1. Introduction

Our society is increasingly dependent on service-oriented applications built from complex pieces of software. Significant questions for achieving high availability in multi-tier service architectures are: (1) Do we gain from separation of code that implements functional and non-functional requirements in a service? In particular, what are the benefits of doing the separation by implementing support for fault tolerance (FT) in the middleware? There are two extreme answers to this question:

- A lot! all FT-related code should be implemented in middleware.
- Nothing! FT-related code can be tailor-made in each application.

The truth could of course be somewhere in between. Some applications will benefit from FT support via middleware, and the implementation of the support could benefit from some application knowledge. The follow up question, assuming that performance is a major requirement in most applications, is: (2) How is performance affected by increased availability? Specifically, will different techniques for achieving fault tolerance have different benefits and penalties, and how are these factors affected if the support is implemented in middleware? Our research agenda is to provide quantifiable evidence in answering these important questions.

In telecommunication systems CORBA is a widespread middleware due to these systems’ long lifetime and interoperability requirements [2]. Moreover, a review of the related work for other middleware shows that CORBA has the most elaborate standardised schemes. Telecom management systems are large and complex pieces of software that are costly to maintain. One factor affecting this complexity is that system availability is typically dealt with at the application level by ad hoc fault tolerance mechanisms. Thus, multiple applications reproduce almost the same recovery routines over and over again. Building fault tolerance capabilities in middleware has the benefit that reliable code (for implementing fault tolerance) is produced once, there is less application code to maintain, and the failover procedures are transparent. The application writer has to write very little FT-specific code, and the client need not even notice that a server goes down whilst the service continues. The remaining question is therefore the adequacy of average response times for serving a client request.

Extending an Object Request Broker (ORB) for dealing with fault tolerance properties affects timing behaviour of an application. Compared to the use of a standard non-FT ORB, we would expect the client to experience longer response times due to continuous saving of information that is recaptured after failures. To measure the response time one could sum up the chunks of time spent by a CORBA request on different segments while travelling from the client to the server object, getting processed and returning as a reply. In an ORB that does not support fault tolerance the time intervals consist of the time spent in the ORB layers and the time spent on the network transport. We will refer to their sum as roundtrip time. In an extended fault-tolerant ORB the time intervals spent on segment $i$ of the named path will increase by a certain amount ($\delta_i$). The sum of $\delta_i$ on all segments gives the so-called overhead. This overhead is encountered constantly while the system is running under no failure conditions.

Similarly, when a failure occurs, all requests sent around the moment of server crash will potentially experience a larger overhead. The reason for the increase is the time spent for reconfiguration of the server, to designate a new request processing replica, and to set its state up-to-date. This time is called failover time.

When quantifying the performance-availability trade-offs, one can either consider only application objects’ failures, or consider eliminating all single points of failure. We have studied both approaches by building and evaluating two extensions of an open source ORB, Exolab OpenORB [4]. One extension closely follows the FT-CORBA standard and has no infrastructure replication. FT-CORBA standard creates new infrastructure objects that unless replicated, can act as central units that become single points of failure. The other, called the fully available platform (FA), uses distributed infrastructure units for dealing with recovery. Failover support is implemented by a combination of leader election and consensus primitives. In our evaluation of the trade-offs we have experimented with a generic service from Ericsson Radio Systems on top of both extensions.

In our experiments we used roundtrip time overhead (in absence of failures) and failover times as performance metrics. Having these quantified, the telecom engineer, depending on the availability requirements of the particular service, can make the relevant decisions for his/her replicated application configuration. The decision will be highly influenced by how critical is the delivery of that service; hence, how fast are the desired reactions to failures. The
decision is also dependent on how much latency can the client tolerate when receiving a reply to a request in a no failure scenario.

2. Terminology

In this section we review some important terms used in the rest of the chapter. These terms are widely used in the fault tolerance literature but we repeat them here to make the text self-contained.

2.1. Replication styles

The most straightforward way to obtain fault tolerance is to replicate processing units, i.e. to distribute the same application functionality on several nodes. Depending on whether only one or all of the replicas process every arriving client request, we can distinguish two main replication styles:

- Primary-backup, known also as passive replication
- Active replication

In the first case, only the primary replica processes incoming requests, while its state is read (checkpointed) and stored in a log from time to time [8]. When the primary fails, one of the backups has to take over its role, using a failover protocol. The last state of the primary will be transferred to the backup and the requests that arrived since the last checkpoint are replayed. In the second case, all replicas receive and process the requests from clients. To ensure that the same result is sent to the client no matter which replica responds, replica computations need to be deterministic[21]. In both cases the client perceives the replica set as a single server, and all state saving and failover operations are transparent to the client.

2.2. Consensus

In some applications distributed servers have to agree on some information, e.g. the outcome to be sent to the client. The algorithm that achieves agreement is called a consensus algorithm. There are two main steps to perform: putting forward the own proposal, i.e. the value to agree upon, and deciding on a common value. There are intermediate steps performed in a number of rounds to achieve this.

A consensus algorithm must guarantee that all replicas decide the same value (agreement), the decided value was proposed by at least one of the involved parties (validity) and that all of the replicas manage to decide within a finite interval of time (termination). Essentially, the properties involve replicas that have not failed.

When failures are expected to happen a major assumption affects the solution:

- The distributed system is synchronous: upper bounds on message delivery delays are known; also, the different computation nodes have a processing rate with a known relation (when a process performs a computation step, every other process performs n computation steps for some n ≥ 1).
- The distributed system is asynchronous: no knowledge about message delivery delays, or computation rates exists.

Fischer et al. presented a major result in this research area in 1985 [7]. It states that there are no general algorithms to implement consensus in the asynchronous setting even if a single node can be expected to crash. Intuitively, the difficulty arises because there is no way to know whether a message from a node is only late or will never arrive (since the node crashed). The theory of unreliable failure detectors was developed to circumvent this impossibility result by making further assumptions (see next section).

