

**Vrije Universiteit**

Computer Science Department

**Ana-Maria Oprescu**

aoprescu@few.vu.nl

Student Number: 1488562

**Replicated Method Invocation  
with Matrix Timestamps**

Master Thesis in Parallel and Distributed Computer Systems

*Master Thesis Supervisors*

**Dr.-Ing. habil. Thilo Kielmann**

**Dr. Jason Maassen**

*Dept. of Computer Science*

Amsterdam, August 2006



## **Abstract**

*We propose a replicated method invocation system for grids, based on Lamport Timestamps Matrix, using an open world protocol. We have built a prototype of the system on top of Ibis Portability Layer which we compare against an approach that uses a central Ticketing Service.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Replicated Method Invocation</b>	<b>5</b>
<b>3</b>	<b>Lamport Timestamps</b>	<b>8</b>
3.1	Matrix Timestamps . . . . .	8
3.1.1	Matrix Time and RepMI . . . . .	9
3.2	Protocol . . . . .	10
3.2.1	Execution . . . . .	11
3.2.2	Heart-Beat messages . . . . .	12
3.2.3	Open World . . . . .	12
3.2.4	A small example . . . . .	14
<b>4</b>	<b>Implementation</b>	<b>16</b>
4.1	Ibis Portability Layer . . . . .	16
4.2	Technical details . . . . .	16
<b>5</b>	<b>Evaluation</b>	<b>19</b>
5.1	Sequencer version . . . . .	19
5.2	Measurements . . . . .	19
5.2.1	OneToMany, Intra-cluster . . . . .	20
5.2.2	ManyToOne, Intra-cluster . . . . .	22
5.2.3	OneToMany, Inter-cluster . . . . .	22
5.2.4	ManyToOne, Inter-cluster . . . . .	24
<b>6</b>	<b>Conclusion</b>	<b>26</b>

# Chapter 1

## Introduction

Object-based programming models have become popular for writing parallel applications for clusters and grids [10]. In these models, communication between processes is based on executing methods on shared objects. In the simplest case, these shared objects are located at a single process, accessible (in Java) via Remote Method Invocation (RMI). This approach is easy to understand and to implement, but it has severe performance limitations, esp. in grid systems where network communication can be slow due to long wide-area latencies.

Object replication is a well-known technique to improve the performance of parallel object-based applications [1]. The underlying assumption is that shared objects are read much more frequently than they are written (or modified). With object replication, read operations become local operations while write operations need to be performed, properly synchronized, on all object replicas.

In [6], a technique called Replicated Message Invocation (RepMI) had been introduced for the Java language. Here, write operations are performed using a technique called *function shipping*: the method invocations, along with the parameters, are sent to all object replicas, where they get executed in a globally unique order. That work was based on the Manta [7] compiler for Java, limiting its broad applicability to heterogeneous platforms like grids. Another limiting factor is the mechanism by which the replicated objects are kept in a consistent state: a centralized *sequencer* node is ensuring a global execution order for all write operations on the replicated objects.

In this thesis, we are presenting a new implementation of RepMI, tailored for grid environments. It improves over the work in [6] by the following features:

1. It improves the performance of write operations by replacing the centralized sequencer node by a completely distributed consistency mechanism, based on Lamport matrix time.
2. It implements an *open world* system, in which processes can join and leave the ongoing parallel computation, being properly included in and removed from the object replica synchronization mechanism.

3. It is a pure Java implementation, using special marker interfaces, and byte-code rewriting for generating the necessary code for invoking write methods on the set of replicas.

This thesis is structured as follows. Chapter 2 introduces replicated method invocation. Chapter 3 outlines Lamport matrix timestamps, and how they can be used to synchronize write operations on replicated objects. Chapter 4 describes our implementation in the Ibis system. Chapter 5 compares the new mechanism to a sequencer-based implementation. Chapter 6 concludes and outlines directions of future work.

## Chapter 2

# Replicated Method Invocation

An interesting spin-off of Remote Method Invocation is Replicated Method Invocation, where the remote object is actually a set of replicas living on different machines spread over a network and represented by a logical object. However, some limitations apply in this case, as complicated dependency patterns between different logical objects may not be supported [5]. All hope is not lost, as there are applications which could require, for performance reasons, a replicated system, but their dependency pattern is very simple (i.e. only one logical object is used by the application). Most often than not, the main requirement of a replicated system is to maintain its replicas consistent. There are many types of consistency and the best one is always relative w.r.t the application using the replicated system.

We aim at building such a replicated system which addresses applications with a need for sequential consistency, that is [2]

”...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

One way to achieve this goal is to use a central service to enforce the global sequential order. It could also be enforced in a distributed fashion, using timestamps. Both approaches could have advantages and disadvantages w.r.t. the type of applications using the replicated system.

It is also important to separate methods that need a write-access to the object from those that only require a read-access. An immediate remark is that invocations of read-access methods need not be considered when ordering operations. They can be locally processed to wait at least until the last local invocation of a write-access method they have seen has been executed.

**Our solution** Our replicated system offers a simple API of basically two methods with the following semantics:

- `processLocalRead(ReplicatedMethod read)` - will block the calling thread until the preceding local write-access method invocation has been executed;
- `processLocalWrite(ReplicatedMethod write)` - will return as soon as the message containing the write-access method invocation is broadcasted throughout the replicated system.

Application built on top of this replicated system need to determine on their own the nature of an invocation: whether is a read-access or a write-access method invocation. Based on the result of this decision, it can then call the corresponding API method and have the invocation processed by the underlying replicated system.

