ELSEVIER

The Journal of Systems and Software 48 (1999) 197–211

The Journal of Systems and Software

www.elsevier.com/locate/jss

A fault–tolerant object service on CORBA

D. Liang [a,*], C.-L. Fang [b], S.-M. Yuan [c], C. Chen [b], G.E. Jan [d]

[a] *Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan, ROC*
[b] *Department of Electronic Technology, National Taiwan University of Science and Technology, Taipei, Taiwan, ROC*
[c] *Department of Computer and Information Science, National Chiao Tung University, Taiwan 31151, ROC*
[d] *Department of Computer Science, National Ocean University, Keelung, Taiwan, ROC*

Received 25 April 1998; received in revised form 10 September 1998; accepted 24 October 1998

Abstract

The Common Object Request Broker Architecture (CORBA), is a major industrial standard for distributed object-based applications. Today's large-scale CORBA applications have to deal with object crashes, node failures, networks partitioning and unpredictable communication delays. Existing efforts to enhance the CORBA reliability can be roughly categorized into three approaches: integration approach, interception approach and service approach. Each approach has its own merits and prices. In this paper, we propose a service approach solution called Object Fault–tolerance Service (OFS). Solutions that adopt the service approach usually specify their service in terms of CORBA IDL interfaces. The implementations of such solutions in general do not modify the ORB infrastructure or IDL language mappings, and thus applications developed with those systems appear to be more portable. OFS differs from other service approach solutions in that OFS does not assume underlying support of reliable group communication. Applications with advance registration can rely on OFS for detection of object and node crashes, and for customized recovery. In this paper, we first present the service specification of OFS. We then give the system architecture of an OFS implementation. This OFS implementation is developed on the Solaris 2.5 platform and with IONA's Orbix 2.0. The performance evaluation of the OFS implementation is also presented. The preliminary experiments indicate that OFS overhead is minimal and client objects experience little response delay when a service object is under OFS surveillance.

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

1. Introduction

With the advance of computer and communication technology, distributed computing systems have become increasingly popular in recent years for their better price-performance ratio. As the number of installation increases and application domains expand, the complexity of the distributed software developed for these systems explodes as well. Many of these distributed softwares are designed to handle critical tasks that involve the safety of human lives. This trend has led to an urgent demand for highly reliable distributed softwares or techniques to develop them. Software fault–tolerance is an approach for this purpose. Since heterogeneous distributed systems tend to be less reliable in nature, software fault–tolerance for distributed softwares becomes an important design issue.

Fault–tolerance design for distributed systems requires comprehensive knowledge of all aspects of system engineering, from detail 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 tools are in great demand. Existing fault–tolerance tool-kits such as ANSA (APM, 1993), ISIS (Birman, 1993), PSYNC (Peterson et al., 1989) and HORUS (Renesse et al., 1992) are now widely available. These products however are all designed for *process-based* distributed applications where process is the basic entity subject to monitoring and recovery.

Recently, object-oriented design (OOD) (Booch, 1991) has been widely applied to software design and development since, it offers greater potential in portability and reusability. Several distributed object-oriented middlewares have

* Corresponding author. Tel.: + 886 2 2788 3799; fax: +886 2 2782 4814; e-mail: drliang@iis.sincia.edu.tw

0164-1212/99/$ - see front matter
PII: S0164-1212(99)00058-8

been proposed lately, notably, Object Management Group's (OMG) CORBA (OMG, 1993), Microsoft's DCOM (Brown and Kindel, 1996; Redmond, 1997) 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, offer adequate support for 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 necessarily 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 runs 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 fault–tolerance support within these distributed middlewares.

In this paper, we propose a fault–tolerance service over CORBA ORB. Though the proposed service in this paper is based on OMG's CORBA model, we believe similar service or concept can be ported to DCOM or JAVA RMI. In Section 1.1, we briefly review the CORBA architecture. In Section 1.2, we compare the existing efforts to enhance the reliability of CORBA ORB or CORBA applications running on top of ORB. Section 1.3 gives the motivation and claims the contributions of this research.

1.1. Introduction to CORBA

The CORBA reference model consists of four major components: object request broker (ORB), common object service specification (COSS), common facility architecture (CFA) and application objects. An object that implements a service specification and exposes its service to clients over the network is called 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 ORBs. A server process may contain multiple service objects. ORB serves as a software interconnection bus between clients and service objects. COSS defines several useful 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 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.

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 a 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 from the client object the complex operations, such as remote request preparation, parameters 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.

1.2. Related works

Numerous research efforts in recent years have been devoted to enhance the fault–tolerance capability of CORBA ORB and/or its applications. They have been categorized into three approaches: *integration approach*, *interception approach* and *service approach* (Narasimhan et al., 1997).

The **integration approach** attempts to layer ORB on top of a reliable multicast communication subsystem. Examples are *Orbix+Isis* (IONA and Isis, 1994) and *Electra* (Landis and Maffeis, 1997). **Orbix+Isis** is the first commercially available system that supports the creation of fault–tolerant CORBA-compliant applications. The fault–tolerance of an object implementation is gained through active replication with reliable multicast from the support of group communication layer at lower level (Birman, 1993). **Electra** is another implementation of reliable ORB but differs from Orbix+Isis mainly with respect to the ease of use and the adaptability of ORB to various communication subsystems. Applications use the multicast ORB via the *application program interface* (API) as it would have normally used in conventional ORB. This approach may require modification of the ORB interface and language mapping. This may lead to a less portable implementation of both server and client applications.

The **interception approach** is used by the **Eternal** system (Narasimhan et al., 1997) to construct a reliable ORB environment. The idea is to capture system calls made by the client object to a low level communication system such as TCP/IP and map these system calls to the reliable multicast subsystem (Moser et al., 1996). Eternal differs from Orbix+Isis or Electra in that Eternal does not modify ORB or language mapping. Thus, it ensures the transparency of fault–tolerance from applications. Another example of the interception approach is the Phoinix system (Liang et al., 1998). Object implementations in Phoinix obtain fault–tolerance by inheriting *fault–tolerance interface* specified by Phoinix at the IDL level. A proprietary IDL compiler (called Extended IDL) parses object's IDL interface and generates object stubs and skeletons with extended fault–tolerance functionality. The Phoinix stub intercepts and analyzes exceptions raised from ORB and performs object rebinds. The Phoinix skeleton, on the other hand, records invocation logs and data checkpointing. The advantages of the Phoinix approach are that it ensures transparency from both client and server implementations, and at the same time it does not require modification to ORB. The drawback though is that it introduces a proprietary IDL compiler, and this might reduce the portability of the Phoinix system.

