Vrije **U**niversiteit **B**russel
Faculteit Wetenschappen
Departement Informatica
Academiejaar 2000 – 2001

CONCURRENCY MANAGEMENT AS A QUALITY OF SERVICE ATTRIBUTE IN DISTRIBUTED ASYNCHRONOUS COLLECTIONS

Pockelé Jannes

Proefschrift ingediend met het oog
op het behalen van de graad van
Licentiaat in de Informatica

Promotor: Prof. Dr. Theo D'Hondt

Abstract

Concurrency is a much-discussed problem in distributed computation. We will investigate if it makes sense to implement it as a Quality of Service Attribute in Distributed Asynchronous Collection.
DAC's are publish/subscribe engines that use asynchronous method calls. They use Quality of Services to provide functionality to their clients. Our quest will be to find out if DAC's are useful, and if concurrency can be implemented as a QoS Attribute of these DAC's

ACKNOWLEDGEMENT

Before we start, I would like to thank the following people without who this work would not have been possible.

Theo D'Hondt,

- The Promotor of this thesis, thank you for giving me the chance of doing the thesis at PROG.

Johan Fabry ,

- Thesis Mentor, who succeeded in motivating me and always helped me with all the problems.

Werner Van Belle,

- His practical aid with BORG and his ideas were invaluable.

Johan Brichau,

- For reading and correcting my thesis many times.

In addition, many thanks to the persons that were kind enough to read this thesis, and correct the many mistakes made in it.

Thanks to: Eva Pockelé, Nico Smets, Nele Dillen, Veronique Volders, Maarten Peeters, Stijn Lavreysen, Annelies Maertens, Koen Huysmans, Bert Heylen, Curt Claeys

I also want to thank all the people of the PROG lab, for providing useful information and feedback at the weekly thesis meetings.

I would like to take this opportunity to thank my parents for giving me the courage and the support I needed to complete my thesis and studies.

On a more informal note, I would like to thank Garfield who, with his wise words: "*Things could be worse*" always encouraged me.

One person did not survive the production of this thesis. Douglas Adams was taken from us on 11 may 2001. It was a terrible loss for the world of literature. So long, and thanks for all the fish.

We've heard that a million monkeys at a million keyboards could produce the complete works of Shakespeare; now, thanks to the Internet, we know this is not true.

Robert Wilensky, University of California

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

> *"If you want to make an apple pie from scratch, you must first create the universe."*
>
> ***- Carl Sagan***

In the early days of computer science, when networks had not been invented, there was no distributed computing since there was no communication between computers. When communication was finally established and you could hear people dream about clusters of computers forming supercomputers, a new research area was created.
Distributed computation is a field of research in Computer Science. It is gaining some popularity since the commercialising of the Internet.
Distributed Systems are defined by Coulouris [1] as follows:

> *A collection of autonomous computers linked by a network, with software designed to produce an integrated computing facility.*

When we are talking about distributed systems or distributed applications we mean programs that are split up into parts that run on different machines. Since these different parts need to work together, there has to be a form of communication between them.
In the early days of distributed computation, the communication had to be implemented on a very low level. This was usually done using sockets. Sockets are data ports, i.e. ports you can write data to, after specifying how much data you are going to write. It was rather primitive and we would rather have something better. Today, more high-level communication tools have been developed such as Java RMI [9] and CORBA [8], which will be explained later.

A widespread example of a distributed system is what we will call '*Distributed Information Gathering'*. This program is built out of different components, spread all over the world. Different companies implement these components, and there are versions for all operating systems. Using search tools that index the distributed information, everyone can locate information that is spread over the world. The '*Distributed Information Gathering'* program serves millions of people at the same time, to get them the information they want. This '*Distributed Information Gathering'* program is also called '*the Internet*'; a system where a group of dedicated applications work together to become one large distributed system. The development of the Internet showed us the power of distributed information storing, and opened the doors to distributed programming.

In the field of distributed systems, there are still lots of problems to be solved. Most problems are caused by the unreliability of the network. Transmission errors, partial failure, package-loss, sending pointer-references over the network… These problems do not occur in non-distributed systems. Concurrency is already a problem in multitasking single-processor-systems, but concurrency issues in distributed systems are far more common

Most abstractions in distributed computation use a request/reply style for their communication, which will be explained later. Recently Distributed Asynchronous Collections (DAC's) [4] were introduced. They use publish/subscribe instead of request/reply. This publish/subscribe style is not new, but it has been believed not to work good with Object Oriented languages.

Some research on distributed computation has been done at the VUB and one of the results of this is BORG [6], an educational language for distributed systems.

We will implement these DAC's into BORG to investigate the usefulness of the publish/subscribe style. When we have implemented the DAC's we will construct some special DAC's with concurrency management.

In wide area networks, there is the possibility of long delays and partial failures. We will try to focus on the concurrency management problem in these wide area networks, using the self-made DAC's.

If it is proven useful, the DAC structure could be added to the BORG core to provide both the request/reply and the publish/subscribe interaction style. The DAC's describes Quality of Service (QoS) attributes. Using these attributes one can select which kind of DAC implementation is suited for your program.

1.1 Overview

1.1.1 Chapter 2: Distributed Computing

Wouldn't it be wonderful if local objects looked exactly like remote objects? Of course, and many have tried to make this work. It turns out that there are some problems when working with this unified model. Since both local and remote objects are supposed to look the same, we either have to extend local objects with the appropriate error handling or ignore the difference and hope for the best. Both options are bad.

In this chapter, we will first explain the main differences between local and remote calls. Second, we will discus the main communication techniques used in distributed programming. Final, we end with a general overview on remote procedure calling.

1.1.2 Chapter 3: Distributed Asynchronous Collections (DAC)

In a paper from Patrick Th. Eugster, Rachid Guerraoui and Joe Sventek on ECOOP 2000 Distributed Asynchronous Collections were introduced.
First, these are collections, meaning an abstraction used to store and retrieve objects. Secondly it is a Distributed Collection, where information (here the objects) is stored and accessed in a distributed manner. The information is spread over different computers on the network. These can access all the information available in the network. As if that isn't difficult enough, it has to be done in an asynchronous way.
We will also explain the different Quality of Services in the DAC's. Distributed Asynchronous Collections implement the publish/subscribe interaction style. The client can tell the server about his interests in the resources on the server. When new information is available about the resource, the clients that have declared their interest in it will be notified. It is the responsibility of the server to notify the clients about these things.

1.1.3 Chapter 4: BORG

Some abstractions were created to make programming of distributed systems easier. The problem with these abstractions is that a lot of extra tools are necessary to make the system work. Naming services, extra compilers, etc. What we would want is a language where basic communication and remote method invocation are implemented into the primitives of the language. We believe that BORG is an excellent example of this. BORG is a research language developed at the VUB. In this chapter we will explain the basics of BORG and show how object oriented programming techniques could be used. To end we will give a short comparison between BORG and other distributed programming languages.

1.1.4 Chapter 5: Distributed Asynchronous Collections in BORG

The natural distributed nature of BORG and the underlying subsystem appeared to be an ideal environment for the implementation of DAC's. In this chapter, we will describe the process of building these DAC's together with some design decisions we had to make. We started by programming some examples to see if the DAC implementation that we had made worked. We will also define Quality of Service here, and give an overview of the QoS Attributes we will be using.
To conclude this chapter we will give an overview of how these DAC's should be used and managed.

1.1.5 Chapter 6: Experiments with DAC’s in BORG.

The experiments are all based on a Stock Market example. This Stock Market was of course implemented with DAC’s in BORG. The Quality of Services of the DAC systems was used to create different types of DAC’s. With the results of this experiment, we improved the implementation of the DAC's. We concurrency was our main concern.

We tried to find some solutions to the problem of concurrency in wide area networks. The problem here is the network latency, causing messages to be delayed. Since the resource is distributed over the network, it could be spread over different computers.

In the DAC system, we know who is interested in the resource. We are going to try to use this advantage to design a locking structure. Once we have the locking, we will implement different Quality of Services, DAC’s that have other primary-service-attributes. For example: a DAC where all the clients are dealt with in an equal way or a DAC that tries to get everything done as quickly as possible.

1.1.6 Chapter 7: Conclusions and Future Work

In this chapter, we will show what we achieved with our implementation. A short overview of the entire document will be made to summarise the conclusions of this document. Afterwards we will explain how this work could be extended.

2 DISTRIBUTED COMPUTING

"In the beginning the Universe was created. This has made a lot of people very angry and has been widely regarded as a bad move."

- Douglas Adams

2.1 Introduction

When people wanted easier abstraction to do some distributed programming, they got the idea of remote procedure calling. The programmer could invoke a procedure on a different machine. Many interesting ideas came about, but some problems were hard to solve.

In this chapter, we will give a short summary of some of the problems with Distributed Computing [2]. Second, some of the well-known abstractions to work with distributed systems will be explained.

2.2 Problems in Distributed Computing

There are two kinds of objects: local objects and remote objects.

Local Object: An object that is in the same data-space of the caller, these are the objects used in normal programming.

Remote Objects: An object that exists in another data-space possibly on another machine on the network. These objects are used in remote method calls.

Throughout the past, many people have been trying to make unified objects, i.e. to see no distinction between local and remote objects. From a programmers' perspective, it would make programming with remote and local objects much simpler. We would be able to call remote methods as if we were calling local methods. Unfortunately, it seems that the *Unified Object theory* [2] contains some flaws. A couple of properties of local objects are totally different form remote objects. Important issues are latency, memory access, partial failure and concurrency. We will describe each of these issues below.

2.2.1 Memory access

An obvious difference between local and remote calls is the access to memory. The remote object is in a different data-space than the local object, so it can't access any of the local variables. When we want to call a remote method with parameters, these parameters must be copied to the remote data-space. When we call a remote method with an object as parameter, the object is copied to the other data-space. If the object contains pointer references, these will not be valid anymore in the remote data-space. Giving pointer references to a remote method call is totally useless. We could solve this using references containing the machine-name. However, this would imply creating a shared memory system, with new problems like access-rules etc.

When an item is sent over the network, to another computer, it is not said that the other computer has the same hardware. It could have a different representation of an integer or a floating point number. The information must first be transferred into a machine independent representation. This is what we call **marshalling**. The receiver of the data must translate the machine independent data into his own representation of the data. This would logically is called **unmarshalling**. These marshalling techniques must be used every time data is send over the network.

2.2.2 Exceptions

Objects in a remote data-space can experience other exceptions than local objects. When calling a remote method, several things can go wrong: the remote machine can fail, the network can fail, and data can be lost or corrupted… These problems do not occur in local computing. It would be ideal if we would know exactly what went wrong, but sometimes there is no way to distinguish, for example, a network failure from a machine failure.

When enormous calculations must be made by the remote object, there is almost no way of knowing if the remote object is still working or if there has been an error and the machine hangs. It becomes very difficult to raise the correct error, or to raise an error at all.

If we would like to use the unified model, there would be no distinction between remote and local objects. All local objects would have to bear the weight of implementing all the remote exceptions, since the programmer does not know if the objects are local or remote.

2.2.3 Latency

According to Szyperski [3]:

> *"The communication cost increases with the network level used. Cross-process method invocations are significantly slower than in-process calls. Cross-machine calls across a LAN are even slower, and so on."*

We can conclude that local call is significantly faster than a remote call and this due to network latency. Although processor speed and network speeds are changing, the difference will always be significant. We can't ignore the difference between a local and a remote call. Implementations that ignore this latency could have serious performance problems.

The difference between local and remote calls is one of the reasons against the unified object model. For example: It would be ideal when the programmer knew how tightly all objects were coupled. If so, the tightest objects can be clustered together, possibly in the same data-space.

2.2.4 Partial failures

When centralised computer systems fail, all the programs that are running on it fail simultaneously. In a distributed system, when a component fails, the other components will normally continue to run. If they depend on the continued operation of the failed component, they may fail later. There are two sorts of failure.

Partial failure: Part of the system can fail while the rest continues to work. Partial failures are not rare at all.

Total failure: Fault condition that has completely interrupted a system.

Designers of distributed applications needs to be aware of the range of possible failures that might occur using a service. A description of the ways a service may fail is called **failure semantics**. The knowledge of the failure semantics of a service can enable a new service to be designed to mask the faulty behaviour of another service.

In distributed computation, each co-operating component could have reached a different point in its execution. To ensure that the permanent data, stored by all of the components, remain consistent, recovery procedures are needed. These must make the data consistent after a failure.

Even if it might be possible to make the latency and the memory access invisible to the user, it would be impossible to hide partial failure from the user.

2.2.5 Concurrency

In distributed systems, it is possible that one component is used by several programs at the same time. Therefore, remote objects should also be able to handle calls concurrently. In the *Unified Object theory* [2], all the objects have to bear the weight of concurrency. The alternative is that no objects handle concurrency. This is obviously not the ideal thing to do. This proves again that there are some problems with the *Unified Object theory*.

There are several solutions for the concurrency problem. One of the solutions is locking. Locking provides a means for serialised data handling. A client can request a lock on a resource. After being granted the lock, the resource is entirely in the possession of that client. There are different kinds of locks; read-locks are used when requesting to read the data, write-locks when you need to write data. Obviously, when a client has a read-lock on a resource, other clients can have a read lock on the same resource, since no one is changing the value. On the other hand, there can be only one client with a write lock.

Locking generates another problem, namely deadlocks. There has been a lot of research on this topic, and several good solutions have been found. Besides locking, there is another solution to manage concurrency: *optimistic concurrency control*. [18] This solution allows a client to continue until all his transactions are done after which then the managing service checks for violations.

In our research, we try to conquer the concurrency problem using locking and the publish/subscribe model. We can use the fact that resources know who is interested in them to handle the locking.

2.2.6 Taking the Difference Serious

If you would like to use the unified object theory, you can have all local objects act like remote objects, or you make all remote objects act like local objects, ignoring latency, partial failure and concurrency. Programming in the unified model would be easier, since all objects look the same, and can be treated in the same way. It would however be foolish to ignore the differences. If we would extend the local objects so they would look as remote objects, we would create a lot of overhead.

