numbers of the primary replicas object-wise, that is, another presentation of the same content as the first one. This time every key stands for the number of a particular object and the relevant HashSet holds the numbers of the nodes where this object is located.

To have these two different presentations of the same content avoids the need to write an inverse function to get both, the numbers of the primary replicas with a certain node number given and the addresses of the nodes hosting a certain primary object replica. Even at the costs of a little more complex update and write operations the approach appears to save work.

Logically the same as for the first two Hashtables applies for the third and the fourth Hashtable. But this time it is the numbers and the node numbers of secondary replicas.

- The third one holds the numbers of secondary replicas node-wise that is the numbers of nodes as keys and HashSets with object numbers as values.
- The fourth Hashtable holds the node numbers of the secondary replicas object-wise that is the keys are represented by object numbers and the HashSets each of which belongs to a certain object number hold the numbers of the nodes where the object is located.

The following example shall not only clarify the above circumstance but also demonstrate the behavior of the `ReplicationMngr` in case a node joins or is cut off.

1.)

The first Hashtable looks like the following one:

Node Numbers (Key)	0	1
Primary Object Numbers (Value)	0	1

And a presentation of the second one:

Primary Object Numbers (Key)	0	1
Node Numbers (Value)	0	1

The third Hashtable holds the node numbers of the secondary object replicas:

Node Numbers (Key)	0	2	3	4	5
Secondary Object Numbers (Value)	1	0,1	1	0,1	0

As does the fourth Hashtable but the other way round:

Secondary Object Numbers (Key)	0	1
Node Numbers (Value)	2,4,5	0,2,3,4

The information that the four above Hashtables represent can be summarized the following way:
Obviously there are 6 nodes residing in the system numbered from 0 to 5 and two objects numbered 0 to 1.
Node number 0 hosts the primary replica of object number 0. In addition it hosts the secondary replica of object number 1.
Node number 1 hosts the primary replica of object number 1 and no secondary replicas.
Nodes number 2 and 4 host secondary replicas of both, objects number 0 and 1.
Node number 3 hosts a secondary replica of object number 1.
Node number 5 hosts a secondary replica of object number 0.

2.)

Assuming that node number 0 gets cut off, e.g. due to a partition split another node has to take over its role. The changes of the Hashtables are displayed below.
The Hashtable holding the primaries node-wise:

Node Numbers (Key)	2	1
Primary Object Numbers (Value)	0	1

The Hashtable holding the primaries object-wise:

Primary Object Numbers (Key)	0	1
Node Numbers (Value)	2	1

The Hashtable holding the secondaries node-wise:

Node Numbers (Key)	2	3	4	5
Secondary Object Numbers (Value)	1	1	0,1	0

The Hashtable holding the secondaries object-wise:

Secondary Object Numbers (Key)	0	1
Node Numbers (Value)	4,5	2,3,4

As soon as the miss of node number 0 is reported by another component of the system a new primary replica of object number 0 has to be selected among all secondary replicas. The selection can be made in any arbitrary way. In the implementation of the `ReplicationMngr` at hand the node with the lowest number that hosts a secondary replica of the object whose primary replica got lost is selected. In this case this is node number 2.
Thus node number 0 is replaced with node number 2 in the first two Hashtables.
The entry with key number 0 and respectively the column with node number 0 are removed from the third Hashtable.
In addition object number 0 has to be removed from the HashSet belonging to node number 2 in the third Hashtable because a node cannot host a primary and a secondary replica at the same time.
Number 2 has to be removed from the HashSet belonging to object number 0 in the fourth Hashtable.

3.)

In the third example the re-entry of a node is simulated. Node number 6 hosts the primary replica of object number 0 and a secondary replica of node number 1. Again the changes in the four known Hashtables are displayed below.
The Hashtable holding the primaries node-wise:

Node Numbers (Key)	6	1
Primary Object Numbers (Value)	0	1

The Hashtable holding the primaries object-wise:

Primary Object Numbers (Key)	0	1
Node Numbers (Value)	6	1

The Hashtable holding the secondaries node-wise:

Node Numbers (Key)	2	3	4	5	6
Secondary Object Numbers (Value)	0,1	1	0,1	0	1

The Hashtable holding the secondaries object-wise:

Secondary Object Numbers (Key)	0	1
Node Numbers (Value)	2,4,5	2,3,4,6

Node number 6 takes over the role of the host for primary replica number 0.
The join of node number 6 is displayed in the first two Hashtables.
Therefore node number 0 has to be downgraded to a host for a secondary replica of object number 0.
In addition node number 6 hosts a secondary replica of object number 1.
The changes in the sets of nodes hosting secondary replicas of objects number 0 and 1 are displayed in the third and the fourth Hashtable. Nothing is changed about status of node number 2 as a host for secondary replica number 1 though.

The `ReplicationMngr` does not only hold the four above mentioned Hashtables but it also provides functions to easily insert a node or remove an object or a node from the system.

It also offers a function that returns the number of the node hosting the primary replica of certain object and a function that returns an integer array containing the numbers of all nodes holding secondary replicas of a certain object.

All other functions implemented in the `ReplicationMngr` are helper functions for transforming a HashSet into an integer array and the other way round.

It is important to keep in mind that an instance of the `ReplicationMngr`-class is located on every node of the system. Thus changes that are observed by a node and that the `ReplicationMngr` is adjusted to by calling the appropriate add- or remove-function have to be propagated to all other nodes as well.

5.5 Invocation Service

The prime target of the `InvocationService` is to offer support for any operation in connection with an object, e.g. the invocation of a method.

It holds a repository with the objects that are instantiated at the startup of the application. The objects are kept as instances of the Java Object-class in a Hashtable. This is because the aim was to specify the objects that are hosted by a certain node in a configuration file without having to modify the source code of the `InvocationService` itself.

Every class of an object is assigned a name to as well as an integer. The `InvocationService` helps to pull together the names and the numbers in an easy way.

The `invokeMethod`-method can be called to invoke a method on a certain object. The arguments that are needed to call the `invokeMethod`-method are

- the name of the method to be called,
- the object that the method is to be invoked on,
- the arguments that the method is to be called with,
- a HashSet containing the addresses of the secondary replicas of the same object as the one whose method is going to be called,
- the address of the primary replica.

The addresses of the secondary replicas and of the primary are needed if the system has to recover from degraded mode and to find out which version list is the one that is kept in normal mode. The `invokeMethod`-method invokes the delivered method on the relevant object by using the java.lang.reflect-package. The advantage of calling a method that way is that the `InvocationService` need not implement different routines depending on what method is called but it can call any arbitrary method in a uniform manner. The method returns a value if there is one and null if there is none.

When the invocation is finished the `InvocationService` adds a version to the version list of the relevant object. A version includes the state of the object after the invocation of the object, the name of the method that has been invoked, the arguments that were needed by the method, a HashSet containing the addresses of all current secondary replicas and the primary's address.

An example of a version list with three entries can be found below:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4, 1, 5	0
1	setParm	myObjectA (1)	5	4, 1, 5	0
2	getParm	myObjectA (2)	null	4, 1, 5	0

Table 2: Version list with three entries in degraded mode

The version numbers are represented by ascending integer numbers. When a new version is generated the way the version list is updated depends on the state of the system. If the system is in normal mode, the first version, i.e. version 0 is just overwritten with the new version. Thus there is just a single entry in the version list whenever the system is in normal mode.

The Copy-of-Object-entry in the above table is of a symbolic nature. In fact it is a complete copy of the relevant object that has been generated after the method has been invoked on it. This is needed in case a roll back has to be performed during the recovery phase of the system.

In the above example the addresses of the secondaries and the primary remain unchanged. Thus there has only been one partition split that occurred before version 0 was generated. If another partition split had occurred after the first one the entries in Secondaries-column or even the address of the primary would have changed.

5.6 Constraint Consistency Manager

The Primary-per-Partition protocol does not contain any requirements concerning the way constraint consistency checks are performed. In fact constraint consistency checking is closely tied to the individual requirements of an application. Constraint consistency may range from simple upper or lower bounds of a single parameter to highly complex inter-object constraints incorporating an arbitrary number of objects.

The key question for the implementation at hand is: How can intra-object and inter-object constraint evaluation be implemented in a simulated environment?

A possible approach would be to tie the implementation of constraint consistency checking to the implementation of the objects whose primary and secondary replicas are hosted by the individual nodes. This approach would have the advantage that the way a constraint is evaluated is clearly defined and can easily be performed whenever an object is reachable.

A disadvantage of this approach would be, though, that the implementation of the constraint evaluation might not be representative for constraint consistency checking in many other applications. Another disadvantage would be that the objects would have to undergo restrictions in order to be able to interact with the constraint consistency manager.

A more generic approach is to detach constraint evaluation from the implementation of the objects in a simulated environment like the one at hand. In a real environment a logic separation of objects and constraint consistency would of course not be reasonable. The core problem of this approach is how to simulate constraint evaluation. A reasonable solution appears to be based on random variables.

A good concept is to consider an evaluation attempt successful whenever a random variable does not exceed a limit that has been defined and written to the configuration

file. This works fine with intra-object constraints, but what about inter-object constraints? The same procedure can be applied with a different limit. That way different probabilities can be set up for intra- and inter-object constraints in a configuration file without having to change the source code.

However another important question has not been answered yet: What if for some reason a constraint evaluation has to be performed twice or more times? This will be the case when the system recovers e.g. after a partition split. A simulation of constraint evaluation based on random variables can not guarantee that the same constraint consistency check returns the same result every time it is performed. Thus in contrast to real constraint consistency checking the simulated one is not idempotent.

