CORBA in the aspect of replicated distributed real-time databases

(HS-IDA-EA-02-109)

Robert Milton (a99robmi@ida.his.se)

Department of Computer Science
University of Skövde, Box 408
S-54128 Skövde, SWEDEN

Final Year Project in Computer Science, spring 2002.

Supervisor: Marcus Brohede

CORBA in the aspect of replicated distributed real-time databases

Submitted by Robert Milton to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

2002-06-06

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: ____

CORBA in the aspect of replicated distributed real-time databases

Robert Milton (a99robmi@ida.his.se)

Abstract

A distributed real-time database (DRTDB) is a database distributed over a network on several nodes and where the transactions are associated with deadlines. The issues of concern in this kind of database are data consistency and the ability to meet deadlines. In addition, there is the possibility that the nodes, on which the database is distributed, are heterogeneous. This means that the nodes may be built on different platforms and written in different languages. This makes the integration of these nodes difficult, since data types may be represented differently on different nodes. The common object request broker architecture (CORBA), defined by the Object Management Group (OMG), is a distributed object computing (DOC) middleware created to overcome problems with heterogeneous sites.

The project described in this paper aims to investigate the suitability of CORBA as a middleware in a DRTDB. Two extensions to CORBA, Fault-Tolerance CORBA (FT-CORBA) and Real-Time CORBA (RT-CORBA) is of particular interest since the combination of these extensions provides the features for object replication and end-to-end predictability, respectively. The project focuses on the ability of RT-CORBA meeting hard deadlines and FT-CORBA maintaining replica consistency by using replication with eventual consistency. The investigation of the combination of RT-CORBA and FT-CORBA results in two proposed architectures that meet real-time requirements and provides replica consistency with CORBA as the middleware in a DRTDB.

Keywords: Distributed real-time database, replication, Real-Time CORBA, and Fault-Tolerance CORBA.

Table of contents

1 Introduction

The project described in this paper focuses on replicated distributed real-time databases. With this kind of databases, the requirements of real-time databases, such as timeliness and predictability, and the problems associated with distributed databases, such as mutual consistency, are combined. In addition, there is also the challenge of a heterogeneous distributed real-time database. What is investigated in this project is whether the *common object request broker architecture* (CORBA) (OMG, 2001) might be suitable as a middleware in a heterogeneous distributed real-time database.

Along with the increased complexity of real-time systems there is an increased need for organized storage of a large amount of data. Databases can provide this service and therefore it is intuitive to think of databases when large amount of data is mentioned. However, a real-time system has the requirement of being *timely*, that is, being able to meet deadlines. Therefore, a real-time database has conflicting demands to fulfill than a conventional database. In order to guarantee the real-time requirements stated by the system, the real-time database must be predictable and timely (Ramamritham, 1993).

Distributed databases are used to increase availability and reliability in a distributed system (Elmasri and Navathe, 2000). The concept of a distributed database is to divide the database into fragments, e.g. logical pieces of the database, and distribute these over the network on different sites. This gives the advantage of data being located on sites where it is needed the most. One way of achieving fault tolerance and availability in a distributed database is to have copies of parts of, or the whole database at different sites. This is called *replication* of the database. Although replication has the advantages of increased availability and fault tolerance there are other problems that have to be addressed. One of the major problems with replication is to maintain *mutual consistency*, that is, all replicas should act like one singe replica (Lundström, 1997).

One problem that can arise in a *heterogeneous* distributed database is that not all sites are built on the same platforms and thereby makes it difficult to integrate these sites. To overcome such problems, the Object Management Group (OMG) defined the standard of the CORBA. CORBA works like a middleware between the application and the underlying details, such as operating system mechanisms and network protocols (Mowbray and Ruh, 1997). Two objects that are using CORBA can communicate with each other although they are not built on the same platform and therefore CORBA is very useful in a heterogeneous environment.

As stated first in this introduction, the project described in this paper focuses on distributed real-time databases (DRTDB), that is, both timeliness and data consistency must be considered. The aim of the project is to in investigate the suitability of CORBA as a middleware in a DRTDB. This is done by investigating Fault Tolerance CORBA (FT-CORBA) as an object replication approach and the possibility of integrate Real-Time CORBA (RT-CORBA) with FT-CORBA to create an architecture for a DRTDB. Two architectures are proposed and analyzed for their functionality in a DRTDB.

The report is structured as follows; chapter 2 provides a background, describing important concepts such as, distributed real-time databases, distributed object computing etc. In chapter 3, the problem is described, and the aim and the objectives are stated. The method used in the project is outlined in chapter 4, and chapter 5 provides the applying of the approach. The proposed architectures, described in chapter 5 are analyzed in chapter 6. Conclusions drawn from the analysis are presented in chapter 7, along with a discussion and possible future work.

2 Background

In this chapter background to this project is provided. In section 2.1 the general characteristics of real-time systems are described. Section 2.2 gives an overview of the characteristics of real-time databases. The basics of replicated databases are given in section 2.3 and in section 2.4 the characteristics of replicated real-time databases are outlined. Finally, section 2.5 and 2.6 describes the distributed object computing (DOC) middleware standards of the common object request broker architecture (CORBA) and the addition for real-time CORBA.

2.1 Real-time systems

Real-time systems are computer systems where tasks are associated with some kind of timing constraints. That is, a real-time system needs to react on an input within a certain amount of time. According to Burns and Wellings (1997, p 2) “the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the result are produced”. In other words, it is not only the result of the computation that is of importance, but also that the computation is made within the associated deadline. To make sure that a real-time system meets its deadlines it is important that the system is predictable and sufficiently efficient. Predictability means that each task has to have a defined time requirement in order to be foreseeable (Brohede, 2000). A real-time system that is efficient enough is a system where each tasks worst-case execution time is lower than the time to its deadline.

Typically, according to Ramamritham (1993), a real-time system consists of a controlling system and a controlled system. The controlling system reacts on input from the environment and then makes adjustments, if necessary, on the controlled system. For example, consider a control system for a tank. If the heat in the tank is too high the pressure has to be lowered and vice versa. The control system interacts with the environment by reading the heat and pressure sensors. By making calculations based on these readings, the controlling system adjusts the pressure valves. Figure 1 illustrates such control system.

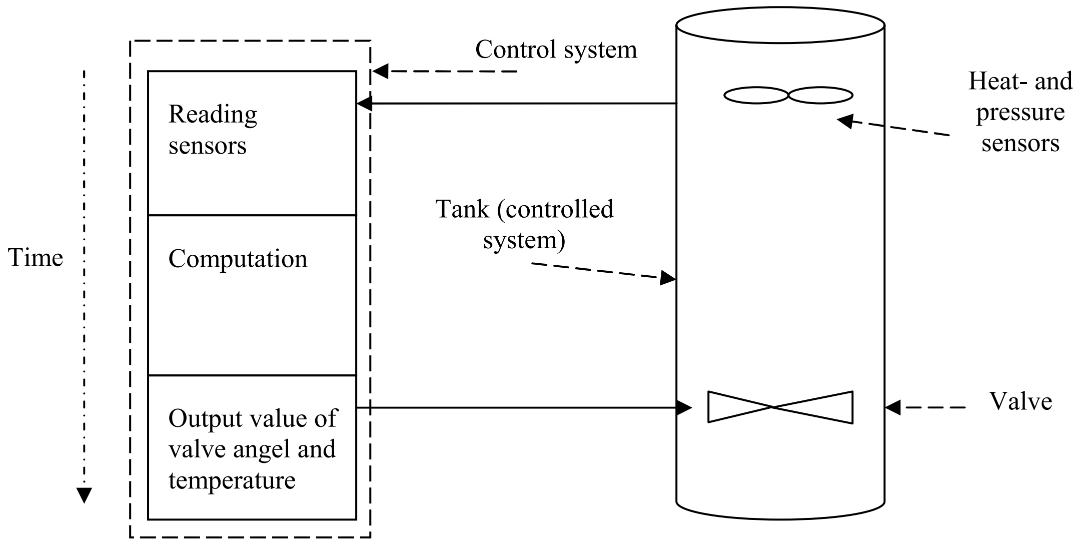

Figure 1: Control system for heat and pressure in a tank.

Real-time systems are often operational in environments that can put human life at stakes if the system will fail. For example, the cooling system in a nuclear power plant is a typical real-time system. If the reactor can not be cooled down in time, there would be a catastrophe. Although many of the existing real-time systems fall under this category, there are also real-time systems where the missing of a deadline will not end in a catastrophe. Burns and Wellings (1997) point out three categories of real-time systems. *Hard* real-time systems are systems where it is absolute imperative that each task makes its deadline. Typical examples of hard real-time systems are control systems for power plants or the landing gear of an airplane. *Soft* real-time systems, on the other hand, are systems where it is still important to meet the deadlines, but if they are missed occasionally the system will still function properly. For example consider a cruise-control of a car. If the system cannot make a measurement of the current speed, the algorithm for calculating the new speed can still use the old value since the difference might be so small that the system still can make a proper calculation of the speed. *Firm* real-time systems are systems where the value of a job completed after its deadline will not be imposed to the system. For example, consider an airline reservation system. If the reservation can not be made, it is not good, since the customer will not get his/hers reservation, but there will not be a catastrophe. A real-time system does not have to consist of only one of these three categories. A hard real-time system can have some actions that have hard real-time requirements but most of the other actions have soft real-time requirements.

2.2 Real-time databases

Real-time systems usually use application dependent structures to manage their data (Kao and Garcia-Molina, 1995), but as these systems evolve and the applications become more complex there will be a need to store more data. To keep track of all data saved at some particular place, there is a need to organize the data and hence a database comes handy. As described in the previous section, real-time systems has requirement of being timely, that is, to make the correct computation within the defined deadline. Therefore, to be able to use databases in real-time systems the databases must be able to keep deadlines. The goal of conventional databases is to achieve a good throughput or good response time (Ramamritham, 1993). To achieve good throughput or good response time, a transaction of a conventional database should, according to Elmasri and Navathe (2000), posses several properties, often referred to as the ACID properties, which are:

- **Atomicity** – meaning that a transaction is performed completely or not performed at all.
- **Consistency** – meaning that a transaction takes the database from one consistent state to another.
- **Isolation** – meaning that a transaction should not be interfered by any other transaction executing concurrently.
- **Durability** – meaning that any committed changes made by transaction must persist in the database.

These properties are fine when dealing with conventional databases where the data always are persistent. By enforcing the ACID properties, transactions can execute concurrently without jeopardizing the correctness of the data. However, insurance of the ACID properties often include protocols for concurrency control and data recovery. Such protocols often diminish the real-time performance through blocking transaction abortion and deadlock which is unacceptable in a real-time database (Kao & Garcia-Molina, 1995).

Since real-time systems often work in dynamic environments, the data used in the system often is temporal (Ramamritham, 1993). This means that after a certain amount of time, the value of the data becomes outdated and of no use to the system. The controlling system often makes decisions based on the value of the data sensed from the environment and, therefore, it is imperative that the actual state of the environment is consistent with the state reflected in the database. This temporal consistency can, according to Ramamritham (1993), be divided into two components.

- Absolute consistency – Data about the environment must be consistent with the state of the environment.
- Relative consistency – Different data items that are used to derive new data must be temporally consistent with each other. All items used to derive new data must be collected within a certain time to be consistent with each other.

Sometimes in real-time systems an approximated value might be more useful then no value at all, since, sometimes it is more important to keep the deadline of a transaction than to produce an exact result (Berndtsson and Hansson, 1995). With an incorrect value there might still be possible to achieve a computation within the deadline. Although the computed value might not be a hundred percent accurate according to the real world, the system will still function properly. For example, take the cruise control system mentioned in section 2.2. If the system can not make a new measurement of the actual speed of the car, the system can use the old speed value since the difference, if the sampling frequency is high enough, between the two measurements would be small. By relaxing the consistency requirement the task will be able to meet its deadline, although with a not completely consistent data.