**A small example** We have imagined a simple remote method invocation system on top of our replicated system, where messages sent between replicas contain method invocations on the replicated object. We used the following toy application of a replicated bank account (see Figure 2.2 and Figure 2.1), where `Replicateable.java` is just a marker interface, to indicate that this object should be replicated, hence its methods should be invoked through our application. The `ReplicatedAccountInterface` indicates which methods require write-access.

```

package repmi.test;

import repmi.protocol.Replicateable;

public interface ReplicatedAccountInterface extends Replicateable {

    public void writeMultiplication(Integer times);
    public void writeAddition(Integer quantity);
}

```

Figure 2.1: ReplicatedAccountInterface.java

```

package repmi.test;

import java.io.Serializable;

public class ReplicatedAccount implements
ReplicatedAccountInterface, Serializable {

    long val = 0;

    public void writeMultiplication(Integer times) {

        val *= times.intValue();
    }

    public void writeAddition(Integer quantity) {

        val += quantity.intValue();
    }

    public Long readVal() {

        System.out.println("readVal: " + val);
        return new Long(val);
    }
}

```

Figure 2.2: ReplicatedAccount.java

## Chapter 3

# Lamport Timestamps

In a distributed world it can be a very hairy business to maintain a global clock system. Moreover, such a system is not always a must for certain types of applications; for instance, in order to maintain a set of concurrently accessed bank accounts in a consistent state only a relative order of operations is sufficient. Lamport Timestamps have been designed with this goal in mind, for applications where the actual time is not of interest, but the relative order of actions is essential to be the same on every entity of the distributed world. Basically, Lamport Timestamps come in three flavours: simple timestamps, vector timestamps and matrix timestamps. Further information about this can be found in [8]. We proceed to present matrix timestamps, which we use in our protocol. Note that we use "entity" to refer to a machine/node/process that is part of the DW.

### 3.1 Matrix Timestamps

The idea behind matrix timestamps is quite simple: each entity A of the distributed world (DW) maintains a matrix of timestamps M, where each row ( $M_A[B,-]$ ) describes the knowledge this entity A thinks another entity B has over every entity of the DW:

- $M_A[A,A]$  - local logical clock of this entity;
- $M_A[A,B]$  - latest logical clock value of B seen by A;
- $M_A[B,C]$  - latest logical clock value of C seen by B, as far as A knows.

Basically, on entity A the column  $M_A[-,B]$  of the matrix indicates what information A has received about how many of the actions of B each entity in the system has seen. In a replication system, based on a combination of this information for all entities (e.g. rows), we can decide how many of the actions currently seen by this entity we can apply on the replicated object.

Updating the local logical clock of A follows two rules:

- R1.** event execution is preceded by a d-step increment on its logical clock:  
 $M_A[A,A] = M_A[A,A] + d, d > 0;$
- R2.** piggyback each message with  $M_A$ ; when receiving a (m,  $M_B$ ) message, take the following actions:
1. update global time (represented by  $M_A$ ):  
 $C \in \Sigma, M_A[A,C] := \max(M_A[A,C], M_B[B,C]);$   
 $C \in \Sigma, D \in \Sigma, M_A[C,D] := \max(M_A[C,D], M_B[C,D]);$   
 where  $\Sigma =$  the entities of the DW;
  2. execute R1;
  3. deliver message m;

### 3.1.1 Matrix Time and RepMI

As further explained in [?], matrix timestamps were employed to discard obsolete information in replicated databases. But in discarding obsolete information, the order in which this information is discarded once it has been marked for deletion is not important; as we already established, order is important for RepMI if we want to maintain sequential consistency. What is not important for sequential consistency is to determine causality between operations issued by different processes. It follows that we can use matrix timestamps for replicated objects, but with some changes to the semantics of the matrix time entries:

- $M_A[A,A]$  - timestamp of last write-access invocation issued by this process;
- $M_A[A,B]$  - timestamp of latest write-access invocation issued by B and seen by A;
- $M_A[B,C]$  - timestamp of latest write-access invocation issued by C and seen by B, as far as A knows.

and the updating rules:

- R1.** write-access invocation buffering is preceded by a d-step increment on its logical clock:  $M_A[A,A] = M_A[A,A] + d, d > 0;$
- R2.** piggyback each broadcast message containing a write-access invocation with  $M_A$ ; when receiving a (write-access invocation,  $M_B$ ) message, take the following actions:
1. update global time (represented by  $M_A$ ):  
 $M_A[A,B] := M_B[B,B];$   
 $C \in \Sigma \setminus \{A\}, D \in \Sigma, M_A[C,D] := \max(M_A[C,D], M_B[C,D]);$   
 where  $\Sigma =$  the entities of the DW;
  2. buffer the write-access invocation;

It then follows that each write-access invocation is uniquely identified throughout the DW by two coordinates: the unique identifier of issuing process and the timestamp given by the issuing process. Adding an order relationship over process identifiers, we obtain an order relationship over write-access invocations.

## 3.2 Protocol

In this section we describe our protocol designed to maintain a sequentially consistent view in a DW over a set of so-called operations, which can be simply thought of as messages. Each entity of the DW can generate:

**”write” operations** which need to be propagated to all other entities present in the DW

**”read” operations** which are resolved locally.

One important point in our design is to uniquely identify w.r.t DW each ”write” operation and to establish an ordering over their unique identifiers. We achieve this goal locally (i.e. without using an external service) by timestamping each ”write” operation at its source and then release it in the DW with an extra ”birth mark” which is the unique identifier of the source.

It is also very important to state the communication model we assume throughout our protocol: broadcasts blocks sending entity until its message reaches all other entities in the DW. We identify three types of operations that an entity has to perform: a ”read” operation (LR), a ”local write” operation (LW) and a ”remote write” operation (RW). LWs and RWs will be buffered until their proper DW time of execution has arrived. For each of these types we define a procedure in our protocol:

**processLocalRead** - blocks until the last local write processed by the application has been executed;

**processLocalWrite** - buffers this operation, based on its unique identifier; increments the local Lamport time; broadcasts the operation to all other entities in the DW, together with current Lamport Timestamp Matrix (LTM).