2.3. Unreliable failure detectors

Formalized by Chandra and Toueg in 1996 [2], an unreliable failure detector is a distributed entity used as an element in consensus algorithms for asynchronous systems. The distributed failure detector provides information about which nodes are “correct” (have not crashed) and can participate in agreement protocols. The information about failed units comes in the form of so-called suspicions. Since the failure detectors are unreliable, from time to time, the information may be wrong in one of the following two senses: either some nodes that failed are not detected (lack of completeness), or correct nodes are suspected to have failed (lack of accuracy). By allowing such mistakes to a certain extent, consensus can be solved even in a failure prone asynchronous system. Chandra and Toueg classified failure detectors in terms of properties that characterise correctness of their detections (in the above two
senses). Then they showed that consensus can indeed be solved in an asynchronous setting for given types of failure detectors. In particular, even using the weakest failure detector consensus can be solved if a majority of the nodes are always correct.

2.4. Broadcast

Consensus algorithms typically use the primitive operation broadcast in order to send the same message to a set of multiple receivers [9]. The abstraction encapsulates the requirement that all destination nodes should receive (and use) the same message. In failure prone systems this is not trivial to obtain. It is possible that the sender fails in the process of sending the message to a bunch of receivers. To ensure that in such a case either all or none of the receivers use a given message, we need to build the broadcast mechanism on top of other primitives: we distinguish between receiving and delivering a message. A broadcast message is only delivered (used in an application) if all the nodes in the destination set have received the message. This notion is then used to define a useful broadcast, namely an algorithm that implements a reliable broadcast. This primitive basically guarantees that all correct receivers deliver the same messages, messages that are delivered have been sent by some sender, and all messages sent by the correct senders are eventually delivered.

3. Background

Bakken defines middleware as “a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems” [1]. One of the complexities in distributed systems arises from potential failures. Thus, there have been several attempts to build support for fault tolerance in middleware. In this section we first briefly review the combination of FT support and middleware in general, and then go on to present related approaches for supporting FT in CORBA.

3.1. Middleware fault tolerance

To replace fault tolerance code at application level, one would expect an FT-supporting to provide at least: client transparency to failures, and automatic replica management with state saving and restoring features. Middleware exist in several flavours: those incorporating component/object as well as communication management (e.g. CORBA, COM/DCOM), those dealing with communication management (e.g. Java RMI), those that support resource allocation in a dynamic way (e.g. Jini), and those that offer containers for objects that can communicate using several offered technologies (e.g. EJB).

Microsoft’s COM/DCOM allows customizing different elements of the remoting architecture\(^1\) in order to accommodate communications with multiple server copies and transparent failover. The COMERA extension by Wang and Lee [24] was needed in order to provide efficient ways especially for replica migration where connections have to be restored in a transparent way and logged messages to be replayed.

Java RMI, as a basic technology does not include fault tolerance features. On the other hand, built on top of this communication infrastructure, middleware such as the Jgroup toolkit [13] or the AROMA [14] system provide transparent fault tolerance to applications. In Jgroup, clients transparently access replica groups as one entity. Replica group members, when processing a query, exchange messages ensuring proper state logging. Failure of a replica is transparently handled by the group. AROMA intercepts RMI messages, and sends them in the same order to all server replicas, thereby assuring consistent states at recovery time. Replication style can be chosen by the application writer based on fault tolerance needs and failover time allowed.

Jini addresses a different approach to fault tolerance. As a technology inherently used for distributed, potentially mobile, networked applications, it does not advocate explicit replication for fault tolerance. Therefore, no automatic application server or client replication is supported; neither is application state saving and restoring. On the other hand, by offering the Lookup Service (possibly replicated) and the lease-based resource allocation approach, failures of different service providers can be made transparent to the client [12,17].

EJB technology uses clustering and transactions to provide transparent fault tolerance to client applications [10]. Still, this is not sufficient for the servers to be available in case of failures. Enhanced with group communication [19], the architecture can provide transparent failover to clients as well as proper state logging and recovery.

---

\(^1\) Infrastructure connecting clients to server objects
3.2. CORBA and fault tolerance

CORBA provides by far the most developed support for fault tolerance in middleware. Prior to the specification of the CORBA extension for FT (April 2000), there are few reported works on alternative augmentations of CORBA with a process (object) group module. Some of these have provided a basis for the development of the standard.

Felber et al. present three different approaches for introducing object groups in CORBA, dependent on the position of the group communication module relative to the ORB [6]. These are: the interception approach, the integration approach, and the service approach. The work covers the third alternative. The object group service (OGS) uses different CORBA objects to implement fault tolerance related services.

Narasimhan et al. developed and implemented the idea of operating system level interceptors to provide fault tolerance to CORBA [15]. The result of research efforts in this direction is the Eternal System. With this approach, an ORB’s functionality can be enhanced for fault tolerance without changes in the ORB, or in the application.

DOORS is a fault-tolerant infrastructure built using the service approach [3,16]. Application objects register to DOORS in order to be made fault-tolerant. Fault tolerance services are realized with two components: ReplicaManager and WatchDog.

A framework for fault-tolerant CORBA services by using aspect-oriented programming is presented by Polze et al. [20]. Their goal is to provide the application writer with the possibility to build a fault-tolerant application by choosing the types of faults to be tolerated: crash faults, timing faults, design faults. The toolkit then chooses the appropriate fault tolerance strategy.

Killijian and Fabre [11] describe a framework for building fault-tolerant CORBA applications by using reflection. They define a meta-object protocol, and use an open compiler, to be able to extend CORBA objects with wrappers of modifier methods. The wrappers always save object state when the wrapped modifier method is called. Therefore, application writers do not have to write state reading or applying methods.

Most of the above works differ from the work we present in this chapter in that they do not provide trade-off studies, or study artificially created applications on top of the platform. None of them follows the actual FT-CORBA standard.

4. Standard support for FT in CORBA

The CORBA standard was extended to support basic mechanisms for fault tolerance in December 2001 [18]. At that time, no major studies documenting the performance/FT trade-offs in an FT-CORBA based implementation existed. To provide some evidence for conditions under which the suggested primitives would be useful in a real application, an experimental platform in compliance with FT-CORBA was built [23]. In section 7 we present the results of these studies. But before that, we provide a short overview of the FT-CORBA standard in this section, and explain the architecture of the implemented platform. This can be seen as an infrastructure over which FT applications can be built with little effort.

