Building and Evaluating a Fault-Tolerant CORBA Infrastructure

Diana Szentiványi and Simin Nadjm-Tehrani
Dept. of Computer and Information Science
Linköping University, Sweden
[diasz,simin]@ida.liu.se

Abstract

In this paper[*] *we explore the trade-offs involved in making one major middleware fault-tolerant. Our work builds on the FT-CORBA specification (April 2000), which is not in widespread use due to lack of quantified knowledge about design rules and trade-offs. Recent results show preliminary overhead and trade-off analysis for a not entirely FT-CORBA-compliant system for embedded applications (Eternal).*

In distinction from Eternal, we have extended an existing open source ORB and combine it with a collection of service objects and portable request interceptors following the FT-CORBA standard. The paper reports on extensive studies relating the timing aspects to different parameters such as the replication style, the number of replicas, and the checkpointing frequency. The experiments were conducted using a realistic telecom application.

1. Introduction

Future data/telecom applications are built on top of object-oriented distributed system platforms that need high levels of dependability. Therefore these industries stress the importance of enforcement of system-level properties such as fault tolerance, timeliness, and security in such platforms. Nevertheless, the ambition is to keep the application writer efforts minimal when adding these features to a certain application.

While there could be more elegant approaches for supporting fault tolerance in runtime systems of a specific language like Java, Ada or Erlang, real systems are typically multi-language, multi-platform. Therefore, building fault-tolerance in a generic middleware is an interesting undertaking. CORBA is one of these types of middleware, and one of the platforms used by our industrial partners at Ericsson Radio Systems.

Although there has been a lot of research work about extensions of CORBA towards fault tolerance, a telecom engineer who so far built fault tolerance as part of the application, needs more information about what are the consequences of using a generic *fault-tolerant* middleware.

Obviously there will be costs associated with support for fault tolerance in any infrastructure. Our goal is to provide indications about the performance trade-offs and the added robustness as a result of the proposed enhancement. Also, the engineer could be interested in whether the old legacy code dealing with fault tolerance inside the application is reusable in the new setting. This work will answer some of these questions in the context of fault-tolerant CORBA, using a realistic application.

Following the FT-CORBA standard specification [1], we have built our infrastructure by combining:

- a service approach seen by the application writer
- an extension to an Object Request Broker (ORB)
- CORBA portable interceptors [2] for requests.

We have tested the infrastructure in a prototype setting to obtain some of the above-mentioned answers. Although we tried to follow the standard as much as possible, where the specification is not complete we had to make our own decisions. Our experiments provide a deeper understanding of the decisions already made in the specification and their consequences. The implementation also provides an insight in how to modify a standard ORB for supporting fault tolerance.

To start with, we have tested this infrastructure with an artificially created server application. This experiment gave an indication of the overhead in the absence of failures. The measured overhead covered the cases of cold passive, warm passive and active replication. Next, a generic service from the telecom domain, provided by Ericsson Radio Systems was used as a test-bed. There were few changes in the application such as the addition of methods for getting and setting the state of the object. This paper explains the results of these studies so far.

The paper is structured as follows. Section 2 presents some background information like some aspects of the FT-CORBA standard. Section 3 will describe the main parts of our infrastructure and some design decisions made. Section 4 presents quantitative evaluation results and discusses the insights gained. Section 5 concludes the paper.

2. Background

In December 2001 the FT-CORBA specification has been added to the latest CORBA standard (version 2.6). Its contents have not changed in a major way since the original introduction in April 2000. In what follows we give an overview of the main components of the specified infrastructure.

This standard was drawn up against the background of several years of work in the fault tolerance research community. In section 2.2 we refer to some of the works which are closely related to the proposals of the standard, and therefore to our work.

