Managing Fault Tolerance Transparently using CORBA Services

René Meier and Paddy Nixon
Dept. of Computer Science, Trinity College, Dublin 2, Ireland
Phone: +353 1-608-1543, Fax: +353 1-677-2204
Email: Rene.Meier@cs.tcd.ie

Abstract. Fault tolerance problems arise in large scale distributed systems because application components may eventually fail due to hardware problems, operator mistakes or design faults. Fault tolerance mechanisms must be employed to reduce the susceptibility of a given system to failure. In this paper, we describe the design of an architecture to overcome potential application component failures, using *CORBA*, a distributed object middleware specified by the *OMG*. Of primary importance to this architecture is *OMG's CORBA Object Trading Service* as the mechanism to advertise and manage service offers for fault tolerant application components. This mechanism enables clients transparently to detect a failed connection to a service object, to discover a similar backup service object and to re-connect to it. This improves overall system stability and enables scalability.

1 Introduction

Application component failures in large scale distributed systems [1][9] are inevitable. *Fault tolerance problems* [7], associated with the use of large scale distributed systems, arise because application components may eventually fail. These failures are caused by hardware problems, operator mistakes or software faults [3]. Within most environments, and in particular within a banking environment, such failures are not acceptable. In our problem domain we are interested in fault tolerant service provision. By fault tolerant service we mean:
A service in a distributed system is called fault tolerant when it behaves according to its specification even in the presents of failures in parts (processor, media, communication link, other service) of the distributed system on which it depends [9].
The author [8] identifies four possible causes of application component failures in our chosen banking environment. All of them can be overcome by providing a fault tolerance mechanism that re-connects a client from a failed service to a similar backup service. This paper introduces such a mechanism, which is described more completely in [8].

1.1 Fault Tolerant Application Components

This paper describes an architecture to make application components in a large corporate banking system, which is described more completely in Section 2, fault tolerant. The banking system is based on a three-tier client-server architecture. It is implemented using CORBA [2][6][10], a distributed object middleware specified by the OMG [10]. It includes several databases, which are kept consistent using a data replication protocol. Servers and their services (service objects) are managed by a so-called Service Manager, which, together with the appropriate hardware, guarantees them to be *fail silent*.

The banking system already provides low level fault tolerance mechanisms. However, it lacks a mechanism that re-connects clients from unavailable master services to backup services. Master services may not be available because of failures or for maintenance reasons. Currently, banking system clients connected to an unavailable master service are not able to retrieve data requested by a user, despite the presence of other similar services that could provide the requested data. The basic assumption is that re-connection delays and even a lower access performance are acceptable, but a total loss of a service is unacceptable.

1.2 Some Requirements of Fault Tolerant Application Components

We propose an architecture to support the development of fault tolerant distributed application components within the banking system. Central to the architecture is the use of the OMG Object Trading Service [10] as the mechanism to advertise and manage service offers of fault tolerant application components.
It is essential that the suggested architecture fits into the existing banking system. Fault tolerant issues must be *hidden* from the client application program and therefore from the user and be Object Request Broker (ORB) [10] *independent*. The suggested architecture must support *on-line* application component management by configuration adjustment, without re-booting the rest of the system. It should provide a highly available system with a good trade off between *system scalability* and *service performance* without resorting to purchasing expensive fault tolerant hardware.

2 Object Trading and a Banking Environment

In this Section, we first introduce OMG's CORBA Object Trading Service [10], which is included in the suggested architecture as the mechanism to advertise and manage service offers for fault tolerant application components. We then present an existing large corporate banking system, into which the suggested fault tolerance architecture must fit.