4.1. The FT-CORBA standard

Applications are supported for fault tolerance through replicating objects. Temporal replication is supported by request retry, or request redirection. Replicated objects are monitored for failure detection. The failover procedure depends on the replication strategy used.

Support is provided for use of active, warm and cold passive, and stateless replication. In cold passive replication, backups are totally passive and are neither updated with primary state nor with information about processed method calls. The state of the primary is checkpointed periodically. Also, information about method calls that change the state of the replica object is stored in a log. These are called update method calls, as opposed to read-only ones that do not modify the state. At failover, all this information is transferred to the backup that has to take over the role of the primary. For warm passive replication, on the other hand, state checkpointing at the primary coincides with transfer of that information to all the backups. Also, information about update method calls executed at the primary are broadcast to the backups and stored in a log, without being executed. At failover, all necessary information is present at the backup that will be promoted to the primary role. Stateless replication is used only when the server does not have state. For active replication, the standard strongly recommends the use of a gateway for accessing the active replica group. This gateway plays the role of a relay for method calls - it broadcasts the method calls to all replicas that have to execute them in the same order.

The basic building blocks in the infrastructure are the Replication Manager, the Fault Notifier, and the Fault Monitor. The Replication Manager implements interfaces that provide operations for setting group properties, managing replicated groups by the application, as well as creating groups on demand, and handling failover.

2 This functionality appears in the Object Factory in the form of replica creation.
The Fault Notifier interface contains methods for registering fault consumers, for announcing faults, as well as for creating event filters. Fault consumers are those nodes in the application or the infrastructure that have to know about the occurrence of faults. An event filter is related to a fault consumer and is used to send a fault notification exactly to those parties that are interested in it.

To be able to manage large applications, the notion of fault tolerance domains is introduced. Each fault tolerance domain contains several hosts and object groups and a separate pair of Replication Manager and Fault Notifier is associated with it. On every host there has to be one Object Factory and one Fault Monitor. The Object Factory creates group replicas on demand. The Fault Monitor has to detect failures of group members running on its host.

Every application object has to implement interfaces for failure detection and checkpointing. Logging and recovery mechanisms, needed for passive stateful replication, are automatically used when infrastructure-controlled consistency and membership is chosen by the application. In active and stateless replication styles, request replies can be logged in order to avoid repeated execution of a method call.

4.2. Architecture units

Figure 1 shows the overall architecture of our prototype FT-CORBA platform. In particular, it depicts the service objects used as building blocks.

![Figure 1 The overall architecture of the FT-CORBA platform, with one replicated server](image)

The building blocks of the infrastructure are:

- A collection of service objects (in the named boxes and ovals in Figure 1)
- CORBA portable interceptors (explained later)
- ORB class extensions (more details in [22])

We now go on to describe the basic building blocks. Note, however, that the elements of the infrastructure are not replicated. The assumption is that the most vulnerable units are application objects (the different shapes in Figure 1), while infrastructure units run in processes or machines that are not failure prone.

4.2.1. Service objects

The service objects consist of a set of CORBA objects for which the roles were clearly defined by the standard; but the standard leaves some implementation choices open. So for the implemented platform each had to be considered separately:

- The Object Factory, Replication Manager and Fault Notifier are described as interfaces in the standard. Therefore, they were implemented as CORBA objects.
- Fault Monitors were implemented as CORBA objects.
- The implementation of the Logging and Recovery Mechanism needed some careful considerations.
- For the active replication style, the standard leaves the choice of a mediator gateway object open. The creation of such an object as part of the infrastructure was chosen as a solution.

Next a more detailed view of the above object categories will be provided.
4.2.1.1. Standard service objects

In a fault tolerance domain, there is one Replication Manager, CORBA object in charge of creating object groups, whenever the application demands it. A more dynamic and important role of the Replication Manager is to coordinate the failover procedure after it receives a failure report from the Fault Notifier. For passive replication the procedure includes the promotion of a backup node to the position of primary. In either case (active and passive), the “failed” replica must be destructed. Note that this applies even if the suspicion is inaccurate. Even if the object was temporarily unavailable, it is permanently removed from the set of replicas. The FT-CORBA platform, thus, does not use unreliable failure detectors.

Object Factories are distributed over every host in the fault tolerance domain in which application object replicas can be created and run. Upon request from the Replication Manager, the Object Factory creates a new instance of an application object and starts it in a different process on the machine. Replicas of an object group are created on different hosts. The list of hosts that can be used is given in the application’s request to the Replication Manager.

Replica failures are detected by Fault Monitors running on each replica’s host. For each object to be monitored, the Fault Monitor CORBA object creates a separate thread. A pull based monitoring style is used, i.e. the Fault Monitor calls the \texttt{is\_alive} method of the application CORBA object periodically to find out whether the replica is still able to process client requests. As soon as this method call times out, or it signals a failure, the Fault Monitor reports this to the Fault Notifier. The latter sends fault notifications to its registered consumers, one of them being the Replication Manager.

4.2.1.2. Gateway for active replication

Instead of the client broadcasting its request directly to all active server replicas, our platform uses a gateway CORBA object. This object is created as soon as an actively replicated group is deployed, and started on one of the server machines. The gateway fulfills a double role in the context of processing requests by the active group.

First and foremost, the gateway broadcasts incoming client requests as method calls to the replicas. Secondly, when the replicas, in their turn, become clients for a third tier server, the gateway acts as a duplicate suppressor. Thus, outgoing method calls from replicas are routed through the gateway, whose reference is known by the replicas’ ORBs. The reply received from the server is sent back to all replicas via the same path. Finally, the client will be sent the reply provided by the fastest active server replica.

The gateway participates in the group reconfiguration procedure when an active replica fails. To begin with, on its creation the gateway registers as another consumer (besides the Replication Manager) of the Fault Notifier. After a failure, the gateway finds out the new configuration of the group and directs the future broadcast requests to the current set of active replicas. When a new replica is created and joins the group, the gateway manages the state transfer procedure. It stops requests that arrive after the start of the state reading, and broadcasts them to the group only after the new replica is successfully set up.