A better approach is to accept that there are differences between these two kinds of objects. Then the programmer has to know if he is sending a message to a local or to a remote object. This might not seem appealing, but it could in fact improve program performance considerably.

2.3 Communication

Now that we have seen some of the problems we can expect in distributed computing, we will look at the tools that were developed for distributed computation. First, we examine the functions for communication when using sockets. Then, a general introduction to remote method calling will be given, after which some examples of remote procedure calling abstractions, namely Java RMI and CORBA, will follow. The lower layers of the network will not be discussed since they are outside the scope of this thesis.

2.3.1 Socket Programming

We believe that an understanding of the socket API provides a good basis for some of the issues and problems of distributed computations.

Socket: A socket is an endpoint used by a process for bi-directional communication with a socket associated with another process.

Sockets are a basic mechanism for process communication on different computer systems connected by local or wide area networks. Sockets can in fact be seen as a “buffer” shared by two processes (for example process *Alpha* and process *Beta*). When process *Alpha* writes data to the socket, process *Beta* can read this data on his side, and the other way around.

There are two ways of communication over a communication channel created with sockets.

1) The communication channel can be like a telephone line. This is what we call **connection oriented**. The sockets are used as telephones over which a conversation takes place.

2) The channel can be as when we send mail. This is called datagram oriented, or **connection-less**. The sockets are used as mailboxes.

For example, if we take al look at TCP/IP, we see that connection oriented communication is reliable, i.e. the system takes care of errors, and the receiver will receive information in the order in which it was sent. Datagram oriented (connectionless) communication is unreliable, i.e. messages can be lost, or they may not be delivered in the order in which they were sent. Of course the convenience of connection oriented sockets comes at a price in performance.

A problem in communication is how to address other processes. When communicating with the telephone, you dial a telephone number. When sending a letter, you write the address on the envelope, and let the post office deliver it. Therefore, we of course need a means to address the other process. The appropriate means to do this is an **IP address** and a

port number. the port number is necessary. It is analogous with an appartment building where each appartment that wants to receive letters must have a mailbox. Likewise, every process on a computer must specify a port number in order to receive and send information. Sometimes you can give a **hostname** instead of an IP-address, but the hostname will be converted to an IP.

Now we have two processes that can communicate with each other using these sockets. (On the condition that they know each other) The next problem is finding a suitable **protocol** for the communication. Both processes could start sending data over the communication channel, but if they don't understand each other, the communication is useless. Just like in a telephone conversation, when you say "Hello" and the reply is in Chinese. The protocol is an understanding between the two processes to guarantee that they will understand each other if they follow that protocol.

A simple example of a protocol is the HyperText Transfer Protocol (HTTP). This consists of three messages: GET, POST and PUT. All these messages can be followed by arguments, for example:

GET /~dvermeir/index.html

All socket communication needs a protocol, and something to translate the information according to the protocol to useful information for the process.

Socket programming is essential for inter-process communication. Middleware has been developed which translates users requests into socket calls. One of the most used high-level abstractions is remote procedure calling (RPC). In an object-oriented language, it is called remote method invocation (RMI) [9], since objects have methods that you invoke instead of procedures. This we will discus later, but first we will discus some communication architectures that are often used with socket programming.

2.3.2 Communication Abstractions

In this section Client/Server and Peer-to-Peer architectures will be introduced. We will need to understand these architectures because they will be used in the remote procedure calling section. First short definitions will be given of Client/Server and Peer-to-Peer and then some examples to make it clear.

Client/Server architecture is a network architecture in which each computer or process on the network is either a client or a server. Servers are powerful computers or processes dedicated to managing databases, disk drives, printers, or other. Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power.

Peer-To-peer is another type of network architecture, where each node has equivalent responsibilities.

Both client/server and peer-to-peer architectures are widely used, and each has unique advantages and disadvantages.

An example of client/server architecture is the communication between a web-browser running on the client side and the web-server running on the server side. The web-server can handle an enormous amount of concurrent calls and serves a lot more web pages than those requested by the client, while the client relays on the resources for the server. There are many more client/server-structured applications. A popular example of a Peer-To-Peer architecture would be a system like Gnutella [10], the mp3 search-tool. Here all the users download from each other and therefore offer the same services as any other user.

2.3.3 Remote Procedure Calling (RPC)

This section will explain the basics of remote procedure calling. Afterwards, we will go into further detail in two cases of remote procedure calling software, Java RMI and CORBA.

Remote procedure calling is a powerful technique for constructing distributed client-server-based applications. It is based on extending the notion of conventional (local) procedure calling. The called procedure does not need to exist in the same address space as the calling procedure. There of course has to be a network link between them. The idea is to have an interaction style comparable to procedure calls, in a distributed context. So we can for example call a procedure that is executed on the remote computer after which the result will be sent back.

Remote procedure calling is analogous to local procedure calling. When a RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Since the parameters must be sent to the remote procedure, they have to pass over the network. Therefore, the parameters must be marshalled and unmarshalled, as explained in section 2.2.1. After executing the function call on the remote machine, the same has to happen to the result. Since we want the remote procedure call to look like a normal call, we don't want to do the marshalling ourselves.

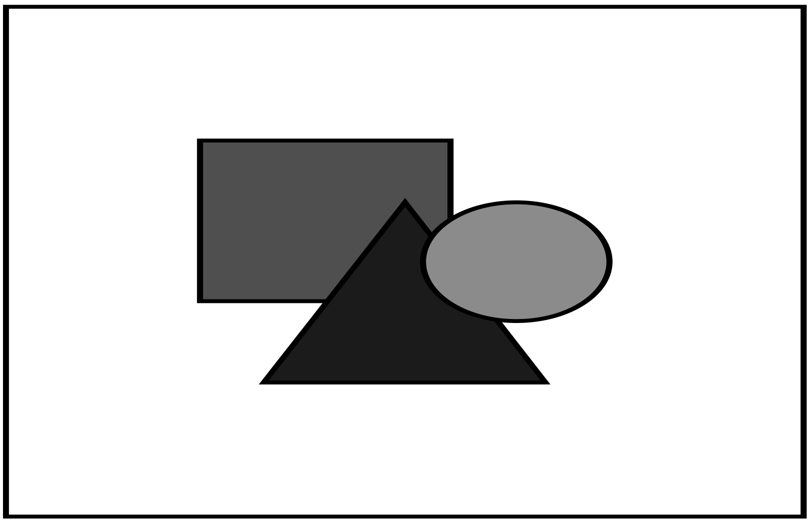

Figure 1 Remote procedure call

In figure 1 we see that a program is split up in two parts. Part *P1,* which is on the local client, and part *P2,* that is on a server somewhere. In part *P1* of the program the procedure *Fac(55)* is called, but this procedure exists in part *P2* of the system. Since we want to be able to call *Fac(x)* as a normal procedure, we define *Fac(x)* in the client stub. The client stub marshals the data and the request for calling *Fac(x)* to the server stub. When the server stub gets this request and data, it un-marshals the data and calls the *Fac(x)* procedure that is local to him. The *Fac(x)* executes, and sends the result back to the server stub. After marshalling and un-marshalling by server stub and client stub, the client stub sends the result back to the calling program using the same technique.

The programmer of course does not have to know about all this. All he has to know is what procedures with which parameters he wants to use. Usually this has to be specified in an interface, representing all possible remote procedure calls. Using special RPC-compile tools the client and server stubs are created from this interface. In the client stub, the remote procedures are implemented as proxy's of the remote procedure. (This means that they forward the call to the remote procedure.)

Another problem is finding the remote procedure. This could be hard-coded, or a naming device could be responsible for it. The option of the naming server would be the better of the two, however finding the naming service itself is also a problem.

A naming device works like a internet-name server; it holds a table of names and references, binding an ip-address to a name. A naming device for remote procedures can be used by the remote objects to register themselves. Caller objects can find them by querying the name-server.

Remote Method Invocations in Java

We will now continue with the first example, that of Java remote method invocation, or short, Java RMI [9]. We will show how the above remote method techniques are used in Java RMI.

Since Java is an Object Oriented language, remote procedure calls are called remote method invocations here. The Java language comes with RMI-class files. We can create a RMI application by extending these pre-defined RMI-classes.

To call a remote method, we must first have an object to call the method on. Since this object must be a remote object, it must extend the Java.rmi.Remote interface, and implement the messages we want to invoke. These messages should be declared in an interface. Java has a special compiler to generate stubs from interfaces. It also has a dedicated naming service called rmiregistry, used for binding and finding remote services.

The stub on the server side is called skeleton in Java RMI. The communication between the stub and this skeleton is hidden in lower levels of the RMI communication protocol.

CORBA (Common Object Request Broker Architecture)

Let's continue with our second example, CORBA [8]. CORBA stands for Common Object Request Broker Architecture.

CORBA is a specification for creating and using (distributed) objects; it is not a programming language. Different vendors provide implementations of this standard. These implementations generate different stubs for the specified programming language. Since all stubs can communicate with each other, remote procedure calls to programs implemented in different programming languages become possible. The Object Request Broker (ORB) encompasses the entire communication infrastructure necessary to identify and locate objects, handle connection management and deliver data. In general, the ORB is not required to be a single component; it is simply defined by its interfaces.

A shared interface definition language is required. This language is the Interface Definition Language, or IDL [11].

For Example: the IDL compiler for Java generates client-side stubs, which represent the remote-CORBA object locally in the Java programming language. Since a shared language is used for the definitions, language specific objects cannot be used because the equivalent of that must exist in all languages. Therefore, the greater usability by offering the possibility to program in other programming languages is limited in expressiveness because the definition language is restricted to be the common denominator of all languages.

2.4 Summary

In this chapter we explained the differences between single processor computation and distributed computation, which we will now shortly summarise.

Memory access is one of the important differences, there is no access to local variables, pointer references are unusable, and different machines could have different data representations. The second problem is with exceptions, special remote exceptions. It must be noticed and it could be undeterminable to find what went wrong. Due to network latency, remote calls are significantly slower than local calls. The next problem rises when a part of the distributed application fails, this is called partial failure. A remote resource could fail or the network could go down. Therefore, the system must be able to handle this. As last problem, we saw concurrency. Since more than one program could be using the remote resource the resource should be protected against concurrency problems.

After explaining these problems we concluded that trying to use remote objects as if they were local objects, like the Unified Object theory says, should be abandoned.

Next, we looked at the communication abstractions used in distributed computation. We discussed socket programming and remote procedure calling.

Socket programming is a low level abstraction where you have to send everything yourself, a protocol is important to understand each other and the programmer has to implement low level error handling yourself.

While remote procedure calling is a high-level communication abstraction. Here the interface must be specified and stubs must be compiled. Most RPC-systems need a naming-service. As a case study of RPC we saw Java RMI, that can only call other Java remote object methods, and CORBA. CORBA can call methods of objects written in many other programming languages. It requires that you specify an interface in the restricted Interface Definition Language.

3 DISTRIBUTED ASYNCHRONOUS COLLECTIONS

There are three kinds of death in this world. There's heart death, there's brain death, and there's being off the network.

Guy Almes

3.1 Introduction

The idea behind the development of Distributed Asynchronous Collections (DAC's) [4] was to create a new communication model for distributed communication, using the publish/subscribe style. The properties of these DAC's will be explained in this chapter and it will be shown that they are useful in Distributed communication.
First, we will give an overview of the DAC's. Second, we will discuss the different possible interaction styles, each with their positive and negative points. Last, the notion of Quality of Service will be explained.

3.2 Basics of DAC's

In object oriented systems, we always use request/reply strategies. This however implies that both parties must be coupled in space and in time. That means that the interacting parties need to know each other, (Coupled in Space) and the interacting parties must be online at the same time (Coupled in Time). Another item is the point-to-point model in normal object oriented systems. There is always one sender and one receiver. Using the DAC model, we could improve this model.
A Distributed Asynchronous Collection is a **collection**. This means that it is an abstraction used to store objects in. There are some differences with a normal collection, being a **distributed** collection means that the collection can be accessed form different points over a network and that information in the collection can also be stored on different nodes. It is also a Distributed **Asynchronous** Collection. The asynchronous nature of the collection is caused by the asynchronous communication model that is used. This will be explained further in the following sections.

3.3 Distributed Asynchronous Collections Overview

We will start by introducing the essential abstractions of the Distributed Asynchronous Collections (DAC), showing the relationship between them and the publish/subscribe communication style. After that, we will demonstrate how to use these abstractions to build several variants of the Distributed Asynchronous Collection Framework.

3.3.1 Object Container

A collection framework is a unified architecture for representing and accessing collections, allowing them to be manipulated independently of their representation. Both Java and Smalltalk [12] contain a rich set of frameworks. These reduce the programming effort by providing useful data-structures and algorithms. A DAC contains an abstraction of a container object. It represents a group of objects, and is a means to save, retrieve and manipulate objects from that group. Unlike conventional collections, DAC collections are distributed collections. Operations on this collection can therefore be invoked form every node of the network. DAC's differ fundamentally from other distributed collections because of their asynchronous and essentially distributed nature. This means that DAC's can be seen as "omnipresent" entities. Every client talking to a DAC is in fact talking to a local proxy, which is viewed as a local collection and hides the distribution of the DAC. DAC's are not centralized on a single host, in order to guaranty the availability despite the failure of some subparts.

3.3.2 Asynchronous

DAC's represent more than just a distributed collection. Since synchronous calls to distant remote objects suffer from latency, they are hardly comparable with local calls. The solution to this problem is asynchronous interaction. When a client calls an operation of a DAC, the client tells the DAC it wants to be notified when the DAC has handled the operation. In the context of DAC's there is the contains operation. When a client calls this operation on a DAC is expresses its interest in a certain kind of objects, and when an object like that is pushed into the DAC, the interested client is asynchronously notified of this fact. Publishers can be seen as information suppliers, and remain anonymous. The DAC's can be seen as a server since they provide the information to all their subscribers.

3.3.3 Publish/Subscribe

The Publish/Subscribe interaction style [13,14] is based on Publishers and Subscribers. Publishers publish items on an information bus and the subscribers are notified about these events. This interaction style in used in Distributed Asynchronous Collections. To explain the differences between this style and the other styles, we will first give an overview of the possible interaction styles.

