Vol. 41 No. 2 Transactions of Information Processing Society of Japan Feb. 2000

Regular Paper

Implementation and Evaluation of a Fault-tolerant Mechanism on Network-wide Distributed Object-oriented Systems

Michiharu Takemoto†

This paper describes the implementation of a fault-tolerant mechanism in a CORBA-compliant object request broker (ORB). This mechanism manages the object replicas, thereby enabling non-stop facilities. It hides the internal structure of fault-tolerant objects and is implemented as an object adapter (OA) on the ORB, not as an ordinary object. Because it combines fault detection using the local timer of each node and maintenance for internal state consistency, the overhead is reduced. Evaluation of the mechanism's performance demonstrated that it is suitable for a distributed object-oriented processing environment because it uses only low-overhead functions, such as message transfer, and not high-overhead functions, such as the dynamic invocation interface/dynamic skeleton interface (DII/DSI). This mechanism was developed for use in the distributed processing environment (DPE) of telecommunication networks in which continuous operation and connectivity to other systems are needed.

1. Introduction

Future multimedia communication services will require processing platforms that afford high performance while reducing the cost of equipment and application development. To support such platforms, my group has been developing a network architecture we call distributed object-oriented network architecture, short for DONA [1]. DONA has many features suitable for future communication services. For example, it is "open" in the sense that objects in other networks can connect with objects in a DONA network. To bring this openness to the DONA distributed processing environment (DPE), we are using CORBA technology [2] to make a DONA object request broker (ORB). In other words, the DONA ORB will be a CORBA-compliant ORB.

An ORB consists of an ORBCore and object adaptors (OAs). The application software on the ORB consists of objects. Each object encapsulates data and its functions (operations). These operations are invoked by request messages. An OA controls the execution behavior of objects. Because an object interacts with other objects only when passing messages, an object cannot directly access the internal space of any other object. The objects provide services by communicating with each other, even though they do not know each other's internal structures. Each object has an object reference that is used to distinguish it from other objects. Each object reference includes a node identifier and information about the object that distinguish it from the other objects in the node. When an object sends a message to another object, it includes its object reference so that the receiver object can identify the sender object.

Because the objective of using a DONA-DPE is to develop multimedia communication systems, it is important for the DONA-DPE to have facilities enabling continuous operation (i.e., non-stop facilities), even if part of the system fails. For example, a node (host) failure should be recovered automatically. Because an extremely large number of objects can be executed simultaneously on one node, we must avoid using non-stop facilities whose implementation requires a large number of objects. In other words, the overhead used to achieve continuous operation should be as small as possible.

Traditional switching systems are distributed and have the non-stop facilities because their execution environments are designed for a special purpose. This concept of specialization does not fit well with using a standard DPE like CORBA.

Our group previously developed a prototype DONA ORB. I have now implemented a mechanism that provides the objects with non-stop facilities on this ORB. This mechanism is based on the concept of "replication". The unit of a replica is an object. In this paper I describe my evaluation of this replica-mechanism and its

† NTT Network Innovation Laboratories

performance.

When replicas are used in a system to improve fault-tolerance, their internal states must be kept consistent. This is because a domino effect among restarting objects may occur in a distributed processing system if an appropriate restart strategy is not used. Our mechanism overcomes this problem by using appropriate checkpoints and replicated messages.

In this paper I assume that all operations of objects are deterministic. A "failure" is a situation in which one or more operations of one or more objects seem to be stopped by the node or the "thread"☆ stops. Failures in the communication layer are out of the scope of this paper. My goal here is to explain how to make a system "non-stop", that is, how to keep a system running even when some of its objects fail.

In Section 2, I describe the requirements for fault-tolerance in telecommunication applications. In Section 3, I describe our replica-management mechanism for meeting these requirements, and in Section 4, I describe its implementation. In Section 5, I discuss the mechanism we used in comparison to other models. In Section 6, I describe our experiment to measure its performance and that of other models. In Section 7, I evaluate the results, and in Section 8, I discuss the relationship to the CORBA specifications. In Section 9, I conclude with a brief summary and a mention of future work.

2. Requirements of Telecommunication Systems

Telecommunication systems have special requirements for non-stop facility related to their characteristics and usage. In this section I discuss the requirements for introducing fault-tolerance to such a system.

2.1 High Performance

Telecommunication systems [3] are real-time systems requiring high performance. Therefore, performance reduction should be avoided when supplying the system with a non-stop facility. For example, a mechanism with a complicated object structure and requiring a number of message passings degrade real-time facility and throughput. It is therefore not suitable for telecommunication systems, even if non-stop facility is achieved theoretically.

Switching systems for conventional telephone-call processing must handle thousands of calls at the same time and respond to most events within tens or hundreds of milliseconds [4]. Consider a system designed to handle 300,000 calls per hour (about 83 calls per second). Assume that the conventional call processing consists of five objects and that half the messages among these objects depend on the mutual connectivity of their ORBs. The ORBs must therefore handle about 200 inter-ORB messages per second, with a round-trip time of about 5,000 microseconds. Although these values are for traditional call processing, future telecommunication systems will have similar values.