4.2.1.3. Logging and checkpointing object

The logging and recovery mechanism is only used for cold and warm passive replication (see Figure 2). The mechanism is implemented on each host using a separate CORBA object. This object’s interface offers operations for logging method call and reply information. Other operations help in retrieving the logged information, when needed at failover. Only information about update method calls that arrived since the last checkpoint must be logged, because only those are needed in the failover procedure.

![Figure 2 Checkpointing and logging in primary-backup replication](image)
We will come back to the logging and recovery mechanism in section 4.2.3.

4.2.2. CORBA portable interceptors

The CORBA standard offers an elegant solution, portable interceptors, for performing operations around a CORBA request at the client side, as well as before the start or after the finishing of the computation performed at the server application object. In the presented platform, CORBA interceptors were used to perform the following operations on the request:

- Adding extra information to the request: the server group reference version known to the client, as well as a request identifier that remains the same even after resending the request, and a request expiration time. This information is placed in a service context at the level of the client portable request interceptor.
- Recording information about a call and its reply in a log. These operations take place in the server request interceptor.
- Tracing of timing information when performance measurements are needed. This operation is done both in the client and the server interceptors.

For different replication styles, application server objects are equipped with different portable server interceptors. For example, in active replication, no logging of method call information is needed. In cold passive replication, method calls are logged only at the primary (see Figure 2).

4.2.3. Logging and recovery mechanism

This section provides more details about how the logging and checkpointing mechanism works and some important implementation decisions.

4.2.3.1. Checkpointing

As shown in Figure 2, the logging and recovery mechanism calls get_state on the primary replica to perform state checkpointing. Execution of get_state has to wait until all update methods that arrived earlier finish their execution. Analogously, execution of update method calls that arrive while the state is being read has to wait until get_state finishes execution. In this context some further questions arise: should checkpointing be done periodically? If yes, how often? When the state does not change so often, it would be better to save it as soon as a change occurs. This would lead to event-triggered checkpointing.

When the primary state is obtained it must be stored for reuse on recovery from failures. Another issue is related to storing only state change records instead of entire state records. This way of checkpointing (get_update, as opposed to get_state) is efficient especially when the state can become large, but through small increments. However, there is a problem with this approach: when the fields of the object are complex structures, a change to the state might be difficult to express. As the get_state and get_update methods have to be written by the application developer, it is important that the demands are feasible. Note that if get_update methods are used, all state update records have to be kept in the log, as opposed to entire state records that are removed as soon as a new checkpoint is performed (in fact, in the latter case, the log always contains one single record). So, there are also issues related to the extensions to the footprint of the middleware that have to be considered.

4.2.3.2. Logging method call and reply information

As well as logging the state information, one needs to log the arriving requests for later replay purposes. Also, one needs to record the evidence that a request was executed once on the server object, at least in the case of update method calls. Executing an update method twice on the server can lead to inconsistent updates of the state, whereas a read-only method reply can potentially be sent twice.

Update method call information has to be logged in a certain order. In our platform this is given by the order of execution of those methods on the primary replica. This is due to the need for later replaying the methods on the backup, in the same order that they were executed on the primary. This order is only available from the log. Further, since logging is done separately from execution, at the level of the server side portable interceptor, the middleware has to enforce sequential method execution, independent of the threading policy in the ORB. The evidence that an update method was executed may consist of a simple note, or, when there is a return value, of the result structure.

The type of a method call (update or read-only) is known in the server side interceptor. When creating a replica of the server group, the Object Factory (written by the application writer) provides this information together with method signature information to the ORB in which the replica is started.
This concludes our overview of the implementation of a platform in accordance with the FT-CORBA standard.

To summarise the characteristics of the platform, it supports three replication strategies, but for active replication, it builds on the replicas being deterministic. It does not support failures in the infrastructure units, and it works in a multi-tier application architecture provided that all the tiers provide support for FT with the same platform. We now go on to describe an alternative platform, also based on CORBA, and set the stage for a comparative evaluation of these in section 6.

5. Adding support for full availability

This section describes the architecture of a fully available (FA) middleware. In this approach the infrastructure units are distributed and may only fail together with application objects, and therefore an algorithm that takes care of application replica failures is sufficient for assuring the availability of the whole system.

We build our fully available middleware on top of CORBA due to its attractive properties e.g. scalability, interoperability. Also, this gives us an opportunity to evaluate FT-CORBA in comparison with another CORBA-based FT support. As an ingredient in the FA infrastructure we employ an improved version of an algorithm (called universal construction) originally developed by Dutta et al. [5]. The emphasis in this section is put on how the algorithm was embedded in a CORBA infrastructure that should provide fault tolerance and transparency to applications. Thus, architecture elements are presented together with CORBA specific implementation issues. The improvement made to the algorithm does not affect its availability aspects (these were already present in the original algorithm), but its performance aspects on recovery from a failure. The last part of this section will describe this improvement for the interested reader.

5.1. Architecture units

The conceptual building blocks of the fully available infrastructure are two distributed components: (1) a consensus service used to ensure the agreement among the replicas on the result to be returned to a client request and the state change enforced on the replicas, and (2) a leader election service used to designate the replica that is supposed to deal with the request.

The algorithm incorporates unreliable failure detectors, and hence the detection of failures is in terms of suspicions. The leader election service might elect different leaders when wrong failure suspicions arise. In this case, the consensus service helps resolve the conflict. The leader election service at each node relies on information from its (unreliable) failure detector oracle (that corresponds to the weakest failure detector in [2]). The failure detector is implemented as distributed modules that exchange heartbeat messages. A server replica that is not leader is simply called witness in the sense that it helps its actual leader resolve a conflict with other possible leaders, and stores the result of the real leader’s actions.

Figure 3 illustrates the deployment of the FA infrastructure units in a CORBA environment. Each server replica uses the following units when executing the steps of the consensus algorithm: the leader election unit (LEU) and the consensus object (CO). Another element involved in the processing of a request is the local copy of the application server object (ASO) within the server interceptor. All operations on the application object are executed at the level of the server side portable interceptor (as explained shortly). The failure detector is set up as a separate server thread, attached to the application CORBA object. As part of the leader election unit, the failure detector server is supposed to receive I-am-alive messages from other LEUs at other application replicas.