The **service approach** defines a set of objects and their interfaces to provide fault–tolerance. As an example, **GCS** (Felber et al., 1996) provides a group communication service with which client's requests and server's replies are coordinated. GCS defines a set of IDL interfaces so that replications of server implementations can form a group with a single group identifier (object reference of the group). A client thus can make invocations and retrieve replies with this single identifier. Group communication is adopted as the communication basis in GCS to achieve fault–tolerance regardless of the nature of fault–tolerance strategy. This implicitly contributes to the non-transparency issue on the client's side and reduces the flexibility at the server's side. Another work that uses the service approach is FTS-ORB (Sheu et al., 1997). In this work, the authors use a checkpoint and message logging approach to provide fault–tolerance to an application. However, the responsibility of failure detection lies with the client and the responsibility of handling its replicas lies with the application itself.

1.3. Objectives and contributions

Software fault–tolerance often requires some degree of redundancy in both time and space domain. Deploying multiple copies of the same application as backups is a common approach. Backup schemes can be categorized in three classes; cold, warm and hot backups (Chen and Bastani, 1992). A cold backup is usually not running in the systems until its primary application fails. In the warm backup scheme, several backups are running as the primary application is running, but only the primary application responds to client's requests. The primary may checkpoint its states to warm backups from time to time. One of the warm backups is promoted to primary if the primary application is found crashed. In the hot backup scheme, all copies of application respond to the client's requests in some type of synchrony. This scheme maintains the highest level of reliability and availability, but also suffers the most performance degradation.

Most of the related research works described above focus on hot backup approach using group communication. However, various research reports suggest that software implemented fault–tolerant systems in many application domains use cold backup (fail-over) or warm backup approaches (Huang and Kintala, 1993). Furthermore, most of the previous research works using integration or interception approach require changes in the ORB layer, in the IDL language compiler, or in the communication layer in order to provide mechanisms for group communication and membership management. As a result, their solutions may be difficult to port to different operating systems (such as UNIX and Windows NT) and to different ORBs (such as IONA Orbix or SUN NEO). In this paper, we propose a fault–tolerance service called *Object Fault–tolerance Service* (OFS) that supports both cold and warm backup schemes. The implementation of OFS does not require any modifications to ORB or IDL language mapping, nor does it assume any underlining group communication support.

The rest of this paper is organized as follows. In Section 2, we present the interface specified in OFS and their functionality. Section 3 presents an implementation of OFS on the Solaris platform using IONA's Orbix. We also discuss a few implementation issues. The performance evaluation of OFS implementation is discussed and presented in Section 4. The concluding remarks are in Section 5.

2. The design of the fault–tolerance service interfaces

2.1. Overview of OFS

We believe a fault–tolerance object service over CORBA platform shall provide a few features that include *failure detection, replica management, recovery handling* and *consistency management of object states*. A primary object or

other manager object may register a list of backup objects to OFS with desired fault–tolerance degree. It is the responsibility of OFS to monitor the liveliness of all active objects, primary or backup, and to maintain enough number of replicas in the networks. The OFS also needs to coordinate the recovery process whenever an object failure is detected. Furthermore, a primary object shall be able to checkpoint its critical internal states to its backups via OFS. OFS shall offer mechanisms with which a primary object can select appropriate checkpointing policy most suitable to that application. These services are realized through three major interfaces in our OFS: *ReplicaManager*, *Primary* and *Secondary*.

ReplicaManager exposes major services offered by OFS. The ReplicaManager object accepts registration of primary and secondary objects, it selects and activates enough number of replicas as required, and it coordinates the recovery process should an object crash be detected. In addition to ReplicaManager, OFS defines two callback interfaces, one for primary objects and one for secondary objects. Objects that support the primary interface allow other backups to access its critical states and message logs whereas objects that support secondary interface allow ReplicaManager to activate them as backups or to promote them as primary. The detailed definition of these interfaces will be covered in the next section.

The interfaces and various objects involved in OFS are shown in Fig. 1. The dark circles represent the objects, including the application objects, application manager, etc. An arrow with a vertical bar is used to show that the target object supports the interface named next to the arrow; and those clients holding an object reference of this type can invoke the operations defined by the interface. The dashed lines define the machine boundaries. We are now ready to describe the three IDL interfaces of OFS.

2.2. Interfaces

In this section, we shall describe the three interfaces defined in OFS. The interface **ReplicaManager** defines the view toward OFS. A primary object may register itself and a list of backups to ReplicaManager via *register_secondary( )*. This operation takes two input parameters and one output parameter. The input parameter *ObjList* specifies a list of object references of backup objects whereas *degree* expresses the desired level of reliability. For example, object O_1 is the primary object and O_2, O_3 & O_4 are its backups. The ReplicaManager probably would activate O_2 and O_3 if O_1 invokes *ReplicaManager::register_secondary*($\{O_2, O_3, O_4\}$, 2, key). The only output parameter *group_key* is used as an index for primary or backup objects to perform group membership-related operations. ReplicaManager defines five such operations: *insert_secondary( )*, *delete_secondary( )*, *get_secondary( )*, *get_active_secondary( )* and *get_primary( )*. The first two methods can be used to update the backup list. The method *get_active_secondary( )* may be invoked when a primary wishes to checkpoint its states or logs to its active backups. (Note that active backups may vary over time due to crashes.) Finally, the method *get_primary( )* can be used to get the current object reference of primary when a backup needs to get the current object state when it is activated as warm backup. The definitions of these methods are listed below.

```
// IDL
interface ReplicaManager {
   Boolean register_secondary(in ObjList secondary_list, in degree, out unsigned short group_key);
   Boolean insert_secondary(in unsigned short group_key, in Object secondary);
```

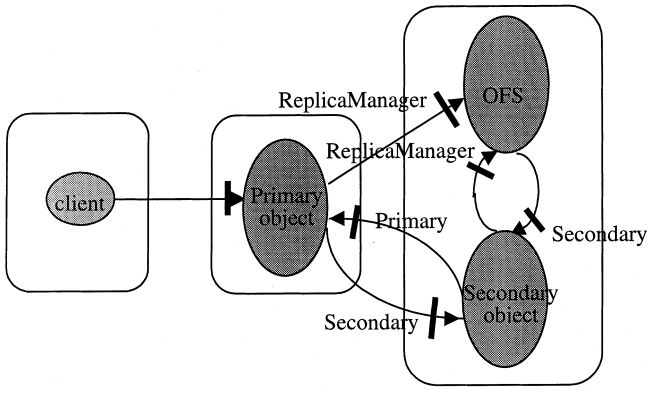

Fig. 1. Structure of OFS.

```
    Boolean delete_secondary(in unsigned short group_key, in Object secondary);
    Boolean get_secondary(in unsigned short group_key, out ObjList secondary_list);
    Boolean get_active_secondary(in unsigned short group_key, out ObjList secondary_list);
    Object get_primary(in unsigned integer group_key);
}
```