2.1. FT-CORBA

To obtain fault-tolerant applications the FT-CORBA approach uses replication in space (replicated objects). Temporal replication is supported by request retry, or transparent redirection of a request to another server. Replicated application objects are monitored in order to detect failures. The recovery in case of failures is done depending on the replication strategy used. Support is provided for use of active replication, primary/backup replication, or some variations of these. The non-active replication styles provided are warm passive, and cold passive (primary/backup). The choice of policy is left to the application writer when initiating the server.

Figure 1 shows the architecture for fault tolerance support according to the standard.

There is a Replication Manager that implements interfaces such as Property Manager, Object Group Manager, Generic Factory. The standard also informally refers to the notion of Fault-Tolerance Infrastructure. This is implicitly the collection of mechanisms added to the basic CORBA to achieve fault-tolerance. The Property Manager interface has methods used to set the above-mentioned policy i.e. the replication style, the number of replicas, the consistency style (infrastructure-controlled or application-controlled), the group membership style (infrastructure-controlled or application-controlled). The Object Group Manager interface has methods that can be invoked to support the application-controlled membership style, at the price of losing transparency. The Generic Factory interface has the create_object and delete_object methods. The Replication Manager's create_object method is invoked when a new object group has to be created. As a consequence, the Object Factories' create_object methods are called. Each object in a group has its own reference, but the published one is the inter-operable object group reference.

Application replicas are monitored by Fault Monitors by means of calling the is_alive method on them. Thus, the FT-CORBA specification mostly focuses on the pull monitoring style. Push monitoring style is also mentioned but not specified in the standard. The fault monitors, as mentioned by the standard, are unreliable (they cannot decide whether an object crashed or it is just slow). Monitoring is initiated by the Replication Manager. Fault Monitors are given indications about the fault monitoring granularity, such as member level, type identifier level or host level.

The Replication Manager is involved in the recovery process of the object group. For this, it needs to register as a consumer at the Fault Notifier. The Fault Monitors announce faults to the Fault Notifier that further announces it towards its consumers.

For checkpointing purposes, application replicas must implement the Checkpointable interface and provide two methods in the application class. These are named get_state and set_state. Checkpointing has to be used in cold/warm passive replication. The standard also requires logging of method calls and replies in those cases.

When using the active replication style the specification strongly recommends the use of a gateway for accessing the group members. There is also an indication about an alternative, namely the usage of proprietary broadcast primitives at the client, and thus, direct access to the group.

To be able to manage large applications, the notion of fault tolerance domain is introduced. Each fault tolerance domain contains several hosts and object groups. There is one Replication Manager associated with it, as well as one Fault Notifier. Object Factories and Fault Monitors are specified as separate entities (objects) running on every host within the FT domain.

2.2. Related work

There have been attempts to build infrastructures to provide application writers with the possibility to construct fault-tolerant application in an easy way.

The Java RMI technology is, for example, used to provide fault-tolerant distributed services in the Jgroup toolkit [4]. Clients interact with these services by an external group method invocation. Group members cooperate via internal group method invocations. This cooperation is achieved by using a group membership service, as well as a reliable communication service.

The Horus toolkit [5] provides a platform independent framework for building fault-tolerant applications. It offers application writers a set of building blocks (such as system and application protocols), to choose from in order to fit system requirements. With Horus the virtual synchrony runtime model is well supported.

Prior to the specification of the FT-CORBA extension (April 2000), few works have studied alternative augmentations of CORBA with a process (object) group module.

Little et.al. present a way of integrating a group communication service with transactions [6]. They start with a system that supports transactions (CORBA), but no process groups. Then, they consider enhancing the use of transactions by introducing process groups.

Felber et.al. show some possible approaches for introducing object groups in CORBA [7]. Three different approaches are presented depending on the position of the group communication module relative to the ORB. These are: the interception approach, the integration approach, and the service approach.

Narasimhan et al. implemented operating system level interceptors to provide fault tolerance to CORBA [8]. The result of their 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.