![Figure 3 Server replica architecture](image)

![Figure 4 Client side architecture](image)

Note that it can be assumed that the leader election unit (and thus failure detection unit) sends “correct” I-am-alive messages on behalf of the application server replica. In other words, the fault tolerance approach properly deals
with crashes of the whole node. Thus, the absence of an I-am-alive message leads to the suspicion of the non-availability of the application server, although the message comes from a separate unit. When trying to introduce the outcome of a request in the total order of outcomes recorded by all replicas, the leader election unit is queried for the present leader. The unit will return the correct process with the lowest index.

Figure 4 depicts the client side of the architecture. The client does not have a consensus object, but it has a leader election unit that identifies the replica that is currently in charge of servicing the client queries, similar to the case of server replicas.

5.2. Infrastructure interactions

A view of major interactions inside the fully available infrastructure is shown in Figure 5. The picture visualizes the message flows between similar units within each server replica, as well as between client and server.

The dashed line boxes containing ASO copy suggest that the server interceptors call methods on the application object, from the interceptor level. The dashed arrows that go between two consensus object boxes, or two leader election unit boxes, represent message flow by using simple socket connections, not involving CORBA calls. The dashed arrows that appear inside the solid "host" box, suggest internal operations within a process (e.g. in the client box, the dashed arrow represents a portion of the flow of a method call from the client to the server, that spans from the ORB to the client interceptor CI). The thick solid arrows represent flow of CORBA calls. The thin ones show method calls that are not performed via CORBA (e.g. leader() or propose()), but are done via socket connections.

I-am-alive messages received from replicas are used to determine who the leader is. The client, in order to enable its leader election unit to receive I-am-alive messages from server replicas, has to register with the server group. Registration is achieved by sending an I-am-in message to all members of the replica group. This message has piggy-backed the address in the client where the future I-am-alive messages have to be sent, i.e. the address of the client's leader election unit.

Let us now concentrate on the solid arrows (thin and thick) in Figure 5. The working of the algorithm in a non-failure scenario can be visualised as follows:

1. arrow from ACO to FA-ORB: client initiates method call \texttt{m1};
2. arrow from FA-ORB to LEU: the LEU is queried about the identity of the current leader, as seen by the client;
3. arrow from client CI to server interceptor SI on Host: client sends request corresponding to method call \texttt{m1};
4. arrow from SI (ASO copy) to ASO: call to \texttt{execute()} that means tentative execution of the request on the application object. Tentative means that no durable change of the state of the object is made so far;
5. arrow from SI to LEU: the LEU is queried about the identity of the current leader;
6. arrow from SI to CO: if the LEU on Host indicates leadership, then \texttt{propose()} is called with the outcome (meaning state update and result) of \texttt{execute()}. 
While performing (6) above, the leader tries to write its outcome in a unique position of a total order of state updates maintained at all replicas\(^3\). It achieves its goal only if a majority of the replicas acknowledge its leadership. The dashed arrow between CO on Host\(_1\) and CO on Host\(_2\) “implements” this by a series of READ, WRITE and ACK/NACK messages.

5.3. Platform implementation

This section provides a couple of details in the implemented platform.

5.3.1. Portable interceptors

The main CORBA specific elements employed to implement the algorithm and the infrastructure are portable interceptors. Client side portable interceptors are used to add unique request identifying information to the client request. Portable server side interceptors are used mainly for intercepting I-am-in registration messages from clients, and for intercepting client requests. Also, all the operations devised in the algorithm are performed in these interceptors. Thus:

- When receiving an I-am-in message in the interceptor (sent by dynamic CORBA method invocation), the call is stopped from reaching the application replica by throwing an exception, handled in a well-defined part of the ORB. Before the exception is thrown, however, the leader election unit is informed about the address where the I-am-alive messages have to be sent later.
- The algorithm operations executed in the server interceptor are those described in section 5.2. The tentative execution of the request on the server replica is simulated by three steps: (a) reading the replica state using `get_state`; (b) executing the method call on the server replica, so that the result, if any, is returned and the state of the replica is changed; and (c) reading of the state update. After this execution, if the current process is a leader, but its proposed outcome is not the decided one, or if it is not the leader, the state read before the method execution is re-instated.

Note that if the request is read-only, the server replica does not execute the consensus algorithm. After the request execution the replica sends the reply to the client and stops, returning to the point where it waits for a new request to arrive.

It is also important to note that the only published references (addresses) are those of the application server replicas. Therefore, the I-am-in messages are sent via CORBA dynamic request invocations to these addresses. I-am-alive and other infrastructure related messages are sent using simple reliable socket connections. Thus, CORBA specific overhead is not encountered for these messages.

5.3.2. State transfer

To clarify our improvement of the original algorithm, let us detail the activities performed by a witness including the moment when it becomes a leader.

During normal processing periods, i.e. when no replica fails or is wrongly suspected to have failed, the leader transfers state updates to the witnesses as soon as a request is processed. These updates are stored at witnesses, possibly in a structure residing in memory. With no further intervention (as in the published algorithm [5]) the storage can be filled up with update structures. Therefore, one limitation of the algorithm was the possible infinite growth of memory needs.

Another limitation is related to the moment when a witness becomes a leader. This can happen either because the previous leader really failed, or because the leader suddenly became very slow and the client as well as the other replicas do not anymore recognize its leadership. After this, when the witness-leader replica receives the first request from a client, it has to bring its state to the value found in the previous leader at the time of its abdication. Before the improvement we made, this state was built in the new leader by incrementing the initial replica state, using the updates stored at the other replicas. Obviously, the time for this procedure grows in an unbounded manner the further we get from the start-up time.

The original paper already proposed a way to solve the problem in the future. When a witness receives a state update from the leader replica, it will immediately apply it on the application object, instead of only storing it. Thus, storage needs are reduced almost to zero. Also, the new leader does not need to reconstruct its state by using information from witnesses anymore, since it has the current state itself.