That is constraint evaluation regarding the same operation and respectively the same object can be successful the first time and fail the next time. This is only acceptable if it does not have an impact on the simulation of the functionality of the Primary-per-Partition protocol which is the case.

Constraint consistency checking is most closely tied to request handling and to reconciliation management. Whenever a remote method is called it is up to the constraint consistency manager to decide whether the invocation of the method may be performed or not. Since the request handler must be able to deal with either case, simulated constraint checking is no problem here. Logically the same conditions apply to the reconciliation management. What may occur, though, is that the same version that was accepted when a certain method was called is rejected in the reconciliation phase due to a negative re-evaluation of a constraint.

Another specification of the Primary-per-Partition protocol has to be considered: The protocol distinguishes between pre- and post-conditions, that is that the former has to be checked before a method is invoked and the latter after the invocation. Only when both conditions are successfully checked a transaction may be committed. If constraint evaluation is based on a random variable like in the case at hand the point when the check is made does not make any difference. Both checks can rather be made before the method is invoked.

So to sum it up what are the parameters needed for a simulated constraint evaluation and that can be defined in the configuration file? First the configuration file contains a value representing the probability that all constraints are met for each object. Second there are values representing the probability that among all constraints an object is involved in there is a critical inter-object one. So in normal mode a constraint consistency check is made by generating a random variable and checking whether it

remains below the first value. In degraded mode it is also checked if a critical inter-object constraint exists for the relevant object. The example below shall make the meaning of the values specified in the configuration file.

Entry in configuration file:	**Meaning:**
0.constraint=0.9	The probability that all constraints concerning object number 0 are met is 90 %.
0.inter.critical=0.1	The probability that among the constraints concerning object number 0 is a critical inter-object one is 10 %.

Table 3: Configuration of constraint evaluation

The constraint consistency manager provides another important method that is needed during the reconciliation phase. When the `ReconciliationMngr` has finished merging two version lists (See also section 5.8.) the constraint consistency manager has to check if the new version list meets all constraints. This is not necessarily the case because version lists of different objects might originate from different partitions. Therefore inter-object consistency checks that succeeded when a method was invoked the first time may now fail because a different version of another object has been installed during the reconciliation phase. The method that is provided by the constraint consistency manager takes as an argument a Hashtable containing the new version lists of all objects with keys being represented by the object names. The method checks the version list of every object by looking for the latest version that meets all constraints. All versions newer than the version found in the previous step are removed from the version list. So the latest version in the old version list that meets all constraints is the last one in the new version list. If no version to meet all constraints is found version 0 is re-installed. This is the last known version to have been generated in normal mode.

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary	Constraint Consistency Check
0	testRoutine	myObjectA (0)	null	4, 1, 5	0	
1	setParm	myObjectA (1)	1	4, 1, 5	0	
2	testRoutine	myObjectA (2)	null	4, 1, 5	0	
3	setParm	myObjectA (3)	5	4, 1, 5	0	pass
4	setParm	myObjectA (4)	0	4, 1, 5	0	fail
5	getParm	myObjectA (5)	null	4, 1, 5	0	fail

Table 4: Version list being checked by the constraint consistency manager

Table 4 displays a version list that is checked by the constraint consistency manager. It starts at the latest version, that is, version number 5 and evaluates all constraints concerning this version. In this case the evaluation fails and version number 5 has to be removed from the version list. The next step is to evaluate all constraints concerning version number 4. The evaluation is again unsuccessful, thus version 4 has to be removed, too. Version number 3 is the latest one to meet all constraints. Thus version 3 is the new current one.

5.7 Request Handler

Although the prime target of the Primary-per-Partition protocol is to allow a configurable tradeoff between consistency and availability the end-user may not care too much about what happens behind the scenes when he launches a request for a remote method call, e.g. by clicking a button of a graphical user interface. Maybe even a programmer that has to integrate a client application with the Primary-per-Partition protocol is not interested too much in what happens in case of a network failure or during the recovery of the system. But he certainly is interested in what methods are provided by the system in order to get access to remote objects. These methods that represent the main gateway of the system are located in the `RequestHandler`-class.

5.7.1 Callback Handler vs. Threads:

Before the request handling is explained in detail some basic considerations have to be made concerning the interaction between the client and the Primary-per-Partition protocol.

The interface that is offered to the client for launching a request for a remote method call may have various forms. The most obvious would be to offer a method that takes as arguments the name of the method to be called, the object that the method is to be invoked on and the arguments that the method is to be called with. This method would transmit the return value of the remote method call directly to the client. The signature of such a method in Java could look like the following one:

```
Object returnValue = callRemoteMethod(String objectName,
                     String methodName, Object[] arguments)
```

A major advantage of this kind of request handling would be that full transparency is provided and the integration with a client application would be simple and straight forward.

But as far as the `RequestHandler` is concerned things become a little complicated. The circumstances are displayed in Figure 6. The `RequestHandler` would take over the control flow from the client posting a remote method call. After starting another thread of control the first one would be blocked e.g. by calling `wait`(). The `RequestHandler` can only provide a synchronous control flow as long as it does not have to yield the control flow to another node and respectively to another thread. In most cases the node that accepts the request from a client will not be able to generate the result itself. It will have to forward it to the node where the primary replica of the relevant object is located. This is the point when it would have to send an (asynchronous) message and drop the control flow afterwards. It would only regain the control flow when it receives a reply-message including the desired return value. In the meantime it would have to block the client request and resume it only when the reply message is received.

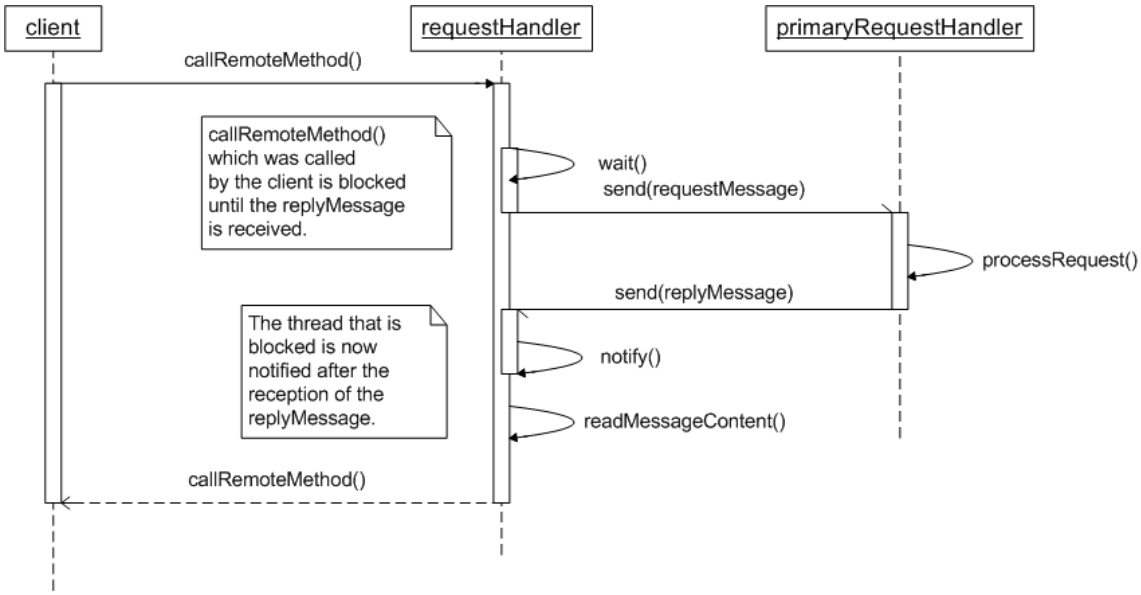

Figure 6: Thread based request handling

The request handling by threading is on the one hand very client-friendly but on the other hand it has some disadvantages that have to be considered:

- A message might get lost and therefore the thread that processes the client request and that is paused at the beginning would never be resumed. Hence a

time-out function would be necessary to insinuate to the client that the request was not successful.

- The simulated environment provided by Neko is based on threading. If threaded request handling was implemented there may be two or more different threads representing a single node. This could have incalculable side-effects on the environment.
- A thread-based approach is more error prone since deadlocks that may occur are hard to find. The implementation of a relatively failure-free threaded approach would therefore be a tedious task.

Due to the above mentioned problems it appears to be more promising to look for another approach and to drop the thread-based solution. A more auspicious way to handle requests may be one that is based on callback functions. The operating mode of a callback-based solution is outlined in Figure 7. First the client has to register with the `RequestHandler` by calling the `registerMeFor`-method. It has to deliver all parameters needed for the remote method call with the arguments of the `registerMeFor`-method. (For lack of space this is not displayed in the sequence chart below.) The `RequestHandler` then finds out the address of the primary and generates the request message. When the `RequestHandler` has sent the request message to the primary the thread of control originating from the client is terminated. The `RequestHandler` regains the thread of control only when it receives the reply message originating from the primary. When it has finished extracting the content and updating its waiting-clients-list it passes the thread of control on to the client by calling the `pickItUp` method of the client. To make sure that the client implements the relevant callback handler function a Java interface is specified (see also section 5.3) which has to be implemented by the client.