Expressing your interest in receiving information of a certain kind can be viewed as a subscription to information of that kind.
A collection is a set of somehow related objects; a DAC can be pictured as an extension of a conventional collection. A DAC can also be seen as a representation of a topic. It is always possible to insert new items in the collection. In the sense of the publish/subscribe style, inserting an object in the collection means to publish that object to all subscribers. Every DAC is a publish/subscribe engine of its own.
Now we will go info further detail on the interaction styles.

3.4 Interaction Styles

When people want to communicate with each other this can happen in many different ways. You might want to telephone you friend to ask him out, send a book to your mother or send a commercial flyer to all your customers. Just like the above examples there are different communication styles available for computer systems. Below, a short explanation will be given of all the major styles and their properties, but first a little word about some of these properties.

3.4.1 Properties

There are three main properties of interaction styles we will discus here, these are space-coupling, time-coupling and multiplicity.

Space-coupling means that the interaction parties need to ‘know’ each other in order to communicate. They need a reference to each other.

Time-coupling means that they have to be running at the same time for interaction to take place.

The **Multiplicity** is the number of receivers in the interaction. The number of receivers in a phone call is one, while the number of receivers of a mailinglist is the number of subscribers on that list.

There are other properties like speed, reliability etc, but these will not be discussed here. Now let us continue with the interaction styles.

3.4.2 Request/Reply

The Request/Reply Communication style is often used in Object Oriented Programming. When objects invoke other object's methods, the caller waits for a reply to his request. The consequences of this are that the sender must have a reference to the object it is requesting something from, and they both have to be available at the same time. This implies space and time coupling. Since the communication is just between two objects, if you want to send information to more objects, you have to make several Request/Reply communication channels. The Request/Reply style, although often used, has the property of making communication rather static and complex, due to the space and time coupling.

3.4.3 Shared Memory

In shared memory systems there is no space coupling as in the Request/Reply style. An object shares a data space with other objects. The objects that are interacting with each other, using this shared space need not know each other. However, the shared memory is available between a number of objects that are running at the same time. This means that there is a time-coupling. In this model, more than one object can read the data written in the shared memory, so the multiplicity of this model is the number of objects that shares the memory. In object oriented languages class-variables are usually shared. This implements a sort of shared memory for all the instances (and subclass-instances) of that class.

3.4.4 Asynchronous send

When using asynchronous send as style, the subsystem will make sure that, when you send a message, the message will be delivered to the addressee. The addressee does not have to be online the when you send the message. This is analogous to sending a package to someone. When sending a package, you have to know the address of the person to whom you send the package. The post-office delivers the package to the addressee, who does not have to be home at delivery. In this example the post-office is our subsystem, we will not go to the house ourselves to deliver the package, but rely on the post service. Since the correspondent does not have to be home at the time of arrival, it is clear that there is no time coupling. However, we still have to know to whom we send it, there is coupling in space. It is also a point-to-point connection, meaning there is one sender and one receiver.

3.4.5 Publish/Subscribe

The publish/subscribe style is analogous to a mailinglist. When there is a mailinglist about an interesting topic, people can subscribe to it. After subscription, they will start receiving messages, and be able to post messages themselves. None of the senders needs to know where the receivers are, and receivers don't have to be online. (They can receive their mail later) This obviously implies that there is space and time de-coupling.

In publish/subscribe there are publishers and subscribers:

The publisher is providing the information, and the subscriber is receiving it. There can be multiple publishers and multiple subscribers, and parties can act both as subscriber and publisher. It is obvious that the multiplicity of this style is equal to the number of subscribers.

3.4.6 Summary

Having seen the most important communication styles, we will give a short summary of their properties in table 1. All these styles have their advantages; for example, the Request/Reply style is used in most Object Oriented Systems because it is very simple to use and guarantees that the message was sent. All the above Interaction styles can implement either a push or a pull structure. This will be explained in the next section

	Time	**Space**	**Multiplicity**
Request/Reply	Coupled	Coupled	1
Shared Memory	Coupled	De-coupled	N
Asynchronous Send	De-coupled	Coupled	1
Publish/Subscribe	De-coupled	De-coupled	N

Table 1 Properties of interaction styles

3.5 Push and Pull interactions

We will explain Push and Pull model using a stock-market example: in the example we have one Stock-Server and several clients. When a Stock-Value changes, the clients that have the Stock are interested in knowing the most recent value. There are several ways to implement this.

3.5.1 Pull model

In the Pull interaction model the server is nothing more than a reservoir of values. Of course it will have a certain amount of concurrency control and other features, but it does nothing more than managing the values. The server provides some methods to let the client get the value of the stock. The client has to take an active part in this, since the client has to find out if the value has changed. Clients will ask the server for the most recent value of the stock on a regular basis. When the value of the stock reaches a certain level, the client might want to sell or buy it. It is therefore very important to know the exact value of the stock near that point. This implies that clients will start asking the value more often, creating a major bottleneck for the server. This model is called **Pull** since the clients have to pull the information out of the server. Variants of this model are pull interactions where the client request is halted until there is a change in value.

3.5.2 Push model

The Push interaction model turns the server into an active information provider. The server holding some stock-values keeps a record of persons interested in the stock-values. The clients can express their interest in a stock-value by telling the server about it. When the value of a stock changes, it is the duty of the server to notify the clients of the change. This would seem like a better idea since all the useless communication that the pull method caused, is no longer necessary. To make sure the clients can receive a notify event from the server, all the clients need to implement a way of being notified. Extending a predefined “notifiable” interface could do this for example. This will be explained later in the section about interfaces.

Another way of being notified could be by telling the server which remote method it is to use when the client subscribes. The provider notify function is then used when the server does a callback, that is, notifying the clients when new data arrives.

Problems with this kind of model arise when clients tend to go offline a lot. The server cannot contact the clients when they are offline, and the clients might not want to be online all the time, they might want to receive the data when they come online. This model puts a lot more responsibility on the server side; the information is pushed to the clients when it is available. Variants of this model keep track of the on/offline status of the clients and remember what was already sent to each client.

Same problems arise when clients move (migrate) to different machines.

3.5.3 Summary

In the above section, we learned that both push and pull models could be useful.
When we are using the pull model, clients decide when they want to receive the information. Clients can update themselves whenever they think it is necessary. The clients do not have to be available all the time. On the other hand, clients will not get an update from the server with the most recent value, but they will have to ask for it themselves. When many clients expect a change in value, they will all start pulling the data from the server causing the server to be overloaded.
If we are using a push method, clients will be notified of changes. Servers will only have to work when values change, and can handle the sending of notifies without being overloaded with requests. The clients however must always be prepared to accept a notify from the server. They always have to be online, they can not update whenever they want and they cannot move wherever they want without telling the server they moved.
If we have to choose between the push and the pull model, we would have to conclude that none is clearly better than the other. Both have proven their use in distributed computation.

3.6 *Communication model and style in DAC*

Now that we have seen different styles and models of communication, we will take a closer look at the communication model of Distributed Asynchronous Collections. DAC's use the publish/subscribe style to implement a push model with a possibility to pull.
Each DAC can be seen as a publish/subscribe engine of it own. When a client queries a DAC for objects, it expresses interest in such objects. This can be seen as a subscription to receive future notifications. The DAC will notify the client about updates on the requested topic.
The publish/subscribe style is a loose communication style: it provides time de-coupling of message-producers and message-consumers. This means that the information providers and receivers remain anonymous to each other. The style also has a multiplicity of N, allowing addressing to several destinations. In most applications, parties will be publishers and subscribers.
The classic publish/subscribe style is based on the notion of topics or subjects. This all will be explained in a later section about topics, but first the notion of services and Quality of Services will be explained.
To allow both the advantages of the pull and the push model, the DAC is also pull-able. For example, clients can also request data from the server without wanting to receive future notifications.

3.7 Quality of Service

Having multiple publish/subscribe implementations allows us to introduce Quality of Service into Distributed Asynchronous Collections. We will first see a couple of examples of services that can be offered using DAC's. In the first sub-section, we will see the difference between various delivery semantics. The second section is about order in delivery, and the last about duplicates. After seeing those services, we will see how they were used in the DAC's to implement quality of services.

3.7.1 Reliability of the delivery

The reliability of the delivery is often limited by the behavior of the deeper communication layers. For example: the properties of the network itself influence the reliability. Therefore, it is not always easy to provide the required services. It is also possible that different types of programs demand different reliabilities. We can divide reliability of delivery into three main groups.

- *Unreliable delivery*: Unreliable delivery gives you few guarantees about delivery. It is a best effort to deliver the messages, in order to guarantee other services. They are often used in applications where throughput is of primary importance, and the loss of certain messages is not fatal for the application.
- *Reliable delivery*: Reliable delivery means that there is guarantee that the messages will be delivered if the client is available. When a client is no longer responding to the messages this is not seen as a failure of the delivery, rather a failing of the client.
- *Certified delivery*: This is a form of reliable delivery where messages are certified when delivered. Even when the client is offline or not responding, the messages for that client are stored in a buffer to be sent whenever the client becomes active again.

3.7.2 Order in delivery

The order of delivery can be important to the client. If we look back at our stock example, we see that it is important to get the latest value last. If a stock value of 49 is rising to 50 and then to 55, we should first get the update 50 and than 55. If this is not the case, we might think that the stock is decreasing instead of increasing.

There are other examples where it is not so important if the messages arrive in order.

For example: if a company is receiving product orders, it is not important in which order they receive them. They could receive order x before order y, or after. Both orders will have to be made and delivered.

It is clear that there are two possibilities, either order is important or order is not important.

3.7.3 Duplicates in delivery

Some low-level protocols do not guarantee that there will be no duplicates in the delivery. For example if a message is sent but the acknowledge of the message was lost the sender might think the message was lost and send that message again, resulting in a duplicate receiving of that message. If we want to guarantee that messages will be received only once, we should do some checking on that. We again have two options, either duplicates can be received or no duplicates are allowed.

When duplicates are allowed, it could be the responsibility of the receiver to filter the duplicates out.

3.7.4 Basics of quality of service

Guaranteeing a QoS costs time and computation cycles. For example, if you had to guarantee a fast transfer for your streaming video file, you would not want the system to request frames that were lost, because by the time they reach you, it will be to late, and they will be skipped anyway.

It is clear that we do not want to have qualities that we do not need since this would make our program slower. In addition, different programs want different qualities.

In order to cope with these needs, we introduce the notion of quality of service into these DAC's. We can build different kinds of DAC’s with different implementations of the publish/subscribe style. We could make a DAC that is fast, secure, certified or a DAC with concurrency management...

To make it possible to use all these DAC’s in the same system, we specify the DAC interface. If an implementation of a DAC has all the methods defined in the interface, the original DAC can be changed into the new DAC without problems. A common interface guarantees that all DAC’s will understand each other. In a next chapter, we will discuss these interfaces.

In figure 2 we see an example of a decision tree, it suggests what kind of DAC a developer needs based on some simple quality of service attributes.

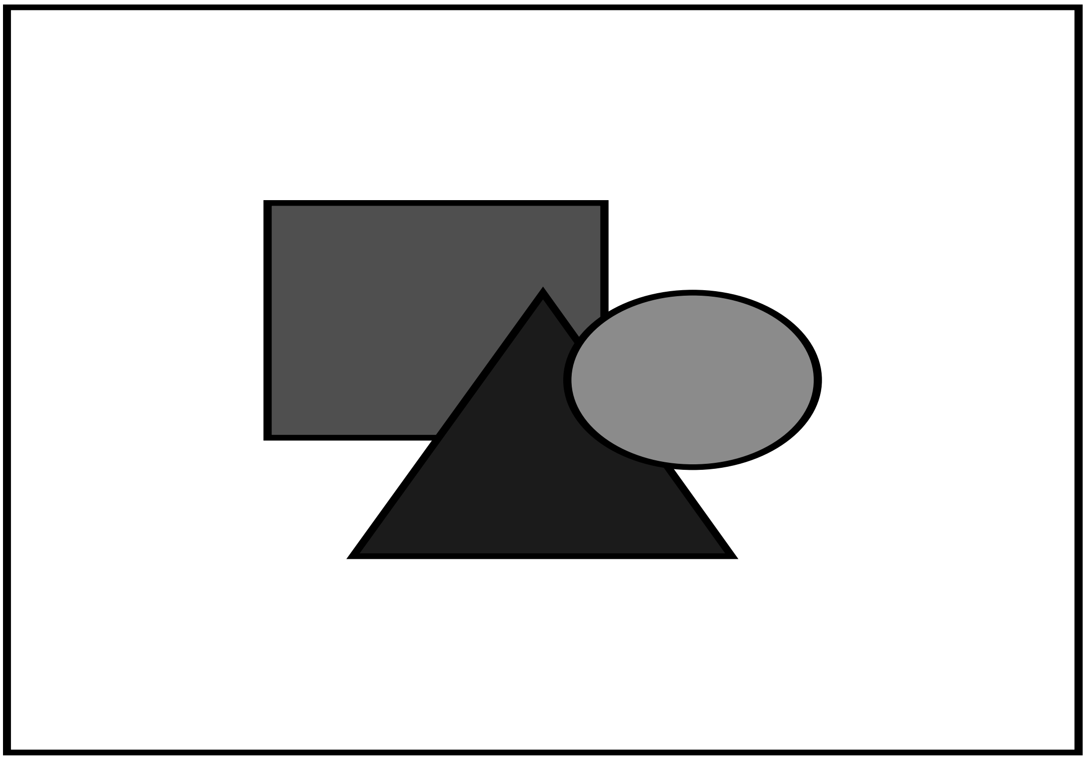

Figure 2 An example of a quality of service hierarcy

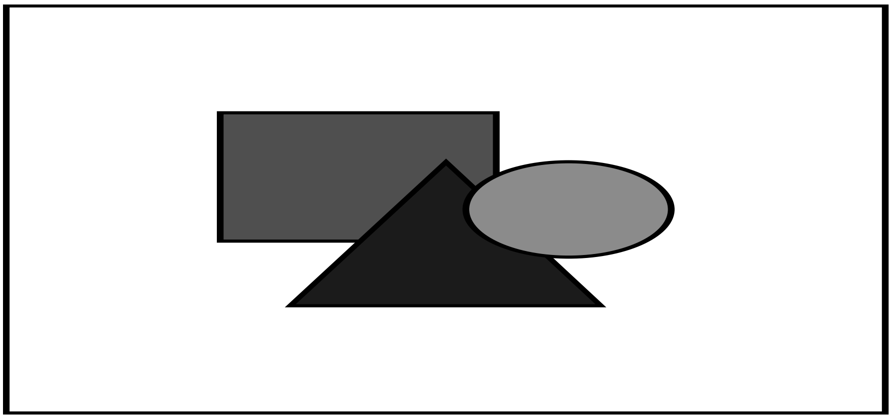

Figure 3 DAC’s and Topics of DAC’s

3.8 Topic-Based DAC's

Most publish/subscribe styles are implemented using topics. A client can subscribe to a topic. This client will then be notified when something is published on the topic.
In the DAC framework, a topic is a DAC, so whenever we talk about a topic this is in fact a DAC representing that topic. The DAC itself could be seen as a collection of DAC's called sub-topics. It is in charge of keeping the posted messages in an archive and notifying the subscribed clients about the postings. It is also responsible for the sub-topic management: making new topics, redirecting messages to the right topic, etc.

