Phoinix — A Fault-Tolerant Object Service in OMA

Deron Liang
Institute of Information Science
Academia Sinica
Taipei, Taiwan, 11529
R.O.C.

S. C. Chou & S. M. Yuan
Department of Information & Computer Science
National Chiao-Tung University
Hsin-Chu, Taiwan 31151
R.O.C.

Key words:

Fault-tolerance, object-oriented programming, OMA, CORBA, distributed computing environment, distributed object services.

Abstract

The Object Management Architecture (OMA) has been recognized as a de facto standard in the development of object services in distributed computing environment. In a distributed system, the provision for failure-recovery is always a vital design issue. However, the fault-tolerant service has not been extensively considered in the current OMA framework, despite the fact that a increasing number of useful common services and common facilities have been adopted in OMA. In this paper, we propose a fault-tolerance developing environment, called Phoinix, which is compatible to the OMA framework. In Phoinix, object services can be developed with embedded fault-tolerance capability to tolerate both hardware and software failures. The fault-tolerance capability in Phoinix is classified into three levels: restart, rollback-recovery and replication; where the fault-tolerance capability enhances as the level increases. Currently, Phoinix is ported on Orbix 3.0 and on SunOS 4.2. Object services provided in the current version of Phoinix are able to tolerate hardware failures with capability up to the level two fault-tolerance, i.e., the level of rollback-recovery. We plan to continue the development of Phoinix so that object services can tolerate not only hardware failures but also software failures, such as process hangs, with all three levels of fault-tolerance.

1. Introduction

As users′ demand for resource sharing grows, distributed systems have become increasingly attractive in recent years. For their excellent price-performance ratio, distributed systems are not only attractive to programmers but also to MIS managers. However, the increasing use of computers in human lives, especially in critical environment, has led to an urgent need for highly reliable computer systems. Fault tolerance is an approach used to increase the reliability of computer systems. Since heterogeneous distributed systems tend to be less reliable in nature, fault tolerance for distributed systems thus becomes an important design issue.

Fault-tolerance design for distributed systems requires comprehensive knowledge from all aspects of system engineering, from detail hardware characteristics to complex software specification and also from low-level communication protocols to high-level distributed computing theory. Therefore, a user-friendly fault-tolerance case tool is highly valuable for application programmers without experience in the fault-tolerant system design. The existing tool-kits are either too primitive to use, such as ANSA[1] and ISIS[5], or lack of portability and/or extensibility (due to the deployment of proprietary protocols in these systems), such as PSYNC[16] and HORUS[17]. In this paper, we attempt to create a software development environment within which software components can be built not only with embedded fault-tolerance capability but also with better support of portability and extensibility in a user-friendly manner.

Object-oriented design (OOD) [6] provides a high-level abstraction to describe the nature of software components. With the class inherence mechanisms, object-oriented technology also offers greater potential in portability and code reusability. To facilitate the development of integratable software objects in distributed environment using OOD, several object interconnection standards have been proposed; notably, Object Management Group's (OMG) OMA[15], Microsoft's COM[7] and IBM's DSOM[8,18]. In this research, we select the OMA standard as our software platform for two reasons: it is an open standard and it is supported by most of the major software venders. Although the OMA specifications provide a higher-level approach for the distributed applications, it does not take the fault-tolerance issue into consideration.

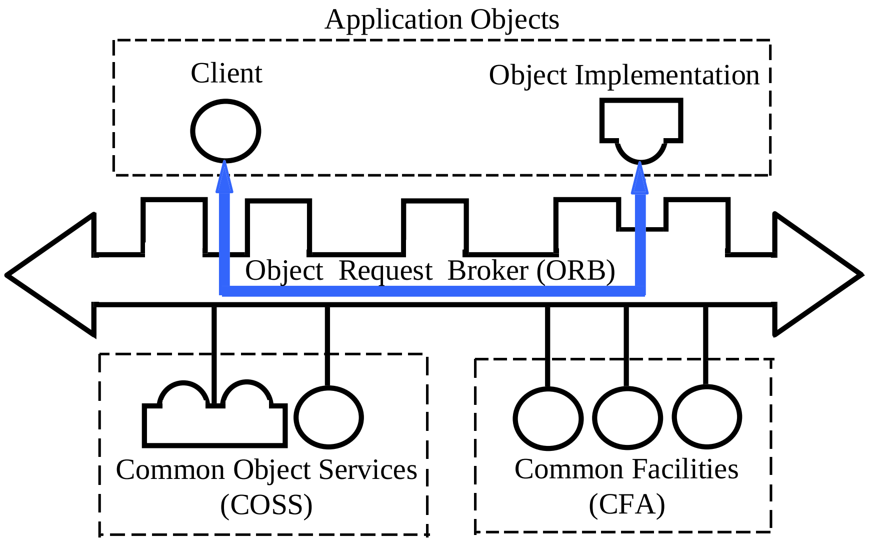

Figure 1. The OMA framework.

As shown in Figure 1, OMA consists of four major components: object request broker, (ORB), common object service specification (COSS), common facility architecture (CFA), and application objects. In OMA model, an object that provides service to clients over network is called object implementation. ORB serves as a software interconnection bus between clients and object implementation. COSS defines several commonly used services in distributed systems, such as transaction service, persistence service, etc. CFA specifies a few facilities that are closer to the application level and are more toward to specific application domain, such as common task management tools and facilities for financing and accounting systems. CORBA (common object request broker) defines the interfaces and functionality of ORB via which client programs may access services from application servers and service provided by COSS and CFA. The objective of this research is to create a fault-tolerant software development environment that is operating in CORBA platform and within which distributed object services can be easily developed to tolerate both hardware and software failures. In this paper, the capability of the fault-tolerance is classified into three levels: *restart service*(level one), *checkpoint-recovery service*(level two) and *replication service* (level three). We notice that the fault-tolerance capability of the object enhances as the fault-tolerance level increases.