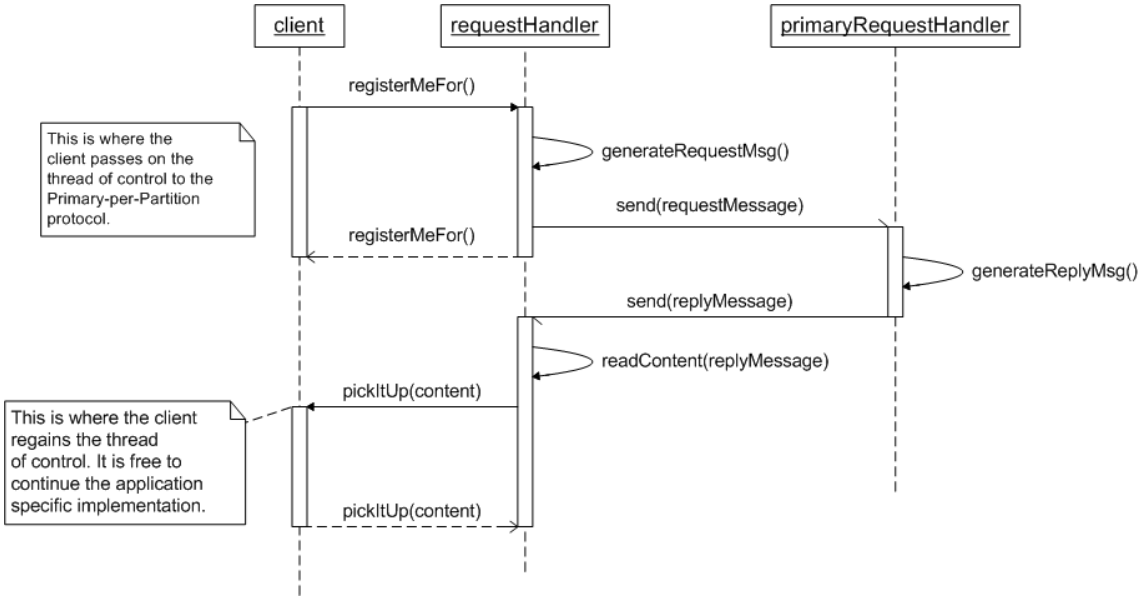

Figure 7: Request handling via a callback handler

This appears to be the more solid approach than the threaded one which is why this is also the one that is going to be explained in detail.

5.7.2 Request handling step by step:

What happens in detail when a request for a remote method call is sent to another node? The following steps are also outlined in the sequence chart below (Figure 8). To better demonstrate the process of request handling in detail the system is divided into three different types of nodes: The node that is designed to serve write method calls and that holds the primary replica of an object, a node that accepts the client request and the nodes holding the secondary replicas of an object. The node that accepts the request from a client may be holding either a secondary replica or a primary one. Aside from the fact that it accepts the client request there is no difference compared to the other two types of nodes, though.

- The client application has to define the name of the method to be called, the object that the method is to be invoked on, the arguments that the method is to be called with and a flag indicating whether this is a read or a write method. These parameters have to be passed on to the `RequestHandler` by calling the `registerMeFor`-method.
- To keep track of all requests that have not been answered yet the `RequestHandler` stores the above mentioned parameters in a Hashtable. This

Hashtable is stored in another Hashtable with a reference to the relevant client as a key. Thus there can only be one request per client in progress. If a client posts another request before the first one has been answered, the first one is overwritten and therefore discarded. Since `RequestHandlers` residing on different nodes do not synchronize their client-Hashtables it is possible for the client to be served several requests at the same time by sending them to different nodes, though.

- The `RequestHandler` has to find out where the object whose method shall be called is located. Depending on whether a read or a write method is called, it is either the address of any arbitrary node holding a replica or the primary's one that is needed.
- The next step is to wrap the parameters that the `RequestHandler` has got from the client into a `NekoMessage` and to send the `NekoMessage` to the primary whose address was obtained in the previous step. The content of a `NekoMessage` is of the type Java Object. In order to send several parameters of different types they have to make up a single Object that is put into a `NekoMessage`. A data structure that can easily be used for storing different types is again the Hashtable of the java.util package.
- A `NekoMessage` does not only contain an Integer array representing the addresses of its recipients and the actual content but also an Integer field marking the type that the message is assigned to. Message types are usually defined as Integer fields and registered with the `NekoSystem` to facilitate the handling of message types. The message type shall help a recipient to classify a message before its content is made subject to further processing. In case of a `RequestHandler` sending a request to another RequestHandler the message type is set to *REQUEST*.
- If the relevant method is a write method the `RequestHandler` of the receiving node has to check if all constraints are met by calling the `checkConstraints`-method of the constraint consistency manager. If the constraint consistency manager does not return a positive result the `RequestHandler` has to cancel the request and notify the sender of the request message about that it was not able to serve the write request. In a real environment the `RequestHandler` would also have to get the post conditions checked by the constraint consistency manager after the request has been served by the `InvocationService`. The way constraint checking is performed in the simulated system at hand the post

condition check can likewise be made before the method call or be skipped at all.

- When the constraint check has succeeded or if the relevant method is a read method the `RequestHandler` of the primary calls the `invokeMethod`-method of the `InvocationService` to call the relevant method and to obtain the return value.
- In order to propagate updates to the secondary replicas the primary forwards the request message to all secondary replicas in view. Since the system undertakes a classification of requests into read and write requests this step can be skipped if a read request has been launched. In this case the primary returns the result of the request to the client and does not perform the following steps.
- The secondaries do not have to perform a constraint consistency check since the primary has already done it.
- Since synchronous update propagation is favored in the Primary-per-Partition protocol the primary has to wait until all secondaries have sent a *REPLY* message. In order to keep track of which secondaries have already sent a *REPLY* message and which have not the primary has to keep a Hashtable with the parameters of the request and a list containing the addresses of all secondaries. Whenever it receives a *REPLY* message the address of the sender is added to another list belonging to the same request and the two lists are compared with each other. When both lists contain the same addresses the update propagation to the secondaries is finished.
- When the primary has received a *REPLY* message by all secondaries it removes the client from the `waitingClients` Hashtable and tells it to pick up the return value by calling the `pickItUp`-method.

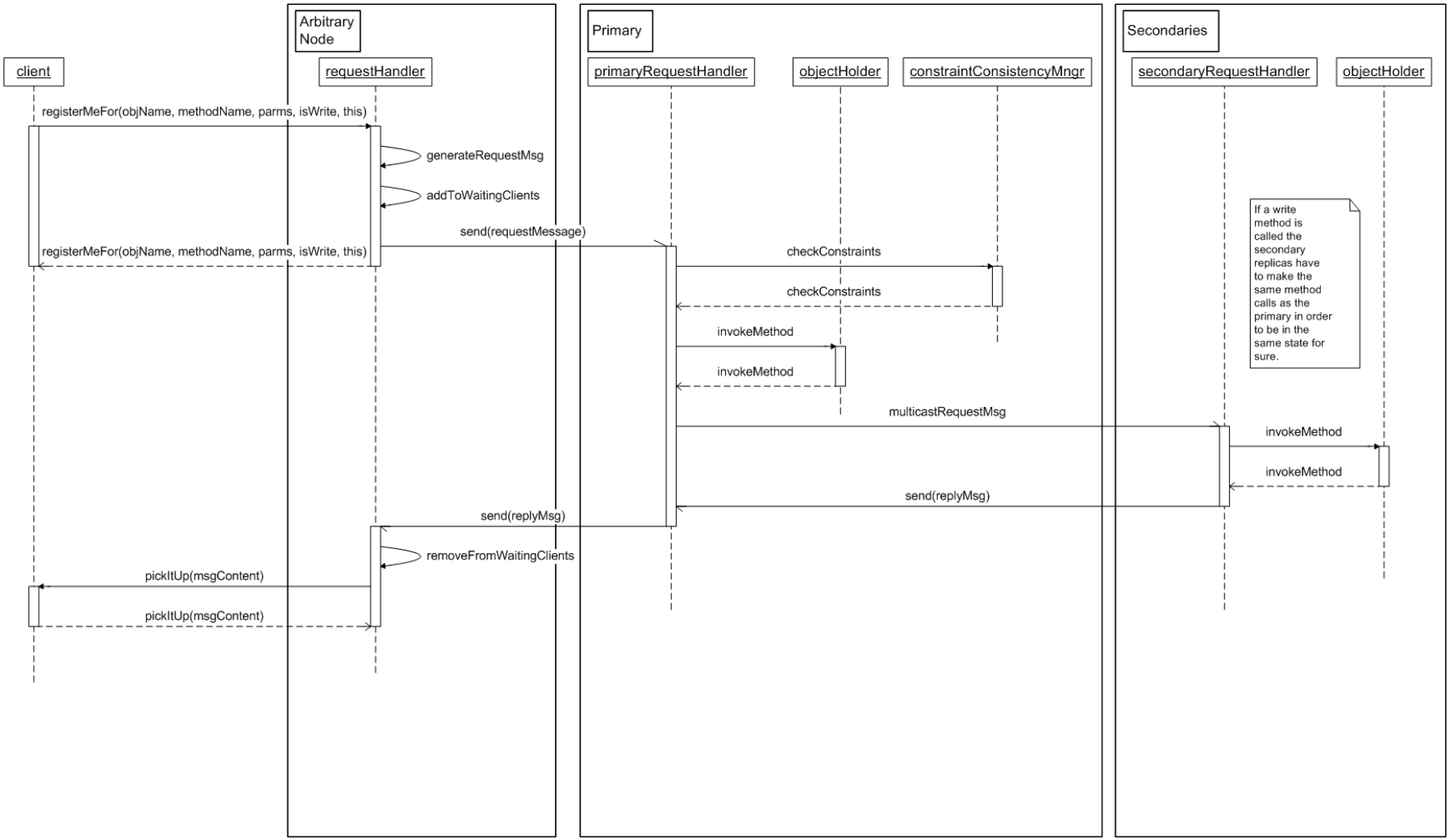

Figure 8: Handling a remote write method call in detail

5.7.3 Nested Invocations

As already mentioned in section 3.2 nested invocations are supported by the Primary-per-Partition protocol. That is the method of a remote object that has been called by a client may launch a remote method call itself. The restrictions that an object has to undergo in this case have been explained in section 5.3.