As pointed out above, the data in a real-time system is often temporal. In addition to this property, there is the requirement of timeliness, e.g. meeting deadlines. According to Ramamritham (1993) these two features gives us an important difference between conventional databases and real-time databases, namely that the goal of a real-time database system is to meet time constraints set by the activities of the system. The need of meeting deadlines in real-time systems makes it important that the transaction response time in a real-time database system is predictable (Krishna & Shin, 1997). If the response time of a transaction is unknown, there is no way to predict whether the deadline of the activity will be met or not. As described in the previous section, it is important that a real-time system is predictable and therefore the transaction response time also must be predictable. Ramamritham (1993) describes four factors that can jeopardize the predictability of transaction response time.

Dependency on data values. The execution of a transaction may depend on the data retrieved from the database and therefore it may not be possible to predict the worst-case execution time.

Data and resource conflicts. If a transaction tries to access some data that are already locked up by another transaction, the first transaction has to wait until the data becomes available. Similarly, a transaction may be hold up while waiting for some resource, such as I/O devices.

Dynamic paging and I/O. Conventional databases are stored on secondary storage, e.g. hard drive, and are divided into pages. When retrieving data, first one or more pages have to be read into main memory from where the data can be used. If the wrong pages are buffered, there will be significant overhead, since new pages must be buffered into the memory. Reading a page that is already in the memory is several orders of magnitude faster than reading the page from the disk (Eriksson, 1997). To deal with this problem, there are researches of putting the whole, or parts of the database into the main memory (Ramamritham, 1993). This would solve the problems associated with paging, but new problems occur. Main memory databases are out of scope in this project, see Ramamritham (1993) for details.

Transaction aborts with rollbacks and restarts. If a transaction is aborted and restarted several times before committing, the total execution time will increase in an unpredictable way. The sources kept by the aborted transaction will be denied to other transactions (Ramamritham, 1993). If there is no upper bound of the number of transaction restarts, the predictability of the transaction response time is seriously jeopardized.

2.2.1 Concurrency control

Concurrency control is important when dealing with database systems, conventional as well as real-time databases. Transactions must be able to execute in parallel with other transactions while ensuring that the effects in the database occur as if the transactions where executed in some serial order. The most popular correctness criterion is serializability (Krishna and Shin, 1997). A schedule *S* of *n* transactions is serializable if it is equivalent to some serial schedule of the same *n* transactions (Elmasri and Navathe, 2000). This means that if the n transactions in some way can be executed in a serial order, the schedule of these transactions is considered serializable. Özsu and Valduriez (1999) list two kinds of concurrency controls that are used in databases, *pessimistic* and *optimistic*. The pessimistic concurrency control ensures that the transaction will not violate the serialization consistency before the transaction is allowed to execute. According to Krishna and Shin (1997) the best known protocol of pessimistic concurrency control is the two-phase locking protocol (Eswaran et. al, 1976). Transactions using this protocol can be divided into two phases, one growing phase, during which all locks needed by the transaction are acquired and one shrinking phase, during which the locks are released. To follow the two-phase locking protocol, all locking operations must precede the first unlock operation (Elmasri and Navathe, 2000). That is, all locks needed by the transaction must be obtained before any locks can be released. There are two major disadvantages of this protocol, the problem with priority inversion and deadlock occurrence. Priority inversion occurs when a lower-priority transaction holds a resource that is needed by a higher-priority transaction. This means that the high-priority transaction has to wait for a low-priority transaction to finish. This is not desirable in a real-time system, where high-priority transactions must be able to finish first. Deadlock occurs when two or more processes are waiting for each other to release a resource (Burns and Wellings, 1997). Deadlock occurrence would bring unpredictability to a real-time system since it is difficult to predict the time it takes to solve the deadlock.

Optimistic concurrency control protocols, on the other hand, execute the transaction and then check whether there is any violation of consistency. They are called optimistic because they assume that little interference will occur and, hence, there is no need to do any checking during the transaction execution (Elmasri and Navathe, 2000). Optimistic concurrency control protocols consist of three phases, a read, a validation and a write phase. During the *read phase* the transaction reads values from the database and the only writing that is done, is done to local copies. In the *validation phase* checking is performed to ensure that the transaction execution does not violate consistency. In the third phase, the *write phase*, the transaction updates are applied to the database if the validation phase was successful. The advantages of optimistic protocols are that they are non-blocking and deadlock free, which is desirable for a real-time system (Kao and Garcia- Molina, 1995). However, there are also some drawbacks, namely the unpredictability of the system if interference would occur. The more interference of transactions, the more transactions have to be restarted (Elmasri and Navathe, 2000) and if a restarted transaction would be interfered it has to be restarted again. In the worst case the restarting of a transaction would be unbounded, which is not acceptable in a real-time system.

2.3 Distributed databases

A distributed database (DDB) is defined by Elmasri and Navathe (2000, p 766) as "… a collection of multiple logically interrelated databases distributed over a computer network". A distributed database management system (DDBMS) is defined as "…a software that manages the distributed database while making the distribution transparent to the user". The general goal of distributed computing is to partition a large, unmanageable problem into smaller pieces and solve it efficiently (Elmasri and Navathe, 2000). By partitioning a large database there will be several advantages. Elmasri and Navathe (2000) highlight four major advantages.

1. **Management of distributed data with different levels of transparency**. A transparent DBMS hides all the details of the physical location of tables or

files from the user and according to Elmasri and Navathe (2000) there are three types of transparency.

- Distributed transparency. It is preferable to hide the details of the network or, if possible, even the very existence of the network from the user (Özsu and Valduriez, 1999). The user should not have to specify where the data is located in order to access it.
- Replication transparency. This refers to the fact that the user should not have to be aware of whether there exists copies of the database or not (Elmasri and Navathe, 2000).
- Fragmentation transparency. By dividing the database into several fragments and treat each fragment as an individual database, advantages such as improved performance, reliability and availability can be achieved. The problem with this is, according to how to handle queries that are made at entire relations but now have to be performed on sub-relations (Özsu and Valduriez, 1999).

The issues of fragmentation and replication are discussed later in this section.

2. **Increased reliability and availability.** In the database community reliability is defined as the probability of the system running at a certain point in time. Availability is commonly defined as the probability of a system continuously being available during an interval of time. With partial or fully replicated distributed databases, the database is located at multiple sites in the network. This improves the reliability since if one site fails, the other sites will still be operational. For the same reason the availability is improved.
3. **Improved performance.** By dividing the database into several fragments and keeping data closer to where it is needed improves the performance since network communication is reduced and local queries gives better performance since the database is smaller (Elmsari and Navathe, 2000).
4. **Easier expansion.** In a distributed environment the expansion of the system by adding more data, increasing database size or adding more processors is much easier than in a centralized database (Elmasri and Navathe, 2000).

2.3.1 Fragmentation and allocation

To distribute the database on different sites in the network, there is a need to decide how to divide the database and where to put it. The former issue is related to as *fragmentation* while the latter is referred to as *allocation* (Elmasri and Navathe, 2000). There are two major advantages of fragment the database according to Özsu and Valduriez (1999). First, most applications do not use the whole relation but just subsets of it. Therefore it is natural to divide the relation into a number of logically units, called fragments (Elmasri and Navathe, 2000) and distribute the fragments to sites where they are of most use. Secondly, by dividing the data into more than on pieces there is a permission of concurrent execution of transactions which will increase the system throughput (Özsu and Valduriez, 1999).

Özsu and Valduriez (1999) points out two ways of how to fragment the data. *Horizontal fragmentation* divides the relation's tuples into a number of subsets of tuples. For example, consider a relation over the employees of a company with the attributes `NAME, NR, ADRRESS` and `SALARY` where `NR` is the primary key. A horizontal fragmentation of this relation could for example be by putting all tuples where `NR` is in the interval of 1 to 20 into one fragment, and those with `NR` in the interval of 21 to 40 into an other fragment. The second way of fragmentation is *vertical fragmentation* which takes one relation R and divides it into R_1, R_2,…, R_n where each sub relation contains a subset of R's attributes and the primary key of R (Öszu and Valduriez, 1999). Consider the example relation above. A vertical fragmentation of this relation could for example result in one relation containing `NR`, `NAME` and `ADDRESS` and a second relation containing the attributes `NR` and `SALARY`.

Once the data is fragmented there must be a consideration of where to put the fragments, on which sites should each fragment be putted. The issues that are addressed during the allocation phase are the choice of sites and the degree of replication (Elmasri and Navathe, 2000). Elmasri and Navathe (2000) points out that how to make these decisions depends on the performance and availability goals of the system as well as the transaction frequency at each site.

2.3.2 Replication

As described above, one important issue of distributed database design is the degree of replication. Replication of data is commonly defined as having more than one copy of the data located on different sites. The main purpose of replication is fault tolerance and availability (Elmasri and Navathe, 2000). There are two types of replication, *full replication* and *partial replication* (Öszu and Valduriez., 1999; Elmasri and Navathe, 2000). A fully replicated distributed database is the most extreme case of replication. This means that the whole database is replicated at all sites. Partial replication, on the other hand, means that the each site has one or more fragments but no site has all fragments. A distributed database could also be *non-replicated* (or *portioned*) (Elmasri and Navathe, 2000). This means that there are no copies of any fragment and each fragment is located at different sites.

2.3.3 Concurrency control

In section 2.2, two types of concurrency control was outlined, namely *pessimistic* and *optimistic*. With distributed concurrency control there are still these two types, although, when dealing with distributed data the problem is more complex than dealing with a centralized database (Elmasri and Navathe, 2000). Elmasri and Navathe (2000) give three categories of problems that arise in a distributed DBMS environment.

- Dealing with *multiple copies* of the data items. Since a database can be replicated over several sites it is important to have concurrency control to maintain consistency among the sites.
- *Distributed commit*. When dealing with a distributed database, one transaction may have to access data located on different sites. If the transaction fails on one site it is important to undo any changes made at other sites to maintain consistency.

- *Distributed deadlock*. Deadlock could occur at more than one site and therefore there is a need for extending deadlock techniques to deal with this kind of situation.

The technical details of how to solve these kinds of problems are out of scope for this paper and the curious reader is referred to Özsu and Valduriez (1999).

2.4 Distributed real-time databases

As described in previous sections, the most important issue when dealing with real-time databases is to support timeliness to assure that the real-time system will meet its deadlines. In a distributed real-time database (DRTDB) this is still the most important request. By replicating parts of or the whole database on different sites, availability of data and improved read performance can be achieved (Ulusoy, 2001). When replicating data, new problems are introduced since the access control are distributed across the sites. Ulusoy (2001) points out the importance of ensuring *mutual consistency* which means that all replicas must act like a single copy.

Lundström (1997) points out that one way of avoiding unpredictable delays in a DRTDB is to allow local commits. This makes it possible to assure real-time requirements on the local site, since any network delay will not effect the local site. The commits are made locally and after that, eventually, the changes are propagated to the other sites. This is called *eventual consistency* (Lundström, 1997). One problem with this type of consistency is that there might be a situation where two different sites make changes to the very same value in their respective replica. To solve the problem with conflicts, Lundström (1997) describes a two-step strategy which includes *conflict detection* and *conflict resolution*. Using an optimistic replication protocol in a distributed real-time database may violate the real-time requirement of timeliness, since, as described before, there are no bounds for the restart of a transaction. To ensure the requirement of timeliness, there is a need to predict the worst-case time for a replication. By using *bounded-delay replication protocol* this can be achieved since Lundström (1997, p 14) defines a bounded-delay replication protocol as a protocol that "… guarantees that any update from node N_x to any other available node N_y is bounded".

2.5 Distributed Object Computing

When creating a distributed system there might be a problem to connect components made from different vendors, since there is not always all vendors use the same standards or techniques for their components. When the system needs to grow there is the problem with connecting new components to the existing system. In the worst case there would be a need to make the new component from scratch to match the existing system, which would increase the cost of the system. To solve these kinds of problems the Object Management Group (OMG) created the standard of the Common Object Request Broker Architecture (CORBA). CORBA is a distributed object computing (DOC) middleware and according to Otte et al. (1996) DOC is a marriage of distributed computing with an object model.

