ELSEVIER

Information and Software Technology 39 (1998) 965–973

INFORMATION AND SOFTWARE TECHNOLOGY

A fault-tolerant object service in the OMG's object management architecture

Deron Liang[a,*], S.C. Chou[b], S.M. Yuan[b]

[a]*Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan, ROC*
[b]*Department of Information & Computer Science, National Chiao-Tung University, Hsin-Chu 31151, Taiwan, ROC*

Received 10 April 1997; revised 10 November 1997; accepted 13 November 1997

Abstract

The object management architecture (OMA) has been recognized as a de facto standard in the development of object services in a 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 an 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 two levels: restart, and rollback-recovery; where the fault-tolerance capability enhances as the level increases. Currently, Phoinix is ported on Orbix 2.0 and on SunOS 4.2. In this paper, the design and implementation of Phoinix is presented and its performance is evaluated. © 1998 Elsevier Science B.V.

Keywords: Fault-tolerance; Object-oriented programming; OMA; CORBA; Distributed computing environment; Distributed object services

1. Introduction

As users demand for resource sharing grows, distributed systems have become increasingly attractive in recent years. For their better 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 environments, 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 of all aspects of system engineering, from detailed hardware characteristics to complex software specification. Fault-tolerant applications designed and implemented from scratch are often complicated, proprietary, and error-prone. Thus user-friendly fault-tolerance case tools are in great demand. Existing fault-tolerance toolkits such as ANSA [1], ISIS [2], PSYNC [11] and HORUS [12] are now widely available. These products are all designed for *process-based* distributed applications where process is the basic entity subject to monitoring and recovery.

Recently object-oriented design (OOD) [3] has been widely applied to software design and development since it offers greater potential in portability and reusability. Later, several distributed object middlewares have been proposed; notably, Object Management Group's (OMG) CORBA [10], Microsoft's DCOM [4], IBM's DSOM [6,13] and JAVA RMI [http://www.javasoft.com]. Though these middlewares greatly enhance the quality and reusability of the distributed object-based applications, they do not, however, ease the pain of developing distributed 'fault-tolerant' object-based applications. The issues of developing fault-tolerant object-based applications differ from that of process-based applications in several aspects. Firstly, a server process often contains many objects. An object death (destroyed from the addressing space) does not necessary lead to a process crash. The detection mechanism for process crash is not sufficient to detect the object crash. Furthermore, many object servers are implemented as multi-threaded servers and each object is running in a separate thread. Thus, the detection mechanism for process hang may not be able to detect an object hang (or a thread hang). As a result, there is a need for object-based

* Corresponding author. Tel.: +886 2 7883799; fax: +886 2 7824814; e-mail: drliang@iis.sinica.edu.tw

PII S0950-5849(97)00055-4

fault-tolerance service. Though the proposed fault-tolerance service in this paper is based on OMG's CORBA model, we believe a similar service or concept can be ported to DCOM or JAVA RMI.

1.1. CORBA introduction

CORBA reference model consists of four major components: object request broker (ORB), common object service specification (COSS), common facility architecture (CFA), and application objects. When an object implements a service specification and exposes its service to clients over a network, this is called an object implementation for that service specification. We feel that the term 'object implementation' is not self-explanatory and sometimes confusing. Instead we use the term 'service object' throughout the paper whenever we feel this term is a more intuitive than 'object implementation'. A service object usually resides in an address space of a server process in most of the current commercial ORB. A server process may contain multiple service objects. ORB serves as a software interconnection bus between clients and service objects. 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 lean more towards specific application domains, such as common task management tools and facilities for financing and accounting systems. CORBA specification defines the interfaces and functionality of ORB via which clients may access services from service objects and/or service provided by COSS and CFA.

Fig. 1 illustrates the complete development of the object implementation and the client. In CORBA, the service offered by a service object is specified in terms of the standard CORBA interface definition language (IDL). The CORBA IDL compiler generates two software components with respect to each IDL specification, namely, the *client stub* and *implementation skeleton*. A client needs to bind via ORB to an service object before it can invoke operations on that service. ORB checks if a service object exists in the networks. If not, an instance of that service object will be activated and an object handle (an instance of the client stub) is returned to the client. The client with the object handle can make invocations on that service object and wait for the reply, all via ORB.