In this paper, we present the design and implementation of *Phoinix*, a fault-tolerance case tool. An object server is called fault-tolerant object provided it is able to continue serving clients in the presence of faults. In the current stage, two levels of fault-tolerance are supported; namely, the *restart service* and the *checkpoint-recovery service.* Objects intend to obtain these two levels of FT capability are realized in the forms of *restart objects* and *logable objects*, respectively. We implement the *enhanced IDL compiler* (EIDL) which parses IDL interface files of the fault-tolerant objects and generates corresponding codes for the failure detection and the recovery triggering. We design a *fault-tolerance library* that serves as the base classes of logable objects. We also implement a *log server* that maintains the run-time states and audit trails of those active logable objects in the system. For the replication service, i.e., the level three fault-tolerance, we have designed but not yet implemented a daemon called *replication manager* and the class libraries for the replicated objects. The platform we adopted is Orbix[9,10,11] running on SunOS 4.2. Our system can be ported to most CORBA compliant platforms with minimal modifications since Orbix is a full implementation of CORBA.

The organization of this paper is as follows. The architecture of CORBA and the related works are presented in Section 2. An overview of Phoinix development environment and the concept of the *fault-tolerant objects* are discussed in Section 3. Section 4 gives the design and the implementation of the *fault-tolerant* objects in Phoinix, whereas Section 5 presents the design and implementation detail of other important components. Section 6 discusses the performance and other extension of Phoinix. Finally Section 7 summarizes the contribution of this work and points out the future research.

2. Background and related works

Phoinix is operating in the CORBA-compliant environment. In this section, we first describe the functionality of major CORBA components and client and server development in CORBA, and it follows by the discussion of two related case tools.

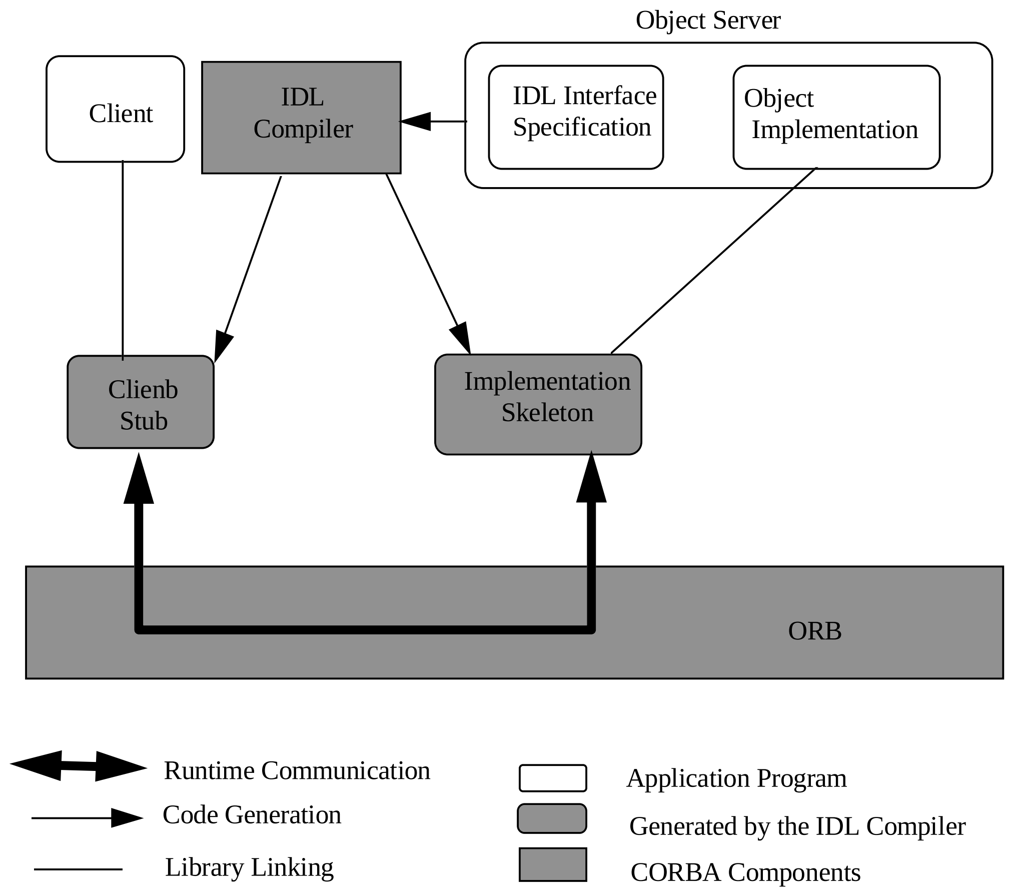

Figure 2. The illustration of Common Object Request Broker Architecture (CORBA).

2.1 Application development in CORBA

As shown in Figure 2, a CORBA environment consists of the following major components: the *client*, the *object implementation* (or server), the *IDL compiler* and the *ORB*. We briefly describe the functionality of each of these components and then discuss the interaction among these components during the program development phase as well as in the run-time. In CORBA, the service of an object server is specified in terms of the application program interface (API) using the standard CORBA interface definition language (IDL). An object that implements the service (or API) according to an IDL interface specification is called an *object implementation* (with respect to that IDL interface). In other words, an object implementation is an excitable entity that is capable of delivering the client the service specified by the IDL interface over the networks. A client makes a request to an object implementation and expects the reply from it all via ORB. In other words, the ORB serves as a software interconnection bus between the client and the object implementation. As shown in Figure 2, a client is able to access the service of an object implementation only if it has the handle of that object, i.e., the *client stub*. The client stub is generated by IDL compiler after it compiles the IDL interface of that object implementation. On the other hand, the object implementation can provide its service over the networks via ORB only if it has the *implementation skeleton* which is also generated by the IDL compiler. Figure 3 illustrates the complete development of the object implementation and the client.