The above figure displays the process of calling a remote request for a write operation in detail. Method calls causing read operations are not concerned because they do not have to be forwarded to other replicas. The figure does not display the functionality of a nested remote method call, though. As already mentioned a write method call is served by the primary replica first and then propagated to all the secondary replicas of an object, that is, replicated as often as there are secondary replicas. To give an example, if the invocation of a write method causes the increment of a parameter by 100 and there are 4 secondary replicas the difference between the parameter before the invocation and the same parameter after all secondaries have completed the invocation will be 500.

Aside from the raise of network load originating from nested calls another aspect that might have to be considered is that whenever a primary forwards a request to its secondary replicas, it waits for their replies before it sends the return value of the method call back to the client. When a secondary processes a call that causes another remote call, it sends the *REPLY* message back to the primary after it has called the `registerMeFor`-method. Therefore the *REPLY* message arrives at the primary before all nested invocations are finished and also the client may get a reply while some nested invocations are still on their way. So synchronous request handling cannot be fully guaranteed when nested invocations are processed.

In literature there cannot be found many general-purpose solutions to the problems caused by nested invocations. A solution is described in [MKK95]. In their approach a replication-aware communication layer is introduced on top of which replicated objects execute. Whenever a replicated object is going to invoke another replicated object, the invocation request is first assigned the same, unique identifier by each replica of the invoking object. A coordinator of the invoking group of replicas forwards the request to all the replicas of the object that is being invoked while the other replicas of the invoking group of replicas hold back their copy of the invocation request. As a result each replica of the object that is being invoked receives the invocation only once.

In the implementation at hand nested invocations are only indirectly tied to the implementation of the Primary-per-Partition protocol. A nested call is part of the

implementation of a certain method of an object. Therefore the only way to limit nested calls is to make restrictions on the architecture of an object.

In order to put the [MKK95] approach into practice every method called in the course of a nested invocation must have an argument representing the unique identifier. The primaries can take on the coordinator's role and keep track of all unique identifiers included in the messages that trigger update propagations. That way every request is processed only once.

The approach foils the idea of keeping objects as generic as possible a little. But it represents a practicable way of getting the growth of network traffic and the unwanted side effects of operation transfer under control.

The question remains what can be done about the lack of synchrony that originates from nested invocations. Due to the callback-architecture of the implementation at hand it must be left to the application to deal with this lack of synchrony which should not pose major problems if the circumstances are clear.

5.8 Reconciliation Manager

The `ReconciliationMngr` is designed to implement a part of the *Replication Protocol* proposed in [OFG+05]. There are two objectives that have to be achieved by the `ReconciliationMngr`. The first one is to merge two version lists considering the specifications of the Primary-per-Partition protocol. This is needed whenever the system recovers from partitioning. When a partition split is fixed and two partitions re-join there are two different version lists per object that have to be merged. One originates from the first primary replica and the other one from the second. As already explained in the Invocation Service-section (Section 5.5) every version list contains among other information the numbers of the nodes in view that host a replica of the relevant object. In accordance with the specifications of the Primary-per-Partition protocol the `ReconciliationMngr` calculates the number of nodes of the latest version of each partition and installs the version list of the partition with the maximum number of nodes. The new primary is the one that originates from the same partition. The resulting version lists are the ones that have to be installed by the relevant nodes.

This is the second objective that has to be achieved. The installation process is performed in the following steps:

- The `updateVl`-function of the `ReconciliationMngr` is called and the new version list is handed over together with a HashSet containing all nodes in view.

- The `ReconciliationMngr` has to check if the current node actually hosts a replica of the object whose version shall be installed. If not the method is exited.
- The next step is to get a reference to the old version list and to drop all versions but the first one.
- Then the method of the second entry of the new version list is invoked on the copy of the object in the first entry of the old version list. This generates the new second entry of the old version list.
- The last step is repeated until the last entry of the new version list is reached. This is the point when both version lists look alike.
- To finalize the installation process the `ReplicationMngr` is updated by adding the new nodes in view and setting the primary of the new version list. Hence the original primary may be replaced with a new one. Since all nodes have the same privileges and all replicas hold the same version list the protocol can deal with a new node hosting the primary replica as well as with the old one.

To clarify things the following example reflects the last steps. Table 5 and Table 6 represent the situation before a new version list is installed.

Version list to be installed:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4,1,5,6,8	0
1	setParm	myObjectA (1)	2	4,1,5	0
2	testRoutine	myObjectA (2)	null	4,1,5	0

Table 5: Version list that is going to be installed at the current node

Table 5 displays the version list that originates from merging the version lists of the joining partition.

Old version list:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4,1,5,6,8	0
1	getParm	myObjectA (1)	null	8	6
2	setParm	myObjectA (2)	5	8	6
3	getParm	myObjectA (3)	null	8	6

Table 6: Version list that is currently installed

Table 6 displays the version list that is currently installed at a node of one of the partitions. The next step is to drop all versions but the first one.

Old version list:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4,1,5,6,8	0

Table 7: Version list with only the first entry left

Table 7 displays a version list at the start of the installation process. The next version is generated by invoking the `setParm`-method of version number 1 of the new version list on copy "`myObjectA` (0)" of the old version list.

Old version list:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4,1,5,8,6	0
1	setParm	myObjectA (1)	2	4,1,5,8,6	0

Table 8: Version list where version 1 has been generated anew

Since the copies "`myObjectA` (0)" represent the same state in both version lists also the copies "`myObjectA` (1)" represent the same state. Additionally the new primary is installed and the new node numbers are added to the ones that are already known. That last step is now repeated to obtain version number 2: The `testRoutine`-method of version number 2 of the new version list is invoked on "`myObjectA` (1)" of the old version list.

Old version list:

Version Number	Method Name	Copy of Object	Arguments	Secondaries	Primary
0	testRoutine	myObjectA (0)	null	4,1,5,8,6	0
1	setParm	myObjectA (1)	2	4,1,5,8,6	0
2	testRoutine	myObjectA (2)	null	4,1,5,8,6	0

Table 9: Old version list after having been completely updated

"`myObjectA` (2)" has been generated by invoking the `testRoutine`-method on "`myObjectA` (1)". Since the old version list has been completely updated it can now as well be denoted "new version list". This is the version list that any further versions are going to be added to as long as the system is in degraded mode.

5.9 Node Observer

The `NodeObserver` is a component whose only task is to ping all nodes in view periodically and to notify the group membership service if there is a node that does not reply.

Therefore the view has to be passed on to the `NodeObserver` by the group membership service. The `NodeObserver` goes online immediately and does not need anymore considerations once it is up and running.

If a node cannot be reached anymore it calls the `leaveCluster`-method of the group management system.

5.10 The Client

Obviously the Client-class impersonates the role that a client application could play. It is not part of the actual implementation of the Primary-per-Partition protocol.

The implementation of the Primary-per-Partition protocol is designed to offer a gateway that is as transparent as possible for the clients. As already outlined in the previous chapter this gateway is located in the `RequestHandler`-class. The key function that is offered by the `RequestHandler` to the client applications is the `registerMeFor`-function. The only requirement that a client has to meet is to implement the `IClient` interface. This is to enable the protocol to notify a client when the return value of the remote method that was called is received.

The implementation of the `Client`-class is rather unspectacular. The `Client` simply calls the `registerMeFor` method for testing reasons and prints out the return value as soon as it is notified. An aspect that has to be considered for the implementation of the client application is that the client has to be aware of whether it calls a read or a write method. So the arguments of the `registerMeFor` method include a flag that indicates whether a read or a write method is called.

Aside from the invocation of remote method calls the `Client`-class has to perform another important task. If a partition split is to be simulated the `Client` has to trigger it by calling the `simulateSplit`-method of the `GMS`. The functionality of the `simulateSplit`-method is similar to the `registerMeFor`-method of the `RequestHandler`-Class in that a callback function of the `Client` is called as soon as the new setup of the system is complete. Although the triggering of the partition split is done by the `Client` here it could as well be done by any other class.

According to the design of the implementation at hand the client application that is represented by the `Client`-class is free to impose any other task to the class that accesses the gateway of the Primary-per-Partition protocol, that is, the `registerMeFor`-method of the `RequestHander`-class. Therefore the `Client` can be enhanced or modified in any arbitrary way if any other behavior is needed.

5.11 Simulation Manager

The `SimulationMngr`-class has on the one hand been left aside so far because it has nothing to do with the actual implementation of the Primary-per-Partition protocol, either. On the other hand the tasks it performs are nonetheless as essential as the ones being performed by the actual components of the implementation. The prime target of the `SimulationMngr` is to provide the information needed by the individual components for starting up the whole system. In addition the way this information is generated when the `SimulationMngr` is started up itself shall leave enough room for enhancements or modifications e.g. for testing purposes. An important aspect of the implementation of the `SimulationMngr` was that it should be possible to specify the key parameters in the configuration file without having to change the source code.