We notice that the client stub acts as a local proxy to the client object on behalf of a service object that provides the actual service. This local proxy shields the complex operations from the client object, such as remote request preparation, parameter marshalling and unmarshalling, and reply delivery. On the other hand, the implementation skeleton acts as the local proxy of client objects that makes the requests where it handles the invocation dispatching and marshalling and unmarshalling of the invocation parameters. With regard 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. Phoinix takes the advantage of this ORB service to detect the anomaly of the service object. This will be covered in detail later in the paper.

1.2. Related works

Current efforts to improve the reliability of ORB or CORBA applications generally fall into two approaches: the system approach and object approach. Electra [9] and Arjuna [14] are the representing systems for each approach.

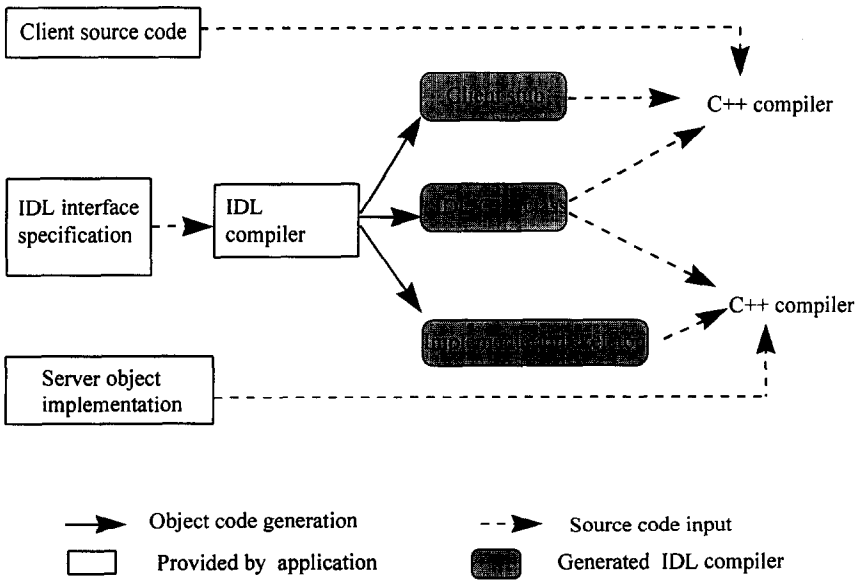

Fig. 1. The application development in CORBA.

1.2.1. 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.) 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 recovery method, and the stub code generated by SDL relies on these methods for marshaling and unmarshalling the state of an object. Furthermore, Electra supports reliable group communication so that service objects in a group receive the same sequence of invocations, thus the internal states of those service objects remain consistent. 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.

1.2.2. Arjuna

Arjuna is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications. 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 uses 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.

1.3. Motivation and objectives of Phoinix

We believe that the fault-tolerance service added to CORBA should be portable in the sense that it requires no modifications to ORB, and it is not based on any proprietary kernel support. Unlike Electra or Arjuna, Phoinix is designed as a CORBAService and thus no modifications to ORB is required and it interacts only with standard ORB interface. Furthermore, a service object may require different fault-tolerance support in different runtime environments. For example, it may require fast recovery if it serves in a real-time environment and only require cold backup in a non-real-time environment. Consequently, a checkpoint recovery scheme should be deployed in the previous case, whereas a restart mechanism is sufficient in the latter. Most existing systems, such as Electra or Arjuna, are based on an underling reliable group communication layer. This implies that replicas are all running as hot standbys. Phoinix, however, allows the service objects to select the kind of fault-tolerant mechanism that is most suitable to the nature of the application domain. The types of the fault-tolerance mechanism in Phoinix can be classified into two levels: *restart service* (level one) and *checkpoint-recovery service* (level two). The fault-tolerance capability increases as the fault-tolerance level increases.

Phoinix provides three IDL interfaces to service objects: *Restartible, Logable* and *LogManager*. A service object server needs to inherit the callback interface `Restartible` or `Logable` in its service IDL specification if it wishes to obtain *restart service* or *checkpoint-recovery service* from Phoinix. Such objects are called *restartible objects* or *logable objects*, respectively. We implement the *enhanced IDL compiler* (EIDL) which parses the IDL interface of the fault-tolerant objects and generates corresponding codes for failure detection and recovery triggering. LogManager (LM) is a CORBA service object that provides the basic checkpointing services for logable objects. Logable objects may reuse a *fault-tolerance library* provided by Phoinix that declares a few base classes for logable objects and implements a few basic operations to interact with the LM at runtime. Orbix [7] running on SunOS 4.2 has been selected to be our ORB platform. We believe Phoinix can be ported to most CORBA-compliant platforms with minimal modifications since Orbix is a full implementation of CORBA.