Chung et. al. present a fault-tolerant infrastructure (DOORS) built using the service approach, on top of CORBA [9]. In this setup, 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 with the use of aspect oriented programming is presented by Polze et al. [10]. 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. Communication between a client and a replicated service is done via an interface object.

3. Our generic infrastructure

We have chosen to use a *Java based* open source ORB, OpenORB, as a basis for our enhanced infrastructure [3]. Figure 1 depicts the architecture of our implemented environment in adherence to the standard.

To be able to build a fault tolerant application using our infrastructure, the application writer needs the following ingredients:

- the collection of service objects (Replication Manager, Object Factories, Fault Monitors, Fault Notifier, and Logging & Recovery Controllers)
- the basic (extended) ORB classes
- the proper Portable Request Interceptors for different replication styles.

These constituents of our infrastructure are now explained in turn.

3.1. Service Objects

The Replication Manager, as specified in the standard, contributes to object group creation and recovery. It was implemented according to the specification.

The Object Factory runs on every host in the fault tolerance domain, and has the role to create a replica for an object group. The standard does not specify exactly how or where the created CORBA object replicas will run. In our implementation every replica runs in a different Java Virtual Machine (a new Unix process).

The Fault Monitor, as specified in the standard, runs on every host and monitors replicas. Besides that, also as a result of the specification, our implementation allows the monitoring at different granularities. Due to the usage of Java language, the most straightforward implementation of the monitor is as a CORBA object with a collection of internal threads. Also, a characteristic of our fault monitors is that they are forced to be reliable, by simply killing an object as soon as it is suspected to have failed.

The Fault Notifier is also implemented conforming to the specification. It receives the fault event from the monitor and it sends it further to the consumers (e.g. Replication Manager).

The FT-CORBA standard does not mention how or where the logging and checkpointing has to be done.

In our implementation, the Logging & Recovery Controller is also a service (CORBA) object. It performs the checkpointing of objects on its host by using a separate internal thread, for every object. The state is recorded in a log located in memory (the location of the log could be easily changed to be on stable storage). Also, the method calls and replies are recorded in the same log, from where they are retrieved later at recovery.

The idea is to have the logging and recovery objects on every host. The usage of these objects is dependent on the replication policy used. In case of cold passive replication the Logging & Recovery Controller at the location of a failed group member is contacted to obtain the latest state and set of method calls. In case of warm passive replication, the logged items are taken from the Logging & Recovery Controller on the new primary's host.

Checkpointing decisions

As mentioned in the specification, checkpointing is done only on objects whose state can change while in service, and in passive replication scenarios. It is obvious that the checkpointing has to be synchronized with execution of update methods on those objects. In this context some interesting questions arise: Must checkpointing be done periodically? In case of periodic checkpointing, how long should be the interval between two checkpoints? It is quite straightforward to see why these issues are important: the recording frequency affects