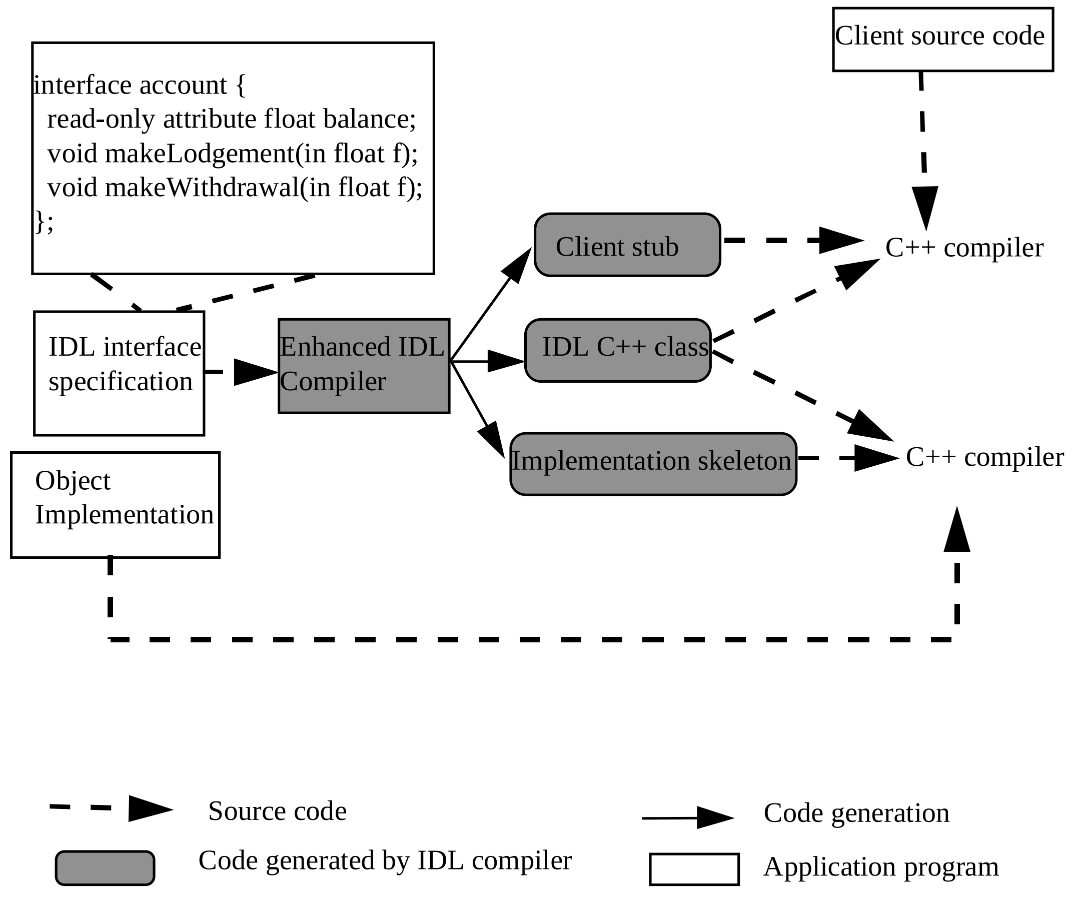

Figure 3. The application development in CORBA.

We now briefly describe the basic concept of how CORBA and its components operate. To invoke an operation on a remote object implementation, a client must first bind to that object. ORB first check if the remote object implementation exists. If not, an instance of the object implementation will be invoked. The object implementation replies the results of the invocation to the client via ORB after the object implementation completes the execution of the invocation. The client may send subsequent requests to the same object implementation without the re-establishment of the communication channel.

Notice that the IDL is a definition language, it is not an operational programming language such as C. Given an IDL interface, there may exists many object implementations for that interface. Moreover, these object implementations may be implemented in various programming languages in different operating systems or hardware platforms. To accommodate the independence of the IDL interface from the object implementation, IDL compiler must be able to compile the IDL interface to produce the client stub and the implementation skeleton into many target programming languages, such as C, C++, FORTRAN and PASCAL, according to CORBA language bindings. We notice that the client stub acts as an local proxy to the client process on behalf of an object implementation that provides the actual service. More precisely, the client stubs shield the complex operations, such as remote request preparation, parameter's interpretation, request invocation and reply delivery from the client. Similarly, the implementation acts as the local proxy of a client that makes the request. Finally, we turn our attention to exception handling. ORB raises an exception signal to the client proxy (the client stub) if the peer object implementation crashes or hangs during the request invocation; the client may react to this signal via an exception handling routine.

2.2 Related Works

In this section, two object-oriented fault-tolerance tool-kits, Electra and Arjuna, are reviewed for comparison. The major difference between Phoinix and the two tool-kits is that both Electra and Arjuna are proprietary while Phoinix is built upon CORBA and is compliant to the open standard.

Electra

Electra is an object-oriented tool-kit providing a set of new abstractions helping to build reliable distributed systems in C++. The tool-kit allows programmers to create C++ objects that can live on different machines in a network and communicate synchronously with other Electra C++ objects. In Electra, services can be defined using the Electra Services Definition Language (SDL) called SNOOPY-SDL. (A service is an abstract definition of what a server is prepared to do for its clients.) The programmer exploits services mainly by communicating with them using Remote Method Calling (RMC) and broadcasting. Electra chooses the Multi-cast Transport Service (MUTS)[12,13] as its basic transport layer. MUTS offers primitives to manage threads, sending and receiving messages, and performing reliable group communication. In Electra, objects derived from the abstract base class *Migratable* can be transmitted over a network. Every migratable object must have an appropriate dump and recover method, and the stub code generated by SDL relies on these methods for marshaling and unmarshalling the state of an object. Because Electra has its own Services Definition Language (SDL) and does not follow an open standard, it is very difficult to port Electra to an existing platform.