**processRemoteWrite** - buffers this operation, based on its unique identifier; updates the local LTM using the LTM that came with the message. Note that there is no need to tick the local logical clock.

**updateLamportMatrix** - entity A received a message from entity B; it uses  $LTM_B$  to ”peak” at the future, assuming B knows more about some of the other entities in the DW, as described in 3.1.1.

By DW time we mean the relative time of the system, given by the ordered set of operation unique identifiers. At any point in the real world time, each entity can compute a minimum Lamport Timestamp (LT) which guarantees

that operations generated in the DW before that time have been received and buffered locally. These operations can be safely executed on the local replica. The minimum LT is computed by entity E locally using the information present in the local LTM, as follows:

$$\min_{A \in \Sigma} LTM_E[E, A],$$

where  $\Sigma$  = the entities of the DW; E = this entity. This basically means the minimum over the values seen by E of all logical clocks in the DW.

### 3.2.1 Execution

We now proceed to explain how operations are executed in the same order throughout the DW. As we already mentioned, each LW or RW operation is buffered locally. The buffering has two phases: "not yet ready" for execution, as there are still interleaving operations to arrive locally, and "ready" for execution, as no entity can send any operation that should be executed with this set. The decision is made by comparing the DW unique identifier of the operation (hence used as a global timestamp) with the current local minimum LT. The execution mechanism is kept separate from the ordering mechanism, a design decision which allows the underlying open world platform to be used with different ordering mechanisms, as seen in 5.1. Let's assume there are two processes in the DW, P1 and P2, P2 just processed a LW and sent a corresponding message to P1; each now have the following state (matrix have been described in 3.1.1):

$$\begin{aligned} \mathbf{P1\ matrix} & \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} \\ \mathbf{operation\ buffer} & \quad op.TS = P2 - 2 \end{aligned}$$

$$\begin{aligned} \mathbf{P2\ matrix} & \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} \\ \mathbf{operation\ buffer} & \quad op.TS = P2 - 2 \end{aligned}$$

But none of P1 or P2 is able to execute this operation, as their limit TS is still 1. Assume P1 also wanted to process a LW, but due to its internal synchronization mechanisms, this LW will be processed after state has been updated following the message received from P2. P1's state will now be:

$$\begin{aligned} \mathbf{matrix} & \begin{pmatrix} 2 & 2 \\ 1 & 2 \end{pmatrix} \\ \mathbf{operation\ buffer} & \quad op.TS = P1 - 2 \quad op.TS = P2 - 2 \end{aligned}$$

and a message is sent to P2 with the current matrix and the LW. P2's state becomes:

$$\begin{aligned} \mathbf{matrix} & \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \\ \mathbf{operation\ buffer} & \quad op.TS = P1 - 2 \quad op.TS = P2 - 2 \end{aligned}$$

. As both P1 and P2 compute a limit TS of 2, they are both now able to execute the operations from their buffers, and in the same order. Note that if P1 would not have a LW to process, none of P1 or P2 would execute the operation buffer. P2 cannot execute  $op.TS = P2 - 2$  right away as there could be a message from P1 containing  $op.TS = P1 - 2$  still traveling through the DW and executing  $op.TS = P2 - 2$  before  $op.TS = P1 - 2$  would validate the global ordering over operations. On the other hand, we should not wait forever for P1 to have a LW for processing. A solution to this problem are heart-beat messages.

### 3.2.2 Heart-Beat messages

Especially as network delay may vary in time, we cannot assume that a node being silent for a longer time has not generated write operations on its local replica and the corresponding messages are not still traveling through the DW. This is the reason for which we need heart-beat messages, indicating that no activity has taken place on the issuing node, thus allowing the DW to continue execution. As we shall see in Chapter 5, the periodicity of such messages can dramatically influence the overall performance of our application. Currently, we use a statically defined period to issue such messages, independent of how many LW this node has issued. These heart-beat messages are represented as special operations: NOPEs.

### 3.2.3 Open World

In order to allow for an open world protocol, where entities are free to join and leave the DW at all times, two special operations have been introduced: JOIN and LEAVE.

An entity NC wishing to participate in this DW can contact a Discovery Service which would indicate an entity A of the DW to be further contacted. Entity A would take the join request and handle it as being a local operation, released in the DW. When A cannot take the join request, the NC will observe it as a timeout and decide to contact another entity in the DW. On execution of this operation (see Figure 3.1), special messages are sent to NC by all members of DW, containing their local operations which have been buffered along with their current LTM, and also the state of the object, in the case of entity A. The global ordering of operations guarantees that the JOIN operation is executed by all entities at the same DW time, hence the state of the object is the same throughout the DW. All local operations will be ordered on NC side, leading to a consistent snapshot of the DW. Before starting its life in the DW, NC will also process all received LTMs.

An entity L wishing to leave the DW will simply release a LEAVE operation in the form of a local operation, and would further refuse joining requests or LW. It will actually leave the DW when executing the LEAVE operation from its execution queue.