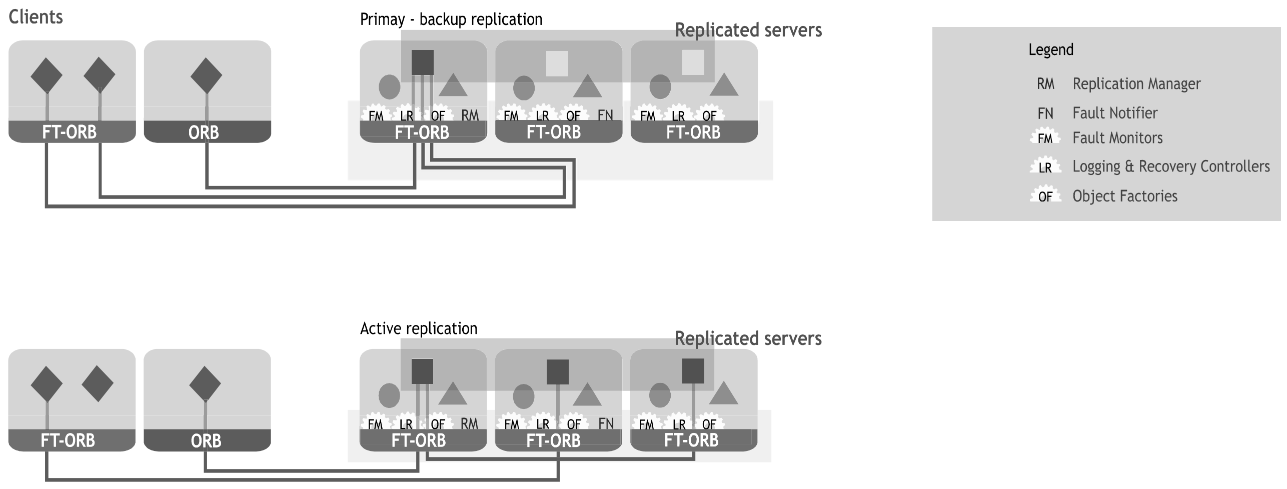

Figure 1: The architecture of our implemented environment

the average time for recovery upon failures and it influences the perceived (average) reaction time of the server as seen by the client. In the next section we will come back to these questions in the context of our experiments**.**

Gateway for active replication

In case of active replication, we chose the gateway approach. The gateway is set up as a separate CORBA Object. One of the gateway's roles is to broadcast method calls to the group members. Besides this, it will be used as a duplicate suppressor for the active object group's outgoing method calls. The gateway is a dynamic part of the infrastructure, as opposed to all aforementioned components.

3.2. Extensions of the ORB

We found that some of the basic ORB classes had to be extended. For example, in case of primary backup replication, the method for choosing the target address for the request had to be changed to be able to find the primary's address (as opposed to the standard non-FT-ORB which had no notion of primary).

The standard recommends the usage of a retention identifier and a client identifier for a certain request, to uniquely identify it. To generate such unique identifiers, the ORB class itself had to be modified. Another extension to the ORB relates to the use of portable request interceptors that will be described below.

3.3. Portable Request Interceptors

The retention identifier and client identifier are sent in a request service context. A group reference version service context is also recommended by the standard. To add these service contexts, at the client side, a portable *client* request interceptor is used. This applies both when the client is just a simple one, and when it is itself replicated. The interceptor is also used, in the latter case, for recording replies to outgoing requests. This avoids unnecessary resending requests if resends are attempted.

Portable request interceptors have an important role at the server side as well. Here, we will talk about *server* request interceptors. Their main function is to contact the Logging & Recovery Controller for method call and reply logging purposes. For different server replication styles, different server interceptors are used. For example, in case of stateless replication style, there is no need for logging, at least not for logging of method calls. In case of warm passive replication style there is need for broadcasting of method calls to the rest of the Logging & Recovery Controllers for the group.

Any changes to the group will cause a group reference version update. Thus, when serving a request one has to verify whether the client has the right version of the object group reference or not. This is also done in the server side interceptor.

Sometimes, a request has to be stopped at the level of the server portable interceptor and the answer sent from that level. For this purpose, a special exception had to be introduced as well as its handling. This was a further extension to the ORB. In addition, a mechanism for sending the reply from within the interceptor had to be devised.

4. Evaluation

In this section we present the experimental studies of the implemented infrastructure and analyse the results obtained.

4.1 Goals

The goal of our work is to obtain a deeper understanding of how the addition of fault tolerance to an existing middleware affects the design of new applications and the behaviour of existing ones. The application writer is primarily interested in overheads incurred due to the usage of a new infrastructure. There is also interest in the effects of the new mechanisms on the size of the code and its maintainability, as well as the application's performance in fail-over scenarios. Thus, the goal of our experiments has been to shed a light on our infrastructure's behaviour in this context.

A second goal is to evaluate the proposals made in the FT-CORBA specification in general, and get an insight into the common weaknesses and strengths of any infrastructure built upon this standard.