Distributed computing is, simple described, two or more pieces of software sharing information with each other (Otte et. al, 1996). These pieces of software could be located on the same computer or they could be placed on different computers connected trough a network. The most common model for distributed computing is the client/server model (Otte et. al 1996). In this model there are two main pieces, the server providing some service or information and the client requesting the services or information provided by the server. Using distributed computing gives, according to Otte et. al (1996) the benefit of a more efficient use of computing resources.

Otte et. al (1996) describes an object model as a concept of object-oriented computing. The basis for an object model is the definition of an object. Mowbray and Ruh (1997) define an object as some encapsulated entity that performs services. An operation of an object identifies an action that can be performed on the object and it is only through the operations that an object can be manipulated. Otte et. al points out the four object-oriented concepts that are the foundation of an object model. First of all, by using *abstraction* the developers can focus on what a certain object is capable of doing something and not how. The idea of abstraction is to get out of the focus of details and make it possible to see the whole picture and how different objects are connected to each other. With *encapsulation* the implementation of the object is hidden from the services it can provide. This gives the advantage of modify the object's implementation without affecting other objects connected to it. *Inheritance* allows one object to inherit the behaviors of some other object. If any changes are made in the inherited object, the changes would immediately be propagated to the objects inheriting its behavior. The fourth and last concept is *polymorphism* that is the ability to substitute objects with matching interfaces with one another at runtime (Otte et. al, 1996). This means that objects with matching interfaces can provide different services depending on what parameters that are sent with the request. Otte et. al (1996) gives three advantages of using an object model.

- Define a system model based on the real world – Once the objects, their behavior, their attributes and what relationship they have with each other are defined the software can be based on the real world.
- Logically separate a system into objects that can perform certain tasks called operations – Each object has a clear definition of what it can be used for which makes it easy for designers and developers to see how to use a certain object.
- Extend the model as requirements change – If all objects necessary is already in the model, new requirements most probably means adding some new behavior to a certain object (Otte et. al, 1996).

The combination of distributed computing with object models results in a DOC middleware that gives, according to Brohede (2000), the advantage of components in a heterogeneous system interacting seamlessly. A DOC middleware is some software that is between the application and the underlying network and operating system. Figure 2 shows the use of a DOC middleware.

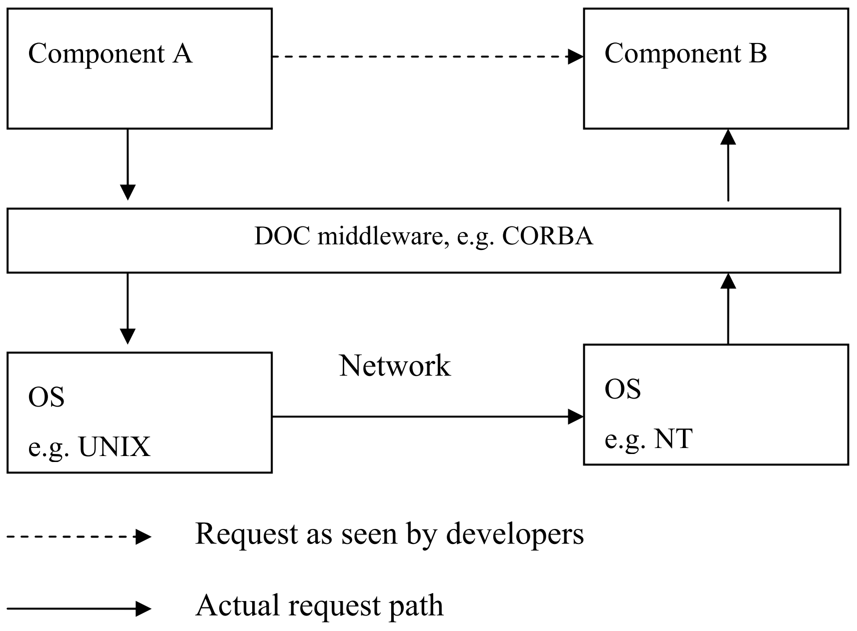

Figure 2: Seamless interaction in heterogeneous environments. (Based on Brohede (2000, p 6)).

2.6 CORBA

The central component in CORBA is the object request broker (ORB). "The ORB functions as a communication infrastructure, transparently relaying object requests across distributed heterogeneous computing environments." (Mowbray and Ruh, 1997, p 36). This means that the ORB is a layer between the client and the server that provide the mapping of a request made by a client to a particular server implementation (Otte et. al, 1996). Since the ORB is handling the communication between the server and the client, there is no need for these two components to know about each others location or implementation. Usually, with a client/server environment there is a one-to-one relationship between server and client. By using a broker, CORBA allows multiple clients to work on a single server or a single client can work on multiple servers (Otte et. al, 1996). Another advantage with using a broker is, according to Otte et. al, that clients can locate and interact with new objects and servers at runtime. To be able to make a request to a server, the client needs to know what service the server can provide and what parameters that are needed in the request. In CORBA, interfaces define the behavior and characteristics of a certain object including the operations a server can perform on the object (Otte et. al, 1996). The OMG interface definition language (IDL) is a definition language introduced by OMG as a standard for interface definition of objects. IDL is not a programming language, but a syntax language to define the software boundaries of an object (Mowbray and Zahavi, 1995). Since the IDL is programming language independent it supports multiple language bindings from a single IDL specification (Mowbray and Ruh, 1997). This gives the advantage of CORBA components being implemented using different languages and still being able to communicate.

The OMG object management architecture (OMA) is the root diagram for all OMG standard activities and it is in the OMA reference model that the role of CORBA is defined. There are four categories of objects in the OMG OMA that communicate through CORBA (Mowbray and Ruh, 1997). How the different components of OMA are related is shown in Figure 3.

CORBAservices: The CORBAservices provide a set of standard functions such as creating objects, tracking objects and abject references and so on (Otte et. al, 1995). In addition to these standard functions, CORBAservices also enables more sophisticated functions such as naming service, transaction service etc. (Mowbray and Ruh, 1997). There are three CORBAservices that are in particular interest for this project, namely transaction services, concurrency services, and query services since these, according to Özsu and Valduriez (1999), are the most important database-related services.

CORBAfacilities: CORBAfacilities represents a higher-level specification which focuses on interoperability issues (Mowbray and Zahavi, 1995). CORBAfacilities is application domain independent and considers the horizontal needs of interoperability (Mowbray and Ruh, 1997). What the CORBAfacilities provide is a set of capabilities for use by multiple different applications such as document management, printing files or accessing databases in distributed environment (Otte et. al, 1995).

CORBAdomains: CORBAdomains is, in opposite to CORBAfacilities, considered with standardizing the interoperability needs close to the needs of the application (Mowbray and Ruh, 1997). The CORBAdomains are vertical market areas such as financial services healthcare or telecommunication and each CORBAdomain considers only its own interoperability needs (Mowbray and Ruh, 1997).

Application objects: The final category is application objects that are not standardized by OMG (Mowbray and Ruh, 1997). According to Brohede (2000), standards developed outside OMG should be sent in to OMG to be introduced into the OMA and thereby providing an overall broader standard.

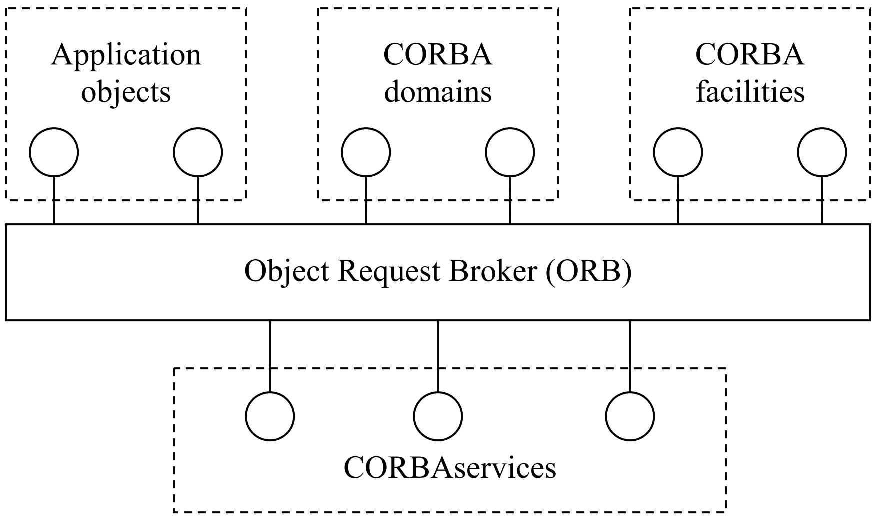

Figure 3: The OMG object management architecture (OMA) (Based on Mowbray and Ruh (1997 p 44)).

2.7 Real-time CORBA

Real-time CORBA (OMG, 2001) was developed to provide end-to-end predictability of activities in real-time systems to support the meeting of real-time requirements, such as timeliness and predictability. To ensure end-to-end predictability, the RT-CORBA specification identifies capabilities that must be both *vertically* and *horizontally* predictable (Schmidt and Kuhn, 2000). With vertically means from network layer to application layer and by horizontally means from application to application over the network. These capabilities are: communication infrastructure resource management, operating system (OS) scheduling mechanisms, real-time ORB end system, and real-time services and applications.

It is to manage these four capabilities that RT-CORBA defines standard interfaces and quality of service (QoS) policies that allow applications to configure and control three types of resources (Schmidt and Kuhns, 2000).

Processor resources For many fixed-priority real-time applications it is important to manage a strict control over the scheduling and execution of processor resources. To allow such strict control, RT-CORBA specification enables client and server applications to (1) determine the priority at which CORBA invocations will be processed, (2) allow servers to pre-define pools of threads, (3) bound the priority of ORB threads, and (4) minimize priority inversion by ensure the semantic consistency of intra-process thread synchronizers.

Communication resources For applications with stringent QoS requirements, location transparency is not always suitable. Therefore the RT-CORBA identifies interfaces that can be used to select and configure certain communication protocol properties. These interfaces added with the feature of client applications being able to make explicit server object bindings, allows applications to control the underlying communication protocols and endsystem resources.

Memory resources By defining a thread pool model, RT-CORBA allows server developers to pre-allocate threads. Thread pools are useful in real-time systems where the amount of memory resources are to be bounded, since by making an upper boundary of threads would make that no further threads can be created and thereby limiting the use of memory resources.

Schmidt and Kuhns (2000) points out a problem with low-level scheduling abstractions in some real-time operating systems. The problem is that there has to be a mapping from the developer's higher-level QoS requirement and lower-level OS mechanisms. This mapping does not come very intuitive for all application developers and therefore RT-CORBA defines a *global scheduling service*. This service is a CORBA object that is responsible for allocation of resources so that the QoS needs of the application can be met.

2.8 Fault Tolerance CORBA

Fault Tolerance CORBA (FT-CORBA) is an extension to CORBA for providing "… a robust support for applications that require a high level of reliability" (OMG, 1999 p 3-11). Fault tolerance is depending on entity redundancy, fault detection and recovery and FT-CORBA supports entity redundancy by providing object replication. The requirements of fault detection and recovery are addressed by supporting several fault-tolerance strategies such as request retries, active replication etc.

With FT-CORBA, replicas of object are created and managed as *object groups*. An object group contains of several object replicas, each with its own object reference. The object group has an *interoperable object group reference (IOGR)* that is a reference to the whole group. This has the advantage that a client can invoke a method on the IOGR and does not have to know about the replication of the object in question (OMG, 2001).