2. Overview of Phoinix

In Phoinix, two types of *fault-tolerant* objects are supported, namely, the *restartible objects* and the *logable objects*. As the names suggest, the restartible object resumes the service from scratch after recovering from failure, whereas the logable object resumes its service from the last check-point. We notice that many common types of hardware failures and software failures can be handled by these two levels of fault-tolerance [18]. The functionality and design of these fault-tolerant objects are presented later in this section and the implementation and interaction of the fault-tolerant objects will be presented in following section.

Phoinix is designed with a few assumptions on the operating environment. Firstly, crashed sites are assumed to operate in a fail-stop manner. (This is a common assumption in many fault-tolerant systems [14,15].) In many systems, we find that site crashes in a distributed environment are relatively infrequent and are usually independent of each other. Logable objects apply checkpoint recovery techniques to ensure the state consistency after failures. This implicitly assumes that the invocations on the logable objects are *piecewise deterministic* [5,8,16,17]. This is

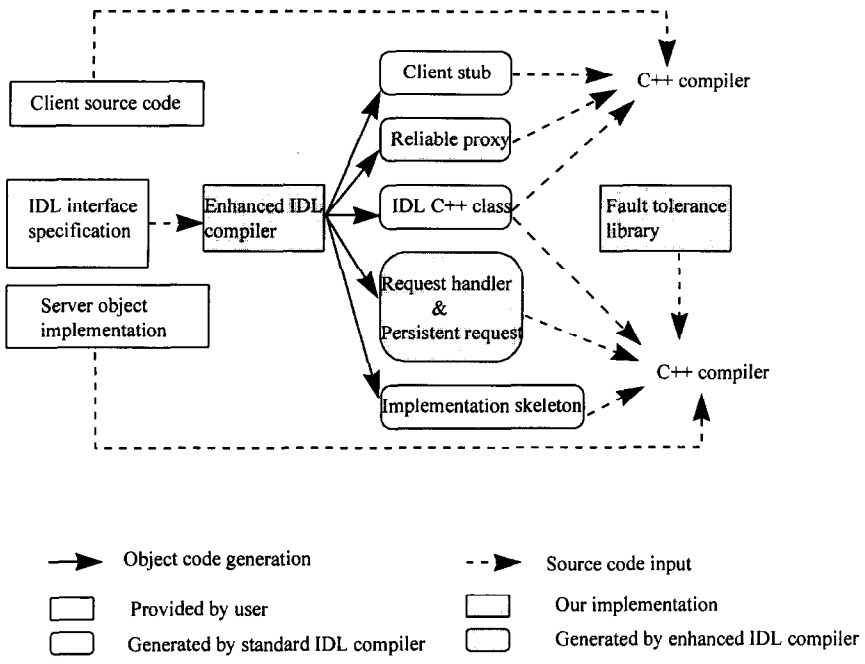

Fig. 2. The enhanced IDL compiler and application development in Phoinix.

also a common assumption in the field. Furthermore, We assume that the client invocation is not nested, i.e. the invoked object does not 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 5.2.

We are now ready to introduce the development of an application fault-tolerant object in Phoinix. Fig. 2 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.

A service object that wishes to become a restartible or logable object has to inherit the callback interface `Restartible` or `Logable` in its IDL declaration. The definition of these two callback interfaces are shown in Figs. 3 and 4. Fig. 5 illustrates the IDL declaration of an 'logable' account object. Given an IDL declaration with the inheritance of either of these callback interface, the *enhanced IDL* (EIDL) *compiler* of Phoinix generates two sets of extra codes in addition to the ordinary client stub and the implementation skeleton. These two sets of programs are the *reliable proxy* that is linked with the client and the

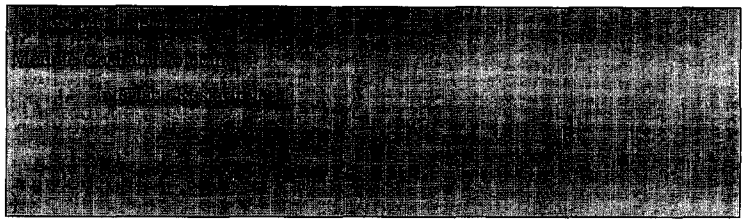

Fig. 3. IDL specification of the interface `Restartible`.