Felber, et al.[5] proposed a model and implemented it as a mechanism that achieves non-stop facility. However, its performance was insufficient for telecommunication systems. This mechanism is evaluated in Section 5.3.

2.2 Space Redundancy and Warm Passive Replication

Making a system fault tolerant requires introducing redundancy. There are several ways of introducing redundancy; I believe the best way is to use space redundancy and warm passive replication.

There are two basic types of redundancy: time redundancy (rollback) and space redundancy (replication). Because the services provided in telecommunication systems operate through the interactions of several objects, when a failure occurs, it is better to switch the executing unit rather than performing a large rollback, which requires selecting an appropriate rollback point from many checkpoints.

There are two alternative models for replica-management: active replication [6] and passive replication [7)~9)].

In active replication, all the replicas of one object execute the same operation in the same order. Therefore, there is no need for special functions to maintain the internal states among the replicas. Moreover the recovery time when a failure occurs is shorter. However, active replication requires extra CPU resources for executing more than one object for each logical object. This extra performance capability is generally wasted during normal operation.

In passive replication, on the other hand, several functions are introduced to maintain internal state consistency among the replicas. While this results in a longer recovery time, the execu-

☆ What is used to implement the "thread" is out of the scope of this paper. An example entity of a "thread" is a process or a POSIX thread of UNIX or a task of RT-OS.

tion cost during normal operation is generally lower. Passive replication is thus preferred because the failure rate of current telecommunication systems is low and because it is important to reduce the overhead for ordinary execution.

Passive replication can be classified into two types depending on how the internal states of the replicas are managed. One type is warm passive replication (no execution but the internal states are kept consistent), and the other is cold passive replication (no execution and the internal states are not kept consistent). Because it is (almost) always necessary to keep the internal states consistent in warm passive replication, the cost of ordinary execution is a bit higher. In cold passive replication, becaust the internal states are not kept consistent, a mechanism is needed to re-create the internal states for the re-execution. Implementing such a mechanism in a network-wide DPE is thus expensive. Therefore, I selected warm passive replication.

Although passive replication is formally a type of space redundancy, failure recovery also needs some time redundancy. I achieve this by setting the checkpoints when maintaining the internal states; a suitable rollback point is selected from these checkpoints when a failure occurs.

2.3 Server-side Implementation

The software used for telecommunication services consists of several objects. If the developers of such objects design them to exchange internal information, problems may occur because objects must sometimes communicate with objects developed by other companies (network providers and service providers). When a server object starts its operation, no one knows which kind of client objects may communicate with the server.

Consequently, if we change the functions of a server to achieve non-stop facility, we should design to objects to reveal their interfaces, not their internal states, to clients. Therefore, for the objects consisting of the telecommunication services, the server-side implementation of fault-detection and fault-recovery is important.

Even if the non-stop facilities are on the server side, objects with fault-tolerant facilities should be able to be handled in the same manner as ordinary objects, i.e., having their internal structure hidden and being identified only by a logical name.

In the oppsite situation, the non-stop facilities are on both the client side and the server side or they use common functions for clients and servers.

An example common functions is to use a daemon that checks the heart beat of thread. Ericsson proposed a mechanism for checking whether the server is "alive" by using a "ping" operation [10]. This mechanism includes a language processor whose function is to add a special operation (ping) to the server. However, it is not suitable for telecommunication systems because it cannot set an appropriate time-out value for each operation. Moreover, determining the appropriate rollback point is complicated [11),12)] because the relationships between objects in telecommunication systems are very complicated.

Another approach is to locate a mechanism on both sides so as to extend the object reference, which includes the structure of the server. Several mechanisms have been proposed to add information to the object reference [13),14)]. When a failure occurs, the client or the ORB of the client explicitly changes the server replica. Therefore, the function for this change must be implemented on the client side. This is not possible in telecommunication systems, in which the server and client sides share only the interface of an object.

2.4 High Concurrency

Telecommunication services generally consists of a number of "small" objects, which are objects whose memory-footprint is small and whose operation execution-time is short. Note that this is different from other applications, such as numerical computation, in which each operation execution-time is long and the number of objects is not so large. Therefore, the effect of mechanisms for fault-tolerance on the concurrency must be as smaller as possible because several objects usually run on each node in telecommunication systems.

As described in Section 2.2, space redundancy is more suitable than time redundancy. If the replica-management mechanism requires expensive execution cost, the total concurrency is low. A replica-management mechanism that does not interfere with the concurrency of the node is preferable. In short, a replica-management mechanism that is simpler and has a smaller execution cost is desirable. For example, a mechanism that minimizes the number of messages associated with increased redundancy will be simpler and less expensive to operate.

If the management entity is an actual ordinary object, assigning one management object to each replica group is not suitable because it increases the number of objects, which degrades concurrency.