2.1 OMG Object Trading Service

There are several possible approaches for service consumers (clients) to retrieve a reference of a service (service object) provided by a server, that may be located somewhere in a distributed system. Such a reference can be available on the client side, e.g. looked up in a table or read from a file or, in a more dynamic approach, can be retrieved from a naming service such as the OMG CORBA Naming Service [10]. In these approaches, in order to retrieve a service object reference, clients need to know the exact name of the desired service object a priori.
The OMG Object Trading Service lets clients dynamically discover service objects based on the type of service they provide. It is like yellow pages for service objects in a distributed system. A server advertises its service objects with a Trading Service. Clients use the Trading Service to discover service objects that match their needs [11]. This is shown in figure 1 and achieved as follows:

1. A server registers (*exports*) its service objects with the Trading Service. By doing so, a server gives all the relevant information about its service objects to the Trading Service. This service offer includes the service object reference, the service object name and the service object properties. Clients use the reference to connect the service object and to invoke on its operation. The name includes the operations to which the service object will respond and their parameters and result types. The properties are name value pairs, which describe the capability of the service object. The Trading Service maintains all the service object information in a repository.
2. A client makes a request (*imports*) for a specific service object type. Based on the properties of the service objects, the Trading Service performs a matching algorithm to return the result to the client.
3. Based on the Trading Services information, it is now the client's responsibility to decide whether or not to *invoke* a service object on a server.

A Trading Service may use an inter-trader protocol to extend its search to other Trading Services. Such a schema is called *federated trading*. We do not consider enhanced trading schemas such as federation in this paper, although the architecture can be extended to use federation easily.

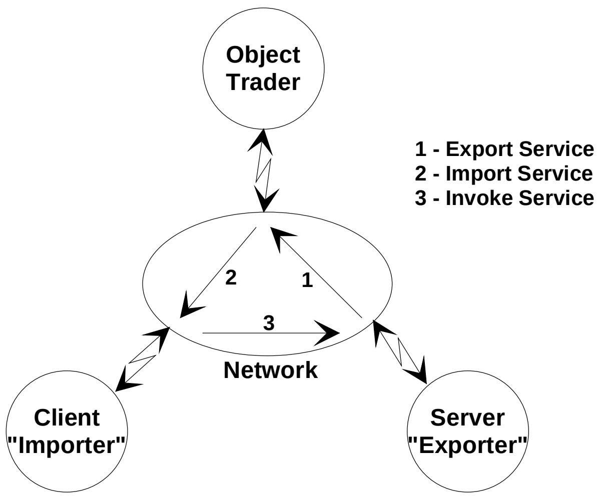


Fig. 1. The OMG Object Trading Service.

Another reason for using an object Trading Service is system stability and scalability. Distributed systems consist of a large number of bindings between clients and servers. Since these bindings are not reliable, re-discovering and re-connecting at run time helps in overall system scalability and stability since the bindings can always be reorganised.

2.2 A Banking Environment

The proposed architecture was designed to manage fault tolerant application components of a particular large corporate banking environment but can easily be applied for any client-server based architecture.

2.2.1 Environment Terminology

The environment consists of:

• Client:	A client is a program, operated by a user, for which a server performs some computation.
• Server:	A server is a physical machine with several processors.
• Service:	A service (service object) is a software application component running on a server.
• Location:	A location is a collection of services running across one or more servers, providing functionality to a community of clients (users).

2.2.2 The Target Banking System

The target banking system is based on a three-tier service based architecture. CORBA is used for the implementation of the communication link between the client-tier and the middle-tier (service-tier).
Clients (users) are grouped together in so-called locations accessing service objects in the middle-tier of the banking system, that may be located on one or more servers. Service objects are logically associated with locations. A server may provide service objects for one or more locations. The servers access a distributed database (database-tier) that is kept consistent using a data replication protocol. This configuration of the system is very flexible. It supports a high performance configuration where few users exclusively access service objects on a server that may be the only one accessing a particular powerful database as well as a configuration where a hundred users spread across several locations share the service objects on a server.
The client-tier and the middle-tier are of particular concern. Within the middle-tier, the features of the Service Manager and the Notification Service will be used in the design.

- The *Service Manager* maintains the status of each of the service objects, using pings, a request that asks if the service is still running. It will shutdown and / or try to restart service objects that failed. The status of each of the service objects will be used by the Notification Service to publish service object status notifications. The Service Manager, supported by the appropriate hardware, guarantees the service objects to be fail silent.
- The *Notification Service* publishes several different types of notifications. A particular notification type is published in the event of a service object status change. To receive notifications, clients register with the notification type they are interested in.