Arjuna

Arjuna is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications[20]. Arjuna provides nested atomic actions for structuring application programs. Atomic actions control sequences of operations upon local and remote objects, which are instances of C++ classes. Operations upon remote objects are invoked through the use of remote procedure calls (RPCs). The computational model of Arjuna is using atomic action controlling operations on persistent objects. In Arjuna, objects are long lived entities and are the main repositories for holding system states. A persistent object can be replicated on several nodes to achieve fault tolerance. Arjuna ensures the consistence of internal states by automatic invocations on objects. The major drawback of Arjuna is that it is not conformed to any standards. It will be more difficult to adapt Arjuna to existing working environments whereas Phoinix can be ported to most CORBA compliant products with minimal modifications.

3. Overview of Phoinix

We state the design philosophy and design objectives of Phoinix in this section. We also sketch the development as well as the run-time environment of the Phoinix system. In the second half of this section, we introduce the key concept in Phoinix, the *fault-tolerant objects*. Two types of *fault-tolerant* objects are introduced, namely, the *restart objects* and the *logable objects*.

As mentioned in the introduction, we are interested in the design of a development environment in which object implementations are constructed with desired fault-tolerance capability in a semi-automatic fashion. We categorize the fault-tolerance capability into three levels: *restart service*(level one), *checkpoint-recovery service*(level two) and *replication service* (level three)[22]. Object implementations in Phoinix with fault-tolerance capability of level 1, 2, and 3 are called *restart objects*, *logable objects*, and *replicated objects*, respectively. As the names suggest, the restart object resumes the service as a fresh server after recovering from failure whereas the logable object resumes its service from the last check-point. For the replicated object, several replicas of the object implementation coexist in the system in some kind of cooperative fashion. The functionality and the implementation detail of these fault-tolerant objects are presented in the current and the following section respectively. We notice that many common types of hardware failures and software failures can be handled by these three levels of fault-tolerance. For example, permanent software failures such as design faults can be managed by replicated objects, such as N-version programming[2]. Lastly, an object implementation is called a fault-tolerant object if it is implemented with a level of fault-tolerance.

Phoinix is designed with a few assumptions on the operating environment. Firstly, crashed sites are assumed to operate in a fail-stop[19] manner. (This is a common assumption in many fault-tolerant systems[9][19][20].) In many systems, we find that site crashes in a distributed environment are relatively infrequently and are usually independent of each other. We assume that the client invocation is not nested, i.e., the invoked object doesn't invoke operations on objects of another object implementation.

This assumption implies that the object implementation is unlikely to hang due to the crash of the client. In order words, the failure detection of the client becomes unnecessary. Possible extension of Phoinix to support the nested invocation will be discussed in Section 6.2.

Next, we discuss the fault-detection and the triggering of recovery process in Phoinix. The detection of an object implementation failure relies on a fundamental service from ORB. Recall that the object implementation failure can be detected by ORB so long as this object is bound by a client invocation. More precisely, the client is able to detect the object implementation failure after it makes an invocation and receives an exceptional signal from ORB.

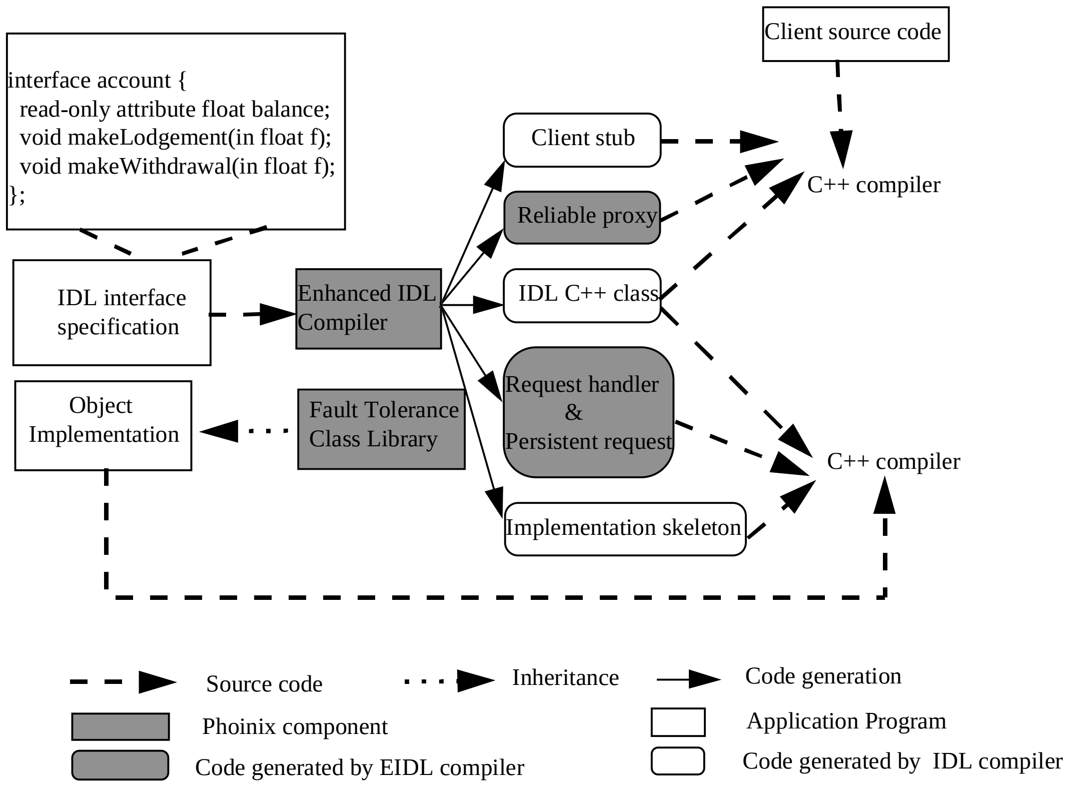
interface account {
read-only attribute float balance;
void makeLodgement(in float f);
void makeWithdrawal(in float f);
};
IDL interface specification
Enhanced IDL Compiler
Object Implementation
Fault Tolerance Class Library
Client source code
Client stub
Reliable proxy
IDL C++ class
Request handler & Persistent request
Implementation skeleton
C++ compiler
C++ compiler
Source code
Inheritance
Code generation
Phoinix component
Application Program
Code generated by EIDL compiler
Code generated by IDL compiler

Figure 4. The Enhanced IDL compiler and the application development in Phoinix.