The Replication Manager is an important component and it inherits three application interfaces: *PropertyManager, GenericFactory* and *ObjectGroupManager* (OMG, 2001). The PropertyManager allows setting some certain properties of an object group including:

- **ReplicationStyle** There are two major types pf replication, *passive* and *active.* If the passive style is choosen, the method invoked to the object group will only be carried out by one member of the group, the primary. All other replicas are standing by for backup. In the second case, active replication, each member executes the invoked method and duplicate requests and replies are suppressed.
- **ConsistencyStyle** There are two possibilities for this property. Either the application is responsible for check pointing, logging, recovery and maintaining consistency, or the fault tolerance infrastructure is responsible for the actions just mentioned.
- **MembershipStyle** Like with the ConsistencyStyle, the MembershipStyle can be one of two values. It is possible to make the application responsible for creating the members of the object group. The MembershipStyle can also be chosen so that the Replication Manager invokes a GenericFactory to create the members of the object group.

The GenericFactory is used by the application to create object groups and is also used by the Replication Manager to create individual members of a group.

2.9 FT-CORBA vs. RT-CORBA

RT-CORBA and FT-CORBA are two extensions to CORBA, developed to be used in the real-time domain and the fault tolerance domain respectively. The combination of real-time and fault tolerance may be difficult to achieve. Decisions made in favor of one, may be disastrous for the other. It is therefore OMG has created two extensions, each developed to solve problems in its respective domain. RT-CORBA is developed to provide real-time developers with end-to-end predictability when using CORBA. This means that no fault tolerance such as fault detection, fault recovery or object replication was concerned when developing the RT-CORBA standard. The important issue that is concerned in RT-CORBA is the meeting of real-time requirements.

In order to provide distributed developers with fault tolerance, the FT-CORBA standard was created. With FT-CORBA the important issue is to always be able to provide the client with an answer. To achieve fault tolerance, FT-CORBA provides fault detection, fault recovery and entity redundancy (OMG, 2001). All replicated objects are managed as object groups, containing a number of group members. FT-CORBA requires strong replica consistency, that is, after each invocation all members have the same state. With FT-CORBA no real-time issues are concerned, in other words, there are no time-limits on transactions or recovery.

3 Problem

As presented in section 2.3, there is a certain amount of demands that are stated for a distributed database system. In addition, when dealing with distributed *real-time* databases (DRTDB), there are demands for timeliness and predictability, as pointed out in section 2.4. The problem that is to be investigated in this project is whether the Real-Time CORBA (RT-CORBA) standard can fulfill the demands of timeliness and predictability.

3.1 Problem definition

Otte et al. (1996) claims that integrating heterogeneous components in a distributed environment is not easy. Hence, it should be possible to claim that it is not easy to integrate heterogeneous sites in a distributed database. The problem addressed in this project is the problems that arise with replication in a heterogeneous distributed real-time database. This project approaches the problem by investigate if CORBA/RT-CORBA is suitable as a middleware in a replicated distributed real-time database and thereby be able to solve problems associated with the communication between the heterogeneous sites.

The need for storage of large amount of data in real-time systems brings us to the use of real-time databases (RTDB). This kind of databases has the requirement of providing its requested operations within specified deadlines. In a heterogeneous system there might be several replicas, all more or less different and maybe developed on different platforms. If there is a need for two or more different replicas to be connected in a distributed environment, it is important that these sites can communicate with each other. By using a distributed object computing (DOC) middleware, for example CORBA, in heterogeneous systems the advantage of components interacting seamlessly can be achieved. Well defined interfaces and the fact that a component does not need to worry about other components' location or implementations (Brohede, 2000) gives the advantage of interchangeability to the system, i.e. that the implementation of one component can be changed but there is no need to change any other components. In a heterogeneous distributed RTDB this would give the advantage of the different replicas to interact without the need to know on what platform or how the other replicas are built. Though, since it is a distributed *real-time* database it is important that the middleware can support timeliness and predictability. If a request from a client to one of the replicas is not carried out in time, there would be of no meaning to use that particular DOC middleware as the communication channel between the replicas in a distributed RTDB. With a *replicated* DRTDB, there is also the problem of keeping all replicas consistent and still preserving timeliness. Therefore, it is of importance to investigate whether FT-CORBA can meet the demands of a distributed RTDB regarding data consistency. It is also of importance to investigate whether RT-CORBA can fulfill the demands of timeliness and predictability. In the specification of RT-CORBA, OMG (2001, p 24-3) points out that "Real-time CORBA brings to Real-time system development the same benefits of implementation flexibility, portability and interoperability that CORBA brought to client-server development".

3.2 Problem delimitation

Since it would be possible to combine the degree of replication and real-time requirements in several ways there is a need to delimit the problem. What will be investigated in this project is whether the RT-CORBA standard can be used to specify a fully replicated distributed real-time database that supports hard real-time requirements. Issues associated with partial replication, as well as firm and soft real-time requirements are out of scope for this project. The database is fully replicated and local commits are allowed. When using local commits, eventual consistency must be considered. In this project it is assumed that eventual consistency is application dependent and therefore issues concerning conflict detection and conflict resolution are not considered. Reliability and recovery is not addressed in this project and therefore it is assumed that no site failure or communication failure will occur.

How a replicated node is implemented is not in focus in this project, since the project aims to investigate how well CORBA serves as a middleware in a DRTDB.

3.3 Hypothesis

The hypothesis of this project is that CORBA, with the extensions FT-CORBA and RT-CORBA, is able to provide timeliness and predictability as well as eventual consistency. The hypothesis is considered false if CORBA does not provide neither of predictability, timeliness and eventual consistency.

The aim is to investigate whether CORBA is suitable as a middleware in a DRTDB, that is, whether CORBA can provide timeliness and predictability as well as eventual consistency.

The problem is addressed by (1) investigating the possibilities of object replication in CORBA since it would be advantageous to use already existing approaches for object replication. (2) Make comparison between replication approaches in order to select one for use in a proposed architecture. The criteria that are investigated are how well the approach follows the CORBA standard, the simplicity to use the approach and whether fault detection and fault recovery is supported by the replication approach. (3) Investigate the possibilities of use RT-CORBA with the chosen replication approach in order to satisfy the requirement of timeliness and predictability. (4) An architecture is created that uses FT-CORBA for object replication and RT-CORBA to make a replicated DRTDB for use as basis for an implemented prototype. (5) A thorough analysis is conducted in order to make sure that this architecture satisfies the requirements of timeliness and consistency.

As stated in section 3.2, the delimitations of this project are to focus on hard real-time requirements in a fully replicated DRTDB. Since the database is fully replicated, local commits are allowed, and eventual consistency must be considered. Any protocols for conflict detection and conflict resolution are not issues addressed in this project. Neither are issues concerning reliability and recovery covered in this project and it is therefore assumed that no site failure or communication failure will occur.

3.4 General benefits

According to Mowbray and Ruh (1999) one key benefit of using object model architecture is improvement in reusability. Szyperski (1997) claims that components defined in a component architecture must have a DOC middleware in order to connect the components to each other and make them communicate. As described before, CORBA is such a middleware and by using CORBA should according to Mowbray and Zahavi (1995) give the advantages of heterogeneity, portability and interchangeability. In addition to these advantages, the use of a broker gives the advantage of defining each interface only once and let the broker handle the subsequent interaction. This gives the system developer the benefit of reduction in complexity (Mowbray and Ruh, 1999). Using CORBA also gives the advantage of transparency of the network. The client does not have to have any knowledge about the network or the operating system.

When dealing with heterogeneous system this is of great importance and since it would make the composing and the management of the system easier. As described in the previous section, distributed real-time systems are often heterogeneous which would make a real-time version of CORBA suitable for distributed real-time systems. Real-time systems must be predictable in order to make them timeliness and to make that happen, all of the system must be predictable. According to OMG (2001, p. 24-4) “The interfaces and mechanisms provided by Real-Time CORBA facilitate a predictable combination of the ORB and the application” which would mean that between the ORB and the application the system is predictable.

4 Approach

With full replication of the DRTDB, the whole database is replicated at all sites. By using this type of replication, local consistency can be achieved. That is, all updates are committed at the local site first and then the changes are propagated to all other sites. To use CORBA as the middleware in a fully replicated DRTDB, there is a need to make the database replicas as objects, since CORBA is based on object computing. In order to assure real-time requirements of the communication between the replicas there is a need to investigate if RT-CORBA can be used to make the communication between the replicas timely and predictable. It is possible that RT-CORBA can not be used in a replicated DRTDB and that would make it important to investigate how to extend the RT-CORBA standard to make this possible.

This chapter is structured as follows; first the method outline is presented in section 4.1 and in section 4.2 the possibility of make a CORBA object of a database replica is investigated. In section 4.3 three approaches for object replication are described and discussed. Section 4.4 provides a description of the semi-active replication, as an approach for combining RT-CORBA and FT-CORBA. In section 4.5 two architectures are proposed and, finally, in section 4.6 it is described how the resulted architectures are verified and analyzed.

4.1 Method outline

The method used in this project consists of several steps and in this section these steps are outlined.

1. Investigate the possibility to make an object of a replica.
2. Investigate replication approaches in CORBA.
3. Integrate RT-CORBA to the replication architecture.
4. Conduct an analysis of the architecture integrating replication and RT-CORBA.
5. Implement a prototype based on proposed architecture in order to test its predictability and efficiency.

The aim of the two first steps in the method outline is to meet objective 1, investigate replication approaches, and 2, select a replication architecture. A database replica need to be implemented as an object, since CORBA is object oriented. To investigate the implementation of a DRTDB replica as an object, the definition of an object in CORBA is compared to the DRTDB replica.

When a database replica can be seen as a CORBA object, the need for object replication in CORBA brings us to the next step, namely, investigate whether there exist different replication approaches in CORBA. The existence of object replication in CORBA could be a way of providing replica consistency without having to implement consistency protocols into the object replicas. By pointing out advantages and disadvantages for each approach and making comparison between them, one approach is selected for the project.

When an appropriate replication strategy is selected the next step is to integrate RT-CORBA to this replication architecture in order to make the communication between the replicas timely. If the integration is successful, this leads to an architecture with RT-CORBA and object replication that can be analyzed to investigate if hard real-time requirements are supported when maintaining mutual consistency.

5 Applying the approach

This chapter provides the investigation of making a database replica as a CORBA object in section 5.1. In section 5.2 the investigations of replication approaches are described. How RT-CORBA can be integrated with FT-CORBA is outlined in section 5.3. Two proposed architectures are described in section 5.4 and finally, section 5.5 provides how the results are verified.

5.1 Basic architecture

This section describes the first step taken in the approach; investigate the possibility of making an object of a database replica.

Consider an example of a typical real-time system, a control system for a car. There are three clients, one for *fuel injection*, one for *Anti Block System (ABS)* and one for *Dynamic Stability Control (DSC)*. To make the control system reliable and available there is a fully replicated database distributed over three sites. Figure 4 illustrates how the network for this system would look like and this is an example of an environment where a DRTDB is useful for predictability and accuracy. In order for the system to work properly each of the clients need data from the others. This data must be provided, with real-time requirements, to the client requesting it. Each database replica is a CORBA object that provides basic database services to the client. According to the CORBA standard an object is "… an identifiable, encapsulated entity that provides one or more services that can be requested by a client."(OMG, 2001 p 1-2). A database replica provides database services to the client and to abstract the internal database from the client, the client does only have to know what services it can request. This makes a database replica an encapsulated entity that provides one or more services, thus, a database replica can be implemented as a CORBA object. All replicas in a replicated database must act the same, that is, data consistency must be maintained. In the following two sections, two architectures that add real-time and fault tolerance features to the basic architecture, are proposed. The architecture illustrated in figure 4 is a basic architecture that is the basis for the proposed architectures described in section 5.4.