Interface Primary provides the methods to access the internal states and logs of the primary object. A secondary object may get its primary object interface through ReplicaManager and synchronize with the primary when it is activated. Two methods are defined: *get_state( )* and *get_log( )* where the method names suggest their functionality. Notice that the input parameters are all *String* types, which implies that semantics of these data are application dependent.

```
//IDL interface Primary
interface Primary{
    Boolean get_state(out String objectstate);
    Boolean get_log(out String message_log);
}
```

Interface **Secondary** provides the view of a secondary object. Upon the registration of the primary object, RM can form a replica group by selecting enough secondary objects. OFS then activates these secondary objects via invoking *Secondary::start( )* and passes the *group_key* for further reference of the group. (See the definition shown in the box below). OFS can activate any of the secondary objects from the list using the same procedure if it detects a crash of an active secondary object. Furthermore, RM can promote an active secondary to primary by invoking *Secondary::turn_primary( )* if it detects that the primary is crashed. To maintain a consistent view, the primary may checkpoint its states or save its current message logs to secondary backups via methods *set_state( )* and *set_log( )*.

```
interface Secondary{
    Boolean start(in unsigned short key);
    Boolean turn_primary();
    Boolean set_log(in String messageLog);
    Boolean set_state(in String state);
}
```

2.3. The interaction of interfaces

We now describe the interaction among all components in our system in this section. We first describe the normal execution flow without failures. Then we go on to discuss how the protocol reacts to keep the execution going when failures occur.

2.3.1. Object registration

A primary that requests fault–tolerance support from OFS requires pre-registration via interface *ReplicaManager*. A primary can register with a list of secondary objects (using their object references) and the degree of fault–tolerance. OFS then activates enough number of secondary objects to form a secondary replica group and to continue monitoring these active objects afterwards. The implementation of the object monitoring in Orbix will be covered in the next section.

Fig. 2 illustrates interactions among objects when a primary registers to OFS. Each vertical line represents the activities of an object, whose type is given on top of each vertical line. Time flows from top to the bottom. A box indicates an operation active inside that object. A horizontal line with arrow indicates object invocation whereas dashed arrows represent service specific operation such as object creation or message delivery.

Suppose a primary object O_1 registers its secondary objects O_2, O_3 and O_4 to OFS and requests a degree 1 fault–tolerance. Upon receiving such invocation, OFS activates O_2 as an active backup by invoking O_2*::start(key)*. The secondary backup O_2 can synchronize its state with the primary via method call O_1*::get_state( )*. OFS shall keep monitoring O_1 and O_2 in order to detect possible object crashes. The failure recovery process coordinated by OFS will be discussed later. During idle periods, O_1 may wish to checkpoint its states or logs to its active backups. O_1 first gets the object references of its active backups from OFS via method call *ReplicaManager::get_active_secondary( )*; in our example, the only active backup of O_1 is O_2. Then O_1 completes the checkpoint by invoking O_2*::set_state( )* or O_2*::set_log( )*.

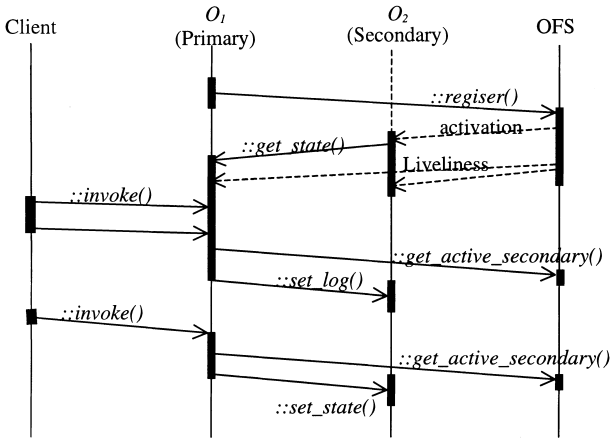

Fig. 2. Object activities during service registration.

2.3.2. The failure recovery

As discussed earlier, one of OFS responsibilities is to detect object failures. Fig. 3 illustrates object interactions involving failure recovery of the primary object. Upon the detection of primary failure, OFS examines the corresponding replica group and finds out that O_2 is the only active backup. OFS thus promotes O_2 from secondary to primary by a method invocation O_2*::turn_primary()*. Furthermore, OFS searches the remaining inactive backup, i.e., O_3 and O_4 and randomly picks O_4 to activate as a new secondary, as shown in Fig. 3 . Object O_4 then synchronizes its state with the current primary O_2, and the system continues. The crash recovery of a secondary backup is similar.

2.4. Programming example

In this section, we demonstrate the code development of an account server that incorporates with OFS to enhance its reliability. The Account server supports the interface "account" whose IDL definition is given below. This interface declares two methods, *makeLodgement()* and *makeWithdrawal()*. Client may save funds into this account or retrieve funds via method calls *::MakeLogement() and::MakeWithdrawal()*.

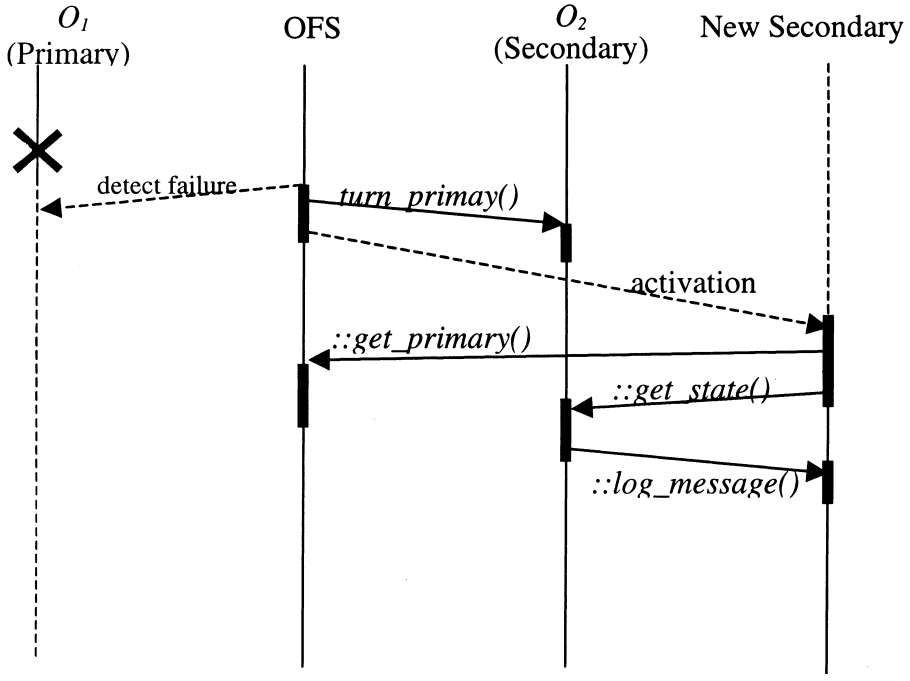

Fig. 3. The crash recovery of the primary object.

```
interface account {
   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
}
```