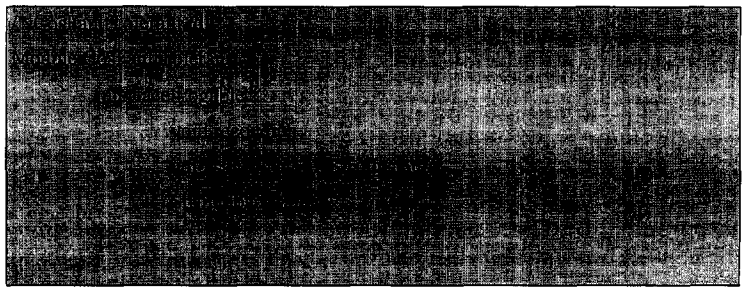

Fig. 4. IDL specification of the interface `Logable`.

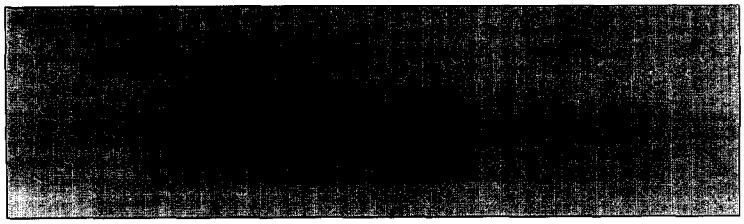

Fig. 5. IDL specification of an account object declared as a logable object.

fault-tolerant skeleton for the service object. The fault-tolerant skeleton includes two classes of objects: *persistent request* objects and *request handler* objects, as shown in Fig. 2.

Once the client object binds to a fault-tolerant service object, the reliable proxy at the client side starts monitoring that service object, and the reliable proxy is responsible for detecting object failures and then triggering the recovery process. The detection of an object failure relies on a fundamental service from ORB. Recall that the object failure can be detected by ORB so long as this object is bound by a client invocation. The client is aware of an object failure if it receives an exceptional signal from ORB after an invocation is made on that service object. Instead of raising this exception signal to the client directly, the reliable proxy intercepts this exception signal, activates another service object via ORB, binds to it, and then triggers appropriate recovery actions according to the type of the fault-tolerant object. The reliable proxy invokes `Restartible::restart()` if the new object is a restartible object, or it invokes `Logable::recover()` if the object is a logable object.

A logable object needs to perform a few fundamental operations in the provision of the recovery from a failure. These operations are: logging and replaying invocations, and saving and restoring its critical states. For each operation defined in an IDL interface of a logable object, two associated classes are needed; a persistent request class and a request handler class. Both persistent request class and request handler class are generated automatically from the EIDL compiler. We discuss the persistent request class first. The constructor of a persistent request class has the same arguments as that operation defined in its IDL interface. A static instance of a persistent request class is constructed when the corresponding operation is invoked the first time on that object implementation. This persistent object will save itself into an audit trail managed by LM automatically each time an operation invocation is completed successfully. Since it is not necessary for logable object to log every performed invocation to LM, a logable object declares only those interfaces or operations with `Logable` interface. Thus, no class of persistent request will be generated for those interface operations that do not require to be logged. A static instance of request handler class is constructed at the same time as its persistent request counterpart is constructed. The functionality of the request handler is the inverse of persistent request, where it retrieves from LM those invocations recorded in the audit trail and re-executes them in sequence when a new logable object is activated after a failure.

For a logable object, its *persistent request* object replicates all incoming invocations to LM before forwarding the invocation to the implementation skeleton. The *request handler* of the logable object will retrieve these invocations from LM and replay them when the logable object is activated by a reliable proxy (via the invocation `Logable::recover()`) to replace a failed logable object.