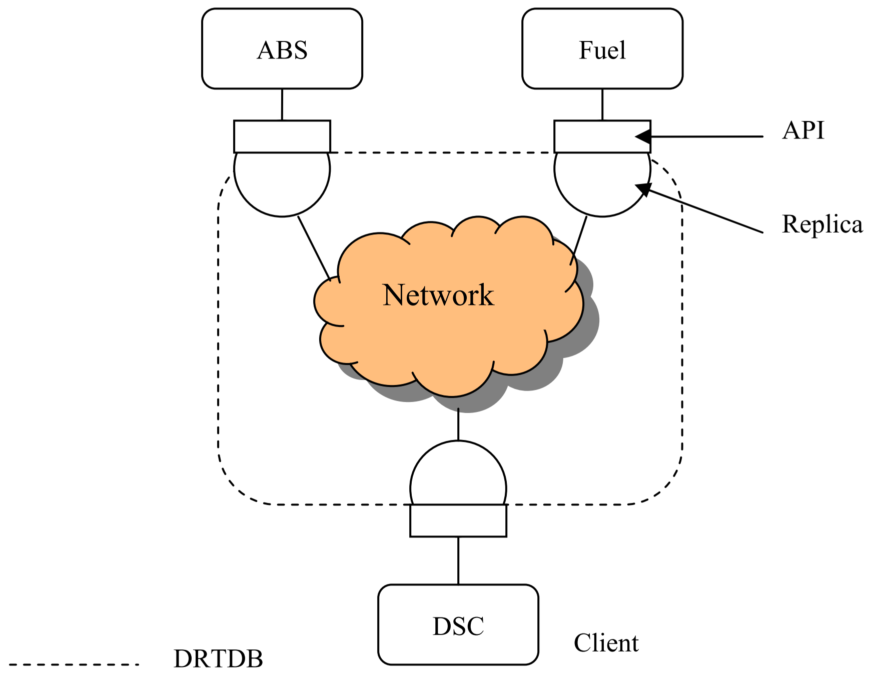

Figure 4: Control system for a car, an example where a DRTDB is useful for predictability and accuracy. Each client needs data from the others to make the correct calculation for the car to function properly. Therefore the database is distributed over three nodes, and to provide the data to the client with real-time demands, local commits are allowed.

5.2 Object replication in CORBA

Besides FT-CORBA, described in section 2.8, a few more approaches have been proposed to extend CORBA with object replication. One approach uses an *object group service* and the second *extends the ORB* in order to create object replication. These two approaches are described in section 4.2.2 and 4.2.3, respectively. This section provides the investigation of object replication approaches in CORBA (step 2 in the method outline).

5.2.1 Object replication using FT-CORBA

With object replication using FT-CORBA the FT-CORBA standard is used to manage replication of objects as well as maintaining consistency among the replicas. In this approach the active replication style is used since all replicas need to execute the request.

A client makes a request to a certain replica that it wishes to get the response from to the group reference. The client specifies the replica in order to use local commit and any changes made by the client is committed to the replica that is pointed out by the client and. Then the changes are propagated out to all the other replicas. This approach is illustrated in figure 5.

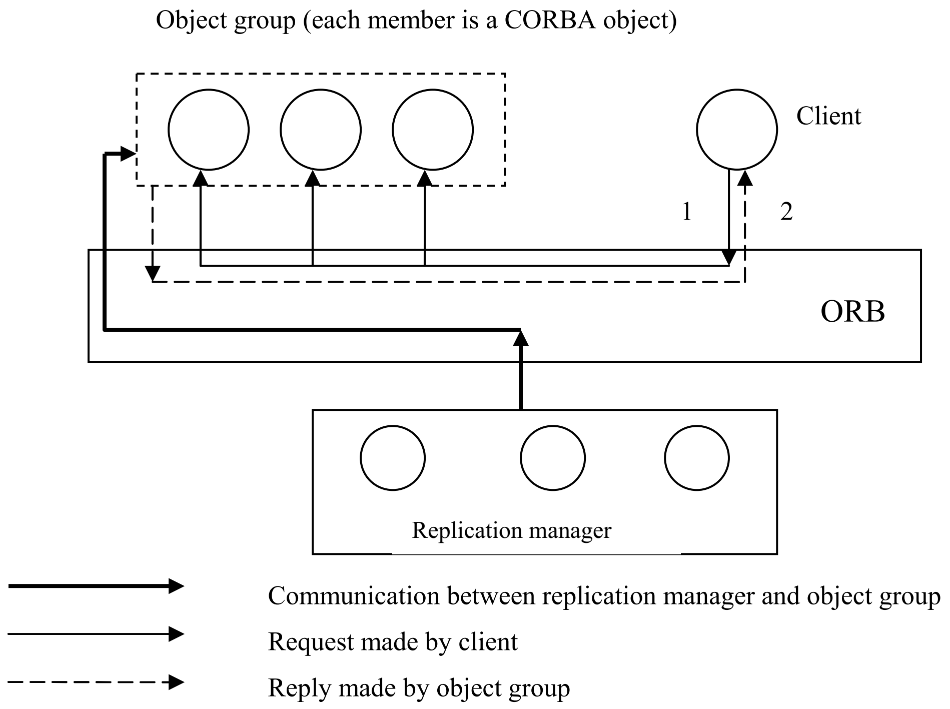
Object group (each member is a CORBA object)
Client
1
2
ORB
Replication manager
Communication between replication manager and object group
Request made by client
Reply made by object group

Figure 5: Object replication using FT-CORBA. First (1), the client makes a request to the object group, the members carry out the request and then (2) the client receives a reply from the object group.

The use of object groups creates an abstraction for the client that does not have to know which member of the object group is invoked. In this project this is not suitable, since, in order to allow local commits the client must be able to choose which server to make a request to. The problem with FT-CORBA is that a client can not invoke a method on an explicit member of an object group unless passive replication is used (OMG, 2001). With passive replication, only one member respond to the request and the others are standing by. In this project local commits are allowed, that is, each client needs to be connected to one specific server in order to be able to fetch data within its deadline. Therefore, FT-CORBA can not be used in a way that each server exists as a member in an object group without making some adjustments to the architecture in figure 5.

The advantages of using FT-CORBA are that FT-CORBA is a standard defined by OMG and that FT-CORBA provides services to maintain consistency between the replicas. The application does not have to care for conflicts or consistency issues.

However, the major drawback is that it is not possible to create an object group of servers and make one client make requests to one explicit server. Unfortunately, this is exactly what this project needs since one objective is to use local commits.

5.2.2 Object group communication using the CORBA Event Service

Object group communication using the CORBA Event Service, as described by Felber et al (1996), uses a service on top of the ORB instead of a part of the ORB (the latter approach is described in the next section). All communication between client and server goes via the object service. A request is made by the client to the object service and then the object service multicasts the request to the server replicas. Figure 6 shows the integration between the application objects and the object service.

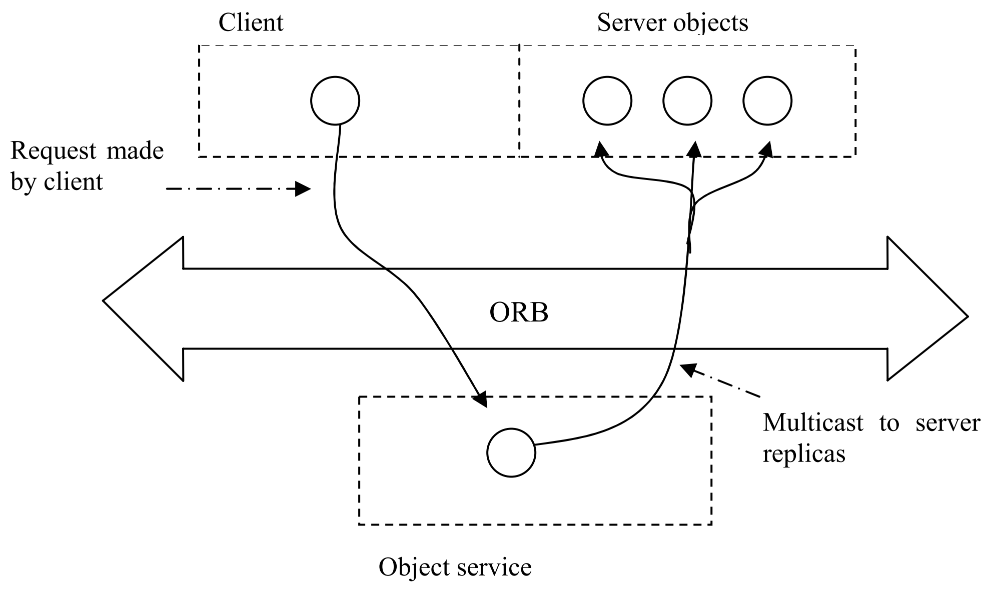

Figure 6: Fault tolerance in CORBA using the Object Service (Based on Felber (1997, p 2)). All communication between client and server goes via the object service. A request made by the client is multicast to the server replicas by the object service.

The approach of object communication with a service on top of the ORB is based on the CORBA Event Service that is a service standardized by OMG. The Event Service decouples the communication between objects by defining two roles for the objects, *consumer* and *supplier* (OMG, 2001b). The consumer process event data and the suppliers produce event data. Communication between consumers and suppliers goes through *event channels*. These event channels are intervening objects that allows multiple consumers to communicate with multiple suppliers (Felber et al., 1997).

The objectives of using the Event Service is service reuse, since the Event Service is specified and some implementations are already done. The natural way of using the Event Service is, according to Felber et al. (1997), to use the *push model*. The push model means that the *supplier* is allowed to initiate the transfer of event data to consumers. That is, the supplier can initiate the transfer of event data and does not have to wait for the consumer to start the transfer. The supplier can push the data to the consumer and force the consumer to receive the data. With this model the client is represented as the supplier and the replicated server is represented as the consumers. A client supply data on the event channel that are processed by the consumers (figure 7).

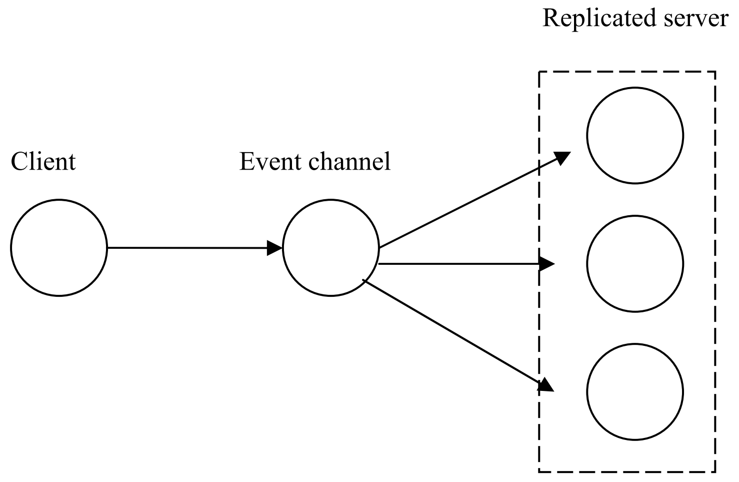

Figure 7: Replication using the Event Service (Based on Felber et al. (1997, p4)). The client invokes some method on the replicated server by making a request through the event channel.

Although the Event Service provides a useful foundation for building replication, it still has some limitations. Felber et al (1997) points out four major issues that has to be addressed.

- **Invocation semantics** The Event Service does not enforce implementations to guarantee concerning atomicity or ordering. This means that messages sent by two clients may arrive in the wrong order, which might compromise the data consistency.
- **One-way vs. two-way invocations** The Event Service only provides one-way invocations, that is, requests made to the replicated server must only have in parameters. This restriction makes it to limited for distributed applications where the client needs some result from the server. Since the client in the project may need to fetch data from the database, one-way invocations are unacceptable. If the client can not receive the asked for, it is of no use to ask for the data.
- **State transfer** When a new replica is added in run-time, there must be a transfer of state to the new replica. Therefore there must exist operations for getting and setting the state of replicas, and this is not part of the Event Service interface. In a non-dynamic environment this would not be a problem, however, this project does not distinguish whether the environment is dynamic or not. The preferable option would be to be able to add new replicas at run-time, since then it would be possible to use the object service approach in both dynamic and static environments.
- **Single–point of failure** One big problem with the Event Service architecture is that it is centralized. The event channel is a standard CORBA object that may be accessed through Internet Inter ORB Protocol (IIOP) by different consumers and different suppliers on different ORBs. IIOP provides no support for multicast communication and therefore interoperable object references identify centralized objects. This implies that the event channel is a single-point of failure (Felber et al, 1997). One of the advantages of using a DRTDB is to avoid a single point of failure and if this is not provided by the Event Service it would be of less meaning to use a distributed database.