Suppose an object wishes to implement the *account* interface and also obtains service from OFS, this object needs to follow a three-step procedure. First, let interface *account* inherit two callback interfaces defined in OFS, *primary* and *secondary*. This inheritance allows OFS to activate an account object either as a primary object or backup object. The IDL definition of *account* now becomes:

```
/**account.idl*/
include ''OFS.idl''
interface account: OFS::Primary, OFS::Secondary{
   void makeLodgement(in float f);
   void makeWithdrawal(in float f);
}
```

The object implementation then needs to implement not only the methods defined in interface *account* but also those methods in *Primary* and *Secondary*. They are *get_state( )*, *get_log( )* (*Primary*); *start( )*, *turn_primary( )*, *set_state( )*, *set_log( )* (*Secondary*). The sample code of implementation is shown in Appendix A. The final step is to register to OFS when the account object is started. The registration can be done by the application itself or other managing AP. A code excerpt from a sample program of account server is shown in Fig. 4. In this example, the AP constructs an account object O_1 and obtains the object reference of two account objects O_2 and O_3 (possibly remote objects). The AP then binds to the RM and registers O_1 as the primary object and O_2, O_3 are secondary backups with fault–tolerance degree 1. After the registration, RM may activate either O_2 or O_3 as the running backup (to satisfy the requirement of fault–tolerance degree). Furthermore, both O_1 and the running backup are under protection against failures.

Fig. 5 illustrates the development of the account server that incorporates with OFS. We notice that a client program invokes a fault–tolerant account object the same way as it would invoke an ordinary account object. The client is not aware of the fact that object O_1 is fault–tolerant.

3. OFS implementation

We implemented an OFS prototype using IONA's Orbix 2.0 (IONA, 1994) and on Solaris platforms. We assume that object crashes or host crashes behave in a fail-stop manner; i.e., a failed entity completely halts in such a way that OFS can detect that this failure has occurred. This is a common assumption adopted in many fault–tolerant systems (Schlichting and Schneider, 1983). We also assume that the Orbix ORB is reliable in the sense that all invocation deliveries via ORB are reliable. In this section, we present the system architecture of this OFS prototype, and later discuss some implementation experience we have gained.

3.1. The system architecture

OFS is designed to protect objects against failures such as object crashes or host crashes. The robustness of OFS implementation itself also needs to be considered in our design. The system architecture is shown in Fig. 6. OFS

```
/**Server.cpp*/
include "account_i.h"
main(){
      account_i O2, O3;
      ReplicaManager_var rm;
      account_i O1=new account_i(0);
      ObjList s_list = new secondary_list(O2, O3);
      rm= ReplicationManager::_bind(":RM","");
      rm->register_secondary(s_list, 1, g_key);
      CORBA::Orbix.impl_is_ready("");
}
```

Fig. 4. The sample code of a server.

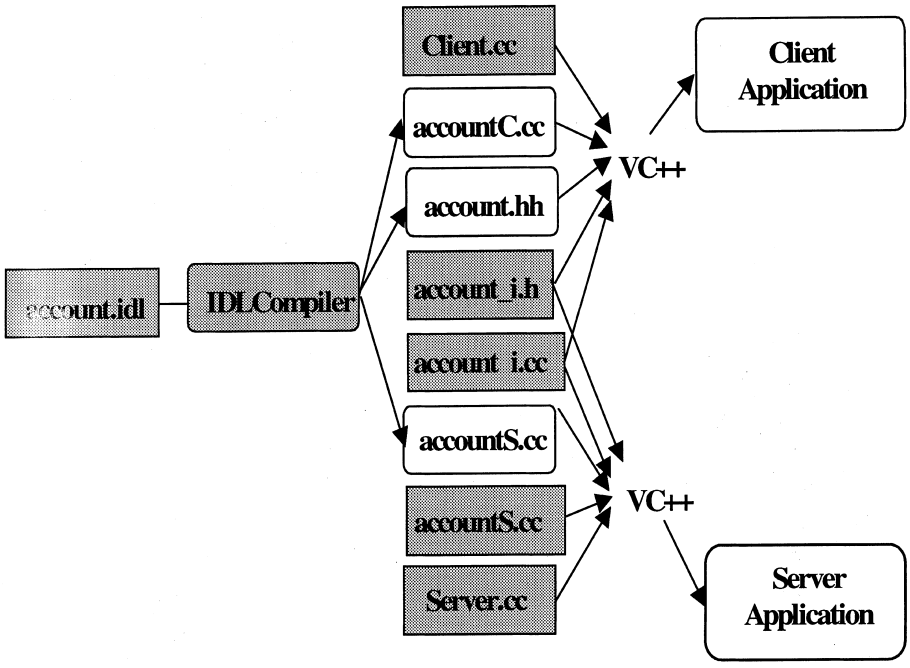

Fig. 5. The application development environment.