The task of distributing the startup information to the individual components can be divided into two sub-tasks: One is to read the configuration file and to extract the parameters defining the system boundaries. This is easily done thanks to the configuration management provided by the Neko framework. Any parameter contained in the configuration file can be returned in the form needed by calling the appropriate method of the Configurations-class of the Neko framework.

The second sub-task is a little more complex to achieve because the form in which information is passed on to the components of the system has to be adapted to the individual component. The configuration file contains the number of nodes that are to be simulated and for each node the names of the classes whose instances it hosts, thus the set of objects located at a certain node. Basically there are two components that need to have this information. One is the `ReplicationMngr` and the other one is the `InvocationService`. To hand the information over to the `ReplicationMngr` first a similar Hashtable than the ones that were shown in section 5.4 is set up based on the parameters of the configuration file. The `GMS` then sets up the Hashtables of the `ReplicationMngr` by node-wise calling the `addObject`-method.

As far as the `InvocationService` is concerned a version list for each node is generated also according to the parameters of the configuration file. Since the system

starts in normal mode, the version list contains only version 0. The `InvocationService` of every node can pick up its initial version list by calling the `getObjects`-function of the `SimulationMngr`.

When the system is up and running the `SimulationMngr` is not needed anymore unless a partition split shall be simulated at a later point. If so the `SimulationMngr` has to read out all necessary parameters to define where the partition split occurs. A good solution is to define boundary nodes, that is, all nodes with a number below the boundary node number form a view and all others form another view.

To sum it up, the flexible configuration handling that is provided by the Neko framework leaves a lot of room for testing simulated environments by writing any arbitrary parameter needed to the configuration file.

5.12 Group Membership Service

The group membership service (`GMS`) is the most extensive component of all. It is the second component beside the `RequestHandler` that can send and receive messages. This is because it has to negotiate with the `GMS` located at the other nodes of the system what happens if a node leaves or re-joins the system. The [OFG+05] proposal for the group membership service component envisions an interface to register handler routines which are called when a view change occurs. Also the `GMS` implementation at hand contains the relevant methods for adding new nodes to or removing missing nodes from the current view. Since the implementation is integrated with a simulated environment additionally a partition split as well as the reunion of two partitions has to be initiated by the `GMS` from inside the system. The routine to remove a node from the current view can also be called by the `NodeObserver` if a node failure is detected.

As long as the system is operating faultlessly the `GMS` simply passes on all incoming messages to the `RequestHandler`. Whenever the `RequestHandler` sends a message it traverses the `GMS` due to the protocol-stack-like architecture of the Neko framework.

Basically there are two cases that the `GMS` takes action on. One is when a node leaves the system and the other is when a node joins the system.

5.12.1 Partition Split

The former of the two above mentioned cases is regarded first. When a node becomes unreachable it may be for several reasons. The two simplest are that for some reason a

node failure has occurred that forces the node to stop operating or that a network failure has occurred that just cuts the node off. In either case the `GMS` would simply remove the node from the system by calling the `removeNode`-method of the `ReplicationMngr`. If the relevant node does not host a primary replica the node is removed from the two Hashtables holding the addresses of the nodes hosting the secondary replicas. Since in an operating network all nodes must have made the same observation, no more measures have to be taken, because it can be assumed that all the nodes have taken the same action.

If the relevant node has hosted a primary replica of an object, another replica has to be promoted to the new primary replica by choosing the secondary replica that is hosted at the node with the lowest available number as the new primary. This is also done within the `removeNode`-method of the `ReplicationMngr`.

Another reason for a partition split may be that it is brought about from outside e.g. because this is an intended feature of an application. This is also the case that is going to be regarded in detail now.

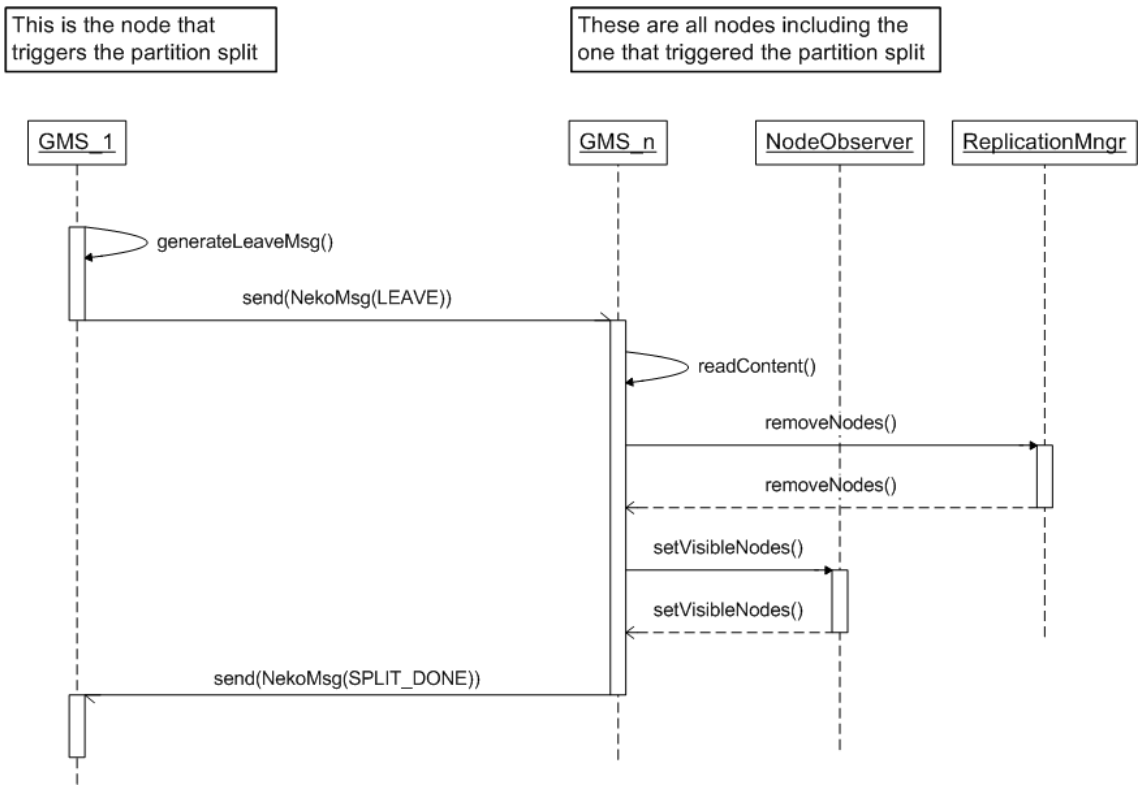

Figure 9: Partition split in detail

Before a partition split is triggered every node has to be assigned to a certain partition. There are several ways how to accomplish this assignment. The one that is chosen because it appears to be the most obvious one is to implement a method in the `SimulationMngr`-class that returns a Hashtable containing the assignments of the individual nodes. This method is called by the `GMS` and two or more NekoMessages are composed. Each one is addressed to the particular set of nodes that form a partition. The content of the message holds a HashSet with all the other node numbers. Before the message is sent off it is typed *LEAVE*. The node that sends off the messages is also included in one of the partitions. Therefore one of the messages is sent to itself.

When a GMS receives a message typed *LEAVE* it reads the HashSet that is wrapped into the message and calls the `removeNode`-method of the `ReplicationMngr` for every node number that is contained in the HashSet. Then the `NodeObserver` is notified about the new view. When this is all done another empty `NekoMessage` typed *SPLIT_DONE* is composed. This message is returned to the node that has triggered the partition split.

When this node has received all *SPLIT_DONE* messages the partition split is fulfilled.

5.12.2 Reconciliation

Although the thought suggests itself that the reconciliation is simply the opposite of a partition split things are somewhat more complicated.

Reconciliation occurs whenever a partition containing at least one node joins another partition. The join of two or more partitions may occur for several reasons. The case that a node recovers from a node internal failure is the simplest one because it can simply be re-added to the view of the other nodes but there does not have to be a reconciliation procedure. The node cannot have any updates because it has not been operating.

The other cases are not as simple because the version lists of the individual nodes have to be compared before it can be decided which versions can be kept and which have to be dropped.

Another reason for reconciliation can be that a network failure has been fixed and that the partition split that has occurred earlier must now be undone.

It is as well possible that the reconciliation of two or more partitions is a feature of the application and that it is therefore triggered from outside. Like for the partition split this is the case that will be simulated and therefore explained in detail.

When the reconciliation of two or more partitions shall be achieved a partition needs to be informed about the address of a node belonging to the new partition.

Thus other than for a partition split no special method is necessary. The reconciliation can be divided into the following steps:

- An arbitrary node launches the reconciliation process by sending the appropriate messages to the nodes of its own partition that host the primary replicas and to any node of the new partition.
- The messages that are sent to the nodes hosting the primary replicas must contain the name or the number of the relevant object. This is because a node might host more than one primary replica. In addition the messages have to be typed *GET_VERSION_LIST*. This is for the receiving GMS to classify the message.
- The receivers of the *GET_VERSION_LIST*-messages send back a copy of the version list of the relevant object and type the message *DELIVER_VERSIONLIST*. It is important to make a shallow copy of the Hashtable that holds the version list. It must be the copy that is wrapped into the message because otherwise it is just a reference to the version list that is sent off which may cause problems in the course of the application. Since the primary replica and the secondary replicas of an object are the same in respect of their content and their version lists the *GET_VERSION_LIST* messages can also be sent to a node hosting a secondary replica but it is more straight forward to always send it to the primary.
- The message that is sent to the node of the second partition is an empty one whose only information is its type namely *GET_FOREIGN_VL*.
- The node in the second partition that receives the message has to send *GET_VERSIONLIST* messages to the nodes in its own partition that host the primary replica of the relevant object.
- Also these nodes send back a copy of the version list of the relevant object and type the message *DELIVER_VERSIONLIST*.
- Whenever either of the nodes that have sent the *GET_VERSIONLIST* messages before receives a *DELIVER_VERSIONLIST* message it stores the version list that is contained in the message.
- The node in the second partition that stores the incoming version lists waits until it has a version list of every object residing in the partition. When the collection

of version lists is complete the node wraps it into a `NekoMessage`. Then it types it *DELIVER_FOREIGN_VL* and sends it to the node that sent the *GET_FOREIGN_VL*-message.