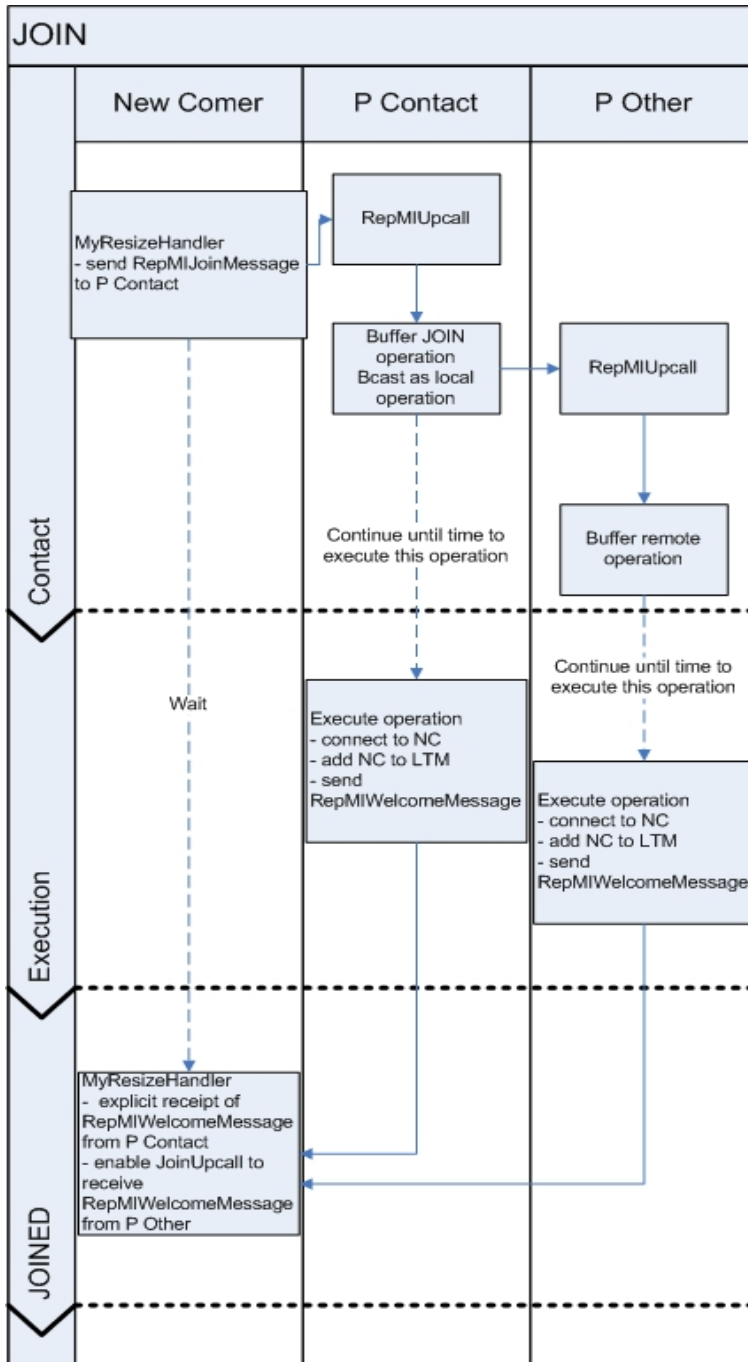


Figure 3.1: JOIN. A new node (NC) joining the DW.

### 3.2.4 A small example

We illustrate our Lamport-based approach with a simple example that would finally involve three nodes. Let P1 be the first node in the DW. Its matrix will look as follows:  $\begin{pmatrix} 0 & \\ & \end{pmatrix}$

Lets assume that next P2 would like to join this one-node DW. It will contact P1 as seen in Figure 3.1 and P1's matrix will become:  $\begin{pmatrix} 1 & \\ & \end{pmatrix}$

with a JOIN operation waiting in the buffer:  $op.TS = P1 - 1$

as the limit TS is now updated to 1, this operation will be executed, hence P1's matrix will become:  $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$  and will be sent to P2, together with the state of the replicated object and with the operation buffer (in this case, empty).

P2 is now part of the DW. Its matrix is  $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ . Assume now P1 has to

process a LW operation; its matrix becomes:  $\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$ , its operation buffer:

$op.TS = P1 - 2$  and the operation together with the current matrix are sent to P2; as a result P2 updates its matrix to:  $\begin{pmatrix} 2 & 1 \\ 2 & 1 \end{pmatrix}$  and its operation buffer

becomes:  $op.TS = P1 - 2$ . But none of P1 or P2 is able to execute this operation, as their limit TS is still 1. Let P2 generate a NOPE; its matrix becomes:

$\begin{pmatrix} 2 & 1 \\ 2 & 2 \end{pmatrix}$  and its operation buffer:  $op.TS = P1 - 2$   $op.TS = P2 - 2$  and

it is sent to P1 together with the NOPE operation; P2 can now execute both operations from its operation buffer as its limit TS becomes 2. On receiving this message, P1 updates its matrix to  $\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$  and its operation buffer to

$op.TS = P1 - 2$   $op.TS = P2 - 2$ ; P1 is now able to execute both operations from its operation buffer, as its limit TS becomes 2. Lets assume now that P3

would like to join this DW. It will contact for instance P1. P1's matrix will become:  $\begin{pmatrix} 3 & 2 \\ 2 & 2 \end{pmatrix}$  and its operation buffer will now contain:  $op.TS = P1 - 3$ ;

the JOIN request is then sent as local operation over to P2 accompanied by P1's matrix. P2 will update its matrix to  $\begin{pmatrix} 3 & 2 \\ 3 & 2 \end{pmatrix}$  and its operation buffer will

now contain:  $op.TS = P1 - 3$ . Assume P2 generates now a LW; its matrix becomes:  $\begin{pmatrix} 3 & 2 \\ 3 & 3 \end{pmatrix}$  and its operation buffer:  $op.TS = P1 - 3$   $op.TS = P2 - 3$ .