In figure 3, we see an example of such a DAC. The root-DAC called "/". This DAC contains two topics called "Quake" and "Chat". IF we now take a closer look at the Quake-topic, we see that this is a DAC of its own, containing its own subscribers and messages. This DAC also contains some sub-topics, called: "Servers", "Insomnia" and "Paintball". In the last part of the figure, we take a closer look at one of the subtopics, to find out that even this is a DAC.

Connecting to a topic requires a directory-like format. For example, "/Quake/Insomnia" is a reference to the topic called Insomnia which is a subtopic of "Quake". The root of the hierarchy is represented by an abstract topic (denoted by "/"). Top-level topics are sub-topics of the abstract root.

The DAC makes the position of the topics on the network transparent. All the topics would seem to exist on your local DAC while they in fact are distributed over the network. The inter-DAC communication is hidden form the user.
For example, when we have a DAC that is distributed over three different machines and on each machine a topic exists. We could have topic 'Topic_one' on machine one, 'Topic_two' on machine two and 'Topic_three' on machine three. Users of these three machines would see all three topics locally, and use them as if they were available on there own machine.
In fact, the location of the topics themselves does not matter. They could all be on a centralised machine, or the topics themselves could be distributed over different nodes of the network. The point is that no matter on which node of the network the DAC is accessed, the topics are still all available.

3.9 Interface

In the section about quality of services, we saw that different implementations of DAC's were possible according to the required service attributes. To make all these DAC's compatible, in the sense of reusable, exchangeable, an interface has been specified. We will use a Java like interface since they are well known by most.
If the DAC framework would be implemented in a programming language without the notion of interfaces, all the messages of the given normal interface should be implemented. This would guaranty compatibility between all DAC's.
There are two parts in this interface since the server (the DAC) as well as the client need to specify methods to be called. This could be seen as a protocol for interaction between the client and the DAC. Both interfaces will be explained below.
Since a DAC is primary a collection, the interface of the DAC will resemble the interface of a collection. The behaviour however will be somewhat different.

3.9.1 DAC methods

In extract 1 we find the main methods of the base DAC interface. We distinguish two types of methods: pull methods and push methods. We will first briefly explain the pull methods.

Pull methods:

- get : Calling this method allows to retrieve objects from the collection.
- contains : Queries the collection for the presence of certain objects. (Only parameter is a specification for an object.)
- add : Add an object to the collection.

Push methods:

- contains(Notifiable) : With this method, the client expresses its interest in objects with these characteristics. (Parameters are a specification for an object and an notifiable interface.)
- containsAll(Notifiable) : Has the same function as the above, but the subscription is generated for all subtopics too.
- remove(Notifiable) : This method tells the DAC that when an object matching the criteria is inserted, this client should be notified after which the object has to be deleted.

- removeAll(Notifiable) : Same as the above but holds for all subtopics as well.
- clear(Notifiable) : Unsubscribes the client.

```
public interface DAC extends java.util.Collection {
    public Object get();
    public boolean contains(Object o);
    public boolean add(Object o);
    …
    public boolean contains(Notifiable n);
    public boolean containsAll(Notifiable n);
    …
    public boolean remove(Notifiable n);
    public boolean removeAll(Notifiable n);
    …
    public void clear(Notifiable n);
}
```

Extract 1 DAC-Interface

3.9.2 Client Methods

With the push method adopted in DAC’s, subscribing entities must register a *callback* object. That callback object must implement a specific interface namely, the Notifiable interface.
Trough the call of the contains method, the DAC notifies the subscriber that it contains new information. The second argument allows us to use the same callback object for several topics.

```
public interface Notifiable {
    public void contains(Object o, String topicName);
}
```

Extract 2 The Client interface (Notifiable)

3.10 Using DAC’s

In this section, we will give a small example of how to use a DAC. The example will use a DAC that implements a stock market. The different topics on the DAC represent different stocks. The Stock server holds a stock

called “L&H”. We will see the example of three persons that are interested in this stock. The value of the stock is automatically updated.
The first two persons (Aram and Bert) subscribe themselves on the topic.

1) They create the collection : DAC stock_LH = new DAC(“L&H”);
2) They subscribe themselves : stock_LH.contains(self);

The third person (Dree) wants to check the stocks whenever he feels like it, so he uses the pull methods implemented by the DAC.

1) He creates the collection : DAC stock_LH = new DAC(“L&H”);
2) He requests the latest value : stock_LH.get();

Then at a certain point the stocks value decreases rapidly. Every time the value of “L&H” is updated both Aram and Bert are notified. They both wisely sell their stocks of “L&H” and unsubscribe using the clear(self) method. Imagine the surprise Dree will have when he checks on his stocks.

3.11 Summary

In this chapter, we explained the basics of Distributed Asynchronous Collections (DAC’s). We started by explaining the interaction styles Request/Reply, Shared Memory, Asynchronous Send and Publish/Subscribe. Next, we explaind the push and pull communication model. Then, we clarified the interaction styles used by the DAC’s and concluded that DAC’s use both the push and pull model. The publish/subscribe style is implemented using attributes form an ordinary collection.
The DAC’s provide a middleware function for the programmer; i.e. they provide a publish/subscribe style on top of the normal request/reply model.
The notion of Quality of Service was introduces in this chapter and proven useful in this context. Through the implementation of various DAC’s with different quality attributes, we can create exactly the DAC we desire. Since every attribute costs in time, we have to make choices.
To be able to change a previous implementation of a DAC with a new one, we specify the interface they have to implement. If clients want to take advantage of the push style of the DAC’s they have to implement a client interface called *“Notifiable interface”*.
We concluded this chapter with a demonstration of these DAC’s.

4 INTRODUCTION TO BORG

"You will become one with the Borg.
You will all become one with the Borg"

The Borg

In this chapter, we will introduce the programming language used to do the programming in this thesis. This programming language is **BORG** [6], a distributed mobile agent programming language.
First, the basics of BORG will be explained. Secondly we will demonstrate how to create objects and how to manage them in an object oriented way. We will also demonstrate different agent interaction possibilities. To conclude this chapter we will give a comparison between BORG and existing languages that support distribution.
This Chapter has been based upon the BORG Chapters in the work of Philippe Demaecker [17] and Werner Van Belle [7].

4.1 BORG basics

BORG was originally developed as a mobile multi-agent-developing environment. It was developed because the field of distributed computation was lacking an environment with enough facilities, like strong migration, asynchronous method calling, synchronisation, etceteras.
BORG is based on the programming language Pico [5]. It provides a very simple syntax with the expressiveness of scheme, which is a Lisp [15] variant. BORG is an extension of Pico: new native functions were added to support mobile multi-agent systems. These native functions can also be used for distributed programming.
BORG is an experimental programming language, therefore it still lacks some functionalities like decent error and exception handling. These will no doubt be added in future releases.

4.1.1 Definitions

Before starting, we will give a couple of definitions of terms that will be used in this chapter. These are the definitions provided by the developers of the BORG language [6], and it is possible that they mean something else in another, non-BORG related context.

Agent:

An agent can be described as a single threaded program with an in-queue and out-queue, respectively used to send and receive messages. The agent can be referred to through his agent-name and has his own code and data space. An agent contains a program that determines his behaviour, which could be seen as autonomous. This behaviour can only be influenced by sending messages to the agent.

Mobile agent:

This is a stand-alone agent, able to communicate with other agents, which can migrate to other places (machines).

Mobile multi-agent system:

A Mobile Multi-Agent System is a system that allows multiple mobile agents to communicate, and work together.

Such a system is the infrastructure, which executes these mobile agents. This system is responsible for the delivery of the messages, for the scheduling of agents and their serialisation.

The term "agent" is also used in the domain of artificial intelligence. Here it usually signifies a matter of autonomous intelligence. The AI-agents are autonomous and learn from their environment. They might have the possibility to migrate. A Mobile Multi-agent system could easily support these kinds of agents. The intelligence of the agent is contained in the program, and that program can migrate transparently in the Mobile Multi-Agent System. BORG mobility is certainly not a problem for AI-agents in Mobile systems.

4.1.2 Creating an Agent

In the previous part we explained that each agent has its own data and code space, these will be allocated when the agent is created. The size of this memory can be dynamically resized. The code-space contains the source that implements the agent, while the data-space contains the variables. It is important to note that other agents cannot access the data and the code space.

An in-queue and an out-queue are reserved for the agent and every agent gets his own execution stack. The stack has a data part and a code/continuations part. The single thread that the agent contains uses this execution stack.

4.1.3 Creating an Object

BORG was not designed to be an Object Oriented language, but it can be used to do some object oriented programming. It is in fact very simple to create an object that encapsulates some data. We will illustrate this using the counter example. We will make a counter object and increment it a couple of times after which we will ask the value.

```
create_counter() :: {
    value : 0;
    get_value() :: value;
    increment() :: { value := value + 1 };
    clone() }
‘<function create_counter>

count1 : create_counter()
‘<dicionary Y>

count2 : create_counter()
‘<dicionary Y>

count1.increment()
‘1
count1.increment()
‘2
count1.get_value()
‘2
count2.get_value()
‘0
```

Extract 3 Example of the counter object

To understand the example the following extremely short introduction to Pico might be handy.

```
a : 5;                          ''Definition of variable a to the value 5.
a := 6;                         ''Assignment of variable a to the value 6.
b :: 0;                         ''Definition of the constant b to value 0.
fun(x) : { display(x); }        ''Definition of function fun with argument x.
obj.method(arg);                ''Calling a method on an object.
for(j:1, j>50 , { display(j);}  ''A simple for loop for(init, cond, exp).
```

Extract 4 Extremely short introduction to Pico

In extract 3 we see that we have to define a counter object, after which two counters are created. *Count1* is incremented two times and then the value is asked. We see that the value of *Count2* is still zero, which signifies that *Count1* and *Count2* are two different objects.

The clone function is called at the end of the definition of the counter. This function will be explained next.

4.1.4 Cloning

The function clone() returns a reference to the local environment. This might seem strange at first, since clone is supposed to mean: *take an exact copy*. We will explain later, using the environment model, why this actually makes sense. Now, we will discuss the clone operator using a more advanced counter example.

```
make_counter() :: {
    value : 0;
    increment() :: { value := value + 1 };
    get_value() :: value;
    make_advanced_counter() :: {
            inc : 1;
            reset()        :: { value := 0 };
            set_value(val) :: { value := val };
            set_increment(in)  :: { inc := in };
            increment() :: { value := value + inc };
            clone() };
    clone()}
```

Extract 5 An advanced counter example

In this example (extract 5) we see an extension of our previously defined counter where a method make_advanced_counter was added. In the environment model drawn in figure 4, we see the global environment as the top rectangle. When the procedure make_counter is executed, a new environment is created where the procedure is executed. The variables of the procedure are bound here, and when a variable is referenced, this is first checked in the make_counter environment and then in the global environment.

The clone operator returns a reference to the make_counter environment. When we call the make_advanced_counter method in the make_counter environment, it creates the make_advanced_counter environment and provides a link to it using the clone operator.

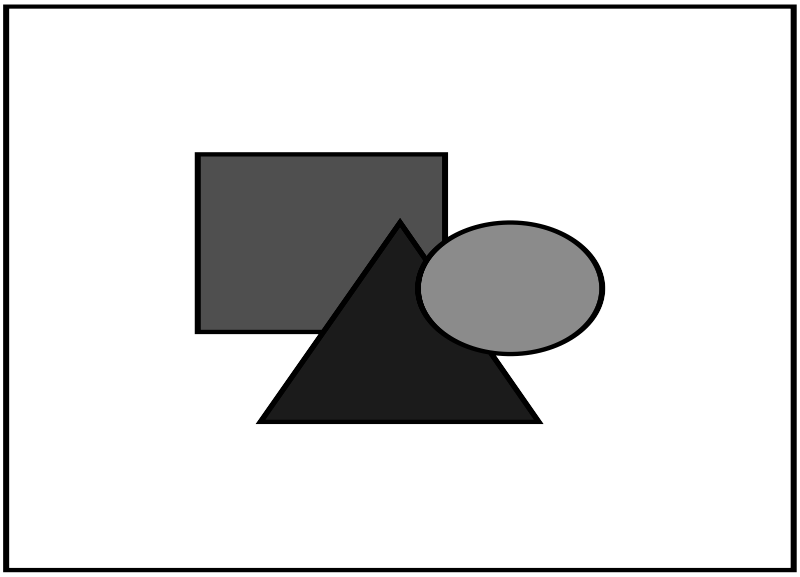

Figure 4 The environment model

If we want to make another counter object, we have to execute the make_counter procedure again which would create another environment with the correct values and the correct functions in it. So there is no problem with overlapping. Using the following piece of code, we will try to demonstrate how the clone operator works. We will use the implementation of make_counter from extract 5 in this code.

```
c1 : make_counter()                 ‘<dictionary Y>
a1 : c1.make_advanced_counter()     ‘<dictionary Y>
a2 : c1.make_advanced_counter()     ‘<dictionary Y>
c1.increment()                      ‘1
a1.increment()                      ‘2
a2.increment()                      ‘3
a1.set_increment(2)                 ‘2
a2.increment()                      ‘4
a2.inc                              ‘1
a1.inc                              ‘2
```

Extract 6 Environments and cloning.