We are now ready to introduce the development of application fault-tolerant object in Phoinix. Figure 4 depicts the development procedure for both the object implementation and client. The development of fault-tolerant objects in Phoinix closely matches with the application development model in CORBA as introduced in the previous section. As described, three types of fault-tolerant objects are identified, and the restart objects as well as the logable objects are implemented.

```
/* account.idl */
interface account : Public Logable {
   readonly attribute float balance;
   void    makeLodgement (in float f);
   void    makeWithdrawal(in float f);
};
```

Figure 5. IDL specification of an account object declared as a logable object.

To declare an object to be a fault-tolerant object, the keyword *Restart* (or *Logable)* has to be included in the IDL specification for the restart object (or logable object). Figure 5 illustrates the IDL declaration of a the logable object. After parsing an object‘s IDL specification, the *enhanced IDL compiler* (EIDL) generates two sets of codes in addition to the ordinary client stub and the implementation skeleton. These two sets of programs are the *fault detection routine* and the for the client and the fault-tolerant skeleton for the object implementation. The functionality of the fault-tolerant skeleton will be discussed in Section 4.2.

For the client program, the client stub with the fault detection routing and recovery process triggering routine are called *reliable proxy*. Recall that this reliable proxy assumes the responsibility to detect the crash of the bound object implementation as discussed previously. Upon the detection of a server failure, appropriate recovery process is triggered by the reliable proxy according to the type of the object implementation. The recovery process of the restart objects and the logable objects are discussed later in this section.

For logable objects, the fault-tolerance library defines the base class of logable objects from which such objects can inherit directly. This base class declares a set of fundamental member functions in virtual form so that logable objects may manage its critical data members in cooperation with the log server. The detail implementation of the fault-tolerance library is given in Section 5. We notice that these member functions are implemented in virtual base, the application designer has the freedom to overload these functions to better manage the critical data of the logable objects in a more efficient manner.

The fault-tolerance architecture is designed as a software layer operating on top of ORB, as shown in Figure 6. This architecture consists of four major components: the enhanced IDL compiler, the fault-tolerance library, the log server and the replica manager. Finally, the replica manager is responsible for coordinating decisions among the replica of an object implementation during request invocations from clients.

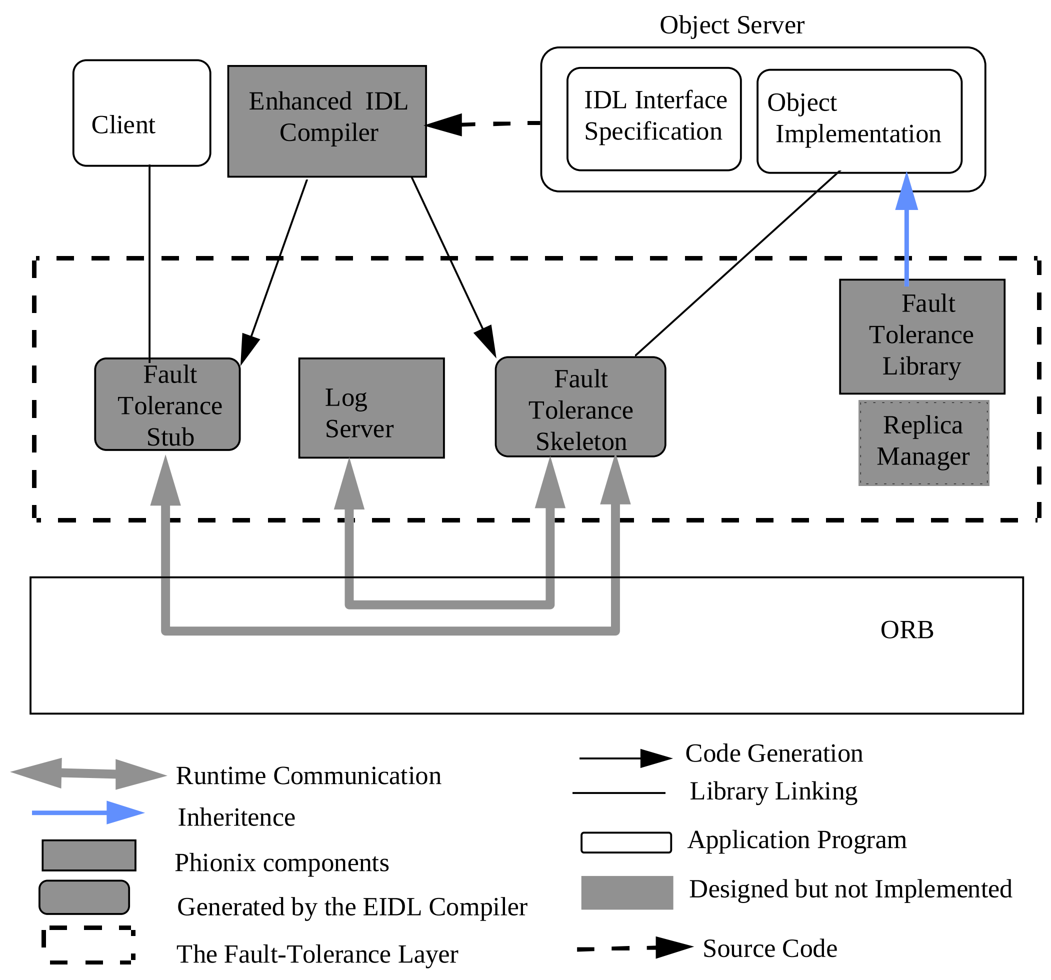

Figure 6. The architecture of the fault tolerance layer in Phoinix.