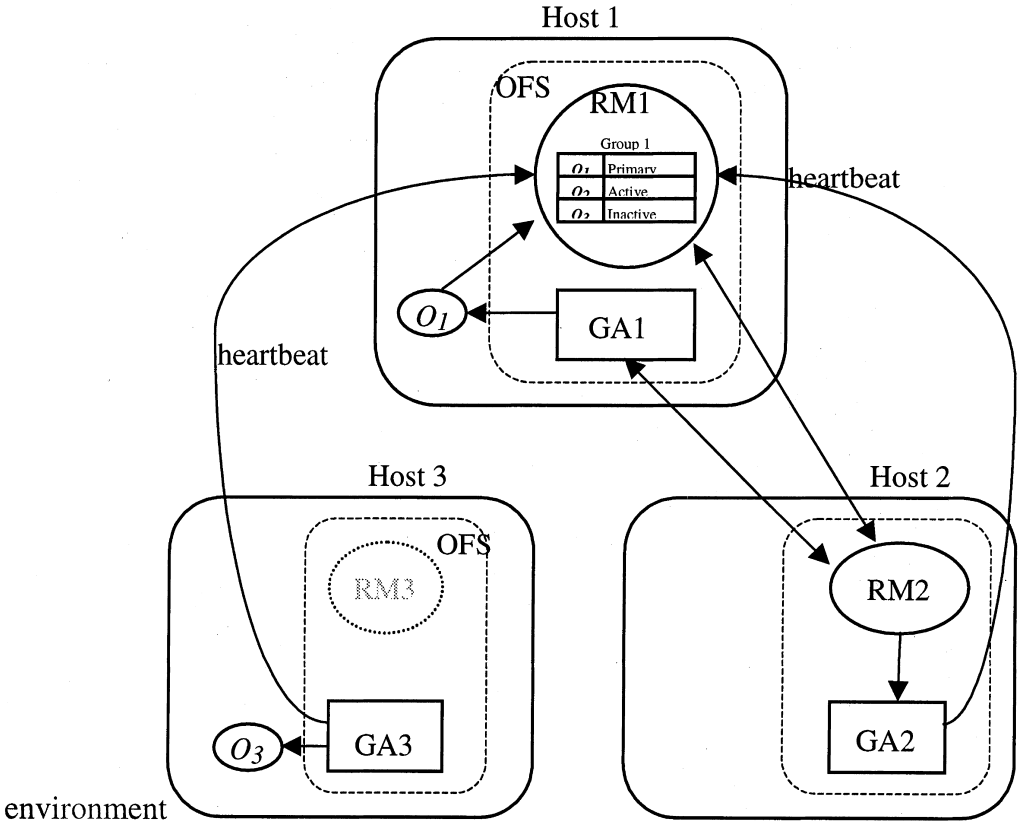

Fig. 6. The system architecture of OFS implementation.

implementation consists of two objects: *RM* and *Guardian Angel* (GA). RM exports the interface *ReplicaManager* and accepts registration from application objects that requires fault–tolerance support. RM also manages each replica group to ensure a large enough number so that enough number of replicas are up and running in the networks. GA is

an internal object and is invisible from external clients. The major functionality of GA includes monitoring active objects, activating backup upon request from RM and monitoring host crashes. As shown in Fig. 6, our implementation installs an OFS on each host. GA in each OFS is actively polling all objects running locally. There are only two active RM in the networks and each is a hot backup of the other. To detect host failures, GA periodically sends heartbeats, on behalf of the host in which it resides, to the primary RM. The host is declared dead if a few heartbeats from the GA to that RM are missing.

3.2. Failure recovery

In the next section, we discuss how OFS handles three types of failures: object crash, host crash and the crash of RM. We use the same example presented in the previous section to illustrate the interaction between RM and GA. When object O_1 registers to RM, RM informs GA in Host 1 to monitor O_1 and decides to activate O_3 as the warm backup. GA in Host 3 now is monitoring O_3. This scenario is shown in Fig. 6.

3.2.1. Failure recovery of an object

Our OFS implementation takes advantage of an Orbix proprietary API *Orbix._non_existent( )* to detect whether a CORBA object is operational. In Orbix implementation, the *basic object adapter* (BOA) of an object registers its live object reference to Orbix ORB when this object is first instantiated. Thus Orbix ORB can detect whether an object reference in the networks is still valid when the method *::_non_existent( )* is invoked. As shown in Fig. 7, GA3 informs RM1 as soon as it detects the crash of O_3 (Steps 1 and 2). RM1 then checks its internal repository and finds that O_3 is a warm backup of O_1. It thus selects O_2 to activate as a new backup via GA2. GA2 activates O_2 via calling O_2->*start( )* so that O_2 can perform its initialization as a backup. Finally, RM1 updates its internal repository synchronously with RM2. We notice that Steps (3)–(5) have to be atomic.

3.2.2. Failure recovery of a host

One of the functionality of GA is to send its heartbeats to RM on a periodical basis so that host crashes can be identified. GA is implemented with self-recovery capability using the techniques proposed in Huang and Kintala (1993). It is assumed that GA is robust. Therefore, missing heartbeats from a GA over a period of time are considered to show that the host on which that GA resides has crashed. As shown in Fig. 8, *RM1* declares that Host 3 has been down if GA3 has been failing to send heartbeats to it. *RM1* then needs to figure out all running objects on Host 3 by checking its internal repository. In our example, O_3 is the only active object on Host 3. RM1 further finds that O_3 is a warm backup of O_1 and thus selects O_2 to activate as a new backup via GA2. GA2 activates O_2 via calling O_2->*start( )* so that O_2 can perform its initialization as a backup. Finally, RM1 updates its internal repository synchronously with RM2. We notice that Steps (2)–(4) have to be atomic.

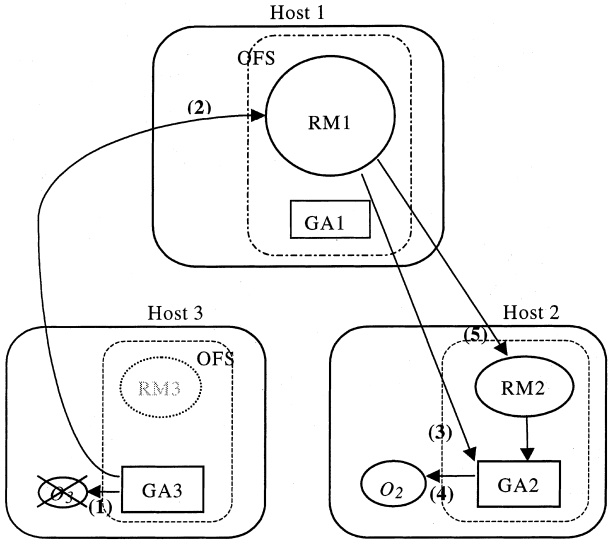

Fig. 7. The recovery of an object failure.

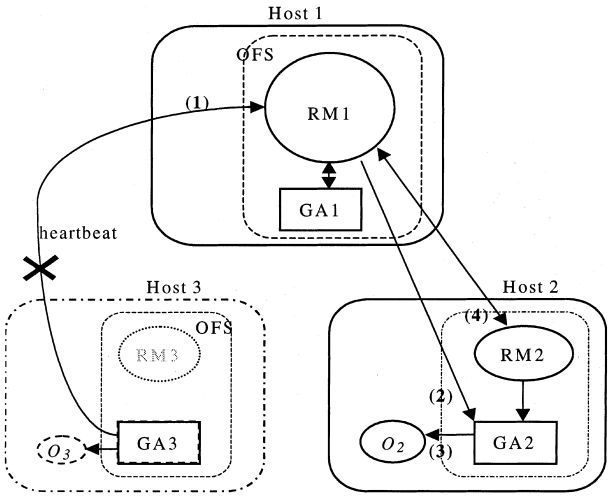

Fig. 8. The recovery of a host crash.

3.2.3. Failure recovery of RM

RM receives heartbeats from GAs at each host to detect host failures. This design leads RM as a single point of failure. Therefore, the design of RM itself must be fault–resilient. RM is implemented with self-recoverable codes, using the techniques proposed in (Huang and Kintala, 1993) as discussed earlier, so that it can bypass software failures. It, however, does not prevent RM failure from host crash. We solve this problem by replicating a hot backup of RM on another host. This is done as illustrated in Fig. 9. GA1 sends heartbeats to RM2 so that RM1 crashes (as do all other active objects on Host 1) can be detected by RM2. If it does happen, RM2 becomes the new RM and it starts the recovery process as shown by steps (2)–(5) in Fig. 9. RM2 first informs all GAs the fact that it is the new RM so that GAs redirect their heartbeats thereafter. RM2 then selects its hot backup. In this example, RM3 is activated. GA2 now sends heartbeats to RM3 so that RM2 will not be the single point of failure. Finally, RM2 start recovering all active objects running on Host1 using the same recovery procedure of host crash as discussed above. Note that Steps (3)–(5) have to be atomic.