3 Managing Fault Tolerance

To solve the fault tolerance problem identified in Section 1, an OMG Object Trading Service is introduced to the banking system. The included component must have appropriate service types, including a sequence of property structures that describe services, to export and import service offers. Property structures include client group identifier lists. Clients with similar needs and location are grouped together and are given a unique group identifier. Servers offer their services to a particular client group by including client group identifiers in a service offer. Clients then query the Trading Service for service offers which include their group identifier.
The basic architecture is a *simple low cost solution* that solves the fault tolerance problem without involving the Notification Service. Clients query the Trading Service for master and backup service offers and cache the retrieved service references. If the service in use

fails, the service user (client) is re-connected to the backup service. During re-connection, the service user is idle. Then, the client starts pinging the original service and re-connects the service user to it as soon as it is back on-line. Pinging is inefficient and causes unnecessary network traffic. To eliminate this, an improvement is proposed, which makes use of the Notification Service. Within the improvement, clients use notifications to detect service failures and therefore need not to ping services anymore. This results in less network traffic and reduced service user idle time. Notifications are also used to automate service offer maintenance and therefore further minimise network traffic. In addition, the trading service's dynamic properties are used to provide load balancing.

3.1 Service Offers

Servers cannot export service offers unless the Trading Service has appropriate service types. The Trading Service can contain a number of such service types that describe services. Service types consist of a type name, an IDL interface ID and a sequence of property structures.
Service offers describe a specific service provided by some server, based on the information defined in a service type. Service offers consist of a reference that is the actual service object reference and a sequence of properties, where each property is a name-value pair.
Each service property of a property structure is qualified by mode attributes. Mode attributes are read-only and mandatory; combinations of both are also allowed. A value of a read-only property can initially be set when its offer is exported to the Trading Service, but cannot be modified afterwards. A value for a mandatory property must always be provided when a service offer is exported to the Trading Service.
When the Trading Service processes a client's query for a service offer, it gathers a sequence of offers together by narrowing down the set of potential offers. The query input is used to determine whether or not an offer is of an appropriate service type. The property constraints are used to determine whether or not the offer matches. The Trading Service's matching algorithm is more fully described in [8][10].

3.1.1 Service Offer Property Sequence

Table 1 shows the core property sequence of the fault tolerant service offers and includes property value examples. In order to keep service offers compatible, the mode attribute of additionally appended properties must not be mandatory.

Property name	**Data type**	**Mode attributes**		**Value example**
		Mandatory	**Read-only**	
ServiceTypeName	String	✓	✓	ft_echoService_if
ServerName	String	✓	✓	EchoServer1
ServiceName	String	✓	✓	Echo Service One
MasterList	String	✓		"0001-0002"
PrimaryBackupList	String	✓		"0003-0004"
SecondaryBackupList	String	✓		"0005-0006-0007"
OfferIsValid	Boolean	✓		True
ServerUtilization	Long			18 %
NumOfUsersOnServer	Long			6

Tab. 1. Property Sequence.

3.1.2 Mode Attributes

The property sequence includes properties that have different mode attributes. ServiceTypeName, ServerName and ServiceName's mode attributes are mandatory and read-only. These properties uniquely identify a service. A property value must always be provided and cannot be changed during service lifetime. If one of these properties must be changed, the original service offer must be withdrawn and then be replaced by the new service offer. The mode attribute of MasterList, PrimaryBackupList, SecondaryBackupList and OfferIsValid is mandatory. These properties describe the target client groups and the status of the offered service. These values may change during service lifetime, e.g. a service may become backup for another client group. ServerUtilization and NumOfUsersOnServer's mode attribute is normal (neither mandatory nor read-only). The feature that supplies the values for these properties may not be supported by some services. Such services ignore these properties simply by not providing a value for them.