In this example, we try to explain how the clone operator works. There are two calls to clone() in the make_counter() function. One is at the end of the make_advanced_counter method, and the other one at the end of the make_counter block. We know that clone() returns a reference to the current environment. The execution of a function constructs an environment where the formal parameters are bound to the actual parameters. All the other objects defined in that block are stored in this environment. It is now easy to see that the value of c1 in extract 6, is the same variable as in a1 and a2. Both a1 and a2 have a different inc value at the end, since they are in different environments.

The technique of cloning is frequently used in prototype based systems.

4.1.5 Agentclone

The agentclone primitive can be used just like the clone primitive, but instead of making an object, by returning a reference to its environment, it creates a new agent, giving it his own data and source space. It returns a reference to the new agent as result. Agentclone takes an object and an agent-name as parameters. In the clone section we saw that an object is nothing more than a reference to the environment of the object. When this reference is given as a parameter to the agentclone primitive, the environment is copied to the position of the new agent, where it is put in its own data and code space.

The problem of the agentclone operator is that the environment of the object is copied to the new location, and every environment that is in a scope-level above is copied as well. The root-environment will not be copied. The root-environment contains the native functions of pico and the communication extensions of BORG. Since the environments are copied, the technique shown in extract 6 will no longer be valid in our example in extract 7.

In extract 7 we see that the value of a1 is different form the value of c1 although it is the same in extract 6. This is because the agentclone operator takes a deep-copy of the agent’s environment, and moves it to the other location. All local objects that are referenced will be copied and move as well.

Sending the deep copy over the network will take longer, as sending a shallow copy with a reference to the remote objects. Both agents behave independent after they were agentcloned, and have their own data and code space. It is important to remember this difference between a local clone and an agentclone, because of their different behaviour.

```
make_counter()  :: {
    value  : 0;
    increment()   :: { value := value + 1 };
    get_value()   :: value;
    make_advanced_counter() :: {
            inc : 1;
            reset()              :: { value := 0   };
            set_value(val)       :: { value := val };
            set_increment(in)    :: { inc := in      };
            get_value(ret)       :: { ret.display(value) };
            increment()          :: { value := value + inc  };
            agentclone(clone(),"adv_count")   };
    clone() }                      '<function create_counter>
c1 : make_counter()                '<dictionary Y>
a1 : c1.make_advanced_counter()    '<reference"/prog/adv_count">
c1.increment()                     '1
a1.increment()                     '<void>
a1.get_value(clone())              '<void>1
c1.increment()                     '2
a1.get_value(clone())              '<void>1
```

Extract 7   The agentclone operator

4.1.6 Creating agents

Agentclone is one way of creating an agent; another way is to create a new session in your BORG environment. (Just by clicking on the “new document” button.) In this newly created session, session is equivalent with agent, you can type and execute code. The evaluation of that code is done by selecting it and pressing ctrl-<enter>. The name of your agent will be in the title-bar of the session. It will have an URL-like structure, like "/prog/Alice" where prog is the name ofythe computer and Alice the local name of the agent. The naming will be explained further in a later section.

4.2 Migration

Before we explain the communication abstractions, we will first explain migration. In the introduction of this chapter, we talked about strong migration, it is this kind of migration that BORG supports. We will start by explaining the difference between strong migration and weak migration. Next, we will continue to explain how migration works in BORG.

4.2.1 Strong migration versus weak migration

Migration means moving processes to other computers or other data-spaces. There are two kinds of migration, there is strong migration and there is weak migration.

Weak migration is done using special states in which the agent is transmittable. The cooperation of the agent is required before the agent can be moved to a new system, since the agent has to be in a “transmittable state”. The state of the agent will be saved in the data. When the agent arrives on his new place, he will call a start method to restart his work.

Strong migration [7] means that agents can be moved whenever the programmer or the agent-system wants them to. The execution stack of the agent must be sent over the network to the other location, where it has to be reinstalled so execution can continue. To be able to do this, the computational state of the agent must be reified and serialized, also the stored data must be serialized and sent to the new position.

4.2.2 Migration in BORG

In the above sections we learned about weak and strong migration, we saw that for strong migration every agent has to migrate its own execution stack. In BORG every agent has its own data and code space, and also his own execution stack. BORG provides some primitive operations to support migration.

The interpreter of BORG is capable of reifying and serializing the computational state and the stored data. Even when an agent is in the middle of a computation, the agent can be moved to a different location where it seamlessly continues the computation. The agent is only moved when the stack is in a consistent state.

Agentmove(agentname) moves an agent to “agentname”. The example we find in extract 11, is an agent that calculates the factorial, and every recursion-step is done on a randomly chosen other machine. The first two displays “factorial” and “moving to” will be executed on the same machine, “Done” will be displayed on the next randomly chosen machine. This example is from the BORG site [6]. In the above comments we also see that the agent will be lost when it is moved to a position that does not exist.

```
‘ machines should contain a list of machines where to jump to
‘ if a machine doesn’t exist the agent is lost
machines:
   [     agent("cubical/Alice&ui"), agent("cubical/Christine&ui"),
         agent("cubical/Bob&ui"),  agent("cubical/Dirk&ui")  ]

fac(i)::  {
          display("factorial " + text(i));
          machine:machines[(abs(random())\\size(machines))+1];
          display(" moving to "+ text(machine)+eoln);
          agentmove(machine);
           display(" Done ");
          if (i > 1, fac(i - 1)*i, 1)}

fac(12)
```

Extract 8 Agentmove function demonstrated

4.2.3 Naming and routing in BORG

The BORG routing system uses some ideas the name-services on the Internet uses. In a collection of BORG systems, there is one agent system that acts like a name-server. This naming server holds the current locations of every agent. The names of the agents are formed using the agent-system on which they were created and their agent named. The agent Alice created on Prog1 would be called “Prog1/Alice”. The naming server would bind the name “Prog1/Alice” with the current location of that agent in the naming table, and update it if it migrates.

For the routing of the messages to the correct agent, the routing system looks for the current location of the agent, using the naming table. The routing system then sends the messages to the agent-system where the agent currently is, and that agent-system delivers the message to the agent that is local to the agent system.

Now, what happens when an agent moves to another position? First of all the name of the agent remains the same. If the agent Bob was created on agent-system Prog1 then the name will always be “Prog1/Bob”. Upon moving the agent to the new location, Prog2 for example, the name-service would be updated to redirect the references for “Prog1/Alice” to Prog2 instead of Prog1. Prog1 would also redirect messages send locally to “Prog1/Bob” to Prog2.

4.3 Communication

There are two ways of communication. The first one is using remote method invocations; the other one is done using the send and receive primitives.

4.3.1 Remote Calls

We can make a remote reference to an agent by using the "agent" primitive. Using this reference you can call the agent's methods just like calling methods of objects.

/prog/Alice	/prog2/Bob

b : agent("/prog2/Bob")
'<reference"/prog2/Bob">
 	
b.say_hello("Hello World")
'true
	say_hello(str) : {
	display(str);
	return(true)}
	'<function say_hello>
	
	'Hello World

Extract 9 Remote Method Invocations

In extract 9 we see that we create a reference to an agent called "/prog2/Bob" and that we can call a method called say_hello that is defined in agent Bob. That is how a remote method invocation is done. We see that remote and local objects and agents have the same calling mechanisms. This reminds us of the unified object theory we talked about in paragraph 2.2.6. The reason why this seems to work here, is because we do not know if an agent is local, remote, or was local and just migrated to a remote position. Therefore, we have to pretend that all agents are remote.

4.3.2 Asynchronous Send and receive

Just like the agent primitive, there are primitives that allow you to send and receive data to other agents. This can be done synchronously and asynchronously. We need the "agent" primitive, above explained, to create a reference to the agent we want to send data to. See also extract 10.

The special word 'any' is used as a wildcard in the pattern to receive the messages. A pattern is usually build as a table. With as first element the id of the receiver, and the rest is data. Extract 10 shows how the keyword 'any' can be used, and patterns are usually build. The messages are delivered using a pattern matcher. The sender sends a message with a pattern, and on the receiver’s side there is a pattern matcher that selects the correct message. This makes sure that the different message senders and receivers of an agent get the correct messages. The pattern could also be used as protocol information, where the pattern specifies what should be done.

When sending information to another agent, the sender does not have to specify neither a port nor the amount of data he is going to send. This makes it all a lot easier than it was in socket programming.

The send and recv primitives are asynchronous, which means they do not wait until the other agent answers. Synchronous send and recv operators are called ssend and srecv. They use the primitive sync operator to ensure message delivery.

/prog/Alice

```
b : agent("/prog2/Bob")
‘<reference"/prog2/Bob">

send(b,[1, “Hello World”]
`void

send(b,[1, “I am Alice”]
`void
```

/prog2/Bob

```
b : agent("/prog/Alice")
`<reference"/prog/Alice">
c : agent(“/prog/Clara”)
`<reference"/prog/Clara">

recv(c,any)
`<no message>
recv(b,[2,any])
`<no message>
recv(any,any)
`[1, Hello World]
recv(b,[1,any])
`[1, I am Alice]
```

Extract 10 Send and recv primitives

4.3.3 Syncing agents

Agents have the possibility to sync with each other. If both agents have to finish something before they continue, the first calls sync and he will be halted until the second calls sync. The sync primitive takes an agent as argument. When both agents have called the sync operator, execution can continue. The sync operator can have an optional parameter, usually a pattern, to specify what the agents may synchronize on. Since the execution is effectively stopped when syncing, this might cause problems when the other agent is not available or not responding. Since we cannot be sure that we can continue if the sync partner is unavailable, this behavior is correct.

4.3.4 Synchronous ssend and srecv

In Extract 11 we see how the ssend and the srecv operators can be constructed from the send, recv and the sync. The communication is now synchronous since we know that the other agent is ready to recv the message. Note that the receiver will effectively wait until the message is received before continuing. This provides a sort of certification of delivery. The pattern specified to the srecv operator will be matched with all the messages form the message-queue.

In the next section strong migration in BORG will be explained.

Operator ssend	**operator srecv**
`Ssend(to, message) : { send(to, message); sync(to, message) }`	`srecv(from, pattern) : { sync(from, pattern); recv(from, pattern) }`

Extract 11 Ssend and srecv primitives

4.4 Comparison

In the second chapter, we explained Java remote method invocations and CORBA. We will now compare these two environments for distributed computation with the BORG system. We will try not to go into to much detail when we are talking about CORBA or Java RMI, since it would not contribute to this thesis. A comparison is however necessary to explain why we used BORG as implementation language.

4.4.1 CORBA

We already mentioned CORBA when we were talking about Remote Procedure Call's in 2.3.3. Now however, we will compare them with BORG.

CORBA defines a standard that allows communication between different components. These different components can be written in different programming languages and can be in different data-spaces. In order to come to such a standard a shared definition (IDL) language is necessary. By defining interfaces and compiling them, stubs are created. These stubs act as the local versions of the remote objects. CORBA allows the communication of components written in different programming languages. This allows communication through remote procedure calling. Commercially it is proven to be very useful. To guarantee compatibility the Interface Definition Language is restricted to the common denominator of all the languages which can be used. Learning to program in CORBA is not easy.

Migration is not implemented in CORBA. First, it is possible that the components do not work on the migrated position. For example: a component compiled on *machine_A* will not necessary execute on *machine_B*. Second, none of the programming languages support migration in their native language, which makes it the responsibility of the CORBA middleware that is supposed to go unnoticed. With remote procedure calls we can pretend that the object we are calling is a stub on the local machine, with migration this becomes more difficult or even impossible.

CORBA seems to fill a gap in the market, since there is need for inter-component communication, in the same address-space or in a different one. CORBA is very successful because it allows different components to work together. The need for CORBA comes forth form the fact that many people know existing languages, and components are already build in these languages. CORBA tries to fill another gap; a request for programming languages for distributed programs. It is clear, that the program-languages of today are not powerful enough to provide such tools themselves, and need middleware like CORBA to extend their functionality. Instead of searching for new and better things, people stick to what they know.

4.4.2 Java RMI

Java-components can be used in CORBA-applications, but we cannot use some Java-types for communication between two components written in Java when using CORBA middleware. This is an enormous restriction to the expressiveness of Java. Luckily, Java contains some abstractions for communication between different virtual machines of themselves. It is therefore not necessary to use CORBA if all you have is Java-components.

Java RMI [9] is a collection of tools to provide communication between two Java components on different Java VM's, possibly on different computers. In the section on remote procedure calling (2.3.3), we already explained how the RMI mechanisms worked. We need extra tools needed to generate stubs, the naming services, etc. The static-ness of the interfaces that have to be specified is awkward. Although some of these problems have a solution, these solutions tend to make the implementation more complex.

Java code is compiled to a machine independent form. The compiled code, called Byte-code, runs faster than interpreted source code. However, Java byte-code still has to be interpreted by the Java VM. The fact that it has to be interpreted and that it is in machine independent form could lead to the conclusion that we might be able to do migration with this code. This however, is not the case. Since we cannot access the runtime stack and runtime data, we cannot serialize it and re-run it. If we would like to do this, we would have to build our own execution stack, or adapt the Java VM.

Java is however, a commercial product and is used all over the world. It has proven its use and filled a gap in the market.

Java also has static typing while BORG has dynamic typing. Static typing has advantages, but in our opinion it makes the language far less expressive and thus less powerful.

4.4.3 BORG

BORG is an educational platform and will therefore never conquer the world as Java has. However, in comparison with CORBA and Java RMI, a language like BORG is promising. BORG offers dynamic typing and communication primitives so you can create true distributed programs.