3.3. Other implementation issues

As discussed earlier, an object is required to register itself and the object references of its backup to OFS if it wishes reliability support. Backup objects may be instantiated or inactive at the registration time. Therefore object references have to be persistent, i.e., a client can connect to an object with its persistent object reference regardless of its in-

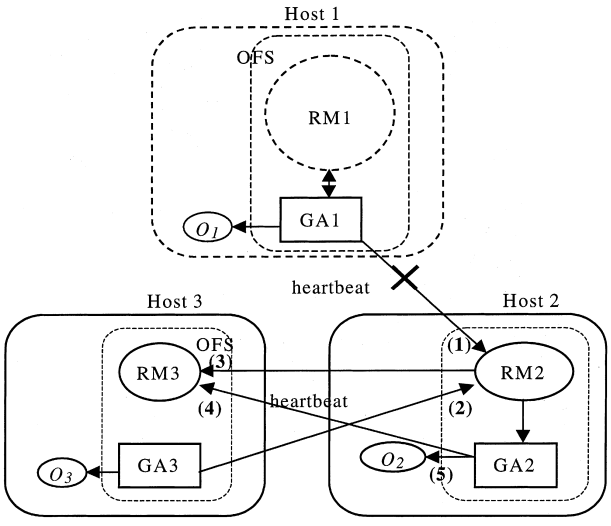

Fig. 9. The recovery of RM.

stantiation or existence in the networks. (Note that object reference of CORBA IOR or DCOM object reference is not persistent.) Orbix ORB supports persistent object reference with two ORB API; *CORBA::ORB::object_to_string()* and *CORBA::ORB::string_to_object()*. The first API is to convert a live object reference to an Orbix proprietary string representation whereas the second API does the opposite. OFS accepts string typed object references when an object does the registration so that GA may activate or instantiate a backup object whenever necessary.

All the objects that register to OFS are under supervision and protection, however, their object references are not fault–tolerant. Consider a client object holds an object reference of a reliable server object which later crashes; this object reference becomes invalid even if this server object is recovered and is replaced by its backup. Exceptions will be raised if the client object attempts to make invocations using that object reference. In the current OFS implementation, automatic fail-over of object references at the client side is not supported. One possible approach to circumvent this deficiency is to install proper exception handling routines in the client program.

Fig. 10 demonstrates this approach using a sample client program implemented in Orbix environment. The client application invokes an object that supports the *account* interface. As shown by the code segment in bold face, the client always requests ORB to discover another valid object reference if it discovers the current object reference of the account object is no longer valid. An operational account object shall soon become available after the primary object crashes if this account object is protected by OFS. We believe it is a good practice to check all possible exceptions after each remote object invocation.

4. Performance evaluation

In this section, we evaluate the system performance of our OFS implementation. The OFS is implemented on Solaris 2.5 and IONA Orbix 2.0. The performance experiments are done using Sparc connected with 10M Ethernet. One copy of OFS is installed in each node in the networks. Objects that implement and export *account* interface are used as server objects in all our experiments. The IDL definition of the interface *account* is given in Section 2.4. The scenario of all experiments is the same. We let an account object to register itself and all backups, one at each node, to OFS with fault–tolerance degree of one, i.e., there exists one primary and one warm backup at any point of time. A client object then binds to the primary account object and makes invocations repeatedly.

```
/** client.cpp – client program that invokes service from account object */
...
int main(int argc, char **argv){
        account_var aVar;
        try{                                        // bind to an object that supports account interface
        }
        catch(CORBA::SystemException &sysEx) {
        }
        catch(...){
        }

        while (True) {                              // make invocations to this account object
                try{
                        aVar->makeLodgement(10);
                }
                catch (const CORBA::SystemException &se) {// detect object crashes
                        aVar = account::_bind();    // rebinds to its replacement until success
                }
                break;
        };
}
```

Fig. 10. The sample client code that deploys exception handling routines.

We measure the performance of OFS in four different categories; the *response time* to detecting a failure (object crashes or host crashes), *system load overhead* due to OFS daemons, *response time delay* of client invocations and *checkpointing overhead* (from primary object to its backups). Each experiment is repeatedly performed numerous times to collect a large enough set of experimental results so that a measure with confidence level of 95% can be obtained using statistics methods. The reason that we apply the statistics method is to enhance the validity of our measured data. It is usually the case that we observe two different measures when we perform the same experiment on the same system twice. This is due to various measurement errors, such as clock resolution, etc. A typical approach to minimize these measurement errors is to repeat the same experiment many times so as to collect a large set of results. We then apply a standard statistics method to obtain a trustable measure, such as an average of that set, and to obtain the confidence level of such a measure. Note that all results reported in this section are obtained with 95% of confidence interval with interval half-widths of less than 3%.

Next, we present the experimental results and discuss the OFS system performance in the context of the four categories. The response time for object failure detection is defined as the time span from the primary object crash to the warm backup object resume the primary responsibility. In OFS design, GA detects the possible object crashes via active polling. Therefore, the polling frequency primarily determines the failure detection time. We vary the polling period from 20 s to 5 min in our experiments. The experiment results show that the failure detection time is roughly half of the polling period, which agrees with the theoretical values.

Next we consider the system overhead to machine load. We recall from Section 3.1 that a GA object is activated at each node with OFS deployment, and there exists an RM object and one hot backup in the networks. All GA send heartbeats to RM on a periodical basis, regardless of the number of application objects (served by OFS). Furthermore, GA actively polls all activated local objects (registered to OFS). These activities all contribute to extra computation load. We therefore design an experiment that has one client object repeatedly making invocations to the account object, and thus CPU utilization approaches near 100%. We then measure the CPU utilization of a node that has the primary RM as well as 1000 active service objects running on it. The GA polling periods ranges from 20 s (a very aggressive polling policy) to 5 min. As shown in Fig. 11, the experimental results show that the overhead due to OFS daemons is no more than 3.3% with 20 seconds polling period when monitoring a very large group (1000) of objects.

Clients experience delay when making invocations to the primary account object if it is checkpointing its internal states to its backups. The efficiency of state checkpointing thus has a great impact on the response time delay of client invocations. The experiment to measure the checkpointing efficiency is done in two steps. The first step is to record the response time for a primary object to checkpoint its backup at the same host, whereas the second step is to checkpoint its backup at a different host. The first step helps us understand the local overhead due to OFS implementation as well as Orbix ORB. The second step identifies the networking overhead. Notice that the purpose of these experiments is to analyze the overhead solely due to the OFS implementation.