---

\(^3\)From the leader’s point of view, the other replicas are witnesses.
Our platform implements a different improvement. In this setting, from time to time, the leader sends entire state information to witnesses. When receiving it, witnesses prune their storages, i.e. replace the set of existing records with only one state record. Thus, the storage is dimensioned so that it accommodates a limited (quite small) number of records. Consequently, the failover process in the new leader will take a limited and relatively short time to execute. It is possible for the leader to trigger the pruning periodically (say every 5 or 10 seconds). A second alternative is to send state after executing a certain number of requests: say after every 20 executed requests, the leader sends its current state to the witnesses.

The second alternative offers the possibility to dimension the storages to accommodate a known number of records. In the present implementation the periodic alternative was used, since it offered a direct way to compare with the passive replication case in the FT-CORBA implementation, where checkpointing of state was done periodically.

6. Experiments with a telecom application

This section will briefly describe the application that was run on top of the two extended ORBs. Also, the set up for the experiments is presented.

6.1. The service

The platforms were evaluated using a real application from Ericsson Radio Systems. The server is a generic service in the operations and management (O&M) part of a future generation radio network and is called Activity Manager. The role of this server is to create jobs and activities to be scheduled immediately or at later times. Also, the server receives activity status and progress reports. There are fifteen operations (methods) offered by the server to fulfill the functionality it promised. The Manager has internal structures where the activity and job objects are stored. These structures make up the state of the server. They are managed according to what the clients (other parts of the O&M network) ask for. This service was chosen by our industrial partners as it has some generic characteristics that are representative for other applications and some methods that implement a multi-tier call structure.

6.2. Experiment setup

The experiments were performed in a host park connected via an IP network with no isolation from other tasks on the hosts. The reason for not performing measurements in a controlled host environment was to mimic the realistic setting in which the service will eventually run. For both platforms and in all experiments the clients were unchanged. There were two goals to performing experiments: measuring roundtrip time overheads, and measuring failover times. As a baseline for comparison, a non-replicated server was employed where no FT support in the ORB was included. Note that such a comparison preserves neutrality to different replication styles (compared to ad hoc application-level replication that may resemble a style). To avoid the probe effect, there were null probes inserted in the non-replicated case in the same position for each interceptor unit.

For the measurements on failover times, the baseline was chosen as the time taken from the crash of a (non-replicated) server to its restart (possibly manually as it is the case in the real setting). In the different platforms, the failover time is affected by the replication style and is computed as follows, for the FT-CORBA and FA-CORBA platforms respectively:

- In cold passive replication, it is the time taken to set the state on the new primary, plus the time needed to replay the requests arrived since the last checkpoint.
- In warm passive replication, it is the time spent in replaying requests.
- In active replication it is the time spent to reconfigure the server group.
- For the FA-CORBA platform, to measure failover times, the current leader is forced to fail after a certain number of processed requests. Then, a new leader is elected and forced to bring its state up-to-date as soon as a next query arrives.

To obtain relevant results the numbers of requests processed before the failure were chosen to be the same for both warm and cold passive replication in the FT-CORBA platform and for the experiments with the FA-CORBA platform.

For measuring overhead, the clients used in the tests were simple: they called six of the update methods of the server in a loop. The clients called the methods in loops of 100 and 200 iterations. Some of the methods were called once per iteration (Method 1, 2, 3, and 5), others two (Method 6), and four (Method 4) times, respectively. The
results are presented as averages of the measurements for every method call computed over runs in both types (100 and 200 iterations) of loops (hereafter called one experiment). The averaging was done in order to take the different network and processor loads at the time of the experiments into consideration.

In both platforms group sizes of 2, 3, 4 and 8 replicas were used. In the FA-CORBA platform, the group size was of special relevance, since it reflects different majority rules. In the FT-CORBA platform three different checkpointing intervals were used: 1, 5 and 10 seconds. In the FA-CORBA platform, the pruning interval was chosen in a similar way to make valid comparisons.

All in all 12 experiments were performed for each of the following styles: warm passive, cold passive, fully available replicas (corresponding to 4 replica configuration and 3 checkpointing intervals). For the active replication four experiments were performed (corresponding to 4 replica configurations).

6.3. Measuring overheads

The slices in the roundtrip time were used to identify dominant parts of the overhead. The slices can be described as follows: the time spent in the client interceptor (when adding information to the request), the time spent on traveling from the client side node to the server side node and until the request is taken up for processing, the time spent in the server side interceptor, when a request arrives at the server and when it returns to the client as reply.

For both platforms the server-side time includes time spent waiting for all earlier arrived update methods to finish execution on the server. In the FT-CORBA platform, the time for logging the method call information is included, as well as the time spent recording reply information. The approximate computation time is counted from the moment of leaving the interceptor at receiving the request, until it is entered again when sending the reply to the client. In the FA-CORBA platform the situation is different: the time spent in the server interceptor includes the method execution time, as well as the consensus execution time and state reading and updating times. The time taken by the reply to travel back from the server to the client node is traced in the same way in both platforms.

7. Trade-off studies

This section presents experimental results for both the FT-CORBA and the FA-CORBA platforms. First, overheads are presented, followed by the time taken for failover.

7.1. Overheads

Table 1 summarizes overheads in case of the FT-CORBA platform (columns three to five) and the FA-CORBA platform (column six). Each row of the table corresponds to one of the six method calls. Column two contains average roundtrip times for the different method calls when the client was calling them on the non-replicated server. The results in the next columns are presented as average percentages. Each average is computed as follows. Consider the highlighted cell in Table 1 that contains the value 62%. Here, the average roundtrip time value for method 3 on the non-replicated server was 65ms. The % means that the average roundtrip time in the replicated experiments was 65+0.62*65=105 ms. This was the lowest roundtrip time measured within the 12 experiments. The highlighted cell shows that the average overhead when calling Method 3 on the server, while using cold passive replication style, ranged between of 62% and 163% for group sizes of two, three, four, or eight replicas and checkpointing intervals 1s, 5s or 10s. In particular, 62% corresponds to a group size of three and checkpointing interval of 10s, while 163% corresponds to a group size of eight and checkpointing interval of 1s.