As we mentioned, switching systems for conventional call processing must handle thousands of calls at the same time, and future telecommunication systems will have to handle the same level of concurrency.

3. Proposed Replica-management Mechanism

To meet these requirements, we previously proposed a replica-management mechanism called FTARO (Fault-Tolerance based on Asynchronous Replicated Objects)[15]. FTARO uses space redundancy (Section 2.2). Because FTARO is on the server side (Section 2.3), the objects on FTARO can connect to any type of client. The replica-management mechanism of FTARO is built with a message-transferring mechanism in the ORB and does not affect object concurrency (Section 2.4). How well it meets the requirement for high performance (Section 2.1) is discussed in Section 6.

3.1 Object Definitions

I defined several objects for use in FTARO. A group of objects that have a fault-tolerant facility and that behave as one object is called a *fault-tolerant object*. An object that is included in a fault-tolerant object and that is a replica for space redundancy is called a *replicated object*. The target operations of a fault-tolerant object are executed by one of its replicated objects. A replicated object that executes the target operations is called an *active replicated object*. The remaining replicated objects are called *stand-by replicated objects*. For the other components (replica-management mechanisms and nodes), *active* and *stand-by* are used with the same meanings.

3.2 Object Execution

The structure of a fault-tolerant object in FTARO is shown in **Fig. 1**. A fault-tolerant object includes one active replicated object (*RO1*), which executes the target operations, and several stand-by replicated objects (*RO2*), which have been created but do not execute the target operations, they only wait. Hereafter, for simplicity, I assume that there is only one stand-by replicated object.

3.2.1 Message Reception

A message from an external object (*EO*) is not directly sent to *RO1*. This is because the node information in the object reference of a fault-tolerant object does not specify on which node *RO1* is located. Instead, it specifies a different node (*N0*). When *EO* sends a message to the fault-tolerant object using the object reference of the fault-tolerant object, the message reaches node *N0*. If the replica-management mechanism (*M0*) of node *N0* determines that the target object is a fault-tolerant object, *M0* obtains the object reference of *RO1* from the conversion table (*T0*), then, rather than distributing the message to the object on it, *M0* transfers the message to node *N1* for *RO1*. In this message-transfer process, *M0* simultaneously sends a replica of the message (*m2*) to the replica-management mechanism (*M2*) of *RO2*.

After the replica-management mechanism (*M1*) of *RO1* receives the message, it distributes the message to *RO1*, and *RO1* executes the target operation.

Because *EO* does not send the message directly to *RO1*, the fault-tolerant object does not need to notify *EO* about any changes of *RO1*. Whether replicated object is an active or stand-by one is internal information, so that information is not shared. This is advantageous in a large distributed system in which each server object must communicate with many types of objects.

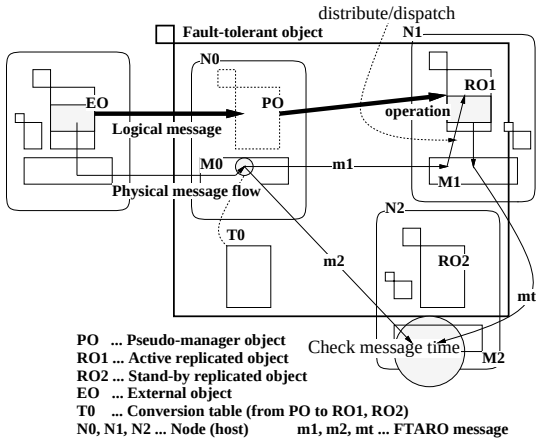

Fig. 1 Fault-tolerant object in FTARO.

3.2.2 Internal-state-consistency Checking and Fault Detection

When *RO1* finishes an operation, control is returned to the ORB. *M1* sends a message (*mt*) to *M2*, that includes at least two types of infor-

mation☆. One type is used for updating the internal states of *RO2*, and the other is used for notifying *M2* of the termination of the operation execution of *RO1*. *M2* uses this information to change the internal states of *RO2*.

By using time-out with these two messages (*m2* and *mt*), a fault in *RO1* or its node can be detected. This detection is processed on the server side. No information leaks from the server side to the client side and no information from the client side is needed.

Because this fault-detection is done by *M2* of *RO2*, when *M2* detects a failure, it changes the stand-by replicated object to an active replicated object and restarts to execute the operation from the beginning. FTARO thus uses two redundancy models—space redundancy (passive replication of objects) and time redundancy (small rollbacks).

3.2.3 Replica Management Model

From the viewpoint of the message flow, an imaginary object can manage replicated objects (*RO1* and *RO2*). Let us call this imaginary object (shadow) a *pseudo-manager object*. In Fig. 1, it is shown as *PO*. When registering a fault-tolerant object to the name service, the object reference and name of *PO* are registered. When *EO* wants to contact the fault-tolerant object, *EO* contacts *PO* via the name service. This enables the fault-tolerant object to be dealt with flexibly, although the actual structure is a group of more than two objects.