4.2 Experiment setup

Earlier works in the area have typically provided experimental results for artificially created applications and test cases (e.g. [7]). Such experiments are easy to set up. Also, isolation of infrastructure properties from the application-induced ones is simplified in this case. We have used such artificial servers for initial tests; but in addition, we have tested a real application provided by our industrial partners. This application was a generic service in the operations and management (O&M) part of the radio networks together with artificial client side test cases.

In the first approach, the client calls the four operations of a replicated artificial server in a sequence, inside a loop. The timing information recorded here is thus summed up and averaged over 200 runs of each method call using this loop. In the second, more complex O&M application, there were six operations within the replicated server subjected o our tests. The server is a so-called *Activity Manager* with the role to create application-related activities and jobs, and schedule them at later times. To do this, it also keeps track of the activities' progress and termination. Operations were called using a test case provided resembling a client application by Ericsson.

To average out the effects of network load and other uncontrollable elements these tests were also performed in loops with varying number of iterations (100, 200, 400 or 800). Although more detailed investigations in this direction would be interesting, for the time being, we think that simple averaging is adequate.

The Activity Manager is not stateless. Neither is the artificial server. So the replication styles tried out were cold and warm passive, as well as active replication for both cases. In all three cases, of course, the client was kept unchanged.

Other parameters in our experiment setup were number of replicas (ranging over 3, 4, or 5 replicas) and the length of the checkpointing interval (ranging over 1s, 5s and 10s).

To estimate overheads, roundtrip times of requests were measured (from recording the request at the client side, transport over the network and CORBA layers, to recording on the server side, performing the operation, checkpointing if needed, as well as sending the result over the network and receiving the response at client side). Then comparisons were made with the same roundtrip delay in the non-replicated case. Failover times were also measured and compared with a reference delay. The reference is given by the time taken when a (non replicated) server crashes and has to be restarted (possibly manually).

The probes were placed in the client, as well as server interceptor methods that are most probably called at the client and similarly at the server side (e.g. send_request, receive_request, send_reply, receive_reply).

The probe effect resulting from the instrumentation of the recordings was reduced by adding interceptors to the non-replicated runs in exactly the same places as above (even if interceptors were not needed for running the non-replicated scenario, and were simply put for measurement purposes).

4.3 Expected results

This section describes our intuitions before running the experimental studies. These will be followed up by the real measurements in the following section.

The number of replicas was expected not to affect overhead or failover times in case of cold passive replication. Of course, if the number of replicas drops under a certain minimum, new replicas have to be created and this can increase the failover time. In case of warm passive replication the expected result would have been that the roundtrip time was slightly influenced by the number of replicas, because of the extra time spent in broadcasting method calls to the other replicas' logging & recovery controllers. But, in fact, the broadcasting was implemented by use of CORBA one-way method calls. Hence, the overhead should not increase dramatically. Similarly, in case of active replication, the number of replicas should not influence the roundtrip time, since the gateway returns the first (fastest) result to the client.

The rate of checkpointing (inverse of the checkpointing interval) can influence the roundtrip time for update operations. This is only relevant for cold and warm passive replication. As mentioned earlier, it is only in the passive style that the call to the checkpointing operation, get_state, is made.

4.4 Results

The measured overheads for the artificial application were seemingly high. For example, for one method call the average over 200 iterations, when the server group size was 3, the checkpointing interval was 5s, and the cold passive style was used, we had 46ms of roundtrip time compared to 24ms for the non-replicated case (92% overhead).

Although this seemed very high at the first glance, we realised that it might not be representative, since a trivial application with low response time will inevitably show a large overhead percentage. Thus, the telecom application was of real interest. In the realistic application we observed considerably lower percentages in general (over all the experiments). For example, for one method call we had in the non-replicated setup a roundtrip time of 106ms, compared to the cold passive with 3 replicas and checkpointing interval at 5s, where the roundtrip time was 156ms (47% overhead).