P2 is now able to execute both operations from its operation buffer, as its limit TS became 3. P2's LW is sent over to P1 with its matrix, and P1 will update its matrix on receiving this message as follows:  $\begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}$ ; P1's operation buffer becomes:

$op.TS = P1 - 3$   $op.TS = P2 - 3$ . P1 is also able to to execute both operations from its operation buffer, as its limit TS became 3, but let's assume a

LW needs to be processed first, hence its matrix becomes:  $\begin{pmatrix} 4 & 3 \\ 3 & 3 \end{pmatrix}$  and its op-

eration buffer:  $op.TS = P1 - 3$   $op.TS = P2 - 3$   $op.TS = P1 - 4$  ; P2 receives the corresponding message and updates its matrix:  $\begin{pmatrix} 4 & 3 \\ 4 & 3 \end{pmatrix}$  and its operation buffer:  $op.TS = P1 - 3$   $op.TS = P2 - 3$   $op.TS = P1 - 4$  . P1 and P2 will first execute the JOIN operation corresponding to the join request of node P3. P1 will expand its matrix to reflect the addition of a new node to the DW:  $\begin{pmatrix} 4 & 3 & 4 \\ 3 & 3 & 4 \\ 4 & 3 & 4 \end{pmatrix}$ ; this new matrix, an operation buffer containing only:  $op.TS = P1 - 4$  (as this is the only LW that P1's operation buffer contains) and the local object representing P1's replica are sent to P3 inside a RepMIWelcomeMessage. P3 will take this matrix as its start matrix and will wait for P2's approval. In turn, while executing the JOIN operation corresponding to P3, P2 will first expand its matrix:  $\begin{pmatrix} 4 & 3 & 3 \\ 4 & 3 & 3 \\ 4 & 3 & 3 \end{pmatrix}$  and will then send a similar RepMIWelcomeMessage to P3, containing the following operation buffer:  $op.TS = P2 - 3$  . Hence, P3's matrix will become:  $\begin{pmatrix} \max(4,4) & \max(3,3) & \max(3,4) \\ \max(4,3) & \max(3,3) & \max(3,4) \\ 4 & 4 & 3 \end{pmatrix}$  P3's operation buffer will contain:  $op.TS = P2 - 3$   $op.TS = P1 - 4$  Now, all three nodes will execute  $op.TS = P2 - 3$  and continue in the same fashion their runs. Note that we have simplified the example by not mentioning the parallel execution thread which is actually moving correctly ordered operations from the operation buffer to its execution queue; this is implicitly understood when we mention execution from operation buffer.

## Chapter 4

# Implementation

We chose the Java language to implement the application, for its "write once, run anywhere" feature which is very important in the context of heterogenous Grids. Furthermore, we make use of the Ibis Portability Layer(IPL) for communication between replicas.

### 4.1 Ibis Portability Layer

Ibis Portability Layer is a part of Ibis, a Java grid software project that is being developed by the Computer Systems group at Vrije Universiteit. The IPL is basically a grid environment oriented communications API and allows for multiple communication models, which makes it the perfect candidate for our application.

As explained in Ibis programmer's manual [4], IPL uses receiveports and sendports as end points of a communication channel. Sendports and receiveports can be connected in a OneToOne, OneToMany and ManyToOne manner. Our protocol requires two of the above modes: OneToMany - for broadcasting the local operation, and ManyToOne - for initializing a new node using information from all nodes already in the system. IPL also allows for both ExplicitReceipt and AutoUpcalls, which again fits well with our model, as our protocol needs to explicitly receive initialization information from the contact node, while upcalls are used to receive remote operations, including initialization information from the rest of the nodes in the system. We use different sendports for introducing a node in the system and for communicating operations between replicas.

The Ibis nameserver is used as a discovery service by entities that wish to become part of this DW.

### 4.2 Technical details

The implementation lies in two Java packages: repmi.protocol and repmi.comm.

The repmi.protocol package is responsible for actions that need to be taken on each replica site, like managing the internal operation queues, proper execution of operations and maintaining a consistent view on the nodes existing in the replication system. The Lamport Timestamps Matrix is represented as a HashMap of HashMaps, to emulate an unique\_id-indexed matrix, as shown in Figure 4.1.

Key = P1	Value =	Key = P1	Key = P2	Key = P3	...	Key = Pi	...	Key = Pn
		Value = M[P1,P1]	Value = M[P1,P2]	Value = M[P1,P3]		Value = M[P1,Pi]		Value = M[P1,Pn]
Key = P2	Value =	Key = P1	Key = P2	Key = P3	...	Key = Pi	...	Key = Pn
		Value = M[P2,P1]	Value = M[P2,P2]	Value = M[P2,P3]		Value = M[P2,Pi]		Value = M[P2,Pn]
⋮								
Key = Pi	Value =	Key = P1	Key = P2	Key = P3	...	Key = Pi	...	Key = Pn
		Value = M[Pi,P1]	Value = M[Pi,P2]	Value = M[Pi,P3]		Value = M[Pi,Pi]		Value = M[Pi,Pn]
⋮								
Key = Pn	Value =	Key = P1	Key = P2	Key = P3	...	Key = Pi	...	Key = Pn
		Value = M[Pn,P1]	Value = M[Pn,P2]	Value = M[Pn,P3]		Value = M[Pn,Pi]		Value = M[Pn,Pn]

Figure 4.1: Lamport Timestamp Matrix.  
 $P_i$  = the unique identifier of a node; for  
 $M[P_i, P_j]$  see 3.1.1.

The internal OpsQueues are implemented as TreeSet maintained in the smallestTimestamp-first order, which guarantees that next operation to be executed on the object is first in queue. The most important class of the package: LTMPProtocol.java offers the API discussed in Chapter 2.

The repmi.comm package deals with interaction between replicas, in terms of upcalls dealing with incoming messages which contain operations (write or join operations). A number of messages can be flow through the DW:

- RepMIJoinMessage - used by a joining entity to address the contact entity; contains three `ibis.ipl.ReceivePortIdentifier`: one for the contact entity to send the special initializing information, one for the other entities to send their special initializing information and one for DW messages to

be sent to; in this approach, we avoid contacting the Ibis nameserver for `ReceivePortIdentifier` lookups, which might lead to extra latency penalty and also make more use of the available bandwidth.

- `RepMILTMMMessage` - used for sending operations between entities;
- `RepMIWelcomeMessage` - used to send the special initializing information to a joining entity.

The `RepMIUpcall` handles incoming messages containing remote operations or joining requests.