In both experiments, we let the primary account object periodically checkpoint data of size from 1 to 512 KB to its backup when there is no pending invocation to this object. This is known as *offline checkpointing* policy. The checkpoint periods range from every 10 s to every 10 min, which is a typical checkpoint frequency. The experimental results are shown in Fig. 12. The results indicate that local checkpointing as well as remote checkpointing are both quite efficient. Considering checkpointing 128 KB data, for example, local checkpoints take roughly 17 millisecond (ms). Recall that a null object invocation in Orbix takes roughly 1.3 ms. Checkpointing the same amount of data to a remote object takes 126 ms over a lightly loaded Ethernet. This indicates that most of the time is network transfer time, thus OFS implementation imposes very light overhead.

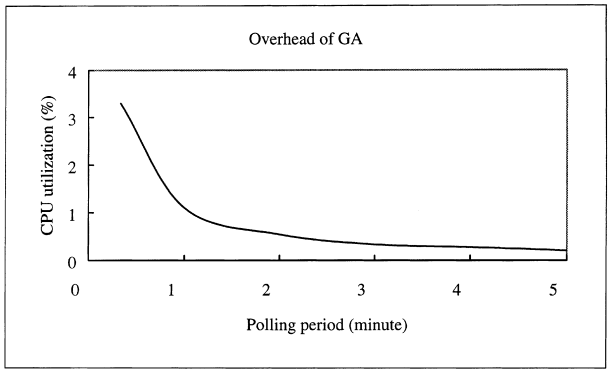

Fig. 11. The overhead of GA.

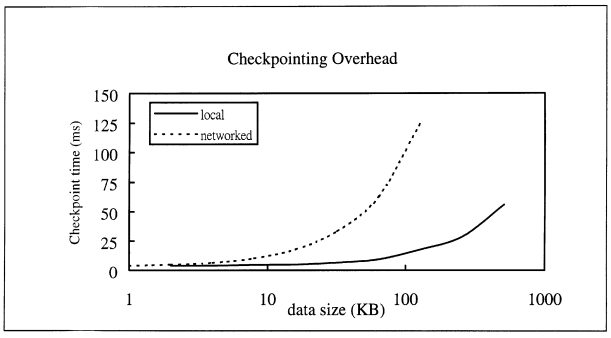

Fig. 12. The checkpointing overhead.

We also perform another experiment where we measure the response time of client invocations with or without OFS support. The experiments are set with CPU utilization from 5% to 50% and network utilization from 10% to 80%. The primary account object checkpoints 128 KB of data to its remote backup every 20 s in those cases with OFS support. The response time of local invocations ranges from 1.32 to 1.35 ms depending on the server load, whereas the response time of remote invocations ranges from 1.77 to 5.64 ms depending on the network traffic. We compare the response time from account object with or without OFS support and find that the difference is less than 3%, almost negligible. We find those factors that contribute to this outcome are *low overhead of OFS daemons*, the *checkpoint frequency*, the *offline checkpoint policy* and the *implementation of account objects*. The time scale of the checkpoint period is in minutes whereas the time scale of the checkpoint time is in milliseconds. This implies that checkpointing over network generates light traffic on an average. Moreover, the chance is very slim that an incoming invocation will arrive during the checkpoint period of a server object, in particular when the server load is less than 50%. Furthermore, the objects providing account services and the object performing checkpoints are implemented as different threads with different priorities in account server. The checkpoint thread is assigned with a lower priority to avoid interference with service threads. We do witness cases where the client invocation experiences extended delay in cases with or without OFS support when server load approaches 100% or the network is very congested for a longer period of time, but these are unusual cases. More importantly, the congested network is not due to the overhead of OFS in those cases.

Furthermore, we compare the response time of client invocations in OFS with other existing fault–tolerant ORB, including Electra and Phoinix. We let the client object and the primary account object reside on the same host. For a fair comparison, we also let the `LogManager` (LM), an important daemon object in Phoinix, run on the same host too. The client invocations are all local invocations in all three systems. The feature of Electra is turned on to emulate the warm backup scenario where each invocation from client is forwarded to all replicas but the invocation is completed as soon as the first reply receives at the client side. The response time is 1.35 ms in OFS, 2.78 ms in Phoinix and 1.55 ms in Electra. We observe that the `RequestHandler` of the account object in Phoinix intercepts each client invocation and checkpoints to LM before it forwards the invocation to the primary account object. Each client invocation in Phoinix in fact incurs two CORBA invocations. This explains why the response time in Phoinix is twice more than that in OFS. The response time in Electra is roughly 15% more than that in OFS. We believe the major overhead occurs at the *multicast group communication* layer of Isis.

5. Conclusions

Existing efforts to enhance the CORBA reliability can be roughly categorized into three approaches: integration approach, interception approach and service approach. Each approach has its own merits and drawbacks. In this paper, we propose the OFS that is specified as a CORBA service. The implementation of OFS does not modify ORB infrastructure or IDL language mappings, and thus applications developed with OFS appear to be more portable. OFS differs from other service approach solutions in that OFS does not assume underlying support of a reliable group communication. The system architecture of an OFS implementation is discussed in the paper. This OFS implementation is developed on the Solaris 2.5 platform and with IONA's Orbix 2.0.

Applications that wish to acquire fault–tolerance support from OFS are required to inherit and to implement two OFS defined call-back interfaces. Applications with advance registration thus can rely on OFS for detection of object and node crashes, and for customized recovery. OFS is also responsible for the replication management. The

performance evaluation indicates that system overhead and client invocation delay due to OFS daemon is minimal. The checkpointing overhead from primary to backup copies with OFS is also evaluated. The preliminary experiments indicate that our OFS implementation is very efficient.

The OFS is designed primarily for applications that desire cold or warm backup strategy. We are currently considering extending the OFS specification to support the hot backup scheme. We also notice that CORBA, DCOM and Java RMI have many features in common. These platforms are all based on the concept of *remote procedure call* (RPC). The service that a server provides is described in the context of *interface description language* in all three platforms, namely, CORBA IDL for CORBA, EIDL for DCOM, and Java language for Java RMI. We believe similar service or concept of OFS can be ported to platforms such as DCOM or JAVA RMI using corresponding IDL language since OFS is described in terms of CORBA IDL.