The measured roundtrip times for the real application showed a clear dominance of the logging component for 5 method calls (the sixth method had the computation time as the dominant component). Actually the logging component of the roundtrip time is captured at a point where all incoming requests are synchronised on the server. So, it is not the very logging method call that takes the time. The component includes the wait time, i.e. the time that the calling (client) method has to wait until the currently running (update or get_state) method on the server finishes, together with all the other (update and get_state) method calls that arrived earlier. So, in fact it is not the logging itself which is taking time, but the synchronisation that we cause in order to be able to recreate the scenario when a new primary has to take over after a crash. For the above scenario the wait component was in fact 74ms of the 156ms (47%). Note that this wait time alone is almost four times the measured overhead for the artificial application.

We now summarise the observed influence of the chosen parameters on the measured overheads.

In the cases of cold and warm passive replication, overheads were pretty much the same, and independent of the number of replicas, as expected. On the other hand, the checkpointing interval was observed to slightly influence the wait time for the update requests and so the roundtrip time. Figure 2 shows the % of wait time out of the round trip time (y-axis) for different experiments performed on one method call, using cold passive style (averages were done over 100 calls, 200 calls, etc). The bars depict the variation relative to the choice of checkpointing interval.

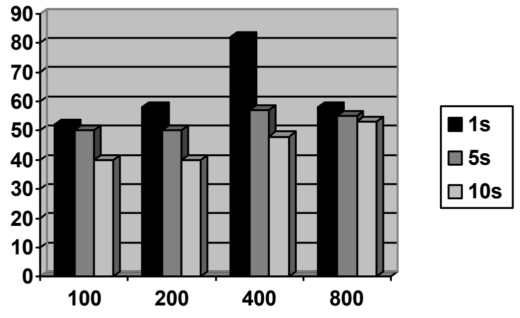

Figure 2: Wait time vs. checkpointing interval

The figure shows a typical chart for experiments on many methods where other parameters were varied. It illustrates that wait time increases as checkpointing frenquency is raised.

The quantitative values for overheads are listed as percentages in the table below. Every row in the table summarises the averaged results over the repeated runs of the method call according to the test loop of 100 iterations (100, 100, 100, 400, 100 and 200 respectively). They also summarize averaged results obtained in the 9 different scenarios (3 dimensions for the replica group and 3 values for the checkpointing interval). For example, 55% - 79% in the first row indicates the collective experience with all the cold passive runs we have had with method 1, over 100 runs. 55% is the lowest overhead % among the 9 scenarios and 79% is the highest overhead %. The first column describes the roundtrip times (ms) for the non-replicated case, and the two following columns differentiate cold and warm passive styles.

	n.r.	Cold passive	Warm passive
Method 1	130	**55% - 79%**	**53% - 66%**
Method 2	61	**77% - 128%**	**110% - 134%**
Method 3	65	**62% - 92%**	**74% - 106%**
Method 4	80	**76% - 168%**	**100% - 151%**
Method 5	133	**44% - 111%**	**59% - 93%**
Method 6	106	**37% - 98%**	**68% - 94%**

Table 1: Summary of primary backup experiments

In case of active replication, for the real application, the situation was somewhat different from the case of the artificial one. In the made-up application there are no outgoing calls from the replicated server. Thus, there is no overhead associated with the duplicate suppression mechanism of the gateway.

The overheads for the made-up application in this case are somewhat dependent on the number of replicas, and remain within acceptable ranges. For example, we had a roundtrip time of 66ms as compared with 24ms for the 3-replica group mentioned earlier in this section.