3.1.3 Names and Values

ServiceTypeName, ServerName and ServiceName contain string values, which uniquely identify a service. A particular service type (ServiceTypeName) can be provided by several servers (ServerName), which can supply several similar service instances (ServiceName). These values are used when clients register with notification channels they are interested in and might also be displayed to the service user.
MasterList, PrimaryBackupList and SecondaryBackupList contain string values that include a list of client group identifiers. Client group identifiers must be unique and need to be separated within the string by a delimiter. A client queries the Trading Service for an offer that includes its group identifier in the MasterList or in one of the BackupLists respectively. The OfferIsValid boolean value marks a service offer as valid or invalid. A service offer is marked as invalid when it is temporary out of service, e.g. for maintenance reasons. Thus, clients expect such services to be back on-line eventually. This is not expected if no service offer was found in a particular list, e.g. there might be no secondary backup service available for a particular client group.
ServerUtilization and NumOfUsersOnServer's long values, which have to be implemented as dynamic properties, may be used to select the best available offer in terms of load balancing. Either the client selects the service offer with the lowest server utilisation and/or number of users or, when configured appropriately, the Trading Service does by exporting only the best available service offer.
Using property names and values, clients query the Trading Service for a particular service type (ServiceTypeName) that is master (MasterList) or backup (PrimaryBackupList or SecondaryBackupList) for its group identifier. The imported service offer includes server and service name (ServerName, ServiceName), the current service status (OfferIsValid) and load balancing information (ServerUtilization, NumOfUsersOnServer).

3.2 Fault Tolerance Architecture

The following architecture introduces the OMG Object Trading Service to an existing banking system as the mechanism to advertise and manage service offers for fault tolerant application components. The *basic architecture* does not require notifications.

The improvements may be build on top of it to increase re-connection performance, reduce maintenance and to include additional features such as load balancing. These improvements require notifications generated by the system’s Notification Service. Even in the presence of a well designed, reliable and highly efficient notification service, there is always a possibility that *clients detect service failures* ***before*** *receiving the corresponding notification*. Clients are enabled to handle this by first implementing the basic architecture and by then providing extensions.
Table 2 shows the tasks that have to be performed to provide fault tolerance within the banking system. The following Sections refer to table 2 when discussing the tasks.

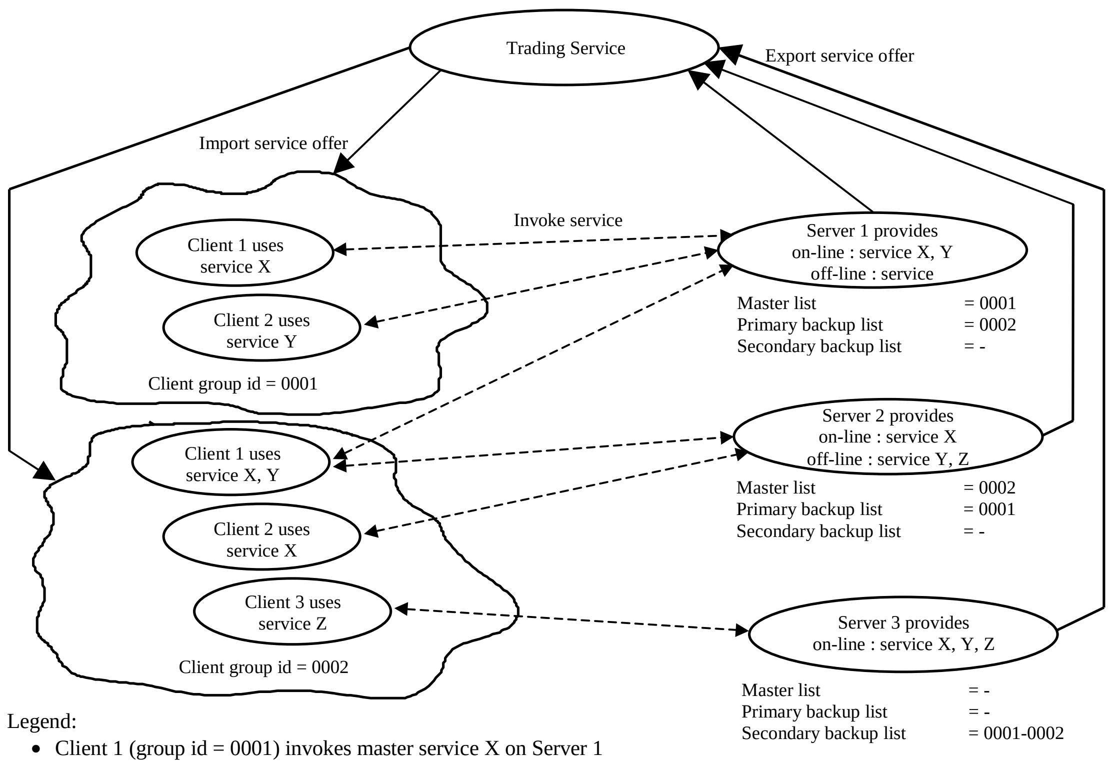
Where	What
Clients	[A] have to detect a service failure [B] need to obtain a backup service reference [C] have to re-connect to the backup service [D] have to re-connect to the original service as soon as it is back on-line
Services	[E] have to be maintained in order to be fail silent
Servers	[F] have to be maintained in order to be fail silent [G] have to provide values for dynamic properties
Trading Service	[H] service offers have to be exported to it [I] temporary invalid service offers have to be marked [J] service offers that are no longer valid have to be withdrawn [K] dynamic properties have to be supported for load balancing