For the moment, BORG has little or no security. Agents can go and do anything they want everywhere they want to. Commercially that is unacceptable, but for research purposes, it can be accepted. We can even be glad not to have to worry about the security issues. BORG also has a very bad error handling. We are not informed about unreceived messages, and even detectable errors are unnoticed. This makes the program a little difficult to use. BORG knows some other problems, like when agents go to a non-existing place, they disappear, and messages to undefined agents keep looping in the routing system. Future versions of BORG will no doubt correct these. The primitive communication abstractions make it easy to program distributed applications. BORG manages the communication between agents and tries to guarantee communication and location transparency.

BORG is a prototype-based language, not class based like other Object Oriented Languages. Prototype-based languages use clones (i.e. copies) of objects to extend them. The only problem we see in prototyped languages is that usually we have to adapt the base class if we want to extend it with extra functionality.

The BORG Virtual Machine interprets the BORG language, just like the PICO VM interprets Pico. Furthermore, every agent has its own execution stack, data and code space. This makes it possible to migrate the agent to a different BORG Virtual Machine. Note that Virtual Machine is a valid term since the Pico interpreter was written in such a way that it would be easy implemented on a stack machine. BORG does not need stub-compilers or interfaces to specify the callable methods. It has a naming service that is build into the BORG-environment. BORG is an excellent example of what a language for distributed computing should look like.

4.5 Summary

In this chapter, we introduced the BORG language and the concept for which it was designed. We have looked at how objects and agents are created and we gave an introduction of how objects and agents are cloned.
Migration and some problems concerning migration were explained next.
We continued with the communication mechanisms of BORG, looking at synchronous and asynchronous messages, syncing and remote procedure calls.
Finally, we concluded with a comparison between some programming environments for distributed programming, and concluded that BORG is a good example of what a language for distributed programming should look like.

5 DAC’S IN BORG

“Common sense is the collection of prejudices
acquired by age eighteen.”
Albert Einstein

5.1 Introduction

In this chapter, we are going to describe our implementation and clarify the decisions we had to make when implementing Distributed Asynchronous Collections (DAC’s) in the BORG language. We will first discuss why we used BORG for the implementation and not, for example, Java. Next, we will give the specifications and requirements of the DAC’s. Finally, we will explain how we implemented them.

5.2 BORG as implementation language

BORG uses asynchronous procedure calling when calling a method on remote agents. Execution can continue locally after a procedure is called on a remote agent. If we would like to achieve this in Java, we would have to use an extra thread for handling the sending and receiving of the calls. Threads can be very useful and powerful, but we would only need them in this case to simulate asynchronous procedure calling.
Note that agents in BORG are single threaded, but there are many agents running at the same time. The communication between the agents is handled by BORG primitives.

In the chapter about DAC’s we learned that a DAC should guarantee asynchrony meaning time de-coupling of publisher and subscriber. Since BORG is asynchronous by nature, it would be a good choice for an implementation language. The BORG primitives do not support multicast or broadcast messages, so the publish/subscribe style will have to be implemented as an iteration of the send command over all the receivers.

5.3 Concept and design

We will start this paragraph by quickly explaining the DAC concept, based on an implementation view. Then we will discuss some design issues. We end this section by introducing the need for Quality of Services.

5.3.1 Concept

We have explained the concept of DAC's in chapter 3. The design of the case in the programming language BORG however, is not straightforward.

A DAC represents a collection that is shared between all its subscribers. Let us first discus the basics.

- The collection must be distributed. Users all over the world must be able to access the collection. This behaviour is similar to a global variable, that all users can see. Note that a distributed collection could also mean that the data of the collection is distributed over different computers. This is however not specified in the DAC-documentation, and it is left to the implementers to decide whether or not to distribute the elements of the DAC.
- Asynchronicity is an essential aspect of DAC's. Method calls are usually synchronous calls, meaning that the caller waits until an answer is received before continuing. In BORG however this is not always so, since we have asynchronous communication operators. Although asynchronous messages can be simulated with synchronous calls, we will see that the BORG asynchronous communication operators will be very useful in our implementation of the DAC's.
- Although it is not described in the DAC specifications, we believe it to be useful, if possible, to have the collection available at all times. Important aspects in distributed computation, like single points of failure, have to be taken into account in order to give better guarantees on availability.

The DAC's have an interface that describes which methods can be called. This interface is an extension of the Collection interface since a DAC is a collection. Due to the asynchronous nature of these DAC's the Clients are required to implement an interface as well, with some methods used as callbacks.

5.3.2 Design

Clients implement the Client-Interface, as explained in the DAC-Chapter, in order to receive notifications from the DAC's when they are subscribed. A DAC will also implement the Client-Interface because, when the DAC is created as Subtopic, it will subscribe on a parent topic to receive the events. So both Subtopics and Clients are addressed using the same interface. We could see the DAC hierarchy as a tree where the Clients are leaves and the DAC's are nodes.

When starting the DAC-subsystem, a Root-Topic will be created. If a user wants to create a Topic, this will be created as a Subtopic of the root Topic. Then the Subtopic will subscribe itself on the DAC Root-Topic.

It is important to know that, in our implementation, Topics and DAC's are the same. A DAC is a representation of a Topic and the Subtopics of that topic are other DAC's. So, when a Topic creates a Subtopic, it creates a new DAC that will handle all Subtopic events.

5.3.3 Implementation issues

One way to create DAC's in BORG is to make one agent as a representation of a Topic. In this way, the creation of a new topic would be equal to the creation of an agent to represent that topic.

In this "one agent per topic" approach, we create a single point of failure. If one agent goes down, the entire topic is down as well. A better way could be to create a local proxy of the DAC upon subscription to the topic. This however entails updating problems, which require complicated updating and consistency management.

Using the idea of Quality of Service, we let the decision, whether we use a "one agent per topic" or a "local proxy" approach, be taken by the users. This implies the implementation of both the "one agent per topic" and the "proxy" way.

The Quality of Service cannot influence the interface between clients and DAC's. The QoS attributes must be seen as something additional to the basic interface of the DAC's, to provide extra functionality. Therefore, the QoS attributes must have an additional specification for client server communication.

The use of agents as a representation of DAC's could be useful if we want to migrate a Topic to another location, close to the Clients, in order to boost performance.

5.4 Quality of Service

In the previous paragraph, we mentioned Quality of Service (QoS) and Quality of Service Attributes. We have already seen QoS in the chapter about DAC's. Here we gave examples of QoS attributes like reliability, duplicates and order of delivery. Besides these attributes, there are, of course, more specialised attributes. For example, if a resource has to be locked for use by one client only, this could be seen as a Service of that DAC.

5.4.1 Concept

Quality of Service is defined on the communication channel between the client and the Topic. Each Client can request the QoS Attributes they need. These Attributes will be passed to the DAC upon subscription. For example: "Topic1.subscribe(self, reliable);", where "reliable" is defined as a constant.

Since there are many clients communicating with a Topic, they can all demand different Quality of Services from that Topic. We will explain this with the stock market example.

Example: A Stock is represented by a Topic, and the updated values are posed on this Topic. Three clients subscribe to this Topic called "Zabena". Our first client is Aram, he requires reliability and no duplicates as QoS. The second client, called Bob, subscribes to "Zabena" after buying stocks from the company. He requires QoS attributes reliability and order. The third client Cecil does not require any QoS attributes. Both Aram and Bob require reliability, but only one of them requires Order. Because the DAC-subsystem has to provide Order of Delivery, all the posted messages have to be identified and numbered for this Client. Therefore, all messages must be gathered at one place, to be able to define an ordering. Because of this gathering messages cannot be send as soon as they are received. The other Clients can just ignore the defined ordering, but will bare the delay of gathering.

5.4.2 Problems

From the above example, it is clear that the DAC's must be able to handle all Quality of Services demanded by the clients. Since the QoS Attributes are defined on the communication-channel between Client and Topic, no interference between different QoS Attributes is expected. However, some attributes might need the entire DAC sub-system to behave differently, as demonstrated in the example about Order of delivery. Note that if the entire subsystem of the DAC's has to change, it is possible that other QoS Attributes become unusable. A Client would not be able to demand Fast message passing, when Order of Delivery is on because of the overhead generated by the Order of Delivery Attribute.

Which QoS attributes should the server use if two clients require uncompatible QoS Attributes? Our solution is that the first client determines witch QoS attributes are important to him, and the subsequent subscribers can only request attributes that are compatible with the attributes requested by the first client. The other QoS attributes, requested by later subscribers, will be ignored. In our implementation all the attributes are compatible which makes things easier.

5.4.3 Inheritance

Different Quality of Service attributes need different implementations in the DAC's, as seen in Figure 2. We will use prototype-based inheritance to implement the different implementations. An example of prototype-based inheritance is given in extract 12.

```
‘ In our base-DAC a method exists to make a Reliable DAC form
‘ the base DAC like the method makeProtected in this stack example
Stack(n): {
        T[n]: void;
       super: void;
       t: 0;
       empty()::   t = 0;
       full()::    t = n;
       push(x)::   { T[t:= t+1]:= x; void };
       pop()::     { x: T[t]; t:= t-1; x };
       makeProtected()::  {
                push(x)::  if(full(), error('overflow'), super.push(x));
                pop()::    if(empty(), error('underflow'), super.pop());
                capture() };
        super := capture() };
S: Stack(10);
if(S.full(), void, S.push(123));
if(S.empty(), void, S.pop());
R: S.makeProtected();
R.push(1);
R.pop()
```

Extract 12 Prototype based inheritance

In our implementation, we will first create a Base-DAC. When a Client subscribes to this Base-DAC, the Client gives the requested QoS Attributes. The DAC will make an appropriate responder to handle the communication with this Client, and add this responder to his responder-list. Every Client gets such a responder. The basic responder will be extended, using prototype inheritance, until the correct QoS Attributes are handled.

For example, a client subscribes in topic “Test” requiring a Reliable and Lockable Topic. The Topic will handle this call and create a basic responder. The basic responder will be extended by calling the makeReliable() method defined on the responder. This would make the responder a Reliable one. Then, by calling the makeLockable() method we would enable Locking. Finally, the reference would be returned.

5.4.4 Watch out when extending

By calling the makeReliable() method, a number of methods of the base-responder are going to be overridden. This could give problems, as we will demonstrate in the following example:

In the previous example there would be three ‘publish’ methods, one from the base responder (p1), one form the Reliable responder (p2), and one from the Lockable Responder (p3). So, p1 would be overridden by p2 which on its turn, would be overridden by p3. Now however the behavior for p2 and p1 would be lost, therefore it is necessary to call the overridden method in the body form the overriding method.

The interface of Clients that want QoS will sometimes need an extra method, this will be described in a later section. When a Client subscribes to a Topic requesting Reliability and the ability to Lock, the client will be obligated to implement the extensions to the normal Client interface, necessary to use the QoS. When a Client requests two different QoS from two different Topics will have to implement both of the interface extensions.

5.4.5 Quality of Service Attributes

We will only deal with a limited set of quality of service attributes. These will be “server reliability”, “order of delivery” and “concurrency”.

Server Reliability

In order to have Reliability, we have to eliminate single points of failure. We can assume that local delivery is more reliable than remote delivery so we will make a local copy of the Topic for every subscriber to a topic. This local copy will be used as proxy. The proxies will communicate with each other and with the main Topic by using a specially designed Inter DAC Synchronization Protocol, which we will explain later. When the main topic fails, the first created proxy-Topic will become the main topic.

Order

In order to provide this quality of service attribute, a numbering of all items posted on the Topic is necessary. Since Topics are single threaded they can only accept one message at a time, which makes a total ordering possible. When reliability is a requested QoS, we have to take the proxies into account. The Clients requesting Reliability will have local proxies, but the messages will be collected and numbered at the main Topic to ensure a total ordering.

Note that the ordering might not always be fair. It will take quite a while for a client allocated on the other side of the world to send a message to the main Topic. A nearby Client however would be able to send the message instantly. So while the message from the other side of the world might be sent earlier, it will arrive later that the message that was sent locally. This means that the ordering is done when the main topic receives it.

Concurrency

We will handle different cases of the Concurrency attribute. (These will also reappear in the experiments Chapter.)

The first case will be an ordinary locking algorithm, where the main Topic will be responsible for the Lock-management.

The second case will be implemented using a distributed locking mechanism on the proxies. These proxies are still copies of the original Topic.

Clients interact with their proxies, receive notifications through them etceteras. Now when a Client wants a lock on an item, that Client will send a lock request to its proxy. When the proxy receives this request, it will lock the currently requested item unless another Client already locked it. After the proxy locked the item, it will first tell the other copies of this Topic to lock the topic, and second, tell the client it can start. The Inter DAC Synchronization Protocol is used for this.

While the client uses the locked item to do calculations, and updates the value, the other copies of the Topic will answer the locking request of the local proxy. If all the copies send that the lock was granted the client is free to do what he has to do with the item, and unlock it when he is done. The unlock is also called a commit, and when this is done, the value will be updated and a message will be sent to unlock all other copies. If the local proxy receives a locking error, meaning that this item is locked in another copy, the operation will be canceled and undone.

This second case was inspired by optimistic concurrency control [18]. The locking algorithm that came forth from this allows the client to continue until he is finished or until a violation is found.

However much research is done on the topic of deadlocks etc, this does not fall within the scope of this research, and will not be addressed.

5.4.6 Quality of Service Summary

In this section, we described the basics of the Quality of Service we will offer in our implementation of DAC's. We also explained the method of prototype inheritance, which we will use in the DAC implementation in order to achieve reuse of existing code and behaviour. We concluded this section by mentioning the Quality of Service Attributes we would use in our implementation, and we explained how these QoS would accomplish their goals.

5.5 The implementation

In the early implementations of DAC's many problems were encountered that were solved in later implementations. The first implementation was not even entirely made according to the interface specified in the DAC-chapter. After some programming, we finished a first implementation of a simple DAC. This would be the base-DAC used as root of the (prototype) inheritance chain.
We will begin this paragraph with explaining how the simple DAC is implemented. We will show some of the problems we encountered, and tell how we solved them. The next section will go into detail about the Inter DAC Synchronization protocol (IDSP). After that the extensions on the client interface for the QoS Attributes will be explained.