In case of the real application, on the other hand, we noticed that the roundtrip time is considerably larger than the non-replicated case, and this is due to the gateway's duplicate suppression mechanism. The fact that servers acting as replicated clients have a higher roundtrip time can be illustrated by the following comparison. The roundtrip times for method 3 (averaged over 100 calls) using active replication ranged between 177ms and 325ms (4 replicas and 5 replicas respectively). This method call does not in turn call other methods (acting as a client) and has a reasonable overhead compared to its non-replicated time (65ms). In contrast, method 4 exhibits a roundtrip time in the range 1698ms and 4678ms (for 3 respectively 4 replicas), which is at least 20 times its non-replicated time!

The down side of long overhead when actively replicating a server is compensated by the faster recovery times. For example the fail-over time for the actively replicated activity manager is in the range of 62ms to 73ms. This should be compared to a manual recovery process that is in the order of minutes, and to the passively replicated group recovery times that are in the range of 1.7s to 18.7s. The time for recovery depends on the checkpointing interval that affects how many earlier calls have to be replayed, and on the size of the state which has to be set on the new primary in case of cold passive replication. The recorded recover times tend to be smaller in case of warm passive replication. Figure 3 below shows how the size of the state to be set in the new primary influences the time for set_state. The picture shows that, the time spent with set_state is not influenced only by the size of the state.

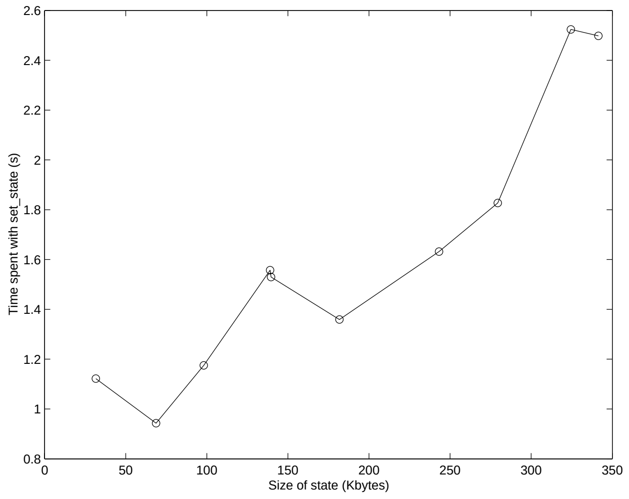

Figure 3: set_state time vs. state size

4.5 Lessons learnt

Average roundtrip time for update method calls is highly influenced by the nature of the client side code. That is, the sequence and placing in time of the method calls affects the wait time mentioned earlier. The wait time for an update method call is built up by execution times of update operations that arrived just before that method call and are executing or queuing on the server.

In our present implementation, because of the way checkpointing of state is done, there is no support for handling of application created threads that can modify the object's state by calling methods directly on the servant. Such modifications are not reflected in the logged method calls, and thus can not be replayed once there is a crash. Therefore, using such design patterns is not advisable in sensitive applications.

Another aspect of interest is the synchronization of update method executions. In the present implementation the granularity of this synchronization is at method level. For this, of course, the server ORB has to know what type a certain method has (is it an update method, or a reader method). Obviously, the granularity of this part of the synchronization, also affects the roundtrip time of a method call. To reduce the round trip time one has to come up with a way to capture application writers' knowledge with respect to mutual exclusive accesses. This is a potential future optimisation case.

4.6 Why use an FT-CORBA infrastructure?

Building and testing our infrastructure has facilitated evaluating the concept of FT-CORBA infrastructure in itself.

On the one hand it is attractive to use a service oriented middleware (CORBA) extension to build fault-tolerant applications. The only thing required from the application writer is the starting the service objects, deploying on a set of hosts, and set the replication policy. Building the infrastructure has shown us the feasibility of extending an existing ORB to support fault tolerance the way the FT-CORBA standard devises it.

On the other hand, building a fault-tolerant infrastructure following the CORBA standard implies using more resources and introducing new single points of failure. For example, the fault monitors, though not fully specified in the standard, are implemented as separate entities (CORBA Objects) as the most straightforward solution. Also, monitoring of an object being done via a CORBA call is somehow waste of resources.