We observed during our experiments that the group size slightly influenced the overheads in warm passive replication. In cold passive replication, the group size had almost no effect on the overheads. The variation in overhead for this replication style is mainly given by the different checkpointing intervals. For lower values of the checkpointing interval, the overhead was slightly higher, due to higher degree of interference between method calls and state checkpointing. Another interesting observation is that the overhead percentages are not much different between cold and warm. For active replication, on the other hand, the overheads are large, and highly affected by group size. The large variations present in Table 1 are due to variation over the number of replicas. The least value in the range corresponds to a group size of two replicas, while the largest to a group size of eight replicas.

The FA-CORBA overheads were also influenced by the group size, but also by the chosen pruning interval. Both parameters influence the average time taken for execution of the consensus primitive which is part of the overhead, in the following ways:

- The group size gives the number of nodes to which messages have to be sent, and from which (their majority) answers must be received, as part of the consensus execution.
• The pruning interval affects the time spent in execution of consensus, due to the pruning action on the registers. Thus, the shorter the pruning interval, the more often the time spent in consensus is larger than in a no pruning situation. So the average roundtrip time grows, and so does the overhead.

<table>
<thead>
<tr>
<th>Method</th>
<th>Non repl.</th>
<th>Cold passive</th>
<th>Warm passive</th>
<th>Active</th>
<th>FA-CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>130ms</td>
<td>55% - 117%</td>
<td>53%-134%</td>
<td>1745% - 5000%</td>
<td>141% - 472%</td>
</tr>
<tr>
<td>2</td>
<td>61ms</td>
<td>77% - 221%</td>
<td>110% - 285%</td>
<td>770% - 3360%</td>
<td>265% - 950%</td>
</tr>
<tr>
<td>3</td>
<td>65ms</td>
<td>62% - 163%</td>
<td>74% -240%</td>
<td>270% - 360%</td>
<td>213% - 861%</td>
</tr>
<tr>
<td>4</td>
<td>80ms</td>
<td>76% - 447%</td>
<td>100% - 550%</td>
<td>1790% - 5100%</td>
<td>308% - 1020%</td>
</tr>
<tr>
<td>5</td>
<td>133ms</td>
<td>44% - 333%</td>
<td>59% - 363%</td>
<td>905% - 4300%</td>
<td>223% - 726%</td>
</tr>
<tr>
<td>6</td>
<td>106ms</td>
<td>37% - 419%</td>
<td>68% - 383%</td>
<td>800% - 3100%</td>
<td>280% - 844%</td>
</tr>
</tbody>
</table>

Table 1 Summary of overhead percentages in the FT-CORBA and FA-CORBA platforms

(FT-CORBA figures after [23] © 2003 IEEE)

Figure 6 gives a bird’s eye view of the overhead percentages for the different methods. We have chosen to show the figures for Method 2 that are representative for other methods too. The chart represents the lower end (min %) of the overhead ranges described in Table 1 as well as the higher end (max %). It shows that:

• The overheads in the FT-CORBA platform for passive replication are lower than those in the FA platform.
• The overheads for active replication are much higher than the overheads in the FA platform.
• The overheads for cold passive replication style are slightly lower than those for warm passive.

The slices of the roundtrip times for the replicated scenarios can be further investigated to find out where the overhead is coming from. It turns out that:

• In the FT-CORBA passive styles, as well as in the FA-CORBA platform the main part of the overhead comes from the wait time when executing methods in a serialized way. This wait time is the sum of times \( t_r \) spent by all requests arrived before the one currently waiting, where \( t_r \) denotes the time from the arrival of the request in the server interceptor until the result of its processing is sent back to the client. In the FA-CORBA platform this wait time, an inherent feature of the algorithm, is even larger because individual \( t_r \)s are increased due to the execution of the consensus primitive (see Figure 7 for the values of the consensus time).
• In the case of active replication the average overhead values are generally large. Since the larger overheads appear only in case of method calls that themselves made outgoing calls, it was deduced that it was due to the gateway’s duplicate suppression mechanism. Method 3 did not call other methods, and thus experienced a lower overhead.

The chart in Figure 7 visualizes the variation with the group size of the average time spent performing the consensus for different methods. Here, absolute timing values were used to depict the results. The tendency is that the higher the number of replicas, the higher the time spent in consensus. A clear difference can be seen for eight replicas where the majority was five nodes, as compared with two nodes in case of two or three replicas, and three nodes in case of four replicas, respectively. Also, it can be noticed, that for a fixed group size, the time spent in consensus stayed fairly constant independently of the called method.
7.2. Failover times

As mentioned in the introduction, support by middleware requires writing little extra code by the application writer, so in that sense it is directly comparable to a non-replicated scenario. In this section we quantify the benefit to the application writer in terms of shorter failover times (compared to manual restarts). We summarize failover time performance in both platforms and relate to the parameters influencing it.

In the FT-CORBA platform there is a significant difference between failover times using passive or active replication. In active replication, the failover time was extremely short: 70 – 75 ms. In passive replication, when using the warm style, the faster failover was due to the fact that no state transfer was needed at the moment of the reconfiguration, as opposed to cold style. In both passive styles, the failover time was dependent on the number of method calls logged since the last checkpoint that had to be replayed on the backup. This dependence was stronger in the warm passive case.

In the FA-CORBA platform the failover time was dependent on the number of state changes that happened in the application object since the last witness register pruning. These state changes (updates) have to be applied at failover on the application object inside the new leader. Let us consider the operations associated with applying an update on the new leader as similar operations to replaying a method in the FT-CORBA platform. These operations consist of reading the update to be applied while performing the consensus primitive as in no failure cases, followed by applying the read update on the application object.