For either restartible object or logable object, the service object needs to implement the functions defined in the callback interfaces as shown in Figs. 3 and 4. Phoinix provides a fault-tolerance library that defines a base class of the restartible and logable objects so that corresponding service objects can inherit and reuse. For example, Fig. 6 depicts a C++ implementation of `Class Logable` in the fault-tolerance library. This base class declares a set of fundamental member functions in virtual form and implements some basic operations so that a logable object may manage its critical data members in conjunction with the LM. The detailed implementation of the fault-tolerance library is given in Section 4. Yet, we notice that when these member

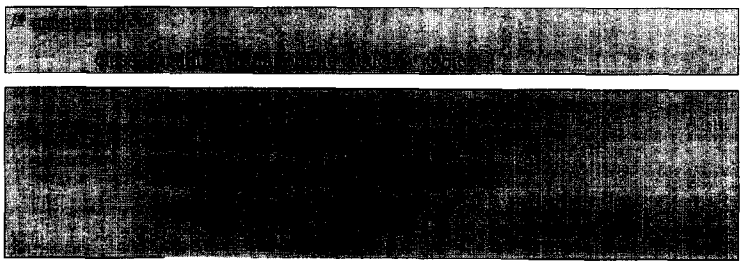

Fig. 6. C++ implementation of the logable class in the fault-tolerance library.

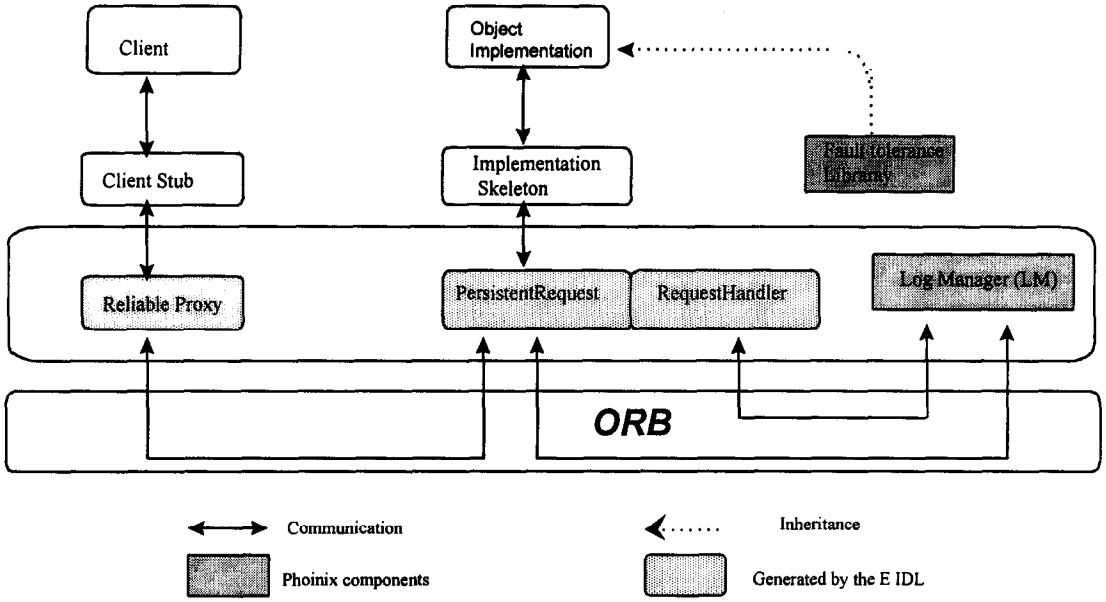

Fig. 7. The architecture of the fault tolerance layer in Phoinix.

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 Fig. 7.

3. The application of Phoinix

We have stated that the restart objects and the logable objects are supported in the 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 is devoted to the design and implementation of logable objects, since the recovery process for the restart objects is rather straightforward. It is impossible to restore the failed objects to their previous states before failures occur, which is the reason we designed another level of fault-tolerance object (the logable object): for the purpose of returning the failed object to its previous state.

3.1. The failure detection

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 is more than one object implementation that is already activated, the client randomly chooses one to bind; otherwise, the client randomly activates one. If the binding is successful, the client proceeds to making request invocations, otherwise, the client repeats the 'broadcast and bind' procedure until the binding is successful. Upon 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 successful, the new object implementation may continue the service to the clients after the appropriate recovery process (see Section 5). In order to make the rebinding operation transparent to the client, we design a *reliable proxy*, as described in the previous section, to perform these operations.