5.5.1 The simple DAC

This basic-DAC-implementation gave us the possibility to create new topics, create sub-topics on those topics, subscribe clients and publish information.

A problem with earlier implementations had been that you would have to know the name of the DAC in order to subscribe to it, while you should only need the topic-name to subscribe to that topic. Another problem was that there could be two separated topics with the same topic-name. The simple DAC solved these problems by using a root-DAC in the topic-hierarchy that controlled all the first level topics. This root-DAC was no different than the other DAC except that the user did not have to create it.

Another problem was the single point of failure. When the collection fails, all subscribers are disconnected until the collection restarts, and all data in the collection will probably be lost. A solution could be to let the clients reconstruct the DAC, but this is not the responsibility of the clients. This problem was tackled by creating proxies at the client side, which generate a local copy of the DAC. In this case the failed DAC could be reconstructed using one of these proxies.

The “proxy method” will however cause a lot of overhead when Topics are big and frequently updated. Entire Topics have to be copied upon subscription, and concurrency problems can occur since all copies of a Topic can not contain the same information at the same time. Therefore this “proxy method” will only be used if the clients require Reliability as QoS attribute.

The communication between the proxy and the Topic, and between sub-Topics and their super-Topics will proceed trough the Inter DAC Synchronization Protocol.

5.5.2 Inter DAC Synchronization Protocol (IDSP)

To get the Subtopics, Topics and the proxies to work together, they have to communicate with each other. We developed the Inter DAC Synchronization Protocol (IDSP) for this purpose. This protocol can be compared to the low-level protocols used in CORBA. IDSP is a protocol designed so that different implementations of DAC’s can communicate with each other. The IDSP consists of an interface providing a number of methods to be implemented and the expected behavior of these methods. Only DAC’s use the IDSP, it is invisible for the user. Its main purpose is to communicate meta-information within the DAC community.

We already explained that the Sub-Topics were subscribed on their super-Topics to receive all their messages, so they behave just like a client would. This is necessary because clients can subscribe on a topic and all its parent topics. The Topic to which the client subscribes handles all the data for that client. Depending on the QoS asked by the client, the Sub-Topic would require that QoS from the parent topic. It is however necessary to get some additional information from the server, in order to keep the locally stored information correct and up to date. The IDSP is used only in the last case. The IDSP will only be used when the interface between the client and the Topic does not provide enough facilities. An example will clarify this.

Example: We have already explained that the Reliability QoS uses proxies to improve reliability. The proxies are copies of the Main Topic and must be kept up to date. The proxies are subscribed to the Main Topic like normal clients, so there is no room for synchronization and updating in the Client interface. The IDSP will fill this gap, by supplying special functions for updating and synchronizing topics and proxies. When a proxy notices that the main Topic is no longer available, it will try to reconnect for a while. If it gets reconnected, it will request an update from the main Topic. If it does not get reconnected, it will try to start a new main Topic, after which the new main topic will ask its proxies to synchronize so the new main Topic can rebuild itself. In the above text we have seen three examples of the IDSP. First there was the checking to see if it was available, next the synchronizing and third the request to start a new main Topic.

5.5.3 Interfaces for QoS attributes

When dealing with QoS attributes like for example locking, an extension of the standard interfaces is required. We want to be able to call a locking procedure, and afterwards a commit. These methods must therefore be added to the DAC-interface, and the client interface must be extended with methods like ‘Commit successful’ or ‘Could not lock’. It is clear that the locking attribute is not the only one that needs an extension of the interface, but in our implementation of just three Quality of Service Attributes, it is.

The QoS attributes like Order and Reliability do not need an adaptation to the Client interface, nor do they need an adaptation to the DAC-interface. It is recommended that the Client and DAC interfaces are not extended with useless methods, unless they really lack some functionality, as is the case with the locking attribute.

The LockableDAC and NotifyLockable interface are specified below.

```
public interface LockableDAC extends DAC {
    public lock(Object o);
    public commit(Object o);
    public void cancel(Object o);
    public void success(Object o);
}
```

Extract 13 LockableDAC-Interface

```
public interface NotifyLocable extends Notifiable {
    public void continue(Object o);
    publid void roleBack(Object o);
    public void cancel(Object o);
    public void success(Object o);
}
```

Extract 14 The interface of a Client with Lockable QoS.

5.6 Summary

This chapter discussed the implementation details of the DAC's. In the first section we argumented that BORG would be an appropriate programming language to create these DAC's because of its asynchronous nature. BORG will provide us with the necessary time and space decoupling.

In the following section, we discussed the concept and design. We noted that a DAC and a Topic are in fact the same, and that both clients and DAC's implement the Client Interface. Since DAC's and clients are addressed with this Client interface, the DAC hierarchy forms a tree where Clients are the leaves and DAC's the nodes. We continued the discussion with determining what the best way would be to represent a Topic. Since the two options both had their positive sides, we concluded to let the users decide this as being Quality of Services they could chose from, and thus introducing the next section.

In this section about Quality of Service, we started by explaining how the concept was implemented in a DAC. We discussed the problem that some QoS needed the entire DAC-subsystem to change. Next, we explained prototype-based inheritance and the applicability of this type of inheritance when using different QoS, to come to some problems caused by using inheritance in this case. Then we discussed the QoS Attributes that we will use in our implementation, being Reliability, Order of Delivery and Concurrency.

Our following point of interest was the implementation of the DAC-system. We described how we made a simple DAC and how QoS Attributes could extend that simple DAC. Next, we described the Inter DAC Synchronisation Protocol (IDSP) which we created. The IDSP is a standard for inter-DAC communication and synchronisation.

6 EXPERIMENTS

"A wise man proportions his belief to the evidence."

-David Hume

6.1 Introduction

In this chapter, we will present a number of experiments we performed with the DAC-system. These experiments use a Stock Market as case study. In this Stock Market, the Stockbrokers will be the clients and the Stocks will be represented as Topics. When a Stock value rises, a new risen value is published on the Stock's Topic. It is not the intention of these experiments to create a model of a Stock Market, just to use it as example to demonstrate some of the facilities of our DAC's.

In our first experiment, we will demonstrate how the DAC's should be used. This experiment also introduces the QoS. The second experiment is done on these QoS Attributes. Here the influence of these Attributes is tested. In a third experiment, the different concurrency techniques are demonstrated and evaluated. In the three sections about the experiments we will start by explaining how the experiment is initiated, then talk about the goals we try to reach, move over to the results and end with conclusions.

We conclude this chapter by examining all the collected data and all the results

6.2 *Basic experiments*

6.2.1 Setup

We will setup a Stock Market by first creating a Market Manager. This Market Manager is a client and acts as Manager of the Stock Market hierarchy. He will be responsible for the creation of topics and their management. When creating a new topic, and thus creating a new Stock value, it will also create a Stock Manager. The Stock Manager will be responsible for keeping the value of the Stock up to date. In real life, computes keep track of offer and demand to calculate the correct prize for the Stocks. In our example, the Stock Manager does this. The Stock Manager, just like the Market Manager, is a client that has specialized event handling.

After the DAC-subsystem has been initialized, a Market Manager will be started. Next the Market Manager will start three Stock-DAC's with their Stock managers. Then all the Clients will subscribe to their Stocks with their requested QoS attributes, and the market will be started.

6.2.2 Goals

The Goals of this experiment is to demonstrate how the DAC's function as a Stock Market and to show that DAC's could be useful in applications. We will first show the general overview of the Stock Market, after whish we will zoom in on certain interactions, explaining how they work.

6.2.3 Results

Our experiment begins with creating the Market Manager. This market Manager is a Client that is subscribed on the Root topic of the DAC hierarchy. He is in charge of creating new Stocks and thus is given the order to create three Stocks. The Market Manager Creates the following Stocks "Zabena", "Euro" and "L&H", and their Stock Managers.

If we take a closer look at this we will see that when we say that the Market Manager creates the Stocks, we in fact mean that this Market Manager asks the Root-Topic to create three Subtopics. When the Subtopics are created, the Market Manager creates three special clients that will act as Stock Managers, and subscribes each one of them to the Topic they have to manage.

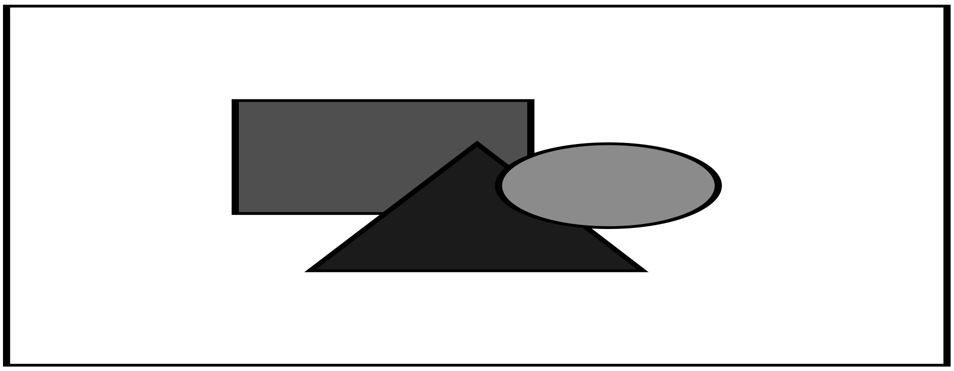

Figure 5 The initial Stock Market

When the Stock Managers receive the message that they have been subscribed to the Topic, they will create three Subtopics. These will be "Sell", "Buy" and "Value". When the three Stock Managers have done this, everything is ready for the stockbrokers.

In figure 5 we see the initial state of the DAC-hierarchy. The white squares represent topics and the colored squares represent Clients. The line is the function "is subscribed to", and is read from bottom to top.

Now we will create some clients and subscribe them to the Stocks. The Clients are specially designed to work on Stocks. They implement the Client-interface, as every client is supposed to, and in addition they use the three Subtopics of the Stock to do their trading business. When a Client subscribes to a Stock, it subscribes to the value of the Stock. When that Stock value changes, the new value will be posted on the value Subtopic. When such a value is posted, the client will determine whether he wants to sell, buy of wait with the stocks. If he would like to sell them, the client would post how many stocks he would like to sell on the "sell" Subtopic. The same goes for the "buy" Subtopic. The client is not subscribed on the "Sell" and "Buy" Subtopics, the Stock Manager is the only one subscribed to this. When something is posted on one of these Subtopics, the Stock Manager will adapt the value of the stock and post it on the "Value" Subtopic.

We will create our first Client and call him Aram. We subscribe him to the "L&H/value" Topic, and Aram does not require any Quality Of Services. Now, let us take a closer look to the subscription of Aram.

Aram contacts the local DAC-subsystem and asks to be subscribed to "L&H/value". The subsystem forwards this request to the Topic "L&H" that refers it to its "value" subtopic. On the "value" Topic a responder will be created to handle all communication with the Client. Since no QoS Attributes are requested, it is only a simple responder that is created.

A reference to the responder is returned in the subscription-succeeded notify message to Aram, and the current value of the Stock is send two.

When our second Client enters, requiring QoS's, the story will be a little different. Only a different kind of responder is created and returned. Let take a closer look at a Client called Bob. The creation of Bob, is the same as the creation of Aram. Bob contacts his local DAC-subsystem and subscribes to the topic "L&H/value" requiring the QoS "Reliability". The subsystem passes the message to the Topic where a responder is created. When the responder extends itself to a ReliableResponder, a proxy is created and liked with the ReliableResponder. This proxy will be initialized with all the data in the Topic. The proxy will migrate to the location of Bob, and Bob will be notified with a reference to this proxy, and with the current value of the Stock.

Now we have two subscribers, we can take a closer look at the Client-Topic communication.

Bob, would like to buy the "L&H"-Stocks while they are low, and publishes on the "Buy" topic how much Stocks he wants. The Stock-Manager is the only subscriber of this topic and is notified. After notifying Bob that his purchase was successful, the Stock-Manager calculates and publishes the new value on the value-topic. The value topic receives this request to publish, and lets the responders contact their clients.

6.2.4 Conclusions

We used this experiment to show how these DAC's are used. We showed how the publish/subscribe mechanism worked and how the responders are used as basis for other QoS. In our implementation, we moved away from the implementation as a strict collection. We made the publish/subscribe interface explicit with the methods "publish" and "subscribe", where in a collection this would have been done using "add" and "contains" as in [4].

The implementation of a stock market was straight forward, and the publish/subscribe style feels natural.

6.3 QoS Experiments

6.3.1 Setup

The Stock Market used in this experiment is analog to the Market used in the basic experiment. It is build with the three topics “Zabena”, “Euro” and “L&H”. We will start by creating a Topic with regular Clients, Clients that do not require QoS. We will introduce some problems in the system, for example a part that stops functioning, and see how the system reacts to this. Next we create a topic where we subscribe a client requiring “Server Reliability” as Quality of Service, and we rerun the test. We repeat these tests with the “Order” attribute.

6.3.2 Goals

In this experiment, we will explain how the Quality of services are implemented and how they manage to provide the required Services. We will also test some QoS Attributes.

6.3.3 Results

As in the previous experiment, we start the DAC-subsystem and the Market Manager. The Market Manager starts the three Topics and their Stock Managers. We first create three ordinary Clients, meaning Clients that do not require any QoS. After some interactions, buying, selling, etc…, we kill the entire Stock, leaving the Clients without a medium to communicate.

Due to the lack in error handling in BORG, the Clients will continue to try to buy or sell their stocks, never receiving an update or an acknowledgement. A client could think his message was lost, if he does not get an acknowledgment, and send it again. This however can be undesirable because the acknowledgement could be lost, or delayed, and than a client would for example buy twice the number of stocks he wanted.

Since the Clients are normal clients, they will have to wait until the Stock is restarted. Then they will notice, if they notice it, that they are not subscribed anymore, because the value of the Stock has been reset. This tragedy could have been avoided.