3.2.4 Reduction in the Number of Messages

The number of messages inevitably increases when redundancy is introduced into a system. To achieve high performance, however, the increase should be minimized. FTARO includes a facility for combining messages. For example, two messages for maintaining internal-state consistency and a pair of messages for fault-detection with time-out can be combined. The number of messages is thus lower than when the functions for passive replication are implemented separately.

4. Implementation of FTARO on CORBA-compliant ORB

4.1 Execution Model of CORBA-compliant ORB

The process of invoking a target operation via

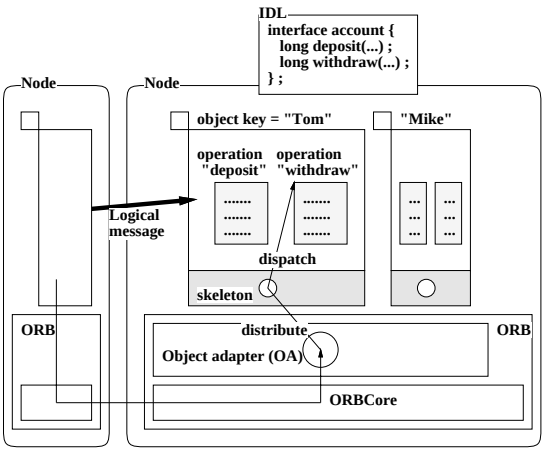

Fig. 2 Execution model of CORBA-compliant ORB.

messaging on CORBA-compliant ORB is illustrated in **Fig. 2**. The object interface is defined using the interface definition language (IDL). The interface name is "account", and the operations are "deposit" and "withdraw". When a message reaches the ORBCore the ORBCore delivers the message to the appropriate OA. The OA analyzes the message header to obtain the *object key* (the name of the object) ("Tom") of the target object. It then distributes the message to the target object. In the target object, the skeleton code is used to analyze the message to obtain the target operation ("withdraw"), then, it dispatches to the target operation.

4.2 Implementation of FTARO

As described in Section 3, FTARO is a message-transferring mechanism invoked on the condition that the target object is a fault-tolerant object. The structure of our current implementation of the FTARO mechanism is shown in **Fig. 3**. It includes the following functions (the numbers correspond to those in the figure).

1. Detect fault-tolerant object. The replica-management mechanism of the pseudo-manager object determins whether the target object is a fault-tolerant object by checking the object key, which is included in the request message header.

2. Convert the object reference. The replica-management mechanism of the pseudo-manager object converts the object reference for the fault-tolerant object to that for the active and stand-by replicated objects. The conversion table can be implemented in the OA because the conversion is based only on the relationships among replicated objects.

☆ The message can also include the load on the node, useful information for load-balancing.

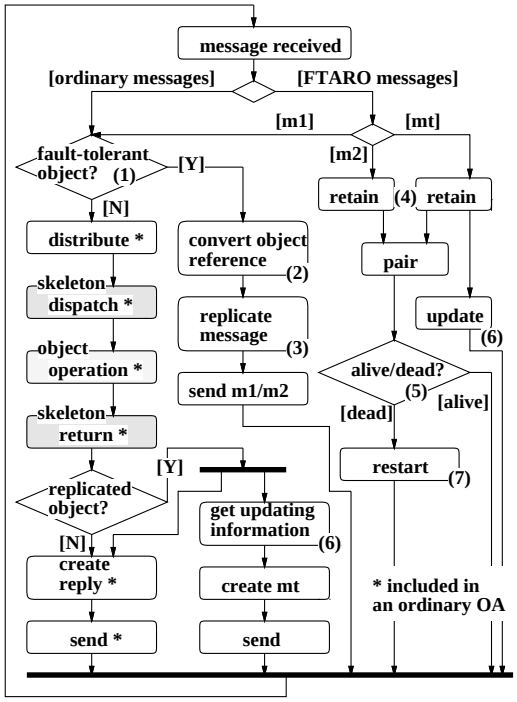

Fig. 3 Structure of current implementation.

3. Replicate and send messages. After replicating the message, the replica-management mechanism of the pseudo-manager object sends one copy to the active replicated object for target operation and the other to the replica-management mechanism of the stand-by replicated object.

4. Retain messages. The replica-management mechanism of the stand-by replicated object stores the replicated message recieved from the pseudo-manager object. It detects whether the message is a replicated one needed for restart by using the object reference of the target object in the message header.

5. Detect failure. To detect the failure of the active replicated object, the replica-management mechanism of the stand-by replicated object measures time since the message from the pseudo-manager object is received until the message from the active replicated object is received. If the latter message is not received before a specified time, it judges that the active replicated object has failed.

6. Maintain internal consistency. To enable restart from the beginning of an operation if a failure occur in the active node, the replica-management mechanisms maintain the same internal states between the active and stand-by replicated objects. At the end of each operation, the replica-management mechanism of the active replicated object sends a message to update the internal state of the stand-by replicated object. The current internal state is reset based on the latest retained message and the previous internal state.