If an object implementation is declared as a restart object, the client only needs to rebind to the target object in a new object implementation, and the object implementation does not need to do any recovery. Notice that this fault-tolerance level only assures that a service object is always available to clients; but it cannot restore a failed object to its previous states before the failure.

3.2. The failure recovery of the logable objects

Now we present the interactions between a logable object, its persistent request object and request handler object, and LM during the failure recovery procedure. A logable object binds to the LM via the interface '`LogManager`' at the time when it is constructed. Fig. 8 illustrates the interface `LogManager` and its operations. The client invocations to this logable object are duplicated and sent to LM (by the *Persistent Request* object) via the

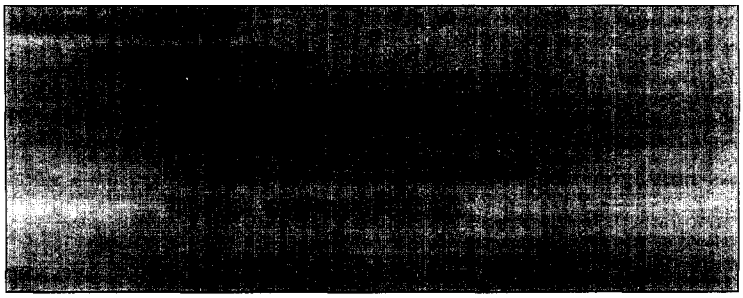

Fig. 8. IDL specification of the interface `LogManager`.

invocation `LogManager::save_invocation()`. The data structure `Invocation` contains the information of the object reference of this logable object, the method name, and the parameters. The logable object also can decide the timing as well as the frequency to checkpoint its critical data to LM via `LogManager::checkpoint()`. A logable object has the domain knowledge as to what internal states to save in LM via the invocation `LogManager::checkpoint()`. Once the checkpoint is done, the LM purges the audit trail and continues service to the logable object. Phoinix provides the base class of the logable class as shown in Fig. 6. A logable object needs to overload the member functions `Logable::save_state()` and `Logable::load_state()` to save the application-dependent object states.

If the object implementation crashes, the reliable proxy of a client object will receive an exception in a successive method invocation. Then, the reliable proxy will re-bind another logable object that implements the same service. The new logable object will first bind to the LM and initiates the rollback recovery process. The new logable object first reloads all the critical states from LM via `LogManager::recover_state()`, and this invocation brings back the last checkpoint. Furthermore, the new logable object needs to redo the requests logged in the audit trail in the original order. The static request handler object is responsible for this action, and it can retrieve the invocations since the last checkpoint via `LogManager::retrieve_invocation()`. The data structure `InvktList` is defined as an array of type `Invocation`, i.e. `sequence < Invocation >` in IDL. The implementation of the request handler includes a handler array. The request handler array maintains request handlers for the object implementation's logable objects. A request handler unmarshals the arguments of a specific IDL interface operation from a logged request, then invokes the operation on the new logable object.

Note that we have assumed that failure occurrence is infrequent in a distributed system. This implies that the possibility that the logable object and its LM both crash at the same time is very slim. The failure recovery of LM will be discussed in Section 4. Note that a logable object may serve more than one client at a time, and it is possible that each client may bind or even reactivate a different object implementations after it discovers the crash of the logable object.

4. Implementation of major components in Phoinix

4.1. EIDL Compiler

The enhanced IDL (EIDL) compiler scans the IDL interface specification file and produces fault-tolerance codes in addition to standard IDL compiler client stubs and implementation skeleton (see Fig. 2). On the client side, the EIDL produces a 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. On the object implementation 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 process for these logable objects.

4.2. Log manager (LM)

LM exports its service using the interface `LogManager`. LM maintains a reliable repository to store the checkpoints for each logable object, and it also maintains an audit trail to record invocations for that object after the last checkpoint. It is possible that there are multiple client objects to invoke the same logable object that is implemented as a multi-threaded server. LM avoids the concurrent access to the same audit trial from different threads of the same object using the *two-phase lock* (2PL) protocol. In other words, an invocation on a logable object can continue only if it is granted the lock to access the audit trail of that object. This implies that a logable object serves one invocation at any time. It is possible to support the concurrent service to multiple clients using a more elaborate implementation, though the current LM does not support this. We hope that the next generation of Phoinix can solve this problem.