- On the one hand the node that has originally launched the reconciliation process has to wait for the version lists of the objects residing in its own partition. On the other hand it has to wait for the node of the new partition to send the *DELIVER_FOREIGN_VL*-message with the collection of the new version lists.
- When both collections of version lists are complete, the actual reconciliation process is started. The node that launched the reconciliation phase passes on both collections of version lists, the one of its own partition and the one that has come from the new partition to the `ReconciliationMngr`. The `ReconciliationMngr` merges the two version lists depending on the number of nodes residing in a particular partition. (See also section 5.8.)
- When the `ReconciliationMngr` has merged the version lists it has them checked by the constraint consistency manager. (See also section 5.6) After constraint consistency check it returns them to the `GMS`.
- The `GMS` has now to distribute the new version lists to all nodes in view. The set of `NekoMessages` that is returned by the `ReconciliationMngr` has therefore only to be sent off by the `GMS`.
- An important aspect is, to keep track of which nodes have received how many updated version lists. Thus the `GMS` must keep a Hashtable that holds an entry for each node with the number of objects that are hosted by this node.
- Every node receives at least one *DO_JOIN*-message containing a version list that has to be installed. This is done by the `ReconciliationMngr`.
- When the installation of the new version list is finished, the node notifies the sender of the version list by sending a *JOIN_DONE*-message.
- When the node that has sent a *JOIN_DONE*-message from every node that has been sent a *DO_JOIN*-message the reconciliation process is finished.

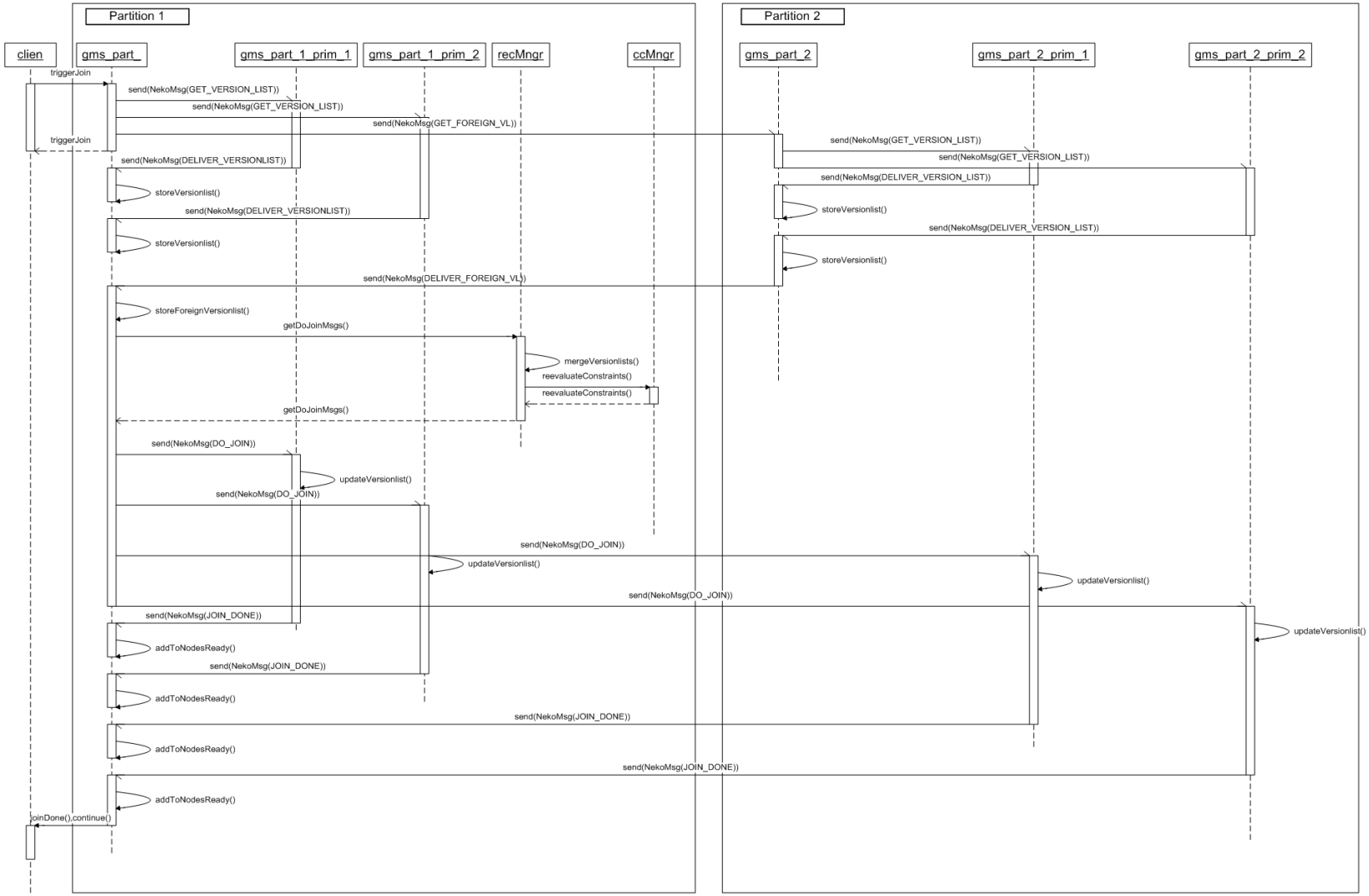

Figure 10: A partition join in detail

The effort that can be recognized by the chart above originates not so much from the complexity of a single operation but rather from the logistic organization that is needed for the reconciliation phase.

But even with this organization it does not produce any major problems.

5.13 UML Chart

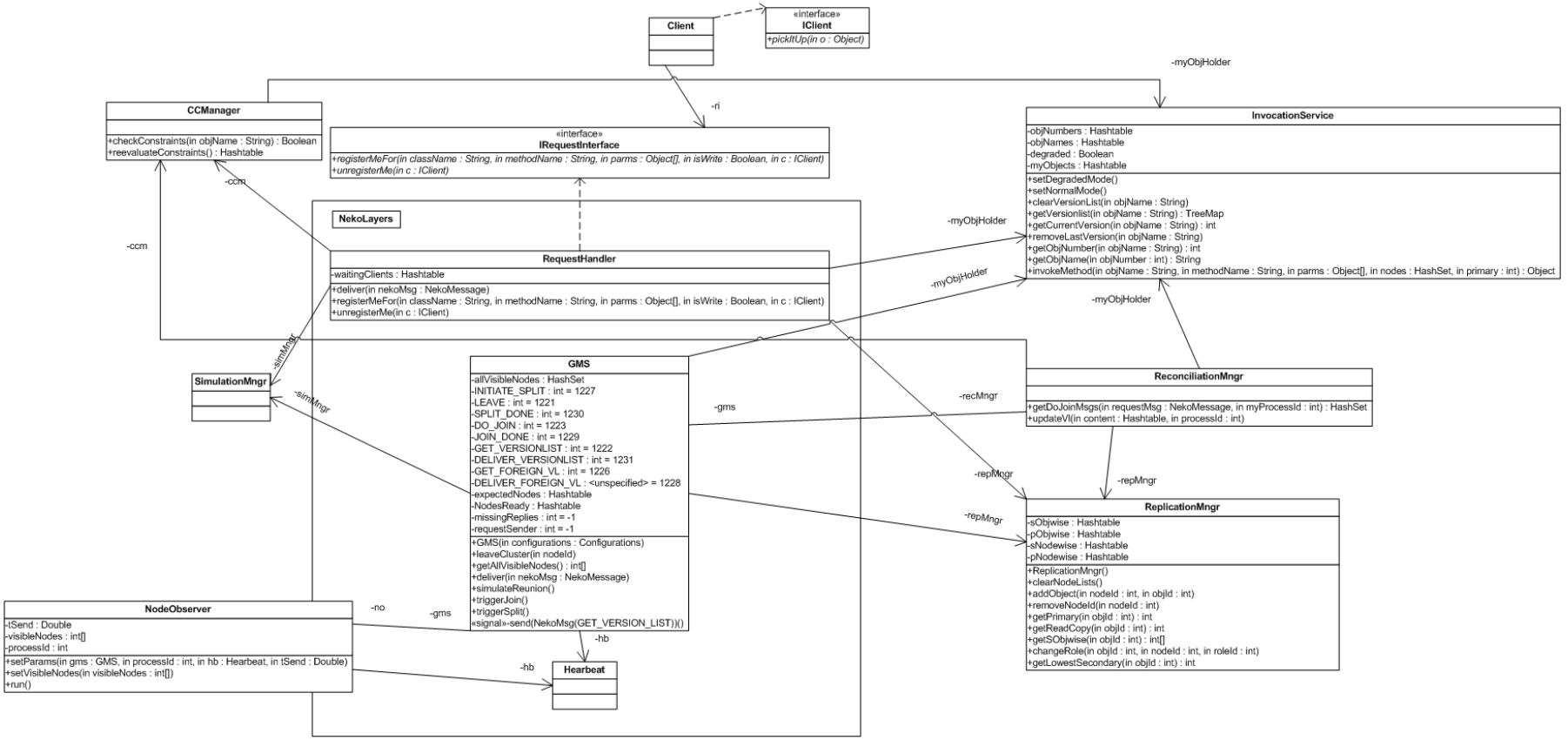

Figure 11: UML chart including the above described components

6 Conclusion

6.1 The Primary-per-Partition protocol

The implementation of the Primary-per-Partition protocol did not pose any major problems and the functionality of the protocol could be fully maintained. In a large part the Primary-per-Partition protocol itself proved to be well thought out and did not cause difficulties. The architecture chosen for the implementation at hand differs only little from the proposals published so far, e.g. in [OFG+05]. Some characteristics of the protocol in conjunction with the implementation are pointed out in the following:

- The functionality of the *Replication Protocol* component contained in [OFG+05] is implemented in the `RequestHandler` component.
- The *Group Communication* component is missing in the implementation at hand because the environment it is integrated with is a simulated one. That is, exogenous disturbances originating from the network are ruled out.
- The architecture of Neko requires to clearly define the components that shall be enabled to send and receive messages by deriving them from the Neko `Layer` class and registering them with a class implementing the `NekoProcessInitializer` Java-interface specified in the Neko framework. In the implementation of the Primary-per-Partition there are two components meeting theses conditions, the `GMS` and the `RequestHandler`. Whenever an incoming message leads to a change of view of a certain node or if measures have to be taken according to the Primary-per-Partition protocol the components that are concerned with it have to be notified by the `GMS` or the `RequestHandler`. To give an example if a view change occurs due to a node failure the relevant method of the `ReplicationMngr` for removing the node from the current partition has to be called by the `GMS`. Due to this role as a central element of the implementation of the Primary-per-Partition protocol the `GMS` has been extended compared to the [OFG+05] proposal. Not only does it provide methods for adding a new node to the current view or removing a faulty one but it also is in charge of the coordination in case of a partition split or a reconciliation phase. The `GMS` inspects every incoming message and decides whether it has to take any further steps itself or if another component has to be charged with it. Considering these circumstances the relatively vast extent of the

implementation of the `GMS` becomes comprehensible and the `ReconciliationMngr` and the `CCManager` can be regarded as helper components of the `GMS`.

- The Primary-per-Partition protocol says that constraint consistency has to be ensured at all times in a healthy system and that it may be relaxed in degraded mode under certain conditions but the protocol does not contain any exact specification of how constraint consistency checking has to be implemented. So the design of integrity constraints is up to the developers of a certain application and may vary widely from case to case. On the one hand this freedom in the design of constraint consistency appears to produce extra work because the implementation may rarely be the same. On the other hand this freedom leaves room for integrity checks that may be exactly coordinated with the needs of a client application. It is possible to put more emphasis on parts where integrity has to be maintained and to relax it where it is not necessarily needed at all times.

  In the implementation at hand constraint consistency evaluation is simulated based on random variables. This does not satisfy the needs of any real client application of course but it allows easy simulations of the behavior of a client application undergoing a certain degree of integrity which can be defined by altering the according parameters.

- The Primary-per-Partition protocol does neither specify which secondary has to be promoted to the new primary in case the original primary fails or gets cut off nor does it give any rule on how a node is to be chosen for serving a read-method request. Therefore the choice of these nodes can be made according to load balancing requirements without violating the functionality of the Primary-per-Partition protocol.

6.2 Differences between the Implementation and the Specification

To round the previous section off, the differences between the implementation and the actual specifications of the Primary-per-Partition protocol shall be outlined again:

Primary-per-Partition Protocol Specifications	**Implementation of the Primary-per-Partition Protocol**
The node that holds the primary replica always regains this role after a partition split is fixed and the system has accomplished the reconciliation process.	The node that has held the primary replica originally may be replaced with a new one during reconciliation. This is because the version list of the object that is located in the bigger partition is the one that is installed during reconciliation. Since all nodes have the same privileges this does not have any negative impact on the functionality of the protocol.
The way constraint consistency checking shall be implemented is not specified in the Primary-per-Partition protocol. The protocol envisions an implementation that is closely tied to the needs of the individual client application.	Constraint consistency checking is based on random variables. Constraint evaluation is configured by defining the parameters specifying the probability that a certain constraint evaluation succeeds with. That way any thinkable situation may easily be simulated.
According to the Primary-per-Partition protocol nested invocations are supported. No further specification of this requirement is given.	In the implementation at hand nested invocations are closely tied to the implementation of the objects whose methods are called in a remote-controlled way by a client or another node. Therefore a potential client application will have to deal with the above requirement when implementing its objects. Hence the idea of keeping the objects as generic as possible is foiled a little.

Table 10: Differences between the specifications of the Primary-per-Partition Protocol and the implementation

6.3 Neko

Neko turned out to be a very handy framework for simulating any kind of distributed algorithm. During the implementation no problems occurred that had to be ascribed to

the Neko framework. The most remarkable characteristics are pointed out in the following:

- Neko is based on the implementation of a protocol stack. Every protocol represents an individual layer of the stack. A protocol can be implemented in any arbitrary way and may have access to all other components of an application integrated with the Neko framework. That is an application may consist of an arbitrary number of Java classes but only the components that are part of the protocol stack can send and deliver messages. This is an architecture that offers an easy to use gateway to the Neko framework and leaves room for the implementation of extensions e.g. by simply placing another layer on the stack.
- Another remarkable feature of the architecture of the Neko framework is the following: In the simulated mode nodes are generated by instantiating the relevant application as often as there are nodes. That is, every instance of the application represents a node of the simulated environment. In other words every component has to contain all the code needed for all different kind of nodes, e.g. client nodes as well as server nodes. The application has to make sure the right code is executed e.g. by requesting the virtual or real id of a node.

  This is an approach that eases file handling and configuration management because even in a real environment the implementation is the same at every node. If the source code is modified the same copy has to be distributed to all the nodes. A slight disadvantage is that the relevant Java classes may become crowded if complex implementations of client and server nodes are put into practice. Therefore it may be hard to keep track of where the individual parts of the source code are located.
- Neko applications may be executed in a real environment as well as in a simulated one without having to change the source code. The configuration of whether the application at hand is executed in a real environment or a simulated one is made in a separate configuration file. Therefore the source code does not even have to be re-compiled if the environment is switched. It is sufficient to add the physical addresses of the nodes to the configuration file and to set a relevant flag if the application is to be run in a real network.
- Neko proved to be well-developed, stable and qualified for the use as a reliable middleware. During the course of the implementation of the Primary-per-Partition protocol no problems occurred that had to be ascribed to the Neko framework.

- The whole setup information for each Neko application is kept in the form of key-value-pairs and stored in a configuration file. The configuration file holds on the one hand information that is needed by Neko in order to run the application. This concerns information e.g. on whether an application shall be run in a simulated environment or a real one. On the other hand developers of Neko applications are free to store their own variables in the configuration file. That way, e.g. parameters can be varied without having to change the source code.
- Although Neko performed well concerning the allocation of a simulated environment for a distributed system no conclusions can be drawn about characteristics regarding its potential qualification as a middleware for the go-live phase of an application. There have not been any experiences e.g. with its long run characteristics or with the behavior under high workload.
- The need for a useful documentation of the Neko framework is urgent. The basics are explained sufficiently on the relevant web site[6] so some simple example applications can be well understood. But the whole package comes with implementations where the efficiency of Neko is demonstrated and any documentation aside from some rather poorly documented API is missing. Some of the above mentioned implementations may be very useful for putting into practice similar applications but they are hard to understand without the appropriate documentation.

Shortly before the thesis at hand was finished a version 1.0 of the Neko framework was released[7] which contained changes in the architecture and which was not compatible with applications integrated with older versions of the framework.

Therefore comments on Neko within this thesis correspond to version 0.9.

[6] http://ddsg.jaist.ac.jp/~urbi/neko/index.shtml

[7] http://ddsg.jaist.ac.jp/neko/

7 Future Work

7.1 Group Communication[8]

Most of the proposals for a system architecture for an implementation of the Primary-per-Partition protocol contain both, a group communication component and the group membership service [OFG+05]. Both of these two components are closely connected with each other. At the first glance the tasks of a group membership service appear a little less complex than the ones of the group communication component. A group membership service is responsible for managing the membership of groups of processes. It reports any change of membership to all group members by providing them with the new view.

The group communication component has to make sure that a message is delivered to either all processes or to none at all. This is a requirement that can be better understood considering the following circumstance: When a process multicasts an update and one of the receiving processes crashes during the execution of the update the update is lost for the relevant process but correctly performed by all other processes. In the course of the further execution of the application the crashed process may recover to the same state that it had before the crash. However, it may have missed several updates. Before the recovering process can now be re-integrated with the remaining processes that have been continuing to operate properly it has to be brought up to date. Therefore it is required to determine which operations have been missed by the recovering process. A reliable determining of these operations is only possible if all processes agree on the current group view at all times, that is, an update operation is always performed either by all nonfaulty processes or by none at all. A special case would be that the sender of a multicast message crashes while the message is in transit. Either all remaining processes would have to deliver the message then or none at all. This is the only case where the delivery of a message is allowed to fail. A multicast implementation restricted to the above described conditions is called virtually synchronous.