4. The fault-tolerant objects

We have stated that the restart objects and the logable objects are supported in current version of Phoinix. In this section, we present the detail implementation of these two types of objects. We first describe the implementation of the failure detection mechanism and recovery invocation of Phoinix in conjunction with the CORBA framework, and then we briefly describe the recovery process for the restart objects. The major part of this section devotes to the design and implementation of logable objects since the recovery process for the restart objects is rather straightforward. It is unable to restore the failed objects to their previous states before failures occur. That's why we designed another level of fault tolerance object (the logable object), for the purpose of returning the failed object to its previous state.

4.1 The failure detection and recovery

The design philosophy of Phoinix is to let the procedures of *failure detection* and *recovery* of the fault-tolerant objects be transparent to the object designer. Before we discuss the real implementation, we briefly review the binding between a client and an object implementation in CORBA. Before binding to an object implementation that provides the desired service, the client first broadcasts to check if an object implementation of that type is already activated in the network. If there are more than one object implementation that are already activated, the client randomly chooses one to bind; otherwise, client randomly activates one. If the binding is successful, the client proceeds to making request invocations, or the client repeats the "broadcast and bind" procedure until the binding is successful. Upon the detection of a failure of a bound object implementation, i.e., the object implementation crashes, the client would try to rebind to an available object implementation that provides the same IDL interface. Once the binding of a new object implementation is successfully, the new object implementation may continue the service to the clients after the appropriate recovery process. (See discussion below.) In order to make the rebinding operation transparent to the client, we design *reliable proxy,* as described in the previous section, to perform these operations.

If an object implementation is declared as a restart object, the client just needs to rebind to the target object in a new object implementation, and the object implementation doesn't need to do any recovery. Notice that this fault tolerance level only assures the object is always available to users; it can not return failed objects to their previous states before the failures occurs.

4.2 The design and implementation of the logable objects

4.2.1 The failure recovery of the logable objects

In this section, we first describe the design concept and the failure recovery procedure for the logable objects, then we discuss the implementation details in the second half of this paper. Through our fault tolerance service, the object implementation has to cooperate the logable server in order to maintain its critical data and subsequent client invocation.

A logable object binds to a log server using the "broadcast-and-bind" strategy as a client, when it is first activated on a node. Under normal operation, the requests sent from the client shall be logged in the audit trail of the bound log server. The bound server will create an audit trail for the log server and a safe storage for the critical data. The logable object shall have the capability to decide whether a request needs to be saved in the audit trail according to the application domain. The logable object also can decide to checkpoint its critical data into the safe manipulated by the log server as necessary. Once the checkpoint is done, the log server purges the audit trail and continues service to the logable object. If the object implementation crashes, the client will receive an exception in the successive method call. Then, the client uses the "broadcast and bind" strategy described before to re-activate a reliever server. Then the logable server is up, the client binds to the new object and informs the corresponding log server. Then the logable object binds to the log server and starts the rollback recovery process. Note that we have assumed that the failures occurrence is infrequent in a distributed system. This implies that the possibility that logable object and its log server both crash at the same time is very slim. In other words, our failure deception and recovery algorithm is robust. The log server and failure recovery will be discussed in Section 5. Notice that a logable object may serve more than one client at a time, and it is possible that each client may bind or even reactivates different object implementations after it discovers the crash of the logable object.

4.2.2 The implementation of the Logable objects

As discussed in the design, a logable object that shall be able to perform a few fundamental operations in provision of the recovery from a failure. These operations are: logging requests, redoing requests, saving objects, and loading objects. Notice that the first two operations are for handling the requests whereas the later two are for the object states. We now describe these operations.

Logging requests

For each IDL interface operation, we need to implement a persistent request class. The persistent request class's constructor has the same arguments as the IDL interface operation. The persistent request class can be generated automatically from IDL interface specification. The persistent request will save itself into the log server automatically when the IDL interface operation call terminates. Since it is not necessary to log every performed request for logable object, we let the user declare the to-be-logged request himself. When the user implements a to-be-logged IDL interface operation, he can create the corresponding persistent request at the beginning of the operation. The persistent request will save itself into the log server automatically when the IDL interface operation call terminates.

Redoing requests

As described previously, during the recovering process, the object implementation should redo the requests logged in the audit trail in the original order. In order to dispatch the logged requests to the target object, we need a request handler array. The request handler array maintains request handlers for the object implementation's logable objects. A request handler unmarshalls the arguments of a specific IDL interface operation from a logged request, then invokes the operation on the target object. Request handler can be generated automatically from IDL interface specification.

Saving objects

In order to make a checkpoint for all its logable objects, an object implementation need to maintains an object table to record all its logable objects. The to-be-saved critical states of the logable object should be defined by its designer. Hence, we design an abstract class *Logable*, which is the base class of all logable objects, as shown in Figure 5. Logable objects are supposed to register themselves to log severs when it is created the first time. Object implementations may overload the member functions *SaveState()* and *LoadState()* defined in the base class, which is inherited from virtual class Logable, to save the user-defined critical states.

Loading objects

As described in Section 3, when an object implementation is activated by the fault-tolerant client stub as the substitute for the crashed object implementation, it shall start the recovery process by reloading all the logable objects saved in the log server by the crashed object implementation since the last checkpoint. However, if a logable object to be loaded is not in the memory, the object implementation should create it in memory first. Hence, we design an object factory for each kind of logable object. Users can declare the object factory for a logable object using the macro we provide. If a logable object has pointers to other logable objects, the loading operation goes in *depth -first- search*.