Figure 8 shows how the average time per method replay (FT-CORBA) or update application (FA-CORBA) depends on the group size in a scenario where the server failed after approximately 600 requests, and the checkpointing (respectively pruning) interval was 5s. The general tendency is that the replay time is the lowest in warm passive replication, followed by the cold passive replication. The highest average time per individual replay (update application) was experienced in the FA-CORBA platform. The explanations are as follows: first, cold passive recovery includes, besides the time for replaying the methods, the time to set the last checkpointed state on the new replica. Second, the average time for applying the stored state changes in the recovering leader (in FA-CORBA) is higher because each such operation implies execution of the consensus algorithm.

Figure 9 shows the general tendency of the average time to execute one update application in the FA-CORBA platform. This average time grows with the size of the replica group (due to the consensus execution). Here we show the experiments with pruning interval of 10s.
Table 2 presents examples of absolute failover times for different group sizes (3, 4 or 8) and checkpointing/pruning intervals (5 or 10 s), when the server failed after approximately 400 requests. All values are given in seconds.

<table>
<thead>
<tr>
<th></th>
<th>3 replicas</th>
<th>4 replicas</th>
<th>8 replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5s</td>
<td>10s</td>
<td>5s</td>
</tr>
<tr>
<td>Cold passive</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Warm passive</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Fully available</td>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2 Example failover times in different platforms and mechanisms (in s)

The above figures are selected from a series of experiments the results of which are summarised below. The scenarios were generated by forcing failures at different points in the request execution sequence. The measured failover times ranged over intervals of:

- In cold passive style, between 2 and 10 seconds
- In warm passive style, between 2 and 11 seconds
- In the fully available platform, between 1 and 16 seconds.

These values should be compared with the ~70 ms failover time of the active replication. Considering the lower overheads, the higher failover times may well be acceptable in a given application. The next question is whether to pay the larger overhead penalty in the FA-CORBA platform for obtaining the full availability. The answer could be yes, since there is no need to worry about extra failure prone infrastructure units.

8. Conclusions

This chapter presented experiences with building two different platforms supporting fault tolerance in CORBA applications. One of them (called FT-CORBA) deals only with failures of the application. The other (called FA-CORBA) uniformly handles failures of the application as well as the infrastructure, hence labelled fully available. Both platforms provide transparency for the application writer so that only very little change has to be applied in the application run on top of the platforms. The main purpose of the study was to find out whether the prospects of the support from middleware would be appealing to the application writer, considering factors such as performance overheads and gains in failover time. At the time of implementation, there were no ORBs available in full compliance with the CORBA standard or with fully available property. The performance related insights are obviously affected by the ORB implementation over which the platforms were built. However, we believe that the wisdom from relative comparison of the different replication styles carries over to other ORBs. A different ORB is expected to increase/decrease the overheads in all styles by similar amounts.

Our findings are supported by measurements on a telecom application using both platforms. Again, one must be careful about drawing general conclusions based on this one application. However, all the earlier published studies on the support for fault tolerance based on CORBA had used synthetic experiments for evaluation. We believe that this work is at least a step in the right direction. Given that the FT technology finds its way into new (commercial) ORBs, the work presented should guide the application engineer in methods for profiling new development environments.

Varying the number of replicas has generated conclusions that are likely to carry over to other middleware. The number of replicas made a significant difference only for active replication and the FA styles. For the FA-CORBA
platform, the group size made a difference in both overheads and failover times. This was due to the need for executing the consensus primitive in both scenarios. The corresponding notion for state transfer, pruning interval, also influenced the two metrics.

To sum up, providing fault tolerance embedded in middleware, and keeping transparency for application writers involves a price. This price is the extra delay experienced by a client while receiving a reply from a replicated server. The price becomes higher in an infrastructure in which failures of non-application units are handled together with those of application objects (employing a distributed algorithm).

The experience with building the FT-CORBA platform shows that it is feasible to implement the FT-CORBA standard by extending an existing ORB. On the other hand, with no other failure handling mechanisms, except the application object related ones, the infrastructure units have to run as processes that do not fail. Ad hoc replication might be employed to cope with the problem. Still, the rigid failure detection mechanism is inherent to this platform: timers are explicit and visible at a very high level.

In contrast, the FA-CORBA platform ensures that failures of infrastructure units do not impair the availability of the application. Besides the higher price paid in terms of overheads, there is another drawback in this approach: a majority of replicated units must be up at all times. This requires some initial hardware investment beyond a redundant pair. Note that if the group has two replicas, none of them may fail! In the FT-CORBA platform, no such restrictions exist. At the extreme (although not really desirable) the system could run with only one replica. Thus, e.g., seven out of eight replicas may fail, and the application will still not block, as opposed to the case in FA-CORBA.

The failover comparisons show that active replication in the FT-CORBA platform is unlikely to be worthy of attention due to its high overheads. In our experiments, the extremely short failover time (~70ms) can not be considered essential in a high-assurance system if the no-failure roundtrip time is around 6 seconds (after adding the overheads). Another limiting factor might be the need to ensure replica determinism. In such a competition, the FA-CORBA platform will win, since non-determinism of replicas is tolerated. In both cases, the available resources further influences the decision, since the number of replicas in the group gives the maximum number of failures tolerated.

When looking at overheads, the two passive replication styles in FT-CORBA score well: no need for determinism, and failover is in range of seconds. Warm passive replication does not exhibit larger overheads than the cold passive style, but the failover time is lower in this case, especially if the check-pointing interval is low.

The bottom line is that the quest for higher availability lies in the likelihood of failure in the infrastructure units. As long as the costs for hardening such units is not prohibitive, the FT-CORBA platform may be considered as adequate since the overheads are smaller and the failover times have comparable values.

Acknowledgements

This work is supported by the European Commission IST initiative and is included in the cluster of projects EUTIST-AMI on Agents and Middleware Technologies applied in real industrial environments (www.eutist-ami.org). It was also supported by the European project Safeguard (IST-2001-32685). The FA-CORBA platform was implemented by Isabelle Ravot on a visit at Linköping university from Lausanne. The authors wish to thank Prof. Rachid Guerraoui for valuable discussions during preparations of a poster presentation for Middleware'03 on an earlier version of the FA-CORBA platform that has influenced the presentation in this chapter. Also, our thanks go to Torbjörn Örtengren and Johan Moe from Ericsson Radio Systems for priceless cooperation.

References