7. Restart operation. When the replica-management mechanism of the stand-by replicated object judges that the active replicated object has failed, it restarts the operation by using the latest retained message with the previous internal state. it only distributes the retained message to the stand-by replicated object.

Because all of these functions are invoked by messages, they can all be implemented on the OA, especially around the message-distributing functions. There is no need to modify the operation-dispatching functions in the skeleton code.

I implemented these functions on a DONA CORBA 2.1-based ORB. The DONA ORB has only one OA, a basic object adapter (BOA), and uses the IIOP-1.1 communications protocol.

I modified the BOA into a FTARO-OA, which is completely upward compatible with the BOA in that it has all the BOA interfaces. The only difference is that the FTARO-OA can invoke several special functions depending on the message. I placed this FTARO-OA on top of the ORBCore.

Because the FTARO-OA has all the interfaces of the BOA with the same semantics, an external object can communicate with an ordinary object and with a fault-tolerant object. A mutual connection can be established even if the communication partner object is running an other CORBA-compliant ORB.

Moreover, application program developers can create fault-tolerant objects on FTARO without being aware of the existence of the replicas. This function is supplied by the language-processing system and works as follows. Directives are given that specify the parameters used by FTARO to convert the original source code, into the specialized code used by FTARO.

The boxes marked with an "*" in Fig. 3 are those used in an ordinary OA. When a message is for an ordinary object, these boxes and the corresponding condition checking are executed.

Note that the current implementation is not

optimized, so the performance is less than that possible. For example, the "replicate message" box (3) in Fig. 3 is implemented by using copies of the memory buffer for the request message. The program code for this box could be optimized.

5. Mechanism

5.1 Alternatives

I focused on two key factors in comparing FTARO with the alternatives. One is the execution entity of the replica-management mechanism and the other is the location of the replica-management mechanism.

When we consider the execution entity of the replica-management mechanism on the ORB, we have only two alternatives: the ORB itself or an actual ordinary object. I selected the former for my implementation. FTARO was implemented on a CORBA-compliant ORB as the OA. The reason for this selection is described in Section 5.3.

When we consider where to locate the replica-management mechanism, we have only two alternatives. One is that one replica-management mechanism is given per fault-tolerant object, and the other is that one replica-management mechanism is given per node. The former means the location is a logical group. The latter means the location is a physical group. I selected the latter: one replica-management mechanism is given per node.

I have analyzed this mechanism model under three alternatives: (1) the replica-management mechanism is an actual ordinary object per fault-tolerant object, (2) the replica-management mechanism is an actual ordinary object per node, and (3) the replica-management mechanism is in the ORB.

5.2 Points to Consider

Let us consider which alternative is best suited for requirements described in Section 2. The space-redundancy (Section 2.2) and server-side implementation (Section 2.3) requirements have already been met. We now need to consider high concurrency (Section 2.4). Although performance (Section 2.1) cannot be precisely evaluated without an actual implementation, it is worth while to also consider performance when comparing mechanisms. In my comparison I thus considered the effect of the replica-management mechanism on concurrency. (The situation in which the replica-management mechanism is in the ORB was found to be a better choice.) I also considered the performance of the replica-management mechanism. This means at which level the replicas are managed. (Achieving the replica-management mechanism as an actual ordinary object was found to be the worse choice. This is because the model needs the operation-dispatching functions to be executed.)

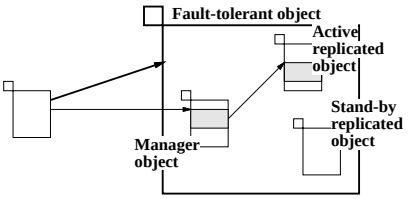

Fig. 4 An actual ordinary object managing replicated objects.

5.3 Comparison

The properties of the models in Section 5.1 are now compared. Implementation examples in a CORBA environment are also described.

(1) As **Fig. 4** shows, an actual ordinary manager object can manage a single group of replicated objects. Because the manager object is an ordinary object, the OA must distribute messages to the manager object and the skeleton code must dispatch the operations. Compared with the replica-management mechanism of the ORB (3), this distribution and dispatching adds a new overhead to the system. Moreover, when the number of groups of the replicated objects is increased, the number of manager objects must be increased. This degrades concurrency severely. Although this model has these two disadvantages, it also has an advantage. The manager object does not need to analyze the interface, including the operation name and the arguments, because it is fixed.

(2) As **Fig. 5** shows, an actual ordinary manager object can also manage all the replicated objects its the node. Because there is only one manager object per node with this model, the effect for concurrency of the manager objects remains small when the concurrency of the other objects increases. This model, however, has the same disadvantage of the first model (distribution and dispatching in the manager object), plus implementation in a CORBA environment entails a large overhead. When implementing this model in a CORBA en-

Table 1 Three models of replica-management.