Tab. 2. Tasks to Provide Fault Tolerance.

Due to the space limit, this paper will not discuss load balancing issues [G], [K]. A more complete description may be found in [8].

3.2.1 Architecture Design

Figure 2 shows the basic solution that does not involve the Notification Service. Clients detect service failure [A] by the time they invoke on a service object, even if the service failed earlier. An invocation on a failed service will eventually return an exception. The service user is idle during failure detection time, which depends on timeouts and network topology. Backup service references [B] were pre-fetched and cached and should therefore be available. Thus, re-connection to the backup service [C] will be efficient. Because of the lack of a cache updating mechanism, backup service references might have become invalid. After detecting an invalid service, clients start pinging the original service and re-connect [D] to it as soon as it is back on-line.
Services and servers are maintained [E][F] by the Service Manager. The maintenance of service offers, such as exporting [H], marking invalid [I] and withdrawing [J], is not automated and has to be performed by IT personnel.
Although this solution suffices, several improvements may be made. Service user idle time can be reduced and the cached service references should be maintained. Pinging the original service before re-connecting to it is not efficient and causes unnecessary network traffic. Furthermore, service offer maintenance could be automated.


Trading Service
Export service offer
Import service offer
Invoke service
Client 1 uses service X
Client 2 uses service Y
Client group id = 0001
Server 1 provides on-line : service X, Y off-line : service
Master list = 0001
Primary backup list = 0002
Secondary backup list = -
Server 2 provides on-line : service X off-line : service Y, Z
Master list = 0002
Primary backup list = 0001
Secondary backup list = -
Client 1 uses service X, Y
Client 2 uses service X
Client 3 uses service Z
Client group id = 0002
Server 3 provides on-line : service X, Y, Z
Master list = -
Primary backup list = -
Secondary backup list = 0001-0002

Legend:

- Client 1 (group id = 0001) invokes master service X on Server 1
- Client 2 (group id = 0001) invokes master service Y on Server 1
- Client 1 (group id = 0002) invokes master service X on Server 2
- Client 1 (group id = 0002) invokes primary backup service Y on Server 1
- Client 2 (group id = 0002) invokes master service X on Server 2
- Client 3 (group id = 0002) invokes secondary backup service Z on Server 3