LM itself is implemented as a regular CORBA object, thus we need to consider the failure recovery itself. In Phoinix, we implement LM as two replicated CORBA objects, and each is the hot standby of the other. LM objects

actively 'poll' each other to detect any anomaly. Each LM can accept invocation from logable objects, and the same invocation is forwarded to the other LM to ensure data consistency within LM. LM activates another replica LM if it detects that its running partner has crashed. We assume that the chance of two LMs crashing at the same time is very small. We also consider the case that a failure occurs in the object implementation or in the LM during the checkpointing process. We implement the *two-phase commit* protocol during the invocations from logable objects to avoid either object implementation or that the LM enters an inconsistent state.

5. 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.

5.1. Performance measurement

We designed a single experiment to explore the performance of Phoinix. The experiment involves account services whose interface definition is illustrated in Fig. 6. The account object provides typical transactional operations such as deposit, withdraw, and balance inquiry. A client object makes 200 invocations on a remote account object over a local area network. During this period, the account object checkpoints it is critical data (roughly 2K bytes in size) onto a LM residing on a different host. The account object, the client program, and the LM are all running on Sun Sparc Workstations connected through a 10B-T Ethernet. Fig. 9 depicts the experimental 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 is not surprising to observe that the overhead due to the checkpoint increases as the number increases. Furthermore, we notice that each checkpoint takes roughly 200 ms, which is a normal network response time. This suggests that the performance of the LM implementation is acceptable. We further notice that the response time for client jumps from 1 to 2.7 s if the account object becomes a logable object. The reason for this is that the persistent request object of the logable object saves each invocation to LM before returning the reply to the client. We believe the response time will be reduced by 50% if these two operations can be done concurrently; i.e. the invocation results are sent back to the client directly, while the persistent request object is saving the invocation to LM.

5.2. Extensions

In this section we will identify 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, one

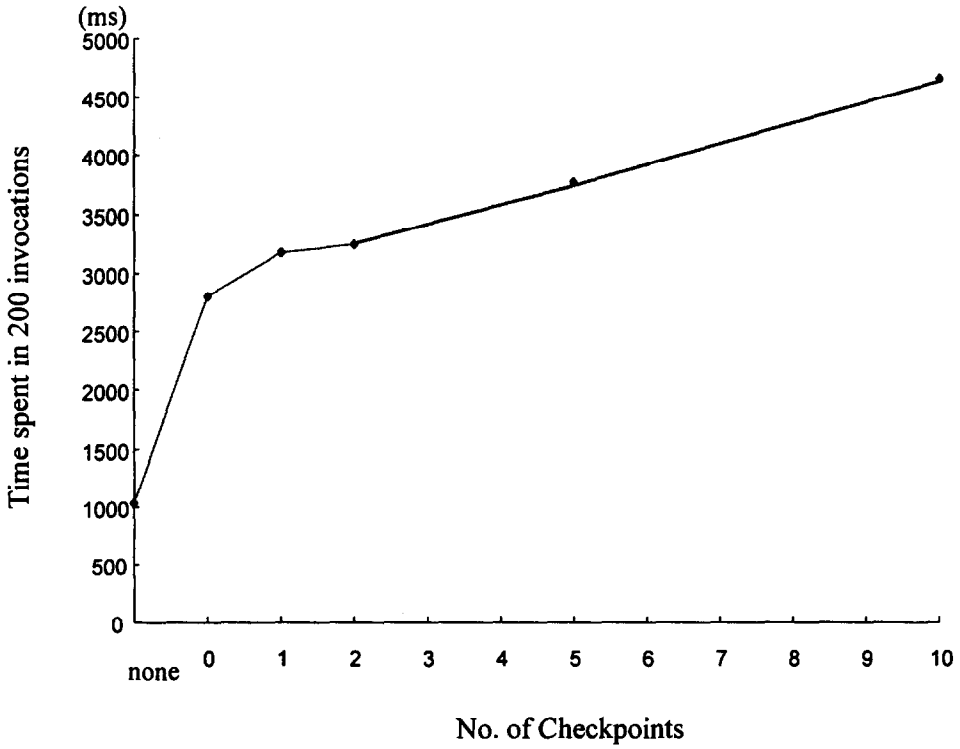

Fig. 9. The response time of 200 invocations under various check-pointing frequencies.

possible way is using a replica manager to coordinate 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 more than once. For masking transient software failures, one can define a *RequestRandomizer* class to re-order the requests saved in the audit trail before redoing requests. A programmer can redefine his own policy to re-order 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 [18]. Thus, object implementations developed with Phoinix will be able to tolerate transient software failures providing the recovery process with this mechanism. One possible approach to support this mechanism in Phoinix is to modify or overload the *PersistRequest()* in the fault-tolerance class library in such a way that the redo order differs from that in the audit trail.