5. Implementation of other components in Phoinix

As outlined in Figure 6, Phoinix consists of three major components: EIDL compiler, the fault-tolerance library, and log server. In this section, we will discuss the design of these components and some implementation issues.

EIDL Compiler

The Enhanced IDL (EIDL) compiler scans IDL interface specification file and produces fault tolerance codes in addition to standard IDL compiler‘s client stubs and implementation skeleton (see Figure 6). For the client side, the EIDL produces reliable proxy for each IDL interface. The reliable proxy is responsible for the failure detection of the object implementation and to trigger the recovery process according to the type of the objection implementation. For the object implementation’s side, the EIDL compiler generates the fault-tolerant skeleton. The skeleton performs the request logging in the normal operations and performs the redo of these requests during the failure recovery recourses for three logalble objects.

Fault Tolerance Library

The fault-tolerance library consists of two families of class libraries for the logable objects. One family is for the fault-tolerant skeleton, where it defines the Persist Request class and Request Handler class as the base classes for request logging and request redo, receptively. The other family defines the base class from which the application logable objects can inherit. This class defines the virtual functions *SaveState()* and *LoadState()* to manage the critical data members of the logable objects. Note that member functions are declared as virtual, so that the application may be implemented in a customized manner by overloading these number functions.

Log Server

As stated briefly in Section 3, an object implementation shall register to a log server upon the invocation from a client. That log server maintains the audit trail and also the reliable repository for that registered object implementation. To avoid the data consistency issue, we implement the *2 phase lock* (2PL) protocol on each object implementation handled by a log server. A client is allowed to bind to an object implementation only if it is granted the lock to access the audit trail of that object implementation. This implies that an object implementation serves a client for a given point of time. To support the concurrent service to multiple clients certainly complicates the design of the log server. This will be considered in the next version of Phoinix and is not of the scope of discussion in this paper. Next, we consider the failure recovery of the log server itself. If the object implementation detects the failure of its log server, using the same technique as the client detects the object implementation failure, the object implementation binds to another log server using the same broadcast-and-bind strategy. When a log server recovers from crash, it must reset itself and recovers all the audit trail it once managed. We also have to consider the case that a failure occurs in the object implementation on the log server during the check-pointing process. To avoid either the object implementation or the log server enters an inconsistent state, we implement the *2 phase commit* (2PC) protocol in the check-pointing process. This protocol ensures the either all audit trail data has been saved into the save area in the log server and the audit trail has been removed, or everything remains intact.

In distributed system, fault-tolerance is always a vital design issue, however, fault-tolerance service has been ignored in the OMA framework despite the fact that increasing number of common services and facilities have been discussed and adopted in OMA. The fault-tolerance capability in Phoinix is classified into three levels: restart, rollback-recovery and replication, where the fault-tolerance capability enhances as the level increases.

6. Discussion

In this section, we first report the performance study of the Phoinix based on a series of experiments. We also discuss possible extension to the current version of Phoinix so that it can fully support our optimal design goal.

6.1 Performance Measurement

We design a single experiment to explore the performance of Phoinix. The experiment involves account services, such as deposit, withdraw, and balance inquiry. A client program makes 200 invocations on a remote account object over a local area network. During this period, the account object checkpoints it‘s critical data onto a log server residing on another host. The account object, the client program, and the log server are all running a Sun Sparc Workstations connected through a 10B-T Ethernet. Figure 7 depicts the experiment results where the X-axis represents the number of checkpoints during the client‘s invocations and the Y-axis represents the time duration over which the client makes those 200 requests. It clearly shows that the overhead increases from the check-pointing data to the log server. It appears that the total overhead from the check-pointing dominates the total overhead from the fault-tolerance measure in Phoinix. This also suggests that performance improvement of the checkpoint implementation deserves more attention in future version of Phoinix.

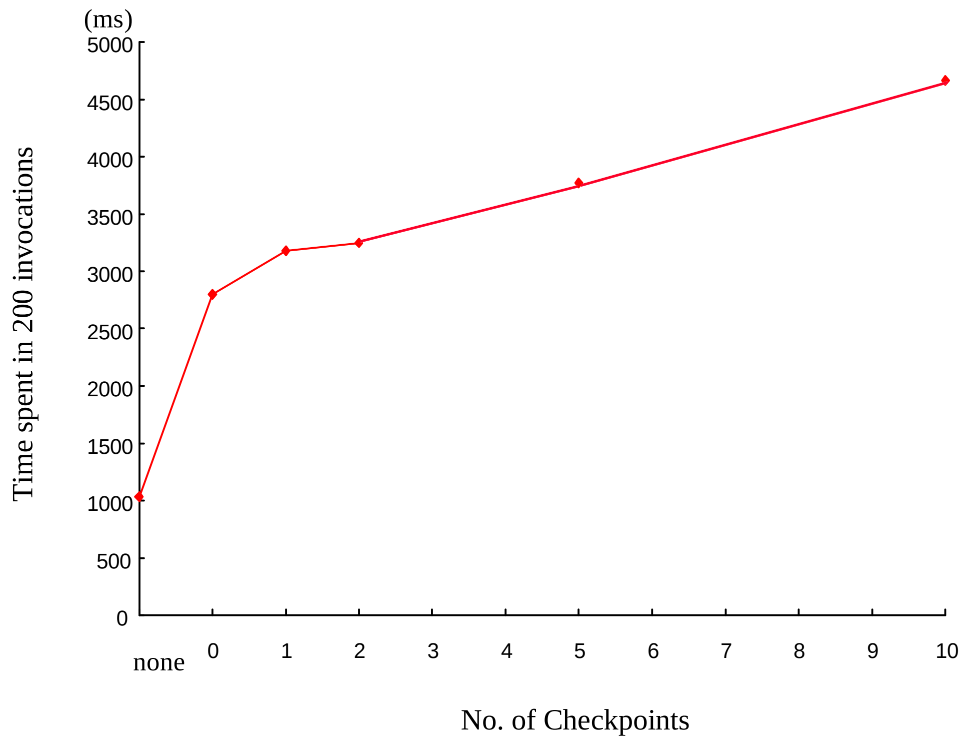

Figure 7. The response time of 200 invocations under various check-pointing frequencies.