We will now do this test again adding one Client that requires “Server Reliability”. This Client keeps a proxy. This proxy is a local copy of the Stock that will update itself when it is necessary, using the IDSP. If now the Stock fails, the proxy will not be able to update anymore. The proxy will send a message to the DAC-subsystem, asking for the creation of a new Stock. The proxy will also include a link to himself when asking this, so the new Stock can synchronize with that proxy. After it has been synchronized, the new Stock can start its work as Stock again. The value was saved, as were the references to the clients, but everything the other Clients did during the crash is lost.

If we add more and more clients with the “Server Reliability” Attribute, we will see that the synchronization between the Clients and the Stock will slow the server down considerably. Therefore it would be ideal if, for example, a total of two Clients guard the “Reliability of the Server”. The other clients can just sit back and relax.

Since the Stock Manager is a special client, we could let him subscribe with the “Server Reliability” QoS Attribute. However, to avoid a single point of failure, the Stock Manager should be on a different machine. Since the Stock and the Stock Manager communicate constanty with eac other, it will slow the system down if they were on different machines. So that is not the ideal solution either.

The “Order” Attribute is constructed by adding a numbering to all the messages. Note that when using the “Order” Attribute all messages sent over that communication channel are numbered, this way we are sure no messages are lost, making the communication reliable too.

We will use the previous setting with three normal Clients and a Client that requires “Server Reliability”, and add one Client with the “Order” attribute. When the topic crashes, the proxy from the client requiring “Server Reliability” will make sure a new Stock is created, like in the previous test, and the Client with the “Order” Attribute will be sure not to have lost anything.

The clients that do not require any QoS’s from the Stock will however have the advantage that their methods are passed directly to theStock, without any delay caused by the QoS’s.

6.3.4 Conclusions

In this experiment, we tested the “Server Reliability” and “Order” QoS Attributes. We noticed that having too many Clients guarding the Reliability of the server did more damage then good. This conatins an important lesson: We should only request the QoS that we require

We have seen in the tests with the “Order” Attribute that numbering the methods is a good solution in many cases. Numbering provides, order, reliable communication and the ability to detect duplicates. We could have learned that from packet-switched networks too.

6.4 Concurrency Experiments

6.4.1 Setup

In this last experiment, we are going to use a slightly different setup. Clients will have to lock the "buy" and "sell" topics before they do their bargaining. These Clients will have to implement the NotifyLockable interface specified in Extract 14, and the Stocks the LockableDAC interface, specified in Extract 13.

The "Concurrency" QoS Attribute is focused on Locking. There are two implementations of locking systems. One will gather all data and handle it sequentially, the other uses a distributed locking algorithm based on optimistic locking control.

6.4.2 Goals

The main goal of this experiment is to see if "Concurrency" could be implemented as a Quality of Service. The second goal is to compare the two Locking Systems.

6.4.3 Results

This experiment starts like all the above by starting the DAC sub-system and the Market Manager. It is important to see that "Concurrency" is not a Service defined between one Client and a Stock, it is a Service between a Stock and all the Clients requiring "Concurrency". This implies that when one Client has single access to the resource, no other Clients can access it until that one client commits the resource. Since "Concurrency" is defined between all the clients and the DAC, the internal structure of the DAC must change to allow locking of the resources, but ordinary Clients and Clients with different QoS attributes must be able to subscribe as well. The internal structure of the DAC must change, but the DAC must stay compatible.

The way in which the DAC solves this compatibility problem, is to create a special group for all the Clients requiring "Concurrency", and to give them permission to get single access tot the resource. Ordinary Clients can still subscribe to this DAC, but cannot have single access to the resource. Both locking systems use this manner.

Centralized locking

Our first locking system is a centralized system, where all lock requests are handled at the main topic. The main Topic is the Topic that is created by the DAC-root. All the lock requests will be processed and granted by this main Topic, that implements the sequential execution of all the Lock-request. When a Client gets single access to the resource, no others are allowed to access it. While the Client that posses the Lock has control over the resource, all the requests are forwarded to the main Topic and put in the Locking Queue. When the Client commits the resource, the next in line can have the resource.

This implementation of the Locking service is simple, correct and straightforward. It suffers a little of unfairness since all Clients requests must be gathered in the main Topic, something also necessary for the order attribute. There is also the problem of the single point of failure introduced by the centralized locking system. We could try to provide better server reliability by using that QoS Attribute for one of the Clients. We will try to accomplish this in the second implementation.

Distributed locking

The implementation of the distributed locking system was created because we wanted to make a locking system that was both faster and more reliable. The techniques used for the “Server Reliability” QoS Attribute will be used here as well. Local proxies will be created when Clients subscribe. These proxies will have a distributed locking system installed. When a Client requests a lock on the resource, the proxy locally locks that resource, and ask the main Topic if the resource is lockable, meanwhile the Client is notified that he could do his calculations on the local value in the proxy. While the Client is doing these calculations, the main Topic checks if the resource is lockable, and if the proxy has the latest value. If the resource is lockable, the main Topic locks the resource and tells the proxy that everything is ok. After this, the main Topic will start to alarm other proxies that the resource is locked, preventing them from issuing locks on that resource themselves. When the resource is locked in the main Topic, the lock requested by the Client is canceled, and the work of the Client reset.

Since the Clients can start working on their data sooner, with a risk on having to cancel their actions, less time will be lost with the startup communication. The local proxies keep an eye on the availability of the server. When they notice that he is no longer available, they alert the DAC-Subsystem to restart and synchronize the main Topic.

In this way the main Topic is still responsible for deciding who gets the lock and who doesn't, and in case that he fails, the service will be halted until the main Topic has restarted. We could try to solve this problem by creating all Topics and proxies equal. In this case, if a Client wants to lock a resource, all the other proxies must be contacted to see if they allow the topic to be locked. If any of the proxies would be down, the system would not be able to continue, thus creating a single point of failure in every proxy.

When the Client has finished using the resource, it will issue the commit method to tell the proxy it can unlock the resource. The proxy will propagate the value and the unlock trough the DAC-structure.

The main Topic will regularly check the proxy that has the lock to see if it is still responding. This to prevent the system from failing when the topic holding the lock fails.

6.4.4 Conclusions

It is possible to create a "Concurrency" QoS, but since it is not based solely on the communication channel between a user and the Topic, the internal DAC system has to be adapted. The interface for both the Client and the DAC will have to be changed in order to handle concurrency.

The two implementations of the "Concurrency" QoS provided in our implementation function correctly. First we have seen the centralized locking system that let all locking be done by the main Topic. We noticed that this induced a single point of failure in our implementation. Our next implementation was inspired on the optimistic concurrency control [18], and it puts the proxies in charge of the locking. The main Topic was there to manage the lock-requests, allowing or disallowing a lock when one was requested. The main Topic was monitored by the proxies so in case of a failure all could be recovered.

6.5 Conclusions

In this Chapter, we used our DAC implementation to dos some experiments. We started by using a simple experiment to explain how DAC's work. Next, we tied to avoid the single point of failure created when using a single DAC. We did this using the "Server Reliability" QoS. We concluded that to many clients requiring "Server Reliability" would make the server slow. Further, we experimented with the "Order" QoS Attribute and noted that "Order" could be a basis of reliable communication.

Our final experiments were done on the "Concurrency" QoS Attribute. We made two implementations of this Attribute, one was a centralised locking system, and the other one was a distributed locking system inspired on optimistic concurrency control. Both implementations work correctly.

Partial failures remain an inherent problem of Distributed Computing, which is not easily conquered.

7 CONCLUSIONS AND FUTURE WORK

“A conclusion is simply the place where you got tired of thinking.”
Anonymous

7.1 Overview

This thesis is situated in the field of distributed systems; therefore, we started our report by introducing the user to certain concepts and problems of this field. We discussed memory access, exceptions, latency, partial failure, and concurrency. Further, we explained the basic communication techniques used in distributed systems.

Our next Chapter talked about Distributed Asynchronous Collections (DAC’s), introducing the first important concept. We looked at the definition of DAC’s, discussed the publish/subscribe style they use as interaction style and explained the pull and push model. Then we came to the second important concept of this thesis: Quality of Service (QoS). The QoS were shown as they were defined in [4]. We ended the Chapter after explaining topic-based DAC’s and their interface.

BORG was the topic of the following Chapter. After a quick introduction into the basic concepts of the language, we explained the communication primitives of BORG and the basics of migration. We concluded this Chapter with a comparison between BORG, Java RMI and CORBA.

The next Chapter handled our implementation of these DAC’s, clarifying the design decisions we made. There we explained how these DAC’s work, which QoS Attributes we used, and how they were implemented.

We discussed the implementation of the Quality of Service Attributes “Server Reliability”, “Order” and “Concurrency”. We continued by explaining the Inter DAC Synchronisation Protocol, used for communication and synchronisation between DAC’s and proxies of these DAC’s. The introduction of QoS created the need for the IDSP. Finally, we described the changes made to the interfaces in order to be able to use the “Concurrency” QoS.

The last Chapter contained some experiments with DAC’s and their QoS Attributes. First some basic experiments were done, then special tests on QoS Attributes, and to conclude we did some tests with Concurrency management.

7.2 Conclusions

On our quest to find the use of Distributed Asynchronous Collections, we came about many problems, from which some were easily conquered and others seemed invincible. Now, when this battle has come to and end, we would like to review what we have accomplished.
We will first talk about DAC's in general, and then talk about the QoS.

DAC's

The both the publish/subscribe style and asynchronous method calling have proven their use in modern computing. DAC's try to combine both concepts, to create a collection that combines both their good properties.
There are some remarks we would like to make about these DAC's.
We believe that, how tempting the notification-system may seem, losing control over the system is an essential problem. Since the user is waiting for the Server to notify him, he might not detect that the server has become unavailable for him. That is a problem with the push model, which can be avoided when using the pull model. The space and time decoupling could make error detection and correction a difficult and slow process.
DAC could certainly be used in applications, since they extend a normal collection and add some behaviour to it. The only question is in how far this new behaviour will be needed and used.

Quality of Service

A good example of Quality of Service would be an Internet connection. That Internet connection should demand a fast and unreliable transfer for streaming movies for example, since the frames must arrive on time to play them. If the frames are too late, they will be useless and thrown away. If there is a small error in the data of the movie, it will hardly be noticed because the frames already contain some noise, and are only displayed very shortly. While correct transmission is not as important as fast transmission for movies and images, programs require a perfect transmission in order to function correctly. Here the Internet connection would require a reliable transmission.
In our implementation, some QoS only affect the communication channel, while others require the DAC internal behaviour to change. If the internal behaviour has to change, this will have consequences for all Clients, while the QoS required by one Client should not influence other Clients.
QoS is something that should not be used if it is not necessary. When misused, it could slow the system down or cause other QoS Attributes not to be usable.
Concurrency is a QoS that requires the internal behaviour to change. It also needs special Clients to use the extended behaviour of the Topic. It is possible to implement Concurrency as QoS, but it will never be as good as a system specially designed for concurrency control.

When we want to implement different Quality of Services, we must implement all the services, and make sure they are compatible. This is a lot of work. In addition, since most users will only use one or two QoS, having a system that supports too much QoS is useless.

7.3 Future work

Since we only looked at three different QoS Attributes, it could be useful to try to implement other services like “Migratability”, allowing clients to migrate to different positions.
BORG could use some good error-detection tools, since they are essential to a good programming language for distributed programs.
When the error detection functions have been installed, BORG could be extended with the possibility to provide QoS Attributes on the communication commands.
In BORG, agents can migrate to every available agent-system. It would be nice to have the possibility to allow or disallow agents to migrate to the system.

BIBLIOGRAPHY

"The secret to creativity is knowing how to hide your sources."
-Albert Einstein

[1] G. Couloris, J. Dollimore, T. Kindberg: *Distributed Systems, Concepts and Design*, 2^{nd} Ed., Addison – Wesley, 1994

[2] Samuel C. Kendall, Jim Waldo, Ann Wollrath, Geoff Wyant: *A note on distributed Computing*, Sun Microsystems Research, technical reports 1994

[3] Clemens Szyperski: *Component Software, Beyond Object-Oriented Programming*, Addison – Wesley, 1998

[4] Patric Th. Eugster, Rachid Guerraoui, Joe Sventek: *Distributed Asynchronous Collections: Abstraction for Publish/subscribe Interactions*, Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, 2000

[5] The Pico homepage by Theo D'Hondt,
http://pico.vub.ac.be

[6] The BORG homepage by Werner Van Belle,
http://borg.rave.org/

[7] W. Van Belle, T. D'Hondt: *Agent Mobility and Reification of Computational State*, Vrije Universiteit Brussel, 2000.

[8] The OMG's CORBA homepage,
http://www.corba.org/

[9] Java^{TM} Remote Method Invocation (RMI),
http://java.sun.com/products/jdk/rmi/

[10] Gnutella is an open, decentralized, peer-to-peer search system that is mainly used to find files. Homepage:
http://gnutella.wego.com/

[11] ISO/IEC 14750:1999, Information technology - Open Distributed Processing — Interface Definition Language.
[ODP, IDL, Formerly DIS 14750-1]
http://www.iso.ch/cate/d25486.html

[12] Smalltalk, The Pure Object Environment and Programming Language,
http://www.smalltalk.org/
http://www.squeak.org/

[13] D. Skeen. Vitria's Publish-Subscribe Architecture: *Publish-Subscribe Overview*, 1998
http://www.vitria.com

[14] Brian Oki, Manfred Pfluegl, Alex Siegel, Dale Skeen : *"The Information Bus - An Architecture for Extensible Distributed Systems",*
http://www.cs.utah.edu/~retrac/cs7460/infobus.pdf

[15] Guy L. Steele: *Common Lisp the Language*, 2nd edition, Digital Press, ISBN 1-55558-041-6, 1990

[16] Johan Fabry: *Full distribution transparency, without losing control – AOP to the rescue*, Master Thesis

[17] Philippe Demaecker: *Combining Components using Control Flow Components*, Licentiate Thesis

[18] Kung and Robinson: *Optimistic methods for concurrency control*, vol. 6, no. 2, 1981