Entity	Location	
	Logical (per FT* object)	Physical (per node)
Actual ordinary object	(1) impairs concurrency	(2) impairs performance, e.g., OGS
ORB	(3) does not impair concurrency and performance, e.g., FTARO	

* fault-tolerant

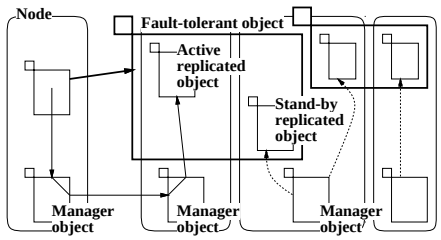

Fig. 5 An actual ordinary object managing all the replicated objects on its node.

vironment, we must use the dynamic invocation interface/dynamic skeleton interface (DII/DSI) mechanism because this type of manager object must receive all types of interfaces. When the operation of the manager object is activated, it gets information about the target object (meaning the replicated object) from the interface repository (IR). This information includes the interface name, the operation name, and the argument types of the target object. Then, it analyzes the information, forms a new request for the target object, and sends the request message. Felber's object group service (OGS)[5] is a typical example of this model. Because this model must use a dynamic interpretation of the arguments. The execution is thus costly.

(3) A replica-management mechanism in the ORB manages all replicated objects on a node. Because the replica-management mechanism is located in the ORB, distribution and dispatch are unnecessary, so concurrency is not affected even when the number of replicated objects is increased. FTARO is a typical example of this model on CORBA. FTARO is only a message-transfer mechanism. It does not need to analyze the interfaces; it simply needs to analyze the message headers in order to transfer messages.

Table 1 summarizes these models. To obtain high concurrency and high performance, the replica-management mechanism should be located in the ORB. These static evaluations thus lead us to conclude that FTARO is suitable for telecommunication systems.

6. Performance

I measured the performance mainly in relation to the requirement for high performance (Section 2.1). It will be measured in relation to the requirement for high concurrency (Section 2.4) in the near future.

6.1 Experiment

In Section 5 I compared the performance of three alternative replica-management models and concluded that the replica-management mechanism in the ORB managing all replicated objects on a node ((3) in Section 5.3) may be suitable for telecommunication systems. However, until I measure the overhead with this model, I cannot conclude that the model is really suitable.

I implemented FTARO on a DONA ORB as described in Section 4.2 in order to measure its performance. Using Felber's OGS distribution☆, I can measure the performance of an OGS on an ORB.

The CORBA IDL interface I used (**Fig. 6**) had very simple operations: *set* (registration of data into a two-dimensional matrix) and *get* (acquiring of data). To evaluate the effect of the number of arguments, I used four additional *set* operations (*set13*, *set23*, *set33*, and *set43*), each with a different number of arguments. The added arguments were dummies and were simply passed to the server, so their behavior are the same as that of *set*.

These experiments were run on a SUN Ultra1 workstation (UltraSPARC 167 MHz, Solaris 2.5.1).

The two models tested were as follows.

- **FTARO** was implemented as the OA of a DONA ORB. The fault-detection ((**5**) in Fig. 3 and Section 4.2) and fault-recovery, which run on the stand-by replica-management mechanisms, were not implemented. However, because all the functions needed on the active replica-

☆ Available at `http://lsewww.epfl.ch/OGS` (Oct. 1998).

```
interface grid {
  void set(in short x, in short y, in long data) ;
  long get(in short x, in short y) ;
  void set13(in short x, in short y, in long data, in long dummy1,,, in long dummy10) ;
  void set23(in short x, in short y, in long data, in long dummy1,,,,, in long dummy20) ;
  void set33(in short x, in short y, in long data, in long dummy1,,,,,,, in long dummy30) ;
  void set43(in short x, in short y, in long data, in long dummy1,,,,,,,,, in long dummy40) ;
} ;
```

Fig. 6 Sample interface (IDL).

Table 2 Measured time of OGS vs. FTARO.

		DONA ORB		CORBA product	
homogeneous	measured time	direct commun.	FTARO	direct commun.	OGS
		433.73	2801.63	899.158	37845.1
	ratio(*)	6.46		42	
heterogeneous (directly communicate with client in another CORBA product)	measured time	direct commun.	FTARO	direct commun.	OGS
		1117.91	3694.09	–	–
	ratio(*)	3.30		–	

unit time: μ sec.
(*) ratio of the round-trip time to that of direct communication.

management mechanisms for ordinary execution were implemented—for example, operation invocation and internal state notification and updating—the latency of the ordinary execution could be measured.

- **Felber's OGS** was implemented on a commercial CORBA product based on the policy that the OA and ORBCore could not be changed. This implementation used IR and DII/DSI.

To reveal the overhead, I measured the round-trip time for operation invocation in a direct-communication situation, i.e., the client and server were connected directly via ordinary CORBA-compliant ORBs. This means invocation was simple without management of the replicas.

To reveal the effect of changing clients, I measured the round-trip time under two types of situations—homogeneous and heterogeneous. In the homogeneous situation, the server and client were running on the same ORB. In the heterogeneous situation, the client was running on a different ORB than the server's one.