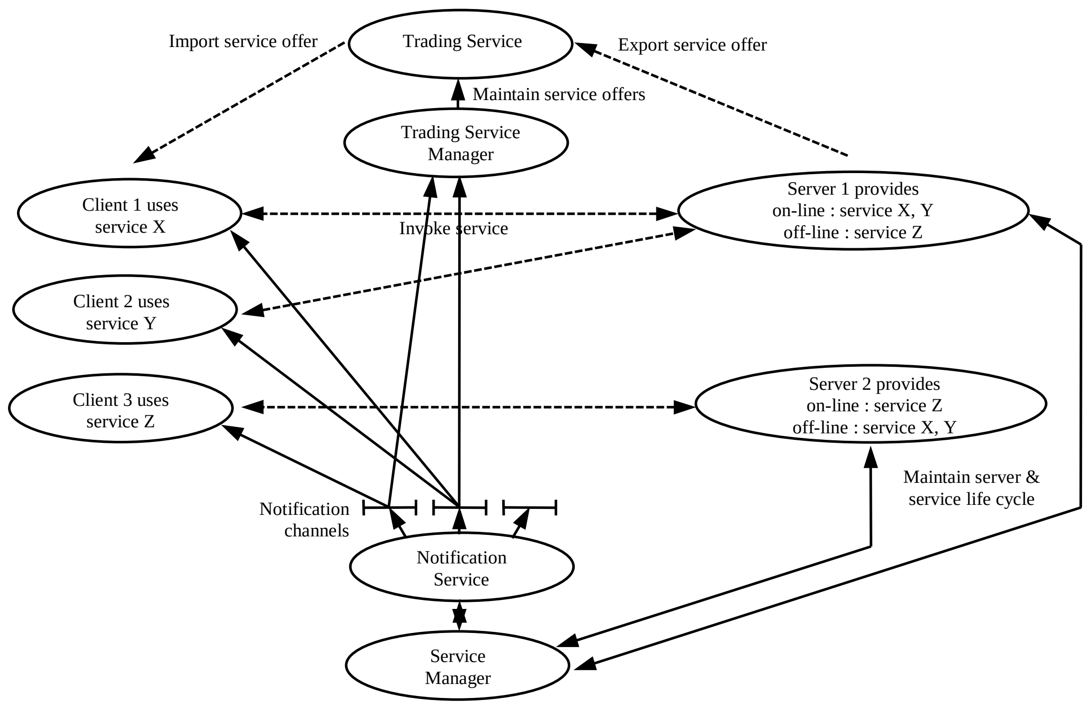
Fig. 2. Basic Architecture.

Figure 3 shows an improvement on the basic architecture that includes the Notification Service and introduces another component, called Trading Service Manager, into the fault tolerance mechanism.

Clients receive notifications whenever a service goes out of service and when it is back on-line. Invalid service references are detected [A] and clients are re-connected [C] to cached [B] backup service references in most cases without service user idle time. There is no need for pinging the original service. Clients re-connect [D] to the original service reference immediately after receiving the corresponding notification.

Services and servers are maintained [E][F] by the Service Manager. The Trading Service Manager also receives notifications. It is responsible for updating Trading Service's service offers, that is to mark temporarily invalid service offers [I]. Thus, clients will recognise invalid service offers when receiving them from the Trading Service; this reduces the possibility of service user idle time further. Servers could export [H] their service offers at start-up time and withdraw [J] them before shutting down. Therefore, IT personnel has to maintain service offers of crashed services, that are not going on-line again, only.

The improvements on the basic architecture include significantly reduced service user idle time due to re-connection in background and improved client cache consistency, and automated service offer maintenance. There is also no need to ping the original service anymore, which results in less network traffic.



Legend:

• Client 1 and Client 2 receive a notification:	"Server 1, Service Z out of service"
• Client 3 receives notifications :	"Server 2, Service X out of service"
	"Server 2, Service Y out of service"
• Trading Service Manager receives notifications :	"Server 1, Service Z out of service"
	"Server 2, Service X out of service"
	"Server 2, Service Y out of service"

Fig. 3. Improved Architecture.

The drawback of the improvement is the introduction of another component, the Trading Service Manager, to the banking system that is, in absence of replication, another single point of failure.
Attention has to be paid to the delivery order of the notifications. When a "service offer updated" notification is generated, the Trading Service must be updated before clients query for the affected service offer. This can be achieved by including a logical timestamp (sequence number) in the notification. This sequence number is added to the service offer by the Trading Service Manager when updating it. Clients are then able to verify whether they received an up to date service offer from the Trading Service. Another way to guarantee notification delivery order is to send them to the Trading Service Manager only, which updates the Trading Service and then forwards them, via the Notification Service, to the clients.
This paper does not address component replication to avoid single point of failure, e.g. in the Trading Service Manager. The banking system has already addressed this issue for its components, e.g. the Notification Service. Thus, we expect this problem to be overcome in a similar way.

3.3 Implementation and Integration

A prototype of the basic fault tolerance architecture was implemented, as shown in figure 4, in the Java Programming Language [5] using Iona's OrbixWeb and OrbixTrader [4].