The most critical point is perhaps that all service building blocks of the infrastructure constitute single points of failure. Thus, we can obtain transparency and modularization, but we have to deal with infrastructure vulnerabilities. In earlier works there are indications about using the infrastructure itself to replicate CORBA objects that are part of the infrastructure [11]. However, it is not clear how the approach deals with the vulnerability. Extra resources are needed but single points of failure are still not removed.

In case of infrastructure components it might be easier to use more efficient replication strategies that do not provide transparency, a property that is important for application object servers, but not for an infrastructure.

5. Conclusion

In this paper we have given an account of our experience in building and evaluating a fault-tolerant middleware based on the FT-CORBA specification. We presented results from extensive tests on application scenarios, using a dummy application as well as a multi-tier activity manager from a real telecom setting. The quantitative results show trade-offs between various design parameters: number of replicas, replication style, checkpointing frequency, and nature of application call patterns.

There are several aspects that can be improved with respect to the performance of our infrastructure. Some of the ideas for improvement are optimisation points specific to our design choices. Others are guidelines to application writers to lead to desired effects in presence of certain policies. Nevertheless, there are weaknesses that are inherent to the FT-CORBA approach.

An interesting point of study is the potential for incorporation of unreliable failure detectors in this infrastructure, and a more clever approach for dealing with retracted suspicions. Also, the work confirms that there is a conflict between compliance to FT-CORBA specification and the wish for distributed agreement on group membership. Hence, the combination of robust algorithms (that do not include managers with single point of failure) is still a viable question to study in the context of CORBA.

Acknowledgments

The authors would like to thank Calin Curescu for preparing the application, and Johan Moe for useful CORBA related discussions.

References

[1] Object Management Group, "Fault-Tolerant CORBA Specification V1.0" available as ftp.omg.org/pub/docs/ptc/00-04-04.pdf

[2] Object Management Group, "The Common Object Request Broker: Achitecture and Specification" - Portable Interceptors

[3] Exolab OpenORB webpage http://www.openorb.com

[4] Alberto Montresor, "Jgroup Tutorial and Programmer's Manual", *Technical Report 2000-13 available at ftp.cs.unibo.it/pub/TR/UBLCS*

[5] Robert van Renesse, Kenneth P. Birman, and Silvano Maffeis, "Horus: A Flexible Group Communication System", *Communication of the ACM*, Vol. 39, Nr. 4, pp. 76-83, April 1996

[6] Mark C. Little and Santosh K. Shrivastava, "Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects", *Distributed Systems*, *Lecture Notes in Computer Science 1752*, *Springer Verlag*, pp.238-253, 2000

[7] Pascal Felber, Rachid Guerraoui, and Andre Schiper, "Replication of CORBA Objects", *Distributed Systems*, *Lecture Notes in Computer Science 1752*, *Springer Verlag*, pp.254-276, 2000

[8] Priya Narasimhan, Louise E. Moser, and P. Michael Melliar-Smith, "Using Interceptors to Enhance CORBA", *IEEE Computer*, pp.62-68, July 1999

[9] P. Emerald Chung, Yennun Hung, Shalini Yajnik, Deron Liang, and Joanne Shih, "DOORS: Providing Fault Tolerance for CORBA Applications", *Poster session at IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, 1998*

[10] Andreas Polze, Janek Schwarz and Miroslaw Malek, "Automatic Generation of Fault-Tolerant CORBA-Services", *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS),* Santa Barbara, August 2000, pp.205-215, IEEE Computer Society Press, 2000

[11] Priya Narasimhan, Louise E. Moser, and P. Michael Melliar-Smith, "State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects", *In Proceedings of the 2001 International Conference on Dependable Systems and Networks*, Göteborg, Sweden, pp.261-270, 2001