6.2 Results

The execution times were measured as the average round-trip time for each operation invocation as seen from the client side.

OGS I used needs an OGS mechanism on both the server and client side, a conventional client on an other commercial CORBA ORB cannot connect to the server on the OGS.

Table 2 shows the result of the performance measurement by using operations with few arguments (*set* and *get*). Each execution time was the average time of *set* and *get*, and was normalized by the execution time for the direct communication, so I obtained the ratio of the round-trip time to that of direct communication. As shown in Table 2, the overhead FTARO (about 6.5) is smaller than that of OGS (about 40).

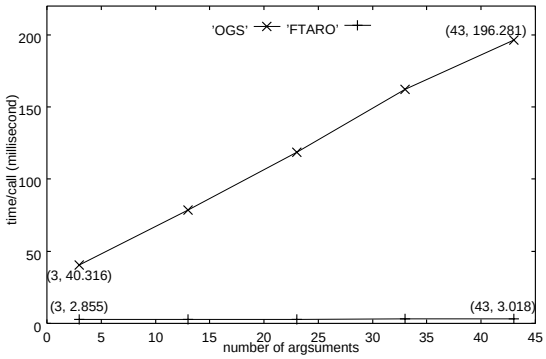

Fig. 7 Increase in the round-trip time with the number of arguments, for operations *set*, *set13*, *set23*, *set33*, and *set43*.

Figure 7 shows the results of the performance measurement by using operations with several arguments (*set*, *set13*, *set23*, *set33*, and *set43*). As shown in Fig. 7 there was little effect on the round-trip time when the number of arguments was increased with FTARO, but, there was a large effect when the OGS mechanisms with DII/DSI and IR were used.

7. Evaluation

The results of my experiment (Section 6) confirmed those of the comparison (Section 5). FTARO imposes less overhead than the alternatives. Although this experiment was performed on a general-purpose workstation and the mechanism has not been optimized, my finding that the round-trip time when using managed replicas was less than 5,000 microseconds (Table 2 and Fig. 7) indicates that FTARO can meet the requirement for high performance (Section 2.1).

My finding that a client on another CORBA-compliant ORB can connect to an object running on FTARO shows that the implementation of FTARO is fully on the server side. FTARO thus meets the requirement for server side functions (Section 2.3).

The ratio of the FTARO round-trip time to the direct round-trip time was less than 10.00 (Table 2) because FTARO is only a message-transferring mechanism in the ORB. As I described in Section 4, FTARO can be implemented in such a way that the whole message is not analyzed in the ORB. Instead, each message is passed through an additional node (a pseudo-manager node). In other words, neither distributing to the target object nor dispatching to the target operation is needed. Therefore, the execution cost of the replica-management—for example, converting the object references and replicating the messages—is small. The small execution cost of the replica-management is linked to the small overhead.

In contrast, the overhead of OGS is huge because it must use the DII/DSI components. Therefore, if we implement the OGS mechanism by using ordinary objects on a CORBA-compliant ORB without modifying the ORB, we must implement the DII/DSI and IR components. This results is huge overhead as follows. When a message reaches the manager object, the manager object must obtain the interface information (the type and value of each argument) for the requested operation from the IR each time. Then the manager object must analyze the sent data with this information and form DII/DSI requests for the active replicated object. This analyzing and forming process is an interface interpreter.

In particular, as Fig. 7 shows, the advantage of FTARO increases with the number of arguments. While another mechanism (OGS) must interpret each argument in the message, FTARO need not analyze all the arguments; it only needs to analyze the message header. In other words, the overhead of FTARO depends on only the size of the request message, so the contents of the request message have little effect on overhead. Even if the message has structures and other types complicated arguments, there is little effect on overhead. This is obvious because the replica-management mechanism of FTARO does not have any functions for interpreting arguments. However, the overhead of the interpreting mechanism depends on the type and size of the operation in the request message. If the client calls an operation whose arguments include several structures, the overhead of the replica-management mechanism is larger.

The FTARO mechanism and its implementation I have described in this paper are independent of the types of arguments. In telecommunication applications, some operations may have very complicated arguments, including structures, unions, and also *Any*s, in which any type of value can be stored. Therefore, FTARO is more suitable for telecommunication applications.

Although the current implementation of FTARO is not yet completed, the results so far indicate that it has sufficient performance.

8. Relationship to OMG Fault-tolerance Specification

Object Management Group (OMG) is currently drafting a specification for "fault tolerant CORBA"[16]. The aim is to describe several fault-tolerant models that require minimum ORB modification and can be implemented using servers. The execution effect is out of the scope of their specification.

Our current implementation of FTARO enables a client that does not follow the "fault tolerant CORBA" specification but does follow the ordinary CORBA specification to access a fault-tolerant server object.

Even though only one fault-tolerant model can be used, it can be changed on-the-fly by changing the pseudo-manager object, that is, the information in the ORB.