Client side fault tolerance algorithms were implemented as smart proxy classes, in order to hide them from the client application program. Unfortunately, the smart proxy feature is ORB vendor specific. A way to implement a fault tolerance algorithm ORB independently is to place it between the IDL interface and the client application program, thus removing transparency.

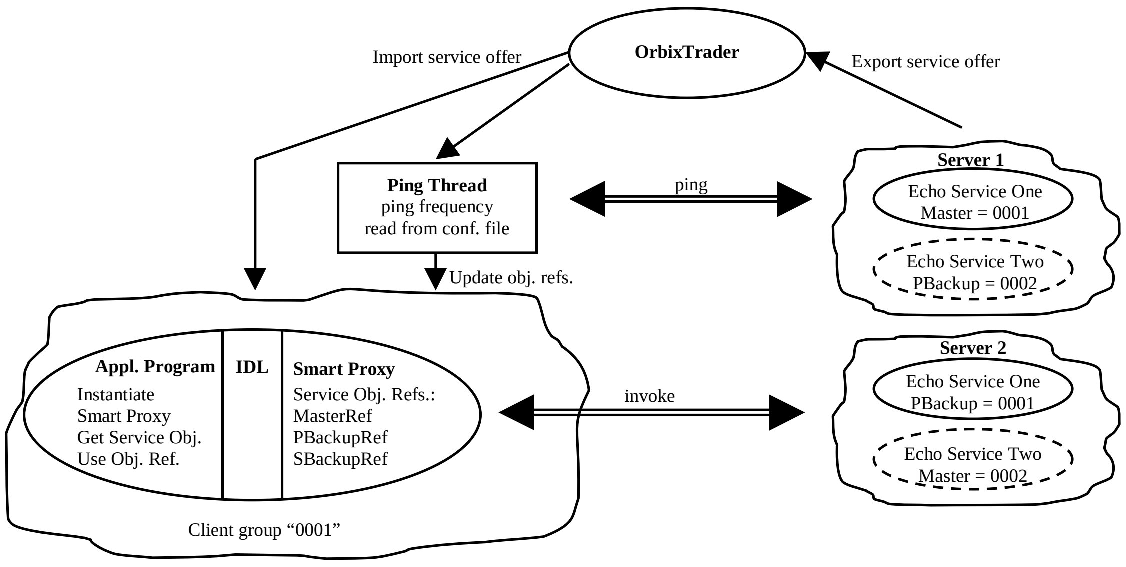


Fig. 4. Basic Architecture Implementation.

Both proposed architectures were designed to fit into the chosen banking system and to solve the fault tolerance problem identified in Section 1, by introducing new software components.
Fault tolerant application components are managed by configuration adjustment, e.g. an out-of-date service offer can be withdrawn and replaced on-line, and by client configuration files, containing a few parameters such as client group identifier, to support system scalability. Performance, in terms of service user idle time and network traffic overhead, increases significantly with the suggested improvement of the architecture. As mentioned above, a trade off between ORB independence and fault tolerance hiding had to be made. Although several ORB’s support a smart proxy feature, that is used to hide fault tolerance from client application programs and therefore from users, it cannot be used without loosing portability in terms of ORB independence.

4 Evaluation

The prototype implementation of the basic fault tolerance architecture was successfully tested on both, Sun Solaris and WindowsNT platforms. During integration testing, services where killed to force client smart proxies to re-connect to backup services and to re-connect to the original service, as soon they were re-started.
The performance of the implemented prototype is evaluated here. The duration of method invocation on a service object running on a remote server, as well as re-connection to backup and re-connection to original service object running on a remote server was measured. The method invocation duration times were taken for 1000 invocations on a simple method that returns an integer value. The results were averaged to get the duration

of a single invocation. The server was connected first using bind and second using the fault tolerance mechanism.
Service re-connection duration was measured by taking the time of an interrupted service invocation. All measurements were made with OrbixDaemon, OrbixTrader [4], client and server process running on the same Sun Solaris Ultra SPARC box and OrbixWeb3.0's [4] default configuration.