The advantage of this approach is that there is no need to make any changes to the ORB and thereby making no interference of the modularity and CORBA compatibility. However, the problem is that the object group service does not provide features for replica consistency, fault detection or fault recovery, which FT-CORBA does.

5.2.3 Object replication extending the ORB

The key idea of creating object replication by extending the ORB is that the group communication is supported by the ORB itself. There are two ORBs that has adopted this approach, namely Orbix+Isis (IONA & Isis, 1994) and Electra (Maffeis, 1995). With the extended ORB model, a group consists of objects of the same type. An object reference of type *T* is mapped to a group of object of type *T* and the ORB is modified to distinguish references to a single object from references to a group (Felber et al, 1997). Figure 8 illustrates how a request is handled in the Orbix+Isis model. A request is made to the ORB that recognizes that the request is made to an object group and the ORB translates the request into a call to an Isis multicast primitive (Felber et al, 1997).

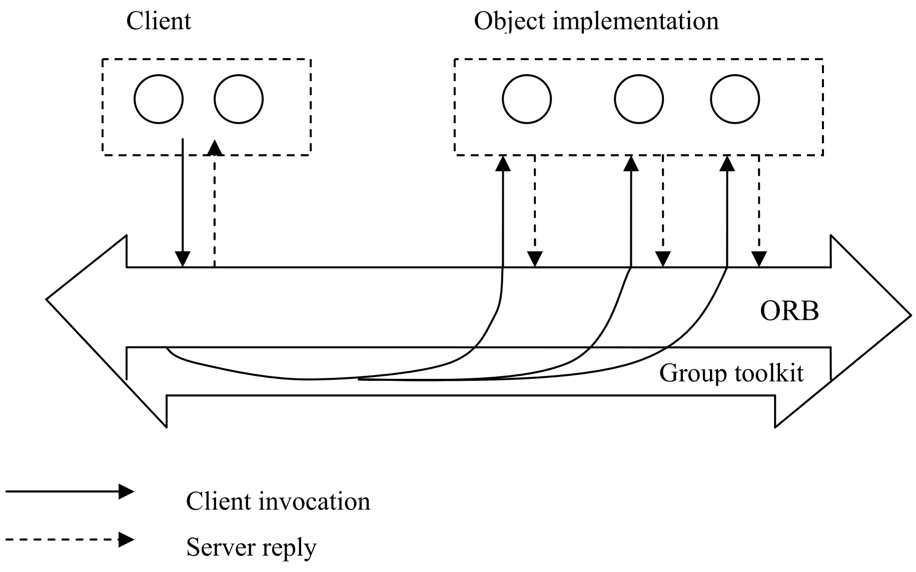

Figure 8: Fault tolerance in CORBA by extending the ORB (Based on Felber (1996, p 4)). The client makes a request to an object group, and the group toolkit recognizes that the request is for an object group and translates the request into a multicast to the server objects.

The advantages of this model is that it is an easy approach since there is no need to create a new group system, there are only some changes to the ORB that has to be made. Transparency is another advantage of this model, that is, an object group is not distinguishable by a client from a single object implementing the same interface (Felber et al., 1997). Although these advantages make this approach an appealing model for object replication there are some disadvantages that makes this approach unsuitable for this project. Felber et al (1996) points out three disadvantages. First, there is a need to change the ORB, which gives the disadvantage of leaving the CORBA standard and thereby loss in portability. The second drawback with this model is that the semantics of CORBA objects are modified since an object reference does not uniquely point out a singleton object. The third drawback is that this model does not support client replication, thus a replicated object is bound to a server role. The approach of extending the ORB only provides an object group communication, that is, the extended ORB translates a request made to an object group into a multicast to the server objects. Neither features for creating and managing replicated objects, or fault detection and recovery exist, as with FT-CORBA.

5.2.4 Summary of replication approaches

This section provides a short summary of the three replication approaches described in section 4.3.

With FT-CORBA all replicated objects are managed as object groups with a set of members. A client makes a request to object group reference and does not need to know the number or the internal state of the members in the group. Within the group strong replica consistency is required, that is, all members have the same state after each invocation. FT-CORBA also provides fault detection and fault recovery which gives the advantage that the application does not have to care for these issues. However, the drawback of this approach, for the project, is the client abstraction that prevents the client from making a request to an explicit member of the object group.

The second approach uses the CORBA Event Service to create object group communication. The Event Service is a, by OMG, standardized CORBA service that defines two roles for the objects, consumer and supplier. The supplier provides the consumer with event data and the consumer process the event data it got from the supplier. All communication between the suppliers and consumers goes via an event channel. The event channel is one of the drawbacks, since, the event channel makes a single point of failure. Another drawback is that only one way invocations are allowed with the Event Service. This means that a client can only make request with in parameters, not requests that requires a reply. The advantage of using the Event Service is that there is no need to change the ORB, thereby preserving modularity and CORBA compatibility.

The last approach for object replication is extending the ORB. The ORB is responsible for distinguish between requests made to a singleton object or an object group. A request made by a client is recognized by the ORB as a request to an object group and the ORB transforms the request into a multicast to the replicas. The advantage of extending the ORB is that it is an easy approach since no new group system needs to be created. However, making changes to the ORB leaves the CORBA standard and thereby loss in portability. Another disadvantage is the modification of the object semantics, since an object reference does not uniquely point out a singleton object.

5.3 Add RT-CORBA to replication architecture

Step three in the method outline is to integrate RT-CORBA with object replication approach. One approach for this is described in this section.

Fault tolerance and real-time are two domains that are not easy to combine. With a real-time system deadlines must be met, but with a fault tolerant system one action may be repeated infinitely in order to make the system fault tolerant. This behavior is not suitable in a real-time system where it is imperative that all actions are predictable and bounded. The problem with this project is that these two domains have to be combined. The database must be kept consistent, but deadlines must not be violated. Schmidt et al. (2002) propose one way to combine the features of FT-CORBA and RT-CORBA for use in mission critical systems that have stringent simultaneous dependability and predictability. It is called *semi-active replication*.

5.3.1 Semi-active replication

Schmidt et al. (2002) outline four challenges when integrating the FT-CORBA and RT-CORBA standard. This section will describe these challenges and provide an answer why semi-active replication is a approach that deals with these challenges.

1. **Non-determinism and expensive replication strategies.**
   One problem when using the active replication in FT-CORBA is that the time needed to synchronize and provide reliable multicast sometimes is unacceptable to use in a distributed real-time environment (Schmidt et al, 2002).
2. **Dealing with semantic incompatibilities between RT-CORBA and FT-CORBA features.**
   There is a significant semantic gap between end-to-end predictability and dependability requirements, which makes the combination of FT-CORBA and RT-CORBA problematic. The ability to engineer a good fault tolerant solution requires tradeoffs that may compromise the system's ability to support real-time behavior, and vice versa (Schmidt et al, 2002). The problem is that the effects of choices made in one area may have unforeseen consequences in the other.
3. **Lack of standards to handle Byzantine and partial failures.**
   The model for failure detection and recovery used by FT-CORBA emphasizes a certain type of failure, namely component failure (Schmidt et al, 2002). With this type failure, a component ceases all interaction with its environment. However, this is not the only type of failure that can occur. Another, more subtle form of failure is one where interactions among components make the system fail. In the light of this type of failure the FT-CORBA model for failure detection and recovery is too limited.
4. **Lack of standard quality of service (QoS) semantics.**
   Although FT-CORBA and RT-CORBA provides the mechanisms required to gain real-time and fault tolerant behavior, CORBA does not give any guarantees that two implementations compliant to the RT-CORBA and FT-CORBA standards will provide equivalent QoS properties (Schmidt et al, 2002). The implementation freedom may cause problems if performance characteristics of two different ORBs, compliant to FT-CORBA and RT-CORBA standard, vary greatly.

The semi-active replication arranges the replicas as a linked list of nodes which is created at startup time. The node at the head of the list is designated as the primary, and if this node should fail, the next node in the list will be the primary. Figure 9 illustrates the semi-active replication architecture. All invocations made by the client to the primary are reliable multicast from the primary to the secondaries. That is, all messages are received by the secondaries in the same order as they were received by the primary. This feature gives the advantage of maintaining replica consistency. These key features, outlined in more detail in Schmidt et al (2002), give the advantages of more deterministic recovery when compared to passive replication. In addition, extra overhead from reliable multicasts is avoided, as well as the complexity and overhead from protocols enforcing inter-replica coordination. Semi-active replication also has a few disadvantages: in case of primary failure the client can be restricted to use the list of references as an ordered list, which is not compliant with the FT-CORBA standard. If a replicas state changes due to operations of other requests other than regular CORBA requests, it is up to application to provide own replica consistency. According to Schmidt et al (2002), these disadvantages have no effect on the determinism of the semi-active replication, which therefore makes this type of replication useful for distributed real-time systems.

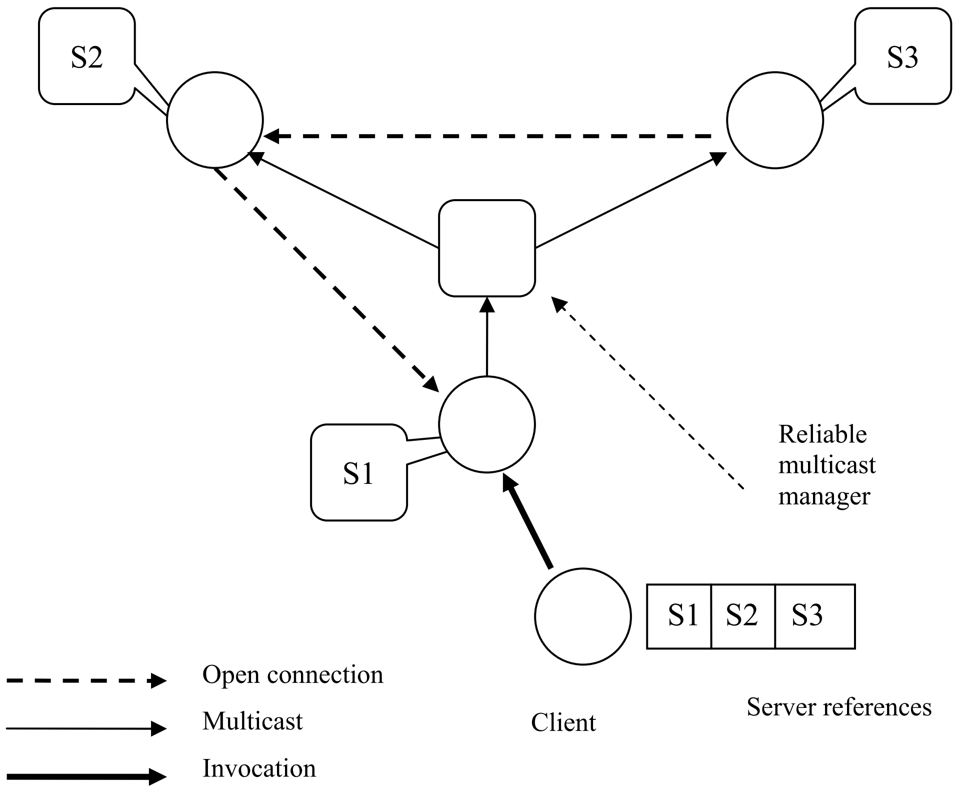

Figure 9: Semi-active replication architecture (Based on Schmidt et al (2002, p 8)). The server replicas are connected with an open connection and in a failure of the primary, the next node connected to the primary is set to primary. All requests are made to the primary and the messages are distributed to the other replicas via the reliable multicast manager.