6. 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 an 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. The fault-tolerance capability in Phoinix is classified into two levels: restart and rollback-recovery; where the fault-tolerance capability is enhanced as the level increases. Currently, Phoinix is ported on Orbix 2.0 and SunOS 4.2. Object services provided in the current version of Phoinix are able to tolerate hardware failures with a capability up to level two fault-tolerance, i.e. the level of rollback-recovery.

In this paper, we have introduced 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 levels of fault-tolerance: restart and rollback-recovery. We have discussed the system architecture of Phoinix, which consists of the following major components: EIDL compiler, fault-tolerance and LM. We have also described the application development environment of Phoinix, within which application object implementation can be developed with the desired level of fault-tolerance. Phoinix was designed to support three levels of fault-tolerance, as described in Section 1, although only two of them were 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 given extensive attention in the current implementation. As the experiments demonstrated, we have identified key areas where performance improvement can be made 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] K.P. Birman, Integrating runtime consistency models for distributed computing, Tech. Rep. 91-1240, Dept. of Computer Science, Cornell University (July 1993).
[3] G. Booch, Object-oriented Design with Applications, The Benjamin Cummings Publishing Company, Inc. (1991).
[4] K. Brockschmidt, Inside OLE, 2nd ed., Microsoft Press, Redmond, Washington (1995).
[5] E.N. Elnozahy, W. Zwaenepoel, Manetho: transparent rollback-recovery with low overhead, limited rollback and fast output commit, IEEE Trans. Computers 41 (5 (May)) (1992) 526–531.
[6] SOMobjects: A Practical Introduction to SOM and DSOM, IBM, International Technical Support Organization (July 1994).
[7] Orbix Programmer's Guide, IONA Technologies Ltd. (November 1994).
[8] R. Koo, S. Toueg, Check-pointing and rollback-recovery for distributed systems, IEEE Trans. Software Eng. SE-13 (1 (January)) (1987) 23–31.
[9] S. Landis, S. Maffeis, Building Reliable Distributed Systems with CORBA, in: Theory and Practice of Object Systems, R. Soley (Ed.), John Wiley, New York (April 1997).
[10] The Common Object Request Broker (CORBA): Architecture and Specification, v 1.2, Object Management Group, Inc. (December 1993).
[11] L. Peterson, N. Buchholz, R. Schlichting, Preserving and using context information in inter-process communication, ACM Trans. Computer Systems 7 (3 (August)) (1989) 217–246.
[12] R.V. Renesse, K.P. Birman, R. Cooper, B. Glade, P. Stephenson, Reliable multicast between microkernels, Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures, Seattle, Washington (April 1992).
[13] J.R. Rymer, IBM's System Object Model, Distributed Computing Monitor 8 (3 (March)) (1993) 1–24.
[14] S.K. Shrivastava, Lessons Learned from Building Arjuna Distributed Programming System, Dagstuhl Seminar on Distributed Systems (1994) pp. 17–32.
[15] R.D. Schlichting, F.B. Schneider, Fail-stop processors: An approach to designing fault-tolerant computing systems, ACM Trans. Computer Systems 1 (3 (August)) (1983) 222–238.
[16] R.E. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Trans. Computer Systems 3 (3 (August)) (1985) 204–226.
[17] R.E. Strom, D.F. Bacon, S.A. Yemini, Volatile logging in n-fault-tolerant distributed systems, in: Proceedings of the Fault-Tolerance Computing Symposium (1988) pp. 44–49.
[18] Y.M. Wang, Y. Huang, W.K. Fucks, Progressive retry for software error recovery in distributed systems, in: Proceedings of the 22nd Fault-tolerance Computing Symposium (1993).