Sample [ms]	1	2	3	4	5	6	7	8	9	10	Average
Bind	3.961	3.946	4.174	4.161	3.922	3.903	3.959	3.942	4.083	4.033	4.008
Fault tolerant	4.021	4.034	4.041	4.006	4.034	4.013	4.061	4.056	4.043	4.051	4.036

Tab. 3. Method Invocation Duration.

Sample [s]	1	2	3	4	5	6	7	8	9	10	Average
Re-connection to backup service	12.227	12.217	14.521	12.166	14.615	12.155	12.229	12.328	12.303	14.568	12.933
Re-connection to original service	1.785	1.695	1.946	1.686	1.695	1.766	1.675	1.957	1.780	1.696	1.768

Tab. 4. Service Re-connection Duration.

The method invocation measurements in table 3 show that invocation time on a bound service object and on a fault tolerant service object are *essentially identical*. The difference between the two is that the invocation on the fault tolerant service object uses the provided smart proxy, whereas the invocation on the bound service object uses the default proxy. These are (almost) identical in absence of a service failure.
The re-connection to backup service, shown in table 4, consists of *failure detection*, which uses OrbixWeb3.0's [4] COMM_FAILURE timeout, *message transmission*, which depends on the network topology and *client algorithms*. The re-connection to original service, also shown in table 4, consists of pinging and therefore starting up the original (failed) service, updating the client's cache, selecting the best available service object reference and killing the ping-thread.
Re-connection to backup service and re-connection to original service may both cause service user idle time. The measurements show that user idle time is within an acceptable range. The worst case is less than 15 seconds, opposed to minutes or even hours due to a temporary or total loss of a service in absence of a fault tolerance mechanism.

5 Conclusions

This paper describes an architecture to transparently manage fault tolerance in a large scale distributed system. The presented architecture is designed to fit into an existing banking system and allows the development of fault tolerant application components. Of primary importance to our solution is the inclusion of the OMG CORBA Object Trading Service into the fault tolerance architecture as the mechanism to advertise and manage service offers for fault tolerant application components. The mechanism enables clients transparently to detect a failed connection to a service object, to discover a similar backup service object and to re-connect to it. The design of the architecture allows application component management by configuration adjustments, without re-booting the

system, supports the different needs of the banking system's clients and adequately addresses the scalability requirements of the system's infrastructure.
A prototype of the suggested mechanism has been realised and evaluated. The implementation shows that performance issues are appropriately addressed and that performance can be improved by completely implementing the architecture. The limitation of the implementation is the trade off that had to be made between fault tolerance hiding and ORB independence. In order to hide fault tolerance algorithms from the client application program, an ORB specific feature, so called smart proxy, was used. This issue is subject of further research. In conclusion, it has been demonstrated that this architecture supports the development of fault tolerant distributed application components, which results in *improved availability*.

Acknowledgements
René Meier completed this project as part of the M.Sc. in Networks and Distributed Systems at Trinity College Dublin and is now a Ph.D. student in the Distributed Systems Group funded by the Higher Education Authority (HEA) of Ireland.

6 References

[1] J. Bacon, *Concurrent Systems.* Addison-Wesley, 1993.
[2] S. Baker, W. Cahill and P. Nixon, *Bridging Boundaries - CORBA in Perspective.* IEEE Internet Computing, Volume 1, Number 5, September/November 1997.
[3] M. Banâtre and P. A. Lee, *Hardware and Software Architectures for Fault Tolerance*. Springer Verlag, 1994.
[4] Iona Technologies PLC, URL = http://www.iona.com.
[5] Java Programming Language, URL = http://www.java.sun.com.
[6] S. Landis and S. Maffeis, *Building Reliable Distributed Systems with CORBA.* Theory and Practice of Object Systems, John Wiley, New York, April 1997.
[7] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice (second edition).* Springer Verlag, 1990.
[8] R. Meier, *A Framework Providing Fault Tolerance Using the CORBA Trading Service.* MSc. Thesis, University of Dublin, Trinity College, September 1998.
[9] S. Mullender, *Distributed Systems.* Addison-Wesley, 1993.
[10] OMG, Object Management Group, URL = http://www.omg.org.
[11] R. Orfali, D. Harkey and J. Edwards, *Instant CORBA.* John Wiley & Sons Inc., 1997.