Schmidt et al (2002) makes a comparison between semi-active and passive replication as well as semi-active and active replication. Since it is active replication that is the main focus in this project this is the only comparison that will be outlined in this paper. The tests conducted by Schmidt et al (2002) is a benchmark to show how the determinism in the time to synchronize the state after each invocation. The result of these experiments indicates that the state transfer time remain tightly bounded for semi-active replication with increased number of replicas. Since the semi-active replication does not possess any other know sources of determinism, the conclusion made by Schmidt et al (2002) is that the semi-active replication is better suited for use in distributed real-time systems than active replication.

5.4 Proposed architecture

This chapter proposes two architectures obtained from performing step three in the method outline, integrate RT-CORBA with object replication approach, which serves to meet objective fou, create an architecture as basis for an implemented prototype.

5.4.1 FT-CORBA architecture

The first proposed architecture, illustrated in figure 10, uses the FT-CORBA architecture. With FT-CORBA, the replicated object is managed as an object group. That is, the object group contains of members, each a database replica. When a client makes an invocation to the replicated object, the invocation is made to the object group reference. With active replication, as used in this architecture, all members receive the invocation and carry out its reply to the client, via the object group reference. For example, consider a client that needs some data from the database. It invokes some method for fetching data on the object group. All members of the object fetch the data, but only the member that is ready first response to the client. FT-CORBA detects and handles multiple replies, so the client only receives *one* reply. If this where to be done without using FT-CORBA, some form of multicast would have to be used to invoke the methods on all server replicas. To make sure that the client does not get multiple answers, either the client must be able to distinguish between the messages or some extra service must handle multiple responses. With FT-CORAB all these problems are already solved. By using a real-time ORB as the middleware in the network, the communication between the replicas is predictable.

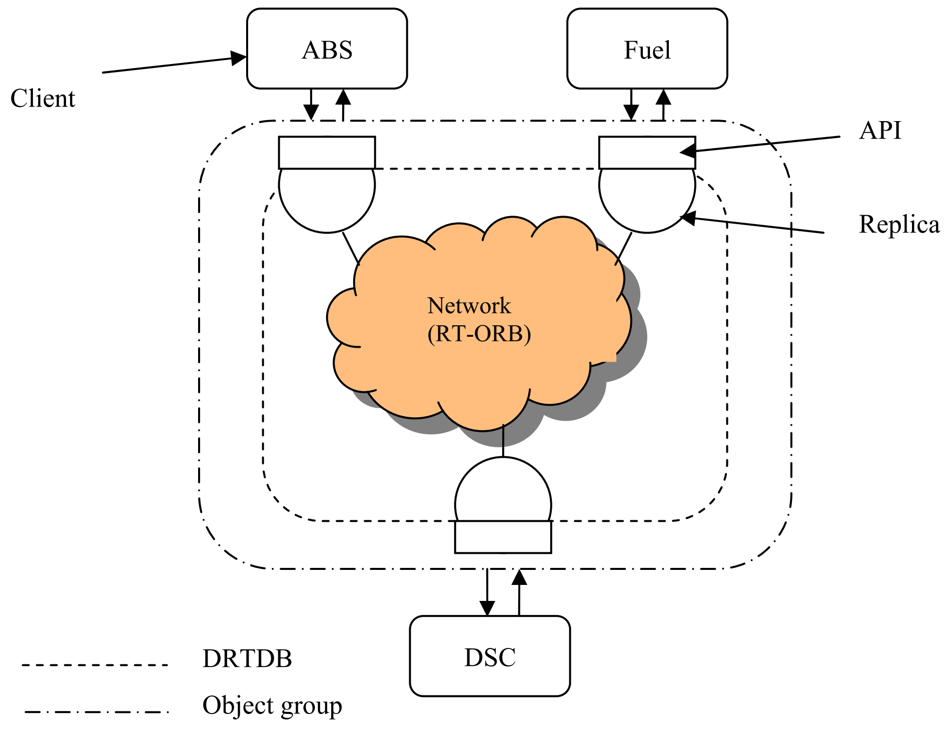

Figure 10: Replication using FT-CORBA. A client makes a request to the object group via the object group reference and receives any response from the same group reference. The object group is managed by the FT-CORBA infrastructure and the network has a RT-ORB to provide a timely communication between the replicas.

FT-CORBA provides the possibility of choosing whether maintaining data consistency should be handled by the application or by the fault tolerance infrastructure. In the architecture in figure 10, the consistency is application dependent. Thus, the replicas has methods for maintaining eventual consistency implemented, thereby making it possible to put an upper bound for the time it takes to detect and resolve a data conflict.

5.4.2 Architecture based on semi-active replication

As described in section 4.4, combining fault tolerance and real-time requirements often lead to that decisions made in favor of one, is disastrous for the other. However, Schmidt et al (2002) propose the *semi-active replication* as a way to combine the features in FT-CORBA and RT-CORBA. The second architecture proposed is based on the semi-active replication, however with some adjustments. In figure 11 the proposed architecture using semi-active replication is illustrated. The replicas are not managed as an object group as in the architecture using FT-CORBA. Instead each replica is a non-replicated CORBA object. The replicas are ordered as a linked list and the node at the head of the list is the primary, just as with semi-active replication. One thing that differs from the semi-active replication strategy is that invocations are not made to the primary only. A client can invoke methods on any replica object and the changes are distributed to the other replicas through the reliable multicast manager. The multicast manager assures that all messages are consumed in the same order by the replicas as the messages were consumed by the multicast manager. By assuring that the multicast manager can receive messages from only one replica at a time, correct order of the messages is assured. For example, consider the network illustrated in figure 12. The Fuel-client updates some info at R2, and the ABS-client updates some other info at R3. R2 contacts the multicast manager before R3 and therefore R2's message will be distributed to the other replicas before R3's messages, even if the multicast manager receives R3's message before multicasting R2's message. Since a client can make invocations to any replica, local commits can be allowed. The fact that messages are consumed by all replicas in the correct order allows for eventual consistency to be maintained.

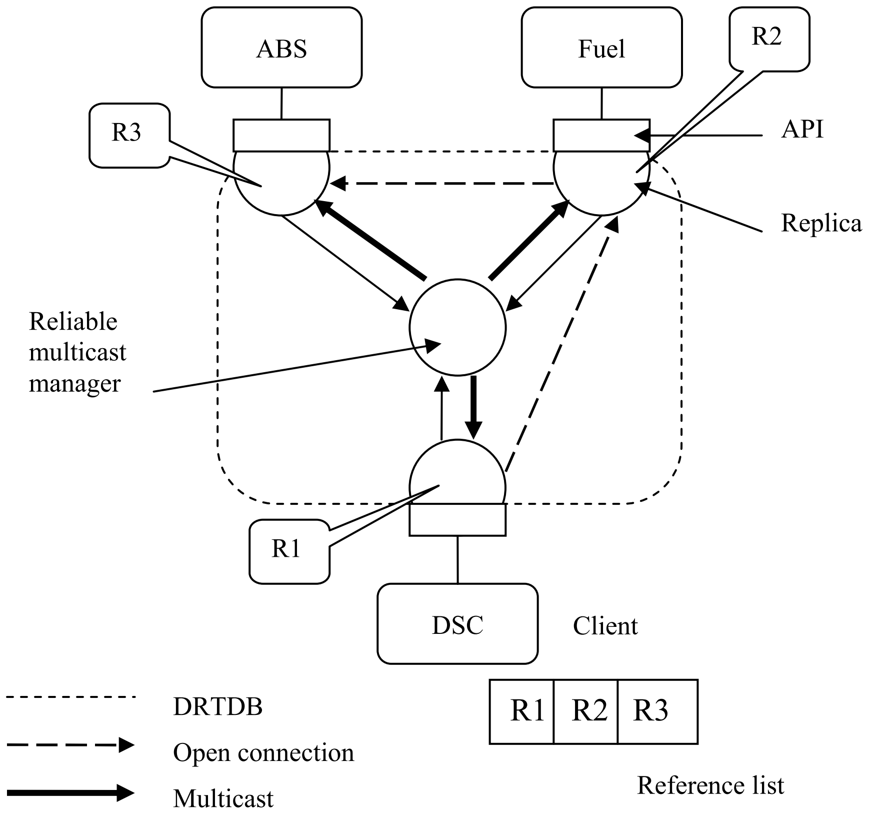

Figure 11: Architecture using semi-active replication. Each client can make a request to any replica and the multicast manager is responsible for maintaining the correct order of the messages. The primary is used in case of any non-primary replica failure. The client requesting the failed replica can redirect its request to the primary and thereby still be able to get an answer.

Each client has an object reference list, containing all replicas in the same order as the nodes are ordered. That is, the first item in the reference list is the primary. The reference list is updated as with semi-active replication. That is, if the connection between the primary and the next following replica fails, the next replica is promoted as primary and the client lists are updated. In the case of a secondary replica failure, the client connecting to the failed replica can invoke methods on the primary. For example, in figure 11, the reference list looks like R1, R2 and R3. This means that R1 is the primary. In case of a failure at R3, the client ABS has the reference list and can invoke methods on the R1 and thereby still be able to function.

In order to provide predictability and efficiency, a real-time ORB is used. Since the replicas and the multicast manager are CORBA objects, these are all connected to the RT-ORB. RT-CORBA provides end-to-end predictability and the communication between the objects is therefore bounded.

5.5 Verifying results

This chapter explains how the proposed architecture is evaluated and analyzed.

5.5.1 Reasoning

To be sure that a real-time application is predictable and timely in all situations testing must be made. However, these tests must be exhaustive in order to be completely sure that the application works properly. Exhaustive tests has the disadvantage of being time consuming and costly. One way of reducing the test cases is to use reasoning about the application when determining the predictability of a real-time application (Brohede, 2002). Provided that the reasoning is completely and sufficiently enough, reasoning about the application conclusions can be made of the predictability of the application in theory. Reasoning provides thereby one way of discard bad solutions without having to test them. Another advantage of reasoning about the application is that some situations may not have to be tested since it is possible that experiments have already been conducted on this very type of situation. For example, an application uses an ORB not implemented by the vendor of the application. If there has been tests conducted by the vendor of the ORB that proves that the ORB provides end-to-end predictability to the objects using it, it would be possible to make the conclusion that the objects connected to ORB in the application will have an end-to-end predictable communication.

6 Analysis

The purpose of this chapter is to provide an analysis of the results of the selection of replication approaches as well as analysis of the two proposed architectures.

This project aimed to investigate the use of CORBA as a middleware in a distributed real-time database (DRTDB). The goal was to find an architecture based on CORBA that combines the features from the real-time as well as from the fault tolerance domain in order to use CORBA in a DRTDB.

6.1 Replication strategy

The first proposed architecture uses FT-CORBA to supply object replication. The reasons for choosing this replication approach are explained in this section.

6.1.1 Extend the ORB

The approach of extending the ORB has the advantages of being simple, since there is no need to create a new group system, and transparency since a client does not distinguish from an object and an object group using the same interface. However, there are two major drawbacks with the approach extending the ORB. First, it is not portable since it deviates from the CORBA standard. Second, the semantics of the CORBA standard are modified since an object reference does not point out a unique object. Therefore this approach is not used in this project.

6.1.2 Object Group Service

The object group service gives the advantage of adding a service to the CORBA standard, which makes it portable and CORBA compliant. Since the major reason for rejecting the approach of extending the ORB is the change of the CORBA standard, the object group service is a more appealing approach. However, one problem with this approach is that the object service is a single point of failure and in order to make it fault tolerant the object group service must be replicated in some way. Another problem, in contrast to FT-CORBA, is that the object group service does not provide any features for replica consistency, fault detection or fault recovery.