# Chapter 5

## Evaluation

In order to evaluate the performance of our system, we compare it to another open world version of a replicated system, we have implemented a simple Sequencer version. We measure performance of both approaches in different scenarios.

### 5.1 Sequencer version

In this version, the DW time is given by the central ticketing service, which guarantees unique, increasing numbers are given to messages released in the DW. Hence, execution happens only when a set of consecutive numbered operations, starting with the expected ticket number, are buffered locally. Again, we maintain an open world policy by treating join and leave requests as local operations; note that the joining entity does not have to request a ticket when addressing the contact entity.

### 5.2 Measurements

We proceeded to make four types of measurements, along two main axis: the "writers"/"readers" ratio and the LAN/WAN nature of communication. Hence, we will compare the two applications in terms of how much time it is needed for an operation to be executed, in four scenarios:

**OneToMany, Intra-cluster** Only one node is writing the replicated object, all others read; the DW is all inside the same cluster;

**ManyToMany, Intra-cluster** All nodes both read and write the replicated object; the DW is all inside the same cluster;

**OneToMany, Inter-clusters** Only one node is writing the replicated object, all others read; the DW is spread over clusters;

**ManyToMany, Inter-clusters** All nodes both read and write the replicated object; the DW is all inside the same cluster;

As top-level evaluating application we use the toy application explained in 2, whose simple classes are presented in 2.2 and 2.1. We considered that a replicated bank account application could require both many "writers" at the same time in the system, as well as periods of "silence", where there is only one "writer"; hence, would be interesting to compare its performances while using the two underlying replicated systems (Vector version and Sequencer version).

In the case of the inter-cluster scenarios we measure for 2, 4, 8 and 16 nodes participating in a distributed world (DW), for both approaches. In the case of the intra-cluster scenarios we have a DW consisting of 2, 3, 4, 5 clusters, each participating with one node.

As both protocols optimize initializing state of a node, the writer could actually buffer all writes before another node A joins, hence sending only the final result to A. As this would not fully illustrate average behavior of both protocols, we implemented a barrier synchronization between all processes before the write operations start. Processes are then left to continue their execution, without forcing a synchronized leave. Each process generates the same number NOPS of "writeAddition" method invocations with the same quantity as argument (i.e. 10) and exits when the internal value of the ReplicatedAccount reaches  $NCPUs * NOPS * 10$ . We measure how much time "elapsedTime" is spent to reach this value, after all processes have joined the DW. We then find the minimum "elapsedTime" over all nodes and divide it by NOPS, and such we compute how many milliseconds per operation are spent. We agree this is an optimistical approach and that there could be a more complex statistical model to compute time per operation, but the main purpose of our evaluation is to establish whether a distributed ordering of messages based on timestamps has any advantages over a centralized one. Note that we use the same approach when measuring time per operation in the Sequencer version. We have used  $NOPS = 1000$  write operations.

We have used the DAS2 [3] system for our tests, described in Table ?? in terms of RTT between nodes inside the same sites and between different sites; entries of the table represent average RTT and have been obtained using `ping -c 10`. We have mostly run our application through the `prun` [9] user interface to the reservation system of DAS2, which guarantees allotted nodes are exclusively used by the scheduled application, as long as users do not circumvent the reservation system.

Note that we always have a separate and dedicated node on which we run the Ibis Nameserver, which plays the role of either Discovery Service - for Lamport-based application or Ticketing Service - for Sequencer application.

### 5.2.1 OneToMany, Intra-cluster

As seen in Figure 5.1, the Sequencer approach functions better, as per operation it has to contact the Ticketing Service and then broadcast the message to all

Table 5.1: RTTs in DAS2 system

	fs0	fs1	fs2	fs3	fs4
fs0.das2.cs.vu.nl	0.142	2.479	2.264	2.871	5.391
fs1.das2.liacs.nl	2.486	0.054	1.706	2.909	3.272
fs2.das2.nikhef.nl	1.906	1.636	0.070	2.280	2.613
fs3.das2.ewi.tudelft.nl	3.073	2.874	2.244	0.059	3.848
fs4.das2.phys.uu.nl	3.362	3.336	2.599	3.834	0.060

"readers", not depending on feedback from the other nodes in the DW. On the other hand, as explained in 3.2, the Lamport-based (Vector) approach depends on periodic heart-beat messages, which announce the "sender" that its "readers" have not written anything on their replicas. This is basically the price paid to have a distributed fashion of keeping the DW consistent versus the centralized way. We chose a 300ms period for the heart-beat messages as this would be the RTT of a ping message to Japan.

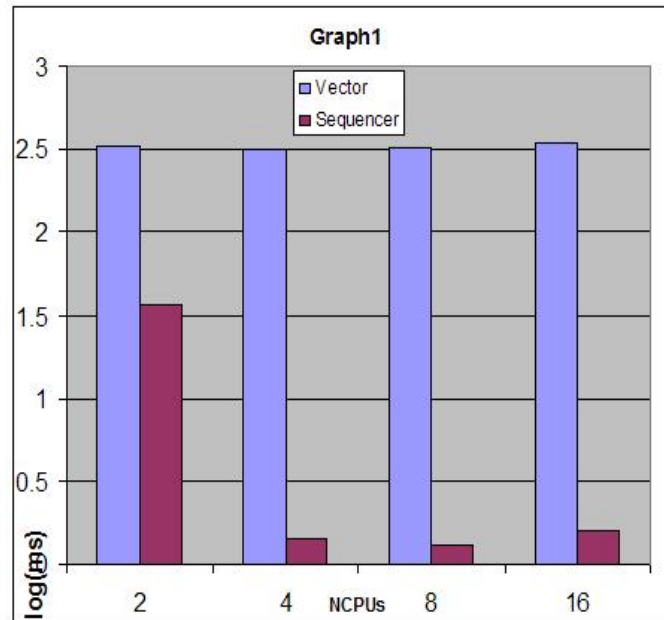


Figure 5.1: OneToMany, Intra-cluster. Measurements taken on fs0 nodes.