In addition to virtual synchrony the group communication component has to support FIFO ordered multicast, that is, messages from the same process have to be delivered in

[8] Based on [TS02]

the same order as they have been sent at every receiving process. Messages sent by different processes may also be delivered in different orders, though.

The implementation within the scope of the thesis contains only a group membership service and not a group communication component. This is sufficient for testing the behavior of the Primary-per-Partition protocol to the exclusion of all exogenous influences. Any network weakness e.g. a partition split has also to be simulated. In a real environment the implementation would run the risk of operating faultily e.g. due to a potential loss of the reliable FIFO ordered multicast support. Therefore the extension of the current implementation by a group communication component appears useful.

7.2 Integration with an Application

The implementation of the Primary-per-Partition protocol described so far is done independently of a client application. This is an approach that holds several advantages:

- Most of the results of the implementation can be regarded as universally valid with respect to a potential integration with an application. Any future integration with an application will maintain the full functionality of the protocol.
- In some parts the implementation could be simplified, e.g. concerning the testing of the gateway for a client application. Since no regard to the needs of a real application had to be paid it was possible to choose a supposably simple substitutional implementation of an application. Therefore it was also possible e.g. to keep the objects small and easy to use and to avoid complications that might origin from protocol-independent complications.

Extensions to the current implementation will be needed when the Primary-per-Partition protocol shall be tested with a real application, though. The lack of these extensions caused some difficulties that came along with the stand alone implementation of the protocol, that is, the role of an application using the service of the protocol was missing. Primarily, future extensions will have to be adapted to the requirements of the application. Some extensions, however, might be unavoidable for almost any application:

- Java offers some data structures that are very convenient e.g. for implementing a version list or for keeping track of missing replies. They work fine in a simulated environment where the amount of data remains manageable and messages contain only placeholders for real objects. However a real application may produce a much larger amount of data that cannot be handled with the

known Java data structures and will therefore demand for expandable and persistent data storage. Some kind of database solution is unavoidable in this case.

- Constraint consistency checking based on random variables will have to be replaced with constraint consistency checks with respect to the application. The way the constraint consistency manager is implemented offers on the one hand large room for tests and experiments. Any imaginable situation concerning the specification of constraints can be simulated by simply adjusting a couple of parameters. On the other hand the application-specific behavior of the Primary-per-Partition protocol can only be tested within the context of the application itself. It cannot be completely ruled out that e.g. the specification of application-dependent integrity constraints might also have unpredictable side effects on the behavior of the Primary-per-Partition protocol that cannot be simulated within the implementation at hand.

7.3 Further Simulations

The simulation and the evaluation of the Primary-per-Partition protocol have to be seen in conjunction with the environment that they are run in. On the one hand there is the environment used for the implementation at hand where all exogenous influences are factored out. That way e.g. the transmission time of a message between two processes is kept constant and node or network failures are ruled out. This can be very helpful if influences of the network on the behavior of the protocol shall be avoided and the implementation e.g. of the above described group communication component is skipped.

On the other hand an implementation has to be able to deal with all circumstances brought about by an error-prone network. So the behavior of the protocol in view of the real network which it is designed for remains still to be examined. One way to carry out this examination would be to run the protocol directly in a real network. Although Neko supports this approach it would be a laborious one for various reasons:

- The relevant test stage would bring about the need for a real network. The setup of this network as well as the setup of the individual nodes and respectively the PCs would be connected with a lot of installation works and would consume a lot of time and possibly become a rather expensive affair.
- Although the Primary-per-Partition protocol would be exposed to the weaknesses of a real network it can not be guaranteed that every network

weakness occurs when it is needed to get information about the relevant potential weakness of the protocol itself. Thus some weaknesses of the network would have to be provoked somehow at a certain point of the test stage e.g. by disconnecting a part of the network. In other words even in a real network some weaknesses would have to be simulated.

Therefore a more reasonable approach appears to not only simulate the behavior of the protocol in a failsafe network but also to simulate the weaknesses of a real network with the help of Neko. Neko provides some features that ease the implementation of such an error-prone network:

- The network layer is completely uncoupled from all other Neko layers.
- The Neko package comes with some helper implementations such as the `CrashEmulator` that allow an uncomplicated simulation of a network failure.

In order to tell the difference between the effects of the Primary-per-Partition protocol and the implementation of the simulated network weaknesses it would be a good idea to completely separate these two parts of the implementation which is easily possible thanks to the protocol-stack-like architecture of Neko.

8 References

8.1 Literature

[BBG+05] Stefan Beyer, Mari-Carmen Banuls, Pablo Galdamez, Johannes Osrael, and Francesc D. Munoz-Escoi: Increasing Availability in a Replicated Partitionable Distributed Object System, 2005, submitted to European Dependable Computing Conference 2006.

[BSM+05] DeDiSys Lite: An Environment for Evaluating Replication Protocols in Partitionable Distributed Object Systems, Stefan Beyer, Alexander Sanchez, Francesc D. Munoz-Escoi and Pablo Galdamez, 2006, to appear in Proc. of the 1st Int. Conf. on Availability, Reliability, and Security (ARES 2006)

[CDK01] George Coulouris, Jean Dollimore, Tim Kindberg: Replication, Chapter 14 in *Distributed Systems*, ISBN 0201-61918-0, Third Edition, Pearson Education Ltd, 2001

[D1.1.1] *D1.1.1 Trade-off: Availability – Consistency*, Deliverable 1.1.1, Rev 2.0, 2005, downloadable from http://www.dedisys.org

[D1.2.1] *D1.2.1 Hybrid Replication Protocols*, Deliverable 1.2.1, Rev 2.0, 2005, downloadable from http://www.dedisys.org

[DSU04] X. Défago, A. Schiper, P. Urbán: *Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR), vol. 36, no. 4,* December 2004.

[Goe05] Karl M. Goeschka:, *Increasing Availability by Sacrificing Data Integrity – a Problem Statement*, Proceedings of the 38th Hawaii International Conference on System Sciences – 2005, downloadable from http://csdl2.computer.org/comp/proceedings/hicss/2005/2268/09/22680291c.pdf

[HK03] Martin Hitz, Gerti Kappel: *UML @ Work*, 2nd edition, 2003, dpunkt.verlag GmbH, ISBN: 3-89864-194-5

[HV00] Haifeng Yu, Amin Vahdat: *Design and Evaluation of a Continuous Consistency Model for Replicated Services*, Computer Science Department, Duke University Durham, NC 27708, 2000 {yhf,vahdat}@cs.duke.edu,

http://www.cs.duke.edu/~vahdat/ps/tact_osdi.pdf

[MGG95] Mazouni K., Garbinato B., Guerraoui R.: *Building Reliable Client-Server Software Using Actively Replicated Objects.* In Graham I., Magnusson B., Meyer B., Nerson J.-M. (eds.), *Technology of Object Oriented Languages and Systems,* Englewood Cliffs, NJ: Prentice Hall 1995

[OFG+05] Johannes Osrael, Lorenz Froihofer, Karl M. Goeschka, Stefan Beyer, Francesc Munoz, Pablo Galdamez, Alexander Sanchez: *A System Architecture for Enhanced Availability of Strongly Coupled Distributed Systems*, 2006, to appear in Proc. of the 1st Int. Conf. on Availability, Reliability, and Security (ARES 2006)

[SG03] Robert Smeikal, Karl M. Goeschka: *Fault-tolerance in a Distributed Management System*: a Case Study, In: Proceedings of the IEEE/ACM International Conference on Software Engineering, 2003, S. 478

[TS02] Andrew S. Tanenbaum, Maarten Van Steen: Consistency and Replication (Chapter 6) and Fault Taulerance, (Chapter 7) in *Distributed Systems, Principles And Paradigms,* ISBN 0-13-121786-0, Prentice-Hall, Inc., 2002

[Urb01] Péter Urbán e. a.: *Neko: A Single Environment to Simulate and Prototype Distributed Algorithms,* 2001. IEEE. in Proc. of the Int'l Conference on Information Networking (ICOIN-15), pages 503-511; downloadable from http://lsewww.epfl.ch/Documents/acrobat/UDS01.pdf

8.2 Figures

Figure 1: Operation transfer vs. state transfer 13

Figure 2: Basics of the Neko Architecture [Urb01] 22

Figure 3: Passive Layers implement the `LayerInterface` [Urb01] 23

Figure 4: Active Layers have their own thread of control [Urb01] 23

Figure 5: Overview of components to implement the Primary-per-Partition protocol .. 31

Figure 6: Thread based request handling 42

Figure 7: Request handling via a callback handler 44

Figure 8: Handling a remote write method call in detail 47

Figure 9: Partition split in detail 55

Figure 10: A partition join in detail 59

Figure 11: UML chart including the above described components 61

8.3 Tables

Table 1: Consistency models not using synchronization operations [TS02] 7

Table 2: Version list with three entries in degraded mode 37

Table 3: Configuration of constraint evaluation 40

Table 4: Version list being checked by the constraint consistency manager 40

Table 5: Version list that is going to be installed at the current node 50

Table 6: Version list that is currently installed 50

Table 7: Version list with only the first entry left 51

Table 8: Version list where version 1 has been generated anew 51

Table 9: Old version list after having been completely updated 51

Table 10: Differences between the specifications of the Primary-per-Partition Protocol and the implementation 64