6.2 Extensions

In the previous section we have identified areas where the current version of Phoinix can be enhanced in order to support the replication service, the nested invocation, and the software failure. The replication service provides for the explicit replication of objects in a distributed environment of replicated copies. In our original design, the replica manager is responsible for the coordination of the interaction among all replicas so that object implementation operates as if there is a single copy in the system. To support nested operation, a persistent request in an IDL interface operation call should also log every outgoing request and response. If the IDL interface operation call is resumed later because of a failure, those outgoing requests which already have logged response should be rolled back to avoid the same requests from being executed for more than once. For masking transient software failure, we define *RequestRandomizer* class to reorder the requests saved in the audit trail before redoing requests. A programmer can redefine his own policy to reorder the logged requests. Huang has suggested that most transient software failures can be masked by redoing the past requests from the last checkpoint to the crash point in the audit trail in a different order[22]. Thus object implementations developed with Phoinix will be able to tolerate transient software failures provided the recovery process with this mechanism. One possible approach to support this mechanism in Phoinix is to modify or overloaded the *PersistRequest()* in the Fault-Tolerance class library in such a way that the redo order differs from that in the audit trail.

7. Conclusions

In a distributed system, the provision for failure-recovery is always a vital design issue. However, the fault-tolerance service has not been extensively considered in the current OMA framework, despite the fact that a increasing number of useful common services and common facilities have been adopted in OMA. In this paper, we propose a fault-tolerance developing environment, called Phoinix, which is compatible to OMA framework. The fault-tolerance capability in Phoinix is classified into three levels: restart, rollback-recovery and replication; where the fault-tolerance capability enhances as the level increases. Currently, Phoinix is ported on Orbix 3.0 and SunOS 4.2. Object services provided in the current version of Phoinix are able tolerate hardware failures with capability up to the level two fault-tolerance, i.e., the level of rollback-recovery.

In this paper, we have introduce the concept of fault-tolerant objects in Phoinix. Two types of fault-tolerant objects are supported, namely, restart objects and logable objects, corresponding to the two level of fault-tolerance, restart and rollback-recovery. We have discussed the system architecture of Phoinix that consists of these major components: EIDL compiler, fault-tolerance and log server. We also have described the application development environment of Phoinix within which application object implementation can be developed with desired level of fault-tolerance. Phoinix was designed to support three levels of fault-tolerance as described in the introduction, though, two of them are implemented. We also plan to extend the recovery mechanism in the logable objects so that software transient failures can be masked and tolerated. The replication service can also be supported with minor extension hardware and software platforms. Performance issues have not been drawn extensive attention in the current implementation. As the experiments demonstrated, we have identified key areas where performance improvement can be mad in the next generation of Phoinix.

References

[1] *ANSAware Version 4.1 Manual Set,* Architecture Projects Management Ltd., Castle Park, Cambridge UK, March 1993.

[2] A. Arizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 12, pp. 1491-1501, December 1985.

[3] J. Bacon, *Concurrent systems*, Addison-Wesley, Inc., New York, 1993.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Welsey, Reading, NJ, 1987.

[5] K. P. Birman, "Integrating runtime consistency models for distributed computing," Tech. Rep. 91-1240, Dept. of Computer Science, Cornell University, July 1993.

[6] G. Booch, *Object-Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Inc., 1991.

[7] K. Brockschmidt, *Inside OLE,* 2nd ed., Microsoft Press, Redmond, Washington, 1995.

[8] “SOMobjects: A Practical Introduction to SOM and DSOM”, IBM, International Technical Support Organization, July 1994.

[9] *IonaSphere Issue 11*, IONA Technologies Ltd., May 1995.

[10] *Orbix Advanced Programmer’s Guide*, IONA Technologies Ltd., November 1994.

[11] *Orbix Programmer’s Guide*, IONA Technologies Ltd., November 1994.

[12] *MUTS Documentation*, The ISIS GROUP, Cornell University, September 1992.

[13] R. Koo and S. Toueg, "Check-pointing and rollback-recovery for distributed systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, No 1, pp. 23-31, January 1987.

[14] R. Ladin, B. Liskov, and S. Ghemawat, "Replication," *ACM Transaction on Computer Systems*, Vol. 10, No 4 , pp. 360-391, November 1992.

[15] *The Common Object Request Broker (CORBA): Architecture and Specification, v 1.2*, Object Management Group, Inc. December 1993.

[16] L, Peterson, N. Buchholz and R. Schlichting, "Preserving and using context information in inter-process communication," *ACM Transactions on Computer Systems* 7, Vol. 3, pp. 217-246, August 1989.

[17] R. V. Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson, "Reliable multicast between microkernels," *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures,* Seattle, Washington, April 1992.

[18] J. R. Rymer, “IBM‘s System Object Model”, *Distributed Computing Monito*r, Vol. 8, No. 3, page 1-24, March 1993.

[19] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An approach to

designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, pp. 222-238, August 1983.

[20] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System," *Computing Laboratory*, University of Newcastle upon Tyne, UK.

[21] R. V. Renesse, *A MUTS Tutorial*, Cornell University, 1991.

[22] Y. M. Wang, Y. Huang and W. K. Fucks, "Progressive Retry for Software Error Recovery in Distributed Systems," *Proceeding of 22nd Fault-tolerance Computing Symposium*, 1993.