We also include the table 5.2 containing the values, expressed in milliseconds per operation. In order to have a better view of the results, the graph presents

the logarithmic values.

Table 5.2: OneToMany, Intra-cluster

NCPUs	Lamport-based (Vector)	Sequencer
2	336.37	37
4	315.35	1.42
8	322.04	1.3
16	345.94	1.58

### 5.2.2 ManyToMany, Intra-cluster

In this scenario, everyone is writing the replicated object; hence, in the case of the Sequencer approach, messages will flow between all nodes of the DW as well. Every broadcast message in the Sequencer approach would first have to receive a ticket from the Ticketing Service and only then it can be sent to all nodes in the DW. This involves extra time spent while waiting for the Ticketing Service to return a correct ticket, which does not appear in the Vector approach (see Figure 5.2). On the other hand, when many nodes are participating in the DW, the Vector approach becomes slower than the Sequencer one, as more computation is needed on each node in order to maintain the global ordering consistent. In the particular case of 2 nodes, the more accentuated difference between the time performance of the two applications can be explained as follows: the Vector approach needs one message to execute a method invocation, while the Sequencer approach would require three messages - one to the Ticketing Service to obtain the ticket, one to itself and one to the other machine.

We again include the table 5.3 containing the results of this measurement.

Table 5.3: ManyToMany, Intra-cluster

NCPUs	Lamport-based (Vector)	Sequencer
2	1.29	3.46
4	0.78	2.17
8	0.91	2.06
16	1.59	1.21

### 5.2.3 OneToMany, Inter-cluster

This scenario is similar to 5.2.1 in terms of how many participants of the DW are write-accessing the replicated object, but this time the DW is spread over a varying number of clusters. As the Sequencer based approach depends heavily on the communication with the Ticketing Service, we considered two subscenarios: (a) where the Ticketing Service is on the same cluster as the "sender", and (b) where they lie on different clusters. As expected, the results show that

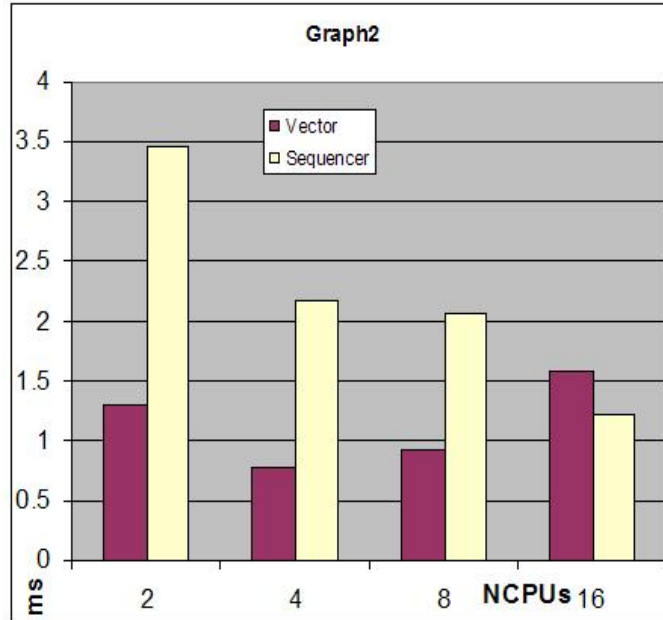


Figure 5.2: ManyToMany, Intra-cluster.  
Measurements taken on fs0 nodes.

indeed the extra latency due to WAN communication adds a time penalty to subscenario (b). Nevertheless, as seen in Figure 5.3, the Vector approach is still dominated by the hear-beat period of 300ms.

Should we take a look at Table 5.4, we note that values for the Vector approach are close to those obtained in 5.2.1 scenario when using a DW of 4 nodes; this shows the Vector approach is stable w.r.t. to LAN/WAN communication, but it "updates" the state of the replicated object slower if there is a small number of "writers". Again, the graph presents the logarithmic values of the measurements.

Table 5.4: OneToMany, Inter-cluster

NClusters	Lamport-based (Vector)	Sequencer (a)	Sequencer (b)
2	313.558	2.63	6.99
3	314.479	2.55	7.02
4	315.989	3.16	6.98
5	317.255	2.59	7.8

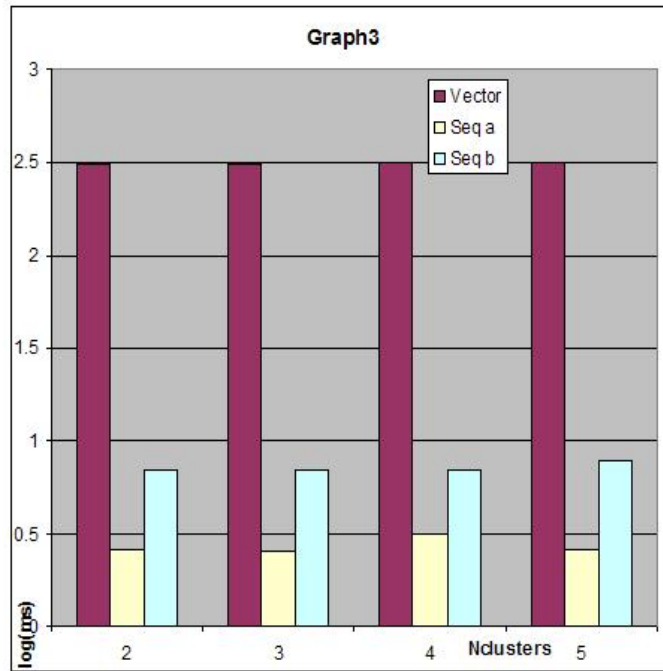


Figure 5.3: OneToMany, Inter-cluster. Measurements taken on DAS2 system.