9. Conclusion

In this paper I described the FTARO replica-management mechanism for fault-tolerant objects in distributed object-oriented telecommunication systems. FTARO can flexibly handle

fault-tolerant objects because it is completely on the server side. It does not effect concurrency.

I also described my implementation of FTARO on a CORBA-compliant ORB. I measured the performance of a fault-tolerant object on FTARO and compared it with that on a different replica-management mechanism. I found that the overhead of FTARO is smaller. Moreover, FTARO is more tolerant of higher numbers of operation arguments. Therefore, FTARO is suitable for telecommunication systems.

We plan to optimize the implementation of FTARO based on the POA-based CORBA-compliant ORB [17] and to evaluate how well it meets the requirement for high concurrency. We will also develop processing-language systems that can automatically generate FTARO fault-tolerant objects

Acknowledgments I thank Mr. Takayuki Nakamura of NTT Network Innovation Laboratories for his assistance with the measurement of the ORB operation.

References

1) Suzuki, S., Yamada, S., Kubota, M., Kogiku, I. and Matsuo, M.: DONA: Distributed Object-Oriented Network Architecture for Revolutionary Network Reconstruction, *Proc. XVI World Telecom Congressn* (*ISS'97*), Toronto, Canada, Vol.II, pp.459–465 (1997).
2) Object Management Group: The Common Object Request Broker: Architecture and Specification. `http://www.omg.org/`.
3) Raguideau, N., Maruyama, K. and Kubota, M.: Telecommunication Service Software Architecture for Next-Generation Networks, *IEICE Trans. Comm.*, Vol.E77-B, No.11, pp.1295–1303 (1994).
4) Maruyama, K. and Kubota, M.: PLATINA: Platform for Telecommunication and Information Network Applications, *Proc. Third Telecommunications Information Networking Architecture Workshop* (*TINA'92*), Narita, Japan, pp.13-2-1–13-2-15 (1992).
5) Felber, P., Guerranoui, R. and Schiper, A.: The Implementation of a CORBA Object Group Service, *Theory and Practice of Object Systems*, Vol.4, No.2, pp.93–105 (1998).
6) Powell, D., Bonn, G., Seaton, D., Verissimo, P. and Waeselynck, F.: The Delta-4 Approach to Dependability in Open Distributed Computing Systems, *Proc. Eighteenth Annual International Symposium on Fault-Tolerant Computing*, (*FTCS-18*), Tokyo, Japan, pp.246–251 (1988).
7) Birman, K.P., Joseph, T.A., Raeuchle, T. and Abbadi, A.E.: Implementing Fault-Tolerant Distributed Objects, *IEEE Trans. Softw. Eng.*, Vol.SE-11, No.6, pp.502–508 (1985).
8) Shima, K., Higaki, H. and Takizawa, M.: Pseudo-Active Replication in Heterogeneous Clusters, *IPSJ SIG Notes*, Vol.96, No.108, pp.31–36 (1996).
9) Speirs, N.A. and Barrett, P.A.: Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing, *Proc. Nineteenth International Symposium on Fault-Tolerant Computing* (*FTCS-19*), pp.184–190 (1989).
10) Ericsson, Iona Technologies PLC and Nortel Networks: Fault tolerant CORBA – Initial Submission (1998). `http://www.omg.org/pub/docs/orbos/98-10-10`.
11) Borg, A., Baumbach, J. and Glazer, S.: A Message System Supporting Fault Tolerance, *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, pp.90–99, ACM (1983).
12) Strom, R.E. and Yemini, S.: Optimistic Recovery in Distributed Systems, *ACM Trans. Comput. Syst.*, Vol.3, No.3, pp.204–226 (1985).
13) Maffeis, S. and Schmidt, D.C.: Constructing Reliable Distributed Communication Systems with CORBA, *IEEE Communications Magazine*, Vol.35, No.2, pp.56–60 (1997).
14) Oracle Corp.: Fault tolerance – Initial Submission (1998). `http://www.omg.org/pub/docs/orbos/98-10-13`.
15) Takemoto, M.: Fault-tolerant Object on Network-wide Distributed Object-oriented Systems for Future Telecommunications Applications, *IEEE PRFTS*, pp.139–146 (1997).
16) Object Management Group: Fault tolerant CORBA Using Entity Redundancy Request for Proposal (1998). `http://www.omg.org/pub/docs/orbos/98-04-01`.
17) Object Management Group: The Common Object Request Broker: Architecture and Specification (Revision2.2) (1998). `http://www.omg.org/`.

(Received May 6, 1999)
(Accepted December 2, 1999)

Michiharu Takemoto graduated from the Department of Information Science, Faculty of Science, the University of Tokyo in 1992, and received the degree of Master of Science from the University of Tokyo in 1994. Since he joined NTT in 1994, his research interests include the high-functional execution environment, such as fault-tolerance or load-balancing. He received a research promotion award from IEICE in 1998. He is currently a Research Engineer, Distributed Network Systems Laboratory, NTT Network Innovation Laboratories. He is a member of IPSJ, IEEE CS and IEICE.