6.1.3 FT-CORBA

The FT-CORBA standard has the advantages of being an OMG standard for fault tolerance and, thereby, provides the developer with services for object replication as well as services for recovery. Another advantage of the FT-CORBA standard is that the standard has no single point of failure. The Interoperable Object Group Reference (IOGR) is an Interoperable Object Reference (IOR) containing multiple IIOP profiles. In case of failure when a client tries to access the object group via one profile, the client is redirected to another profile and thereby supporting client redirection transparency.

However, invocations can only be made to the object group and not to an explicit member of the object group. This is a problem, since the client may need to be able to choose which server that should carry out the request for the client. For example, if one vital resource resides on one node, changes to its local replica may have to be committed without making an update to all replicas first.

Although explicit invocations can not be made, FT-CORBA has the advantage of no single point of failure and there is no need to make any changes to the CORBA standard. Therefore FT-CORBA is chosen to be used in the first proposed architecture.

6.2 Reasoning about the FT-CORBA architecture

The negative effect of managing the replicas as an object group is that the FT-CORBA standard does not allow the client to make an invocation to an explicit member, when active replication is used. In this project local commits are used to satisfy predictability and efficiency at the local node. If the client can not make the invocation to its local node, it might not receive a response within the specified deadline. If all group members are connected to a real-time ORB, this might not be a problem, since RT-CORBA provides end-to-end predictability and the time for a member to response is therefore bounded. However, the fact that FT-CORBA requires strong replica consistency, that is, all members has the same state after each invocation, might jeopardize the predictability since a new invocation can not be made before all replicas has the same state.

6.3 Reasoning about semi-active replication

The main goal of the semi-active replication is to provide the right answer in the right time. The tests conducted by Schmidt et al (2002) shows that the semi-active replication is better suited for use in a distributed real-time environment than an ordinary active replication. In the experiments conducted, the TAO Real-time Event Channel was used to propagate information to the replicas. The TAO ORB (Schmidt et al, 1997) is a real-time ORB supporting hard deadlines, and using TAO would provide predictable communication between the objects.

However, with this solution there is the problem with failure of the multicast manager. As the architecture is described, there is nothing that prevents the multicast manager to fail, and thereby making it impossible for the replicas to distribute any changes made at the local database. An approach for dealing with this problem is discussed in chapter 8.

The proposed architecture based on semi-active replication has the drawback of a single point of failure, although, this architecture is better suited for solving the problem than the proposed architecture based on FT-CORBA. No *strong* replica consistency is required which makes it possible to use eventual consistency. All messages from the replicas to the multicast manager are distributed in the same order as the multicast manager received them. Thus, all updates are made by all replicas in the same order, thereby providing replica consistency. By using a RT-ORB, for example TAO, the communication between the objects are predictable with the ability of meeting hard deadlines.

7 Conclusion

This chapter provides the contributions of the project in section 7.2 and in section 7.3 a discussion of the work is performed. Finally, in section 7.4, possible future work is presented.

7.1 Summary

The problems associated with communication with heterogeneous components in a distributed environment made OMG create a standard for a DOC middleware, namely CORBA. The use of CORBA as a middleware in a distributed environment provides transparent communication between the components. With a distributed real-time database (DRTDB), there are requirements of predictability and efficiency as well as requirements of data consistency. The aim of the project presented in this report is to investigate the suitability of COORBA as a middleware in a DRTDB, that is, investigate whether CORBA satisfies the requirements of timeliness and consistency.

The hypothesis of this project is, as described in section 3.3, that CORBA is a suitable middleware for use in a DRTDB in the sense that CORBA satisfies the requirements of timeliness and replica consistency. In order to make a statement of whether this hypothesis is true or false, conclusions of the objectives must be made.

7.2 Contributions

The CORBA middleware must be able to satisfy the requirements of timeliness and predictability in order to be able to use CORBA in a real-time environment. The extension RT-CORBA standard provides end-to-end predictability both vertically, and horizontally, thereby satisfying the real-time requirements.

In order to manage object replication, different approaches proposed was investigated. The conclusion of this investigation is that FT-CORBA provides fault tolerance through object communication, fault detection and fault recovery. The other approaches investigated has drawbacks such as derivate from the CORBA standard or lack of fault detection and recovery features. The fact that FT-CORBA also provides the possibility of choosing whether the replica consistency should be application or infrastructure dependent gives the conclusion that FT-CORBA is the well suited replication approach for this project.

Using FT-CORBA as the replication strategy and use a RT-ORB to provide predictability might not be a solid solution, since there might not exist an upper bound for the time to recover from failure or resolve data conflicts. Although the communication between the replicas is predictable, this might not be enough.

Combining the FT-CORBA and the RT-CORBA standard is not easy, since making decisions in favor of one, could be disastrous for the other. Therefore, a proposed alternative replication strategy is used in the second proposed architecture. The semi-active replication has the advantageous of diminishing overhead associated with fault tolerance and providing reliable, bounded synchronization of the state after an invocation. The conclusion made by Schmidt et al (2002) is that this type of replication is better suitable than the active replication in FT-CORBA and therefore a variant of the semi-active replication is used in one of the proposed architectures.

The hypothesis can not be considered false, since the proposed architectures do meet the requirements of real-time and replica consistency. However, the architecture based on the semi-active replication does not provide a fault tolerant approach but it is a step in the right direction.

7.3 Discussion

The project described in this paper aimed to investigate the suitability of CORBA as a middleware in a DRTDB. The assumptions made for this project, as stated in section 3.2, is that local commits are allowed, no site failure will occur and conflict detection or conflict resolution is not concerned. In a real-world, these assumptions would not hold, since, site failures do occur.

The conclusions drawn from the proposed architectures show that FT-CORBA is the best suited object replication approach, since it provides fault tolerance along with object replication. In the real world where it would be possible of site failure, this conclusion is accurate, however, with the assumptions made of no site failure it would be possible that the replication approach based on the CORBA Event Service is better suited. Since no site failure would occur there would be of little interest to use the FT-CORBA infrastructure for fault detection and fault recovery. Perhaps the Event Service approach would provide a more appealing approach, since all communication goes via the event channel and it would therefore be possible to extend the event channel in such a way that data conflicts will occur. The event channel is an object that handles the multicasts to the replicated servers, and it might therefore be advantageous to implement a function in the event channel that recognizes data conflicts in the requests made by the clients.

In the second proposed architecture, based on semi-active replication, there is the problem with data conflicts. According to the assumptions, this problem is application dependent and it is therefore up to the replicated objects to solve conflicts. However, since all changes are propagated to all replicas via the multicast manager a similar approach for conflict detection as mentioned above would be advantageous to use. By solving data conflicts in the multicast manager, the servers might be less complex to implement and manage.

Only reasoning about a solution might not give a complete proof of the suitability of the proposed architecture and therefore implementing a prototype would be of interest. It is of interest to implement a prototype based on the semi-active replication to be tested in order to evaluate the real-time requirements. It would also be of interest to implement a prototype of the architecture based on FT-CORBA and make a comparison between the two proposed architectures.

7.4 Future work

The proposed architecture based on FT-CORBA has the drawback of a client not being able to invoke methods on an explicit member. In a future work this would be desirable to solve in order for client to commit locally. Another drawback with FT-CORBA, from a real-time point of view, is that strong replica consistency is required. In a future work it is desirable to investigate the possibilities of relaxing this requirement and make FT-CORBA able to provide eventual consistency.

With the architecture based on semi-active replication it is desirable to solve the problem with the multicast manager being a single-point of failure. One approach might be to implement the multicast manager as a replicated object and thereby making it fault tolerant.

No prototype was implemented and therefore, no feasibility test could be conducted. It is desirable to implement a prototype and to test the prototype in order to show that CORBA can be used in a distributed real-time database.

8 References

Berndtsson, M. & Hansson, J. (1995) Issues in active real-time databases. In: M. Berndtsson & J. Hansson (Eds.), *Active and real-time database systems (ARTDB-95), Skövde 1995,* (pp 142-157). Springer-Verlag

Brohede, M. (2000) *Component decomposition of distributed real-time systems.* Unpublished final report from University of Skövde.

Burns, A. & Wellings, A. (1997) *Real-time systems and programming languages* (2nd edition). Addison-Wesley.

Elmasri, R. & Navathe, S.B. (2000) *Fundamentals of database systems* (3rd edition). Addison-Wesley.

Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L (1976) The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM, 19*(11) 624-632

Felber, P., Garbinato, B. & Guerraoui, R. (1996) The design of a CORBA group communication service. *In Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, (pp 150-159).

Felber, P., Guerraoui, R., & Schiper, A. (1997) Replicating Objects using the CORBA Event Service**.** In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis.

Eriksson, J. (1997) Real-time and active databases: A survey. In: S.F. Andler & J. Hansson (eds.), *Active, real-time and temporal database systems*. Springer-Verlag

IONA & Isis (1994) *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc.

Kao, B. & Garcia-Molina, H. (1995) An overview of real-time database systems, In: W.A. Halang & A.D. Stoyenko (eds), *An Overview of Real-Time Database Systems, in Real-Time Computing*, (pp 261-282). Springer-Verlag

Krishna, C.M., & Shin, K.G. (1997) *Real-time systems.* Mac Graw Hill, Inc.

8 References

Lundström, J. (1997) A conflict detection and resolution mechanism for bounded-delay replication. Technical report HIS-IDA-TR-97-10. University of Skövde, Sweden.

Maffeis, S. (1995) *Run-Time support for object-oriented distributed programming.* PhD thesis, Univeristy of Zurich.

Mowbray, T.J. & Zahavi, R. (1995) *The essential CORBA – system integration using distributed objects.* John Wiley & Sons.

Mowbray, T.J. & Ruh, W.A. (1997) *Inside CORBA.* Reading, Massachusetts: Addison-Wesley.

Object Management Group (1999) *Fault Tolerant CORBA Joint revised submission.* ftp://ftp.omg.org/pub/docs/orbos/99-10-05.pdf [Fetched 2002-04-03].

Object Management Group (2001) *The Common Object Request Broker: Architecture and Specification* (2.6 edition). ftp://ftp.omg.org/pub/docs/formal/01-12-35.pdf [Fetched 2002-02-20].

Object Management Group (2001b) *Event service specification version 1.1.*

ftp://ftp.omg.org/pub/docs/formal/01-03-01.pdf [Fetched 2002-04-18].

Otte, R., Patrick, P. & Roy, M. (1996) *Understanding CORBA the common object request broker architecture.* Upper Saddle River, New Jersey: Prentice-Hall

Ramamritham, K. (1993) Real-time databases. *International Journal of Distributed and Parallel Databases, 1*(2), 199-226

Schmidt, C.D., Levine, D.L., & Mungee, S. (1997) The Design of the TAO Real-Time Object Request Broker. *Computer Communications Journal.*

Schmidt, D.C. & Kuhns, F. (2000) An Overview of the Real-time CORBA Specification. *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing, 33*(6), 56—63

Schmidt, D.C., Gokhale, A.S., Natarajan, B. & Cross, J.K. (2002) *Towards dependable real-time CORBA middleware.* http://www.cs.wustl.edu/~schmidt/PDF/CCJ-2002.pdf [Fetched 2002-04-19]

Szyperski, C. (1997) *Component software – Beyond object-oriented programming.* Addison-Wesley

Ulusoy, Ö. (2001) Data replication and availability, In: K-Y., Lam & T-W., Kua (eds.), *Real-time database systems – Architecture and techniques*, (pp 217-225). Kluwer Academic Publishers.

Ulusoy, Ö. (2001b) Distributed concurrency control, In: K-Y., Lam & T-W., Kua (eds.), *Real-time database systems – Architecture and techniques*, (pp 206-215). Kluwer Academic Publishers.

Özsu, M.T. & Valduriez, P. (1999) *Principles of distributed database systems* (2nd edition). Upper Saddle River, New Jersey: Prentice-Hall