### 5.2.4 ManyToMany, Inter-cluster

In the last scenario we proposed, we expect the Lamport-based (Vector) approach to behave better than the Sequencer approach, as everyone is "writing" and "reading" the replicated object, hence messages flow both ways between all nodes of the DW, spread over a number of clusters. Indeed, Figure 5.4 shows that the Vector approach pays off in terms of performance, as there are now write operations generated by all nodes in the DW, and this forces quicker "updates" of the replicated object's state.

Table 5.5: ManyToMany, Inter-cluster

NClusters	Lamport-based (Vector)	Sequencer
2	1.84	4.8
3	1.23	4.64
4	1.25	3.55
5	1.15	3.08

Looking at the values from Table 5.5, we can see now that the extra delay

owned to communication between clusters rather than inside the same cluster (see 5.2.2) does impose a small penalty over the performance of the Vector approach.

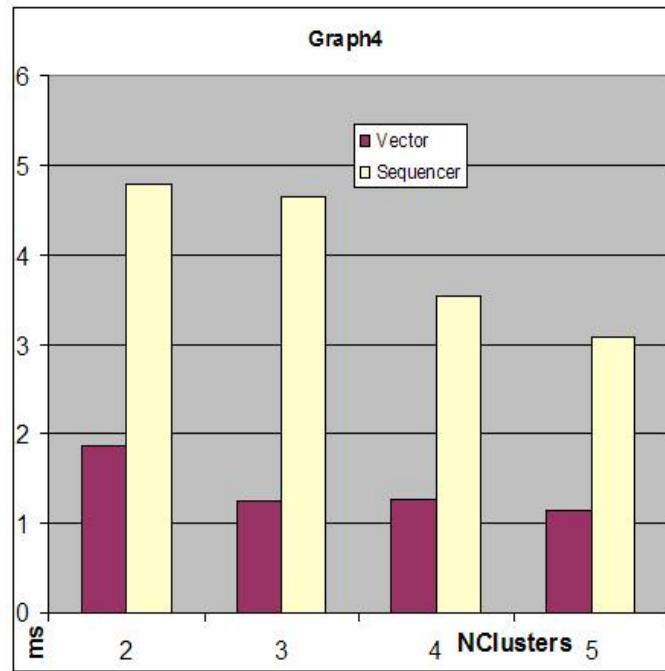


Figure 5.4: ManyToMany, Inter-cluster.  
Measurements taken on DAS2 system.

## Chapter 6

# Conclusion

**Summary** We have presented an implementation of Replicated Method Invocation based on Lamport Matrix Time. We have adapted the Matrix Time to help achieve proper synchronization of write operations execution. The underlying communication has been implemented using Ibis, as well as a Discovery Service and a Ticketing Service. When comparing our approach to the traditional, sequencer-based approach, we have seen that it can perform better, especially on wide-area communication which is specific for grids.

Experience with implementing the Sequencer approach showed that the open world platform built for the Vector approach can be easily reused with different mechanisms for message ordering. Its main feature is the special operations introduced for joining and leaving the distributed world (DW). A consistent snapshot of the messages flowing in the DW is obtained by filtering only local operations when constructing the welcoming message for the joining node. This way, the newly added node has a concentrated history of what happened before its arrival in this DW.

The toy application presented in 2.1.2.2 proved it would be actually easy to add a module for generating the code which would wrap the application-level method invocations inside the corresponding repmi protocol calls, as described in 4.2. Using the marker interface, "replicateable" objects as well as invocations of their read-access or write-access methods could be identified at compile time and corresponding code can be generated. Moreover, even the interface indicating which methods require write-access (e.g. 2.1) could be automatically extracted by analysis of the methods in the "replicateable" class.

Another interesting point is the ability of the DW constructed using the Vector approach to survive failure of the Discovery Service, becoming a closed world, but still functional; on the other hand, if the Ticketing Service fails, the Sequencer approach cannot function anymore.

Finally, evaluation of both approaches (see 5.2.1, 5.2.3) yielded that dynamically adapting the period of heart-beat messages is of crucial importance for the performance of the Vector approach in scenarios where applications using the replicated system are mostly read-oriented.

**Future Work** There are several directions which we believe are worthy for further study:

- adaptive heart-beats impact on performance of the Vector approach, especially in scenarios as the one presented in 5.2.3; our measurements suggest adapting the heart-beats to the activity rate of the node could crucially influence the overall performance of the replicated system;
- when the underlying network is characterized by large bandwidth and varying-large delay, improving the update rate of the limit TS by gossiping operations between nodes (i.e. if node A notices based on its Matrix Time that node B has not seen yet several operations from other nodes, it can piggyback those when sending a message to B; if by the time B receives this message has already seen these operations it will simply discard them);
- the above suggested improvement could also improve fault tolerance at application level, though we are currently relying on Ibis TCP-mode for communication between nodes;
- analyze performance of the replicated system when there are more shared objects. To allow for more shared objects to be alive at the same time in the replicated system, only one method needs to be added to API presented in 2: `lookup(String name)`, where the naming scheme could be similar to the one used by RMI.

# Bibliography

- [1] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Lan gendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.
- [2] [http://en.wikipedia.org/wiki/Sequential\\_consistency](http://en.wikipedia.org/wiki/Sequential_consistency).
- [3] <http://www.cs.vu.nl/das2/>. DAS2.
- [4] <http://www.cs.vu.nl/ibis/progman/>. *Ibis Programmers Manual*.
- [5] Jason Maassen. *Method Invocation Based Programming Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit, 2003.
- [6] Jason Maassen, Thilo Kielmann, and Henri E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8–9):681–712, 2001.
- [7] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java’s Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’99)*, pages 173–182, Atlanta, GA, May 1999.
- [8] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.
- [9] <http://www.cs.vu.nl/das2/das2-run.html>. PRUN.
- [10] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Cerial Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, 2005.