Appendix A. Sample implementation code of account interface

```
// accounti.h
#include ''account.hh''
class account_i: public virtual accountBOAImpl, public virtual primary, public virtual secondary {
public:
   account_i(const char *name):
   ...
   void makeLodgement(CORBA::Float f, CORBA::Environment &ITenv);
   void makeWithdrawal(CORBA::Float f, CORBA::Environment &IT_env);
   CORBA::Boolean get_state(char *& os, CORBA::Environment &IT_env);
   CORBA::Boolean get_log(char *& ml, CORBA::Environment &IT_env);
   CORBA::Boolean start(CORBA::Short k, CORBA::Environment &IT_env);
   CORBA::Boolean turn_primary(CORBA::Environment &IT_env);
   CORBA::Boolean set_log(char *& ml, CORBA::Environment &IT_env);
   CORBA::Boolean set_state(char * is, CORBA::Environment &IT_env);
   ....
}
/** account_i.cc - object implementation of account*/
#include ''account_i.h''
....
void account_i::makeLodgement(CORBA::Float f, CORBA::Environment &IT_env){
   try{
   ...;
   }catch(const CORBA::SystemException &se){
}
void account_i::makeWithdrawal(CORBA::Float f, CORBA::Environment &IT_env){
   try{
     ...
   }catch(const CORBA::SystemException &se){
};
CORBA::Boolean account_i::get_state(char *& os, CORBA::Environment &IT_env){
   ....
};
CORBA::Boolean account_i::getlog(char *& ml, CORBA::Environment &IT_env){
   ...
};
CORBA::Boolean account_I::start(CORBA::Short k, CORBA::Environment &IT_env){
   ....
};
CORBA::Boolean account_I::turn_primary(CORBA::Environment &IT_env){
   ....
};
CORBA::Boolean account_i::setlog(char *& ml, CORBA::Environment &IT_env){
```

```
    ....
};
CORBA::Boolean account_i::set_state(char * is, CORBA::Environment &IT_env){
    ....
};
```

References

APM, 1993. ANSAware Version 4.1 Manual Set, Architecture Projects Management Ltd., Castle Park, Cambridge, UK.

Birman, K., 1993. Integrating runtime consistency models for distributed computing, Tech. Rep. 91-1240, Department of Computer Science, Cornell University.

Booch, G., 1991. Object-Oriented Design with Applications, The Benjamin Cummings Publishing Company.

Brown, K., Kindel, C., 1996. Distributed Component Object Model Protocol - DCOM/1.0, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

Chen, I.R., Bastani, F.B., 1992. Reliability of fully and partially replicated systems. IEEE Trans. Reliability 41 (2), 175–182.

Felber, P., Garbinato, B., Guerraoui, R., 1996. The design of a CORBA group communication service, in: Proceedings of the 15th IEEE Symposium on Reliable and Distributed Systems (SRDS'96), Canada, pp. 150–159.

Huang, Y., Kintala, C., 1993. Software implemented fault tolerance, in: Proceedings of 22nd Fault-tolerance Computing Symposium, pp. 2–10.

IONA, Isis, 1994. An Introduction to Orbix+Isis. IONA Technologies Ltd. and Isis Distributed Systems.

IONA, 1994. Orbix Programmer's Guide, IONA Technologies Ltd.

Liang, D., Chou, S.C., Yuan, S.M., 1998. Adding Fault-Tolerant Object Services to CORBA, Information and Software Technology, pp. 20–34.

Landis, S., Maffeis, S., 1997. Building reliable distributed systems with CORBA, in: R. Soley (Ed.), Theory and Practice of Object Systems, Wiley, New York.

Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., Lingely Papadopoulos, C.A., 1996. Totem: A fault-tolerant multicast group communication system. Commun. of ACM 39 (4), 54–63.

Narasimhan, P., Moser, L.E., Melliar-Smith, P.M., 1997. The interception approach to reliable distributed CORBA objects, in: Proceedings of USENIX Third Conference of Object-Oriented Technologies and Systems (COOTS'97), pp. 245–248.

OMG, 1993. The Common Object Request Broker (CORBA): Architecture and Specification, vol. 1.2, Object Management Group.

Peterson, L., Buchholz, N., Schlichting, R., 1989. Preserving and using context information in inter-process communication. ACM Trans. on Comput. Syst. 7 (3), 217–246.

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

Redmond, F.E., 1997. DCOM: Microsoft Distributed Component Object Model, IDG Books.

Sheu, G.W., Chang, Y.S., Liang D., Yuan S.M., Lo, W., 1997. A fault-tolerant object service on CORBA, in: Proceedings of International Conference on Distributed Computing Systems, pp. 393–400.

Schlichting, R.D., Schneider, F.B., 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Trans. on Comput. Syst. 1 (3), 222–238.

Deron Liang received a BS degree in electrical engineering from National Taiwan University in 1983, and an MS and a Ph.D. in computer science from the University of Maryland at College Park in 1991 and 1992 respectively. Since 1993, he has been an assistant research fellow with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, Republic of China, where he is now an associated Research Fellow. Dr. Liang's current research interests are in the areas of distributed computing systems, object-oriented technology, and system reliability analysis. Dr. Liang is a member of ACM and IEEE.

Chen-Liang Fang was born in Taichung, Taiwan, ROC. He received the MS Degree in computer science in 1992 from New Mexico Institute of Mining and Technology, Socorro. He is currently working toward the Ph.D. degree in the Department of Electronic Technology at the National Taiwan University of Science and Technology, Taiwan. Currently, he is on the faculty of Jinwen Institute of Technology, Taipei, Taiwan. His research interests include distributed system, fault-tolerance, and high-speed networks.

Shyan-Ming Yuan was born on 11 July 1959 in Mauli, Taiwan, ROC. He received the B.S.E.E degree from National Taiwan University in 1981, the MS degree in Computer Science from University of Maryland Baltimore Country in 1985, and the Ph.D. degree in Computer Science from University of Maryland College Park in 1989. He joined the Electronics Research and Service Organization, Industrial Technology Research Institute as Research Member in October 1989. Since September 1990, he had been an Associate Professor at the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan. He was promoted as a Professor in June 1995. His current research interests include distributed system design, fault-tolerant computing, web Technologies, distance learning environment. Dr. Yuan is a member of ACM and IEEE.

Chyouhwa Chen received the B.S.E.E. degree from National Taiwan University, Taipei, Taiwan, in 1983, and the Ph.D. degree in Computer Science. From the State University of New York at Stony Brook in 1991. He joined the department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan in 1991. He is an Associate Professor now. Dr. Chen's research interest lies int he areas of multimedia systems, real-time systems, and system performance analysis.

Gene Eu Jan received the B.Sc. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, in June 1982, and the MS and Ph.D. degrees in electrical engineering from the University of Maryland, College Park, in May 1988 and August 1992, respectively.
He was a Visiting Assistant Professor in the Department of Electrical and Computer Engineering at the California State University, Fresno, CA between August 1991 and August 1992. Since 1993, he has been an associate professor with the Departments of Navigation, and Computer Science, National Taiwan Ocean University, Keelung, Taiwan. His research interests include parallel computer systems, interconnection networks, and VLSI system design.