Regular paper

The implementation and analysis of OCI-based group communication support in CORBA

Dukyun Nam, Dongman Lee, and Chansu Yu
School of Engineering
Information and Communications University
Taejon, Korea
Tel: +82-42-866-6163 Fax: +82-42-866-6222
{paichu, dlee, cyu}@icu.ac.kr

Abstract

Object replication is a technique to enhance fault tolerance and high availability. Group communication is a useful mechanism guaranteeing the consistency among replicated objects. We propose a generic group communication framework that allows transparent plug-in of various group communication protocols with no modification of existing CORBA. For this, we extend the Open Communications Interface (OCI) to support interoperability, reusability of existing group communication, and independency on ORB and OS. The proposed scheme is composed of group membership component, group IOR component, and group multicast component. Experiment results show that group object invocations using the proposed scheme produce almost constant latency regardless of the number of members while the latency of multiple object invocations using IIOP constantly increases.

Keywords: Group Communication, CORBA, Open Communications Interface, Multicast, Group Communication Inter-ORB Protocol

1. Introduction

Object replication is a technique to enhance fault tolerance and high availability [4]. An object group is a collection of object replicas that cooperatively works for a common task [9]. Here consistent state of an

object group needs to be maintained for constructing reliable and fault-tolerant distributed applications [14]. Group communication service (GCS) is a useful mechanism guaranteeing the consistency of the states of all the member objects. It maintains a view, a list of the currently active and connected members in an object group, and also informs the running objects of the updated view whenever it changes. The consistency can be guaranteed by reliable delivery of messages to the members in the current view.

CORBA provides many attractive features like portability and interoperability that allow distributed objects to transparently interact with each other in heterogeneous and distributed environment. As such, a single group communication protocol would not satisfy all the needs of CORBA group applications. There have been several approaches for supporting group communication service in CORBA [2, 8, 10, 12, 13]. However, the approaches do not support transparent plug-in of group communication protocols into CORBA, and thus CORBA application programmers cannot directly exploit the protocols. Therefore, there must be a generic group communication framework that allows transparent plug-in of various group communication protocols via a standard CORBA interface.

We propose a mechanism that allows such framework with no modification of existing CORBA. The proposed mechanism leverages the OCI (Open Communications Interface) [6] with Group Communication Inter-ORB Protocol (GCIOP) as a group communication instantiation of General Inter-ORB Protocol (GIOP) and assigns the CORBA object to each process member in the group communication protocol. Group name, ordering type, and state as attributes are added to the Info Object, and the interfaces of the OCI were extended to support group semantics. We implement the proposed scheme on top of our group communication protocol using an ORB supporting OCI. Experiment results show that group object invocation using the proposed scheme has almost fixed latency regardless of the number of members while the latency of multiple object invocations using IIOP constantly increases.

The reminder of the paper is structured as follows. Section 2 discusses the existing approaches for group communication in CORBA. Section 3 explains internal procedures and implementation of each

component in the proposed scheme. And Section 4 analyses the performance of the extended OCI. Conclusion follows in Section 5.

2. Related Work

Group communication services for CORBA can be categorized into three approaches such as integration approach, service approach, and interception approach [3, 13]. Examples are Electra [8], Object Group Service [2, 3], Eternal [10], and respectively. Electra integrates group communication service with CORBA by extending Basic Object Adapter (BOA). The implementation and performance of the system are efficient since there is no intermediate object between ORB and group communication system. However, this approach needs to modify ORB for creating group reference and supporting group communication. Object Group Service (OGS) [2] provides a new CORBA Object Service for group communication. This approach is independent of ORB and guarantees portability with CORBA Object Service. However, it cannot utilize existing group communication systems and has potential drawback on the performance. Eternal [10] and Eternal interceptor [14] capture and transmit Internet Inter-ORB Protocol (IIOP) messages to the replication manager, which maps the messages onto the group communication system. This approach need not modify ORB as in the case of using interceptor. However, it is dependent on the OS since interceptor is implemented in the system call level.

Open Communications Interface (OCI) [6] provides plug-in protocol interfaces for CORBA. It implements the Acceptor/Connector module [18] in ORB. The module distinguishes connection establishment from service initialization occurring in the communication of a client/server model. The interfaces are Buffer, Acceptor, Transport, Connector, Connector Factory, Registries, and Info objects [15]. A Buffer holds data in an array of octets and maintains a position counter, which are used in communications between client and server. Info objects provide information on Acceptor, Transport, Connector, and Connector Factory. Figure 1 shows how OCI incorporates a given transport protocol with

ORB. As client and server are activated, an Acceptor Registry in the Object Adapter (OA) activates an Acceptor, and a Connector Factory Registry in ORB creates a Connector Factory. Then at the server side, an Acceptor creates profile information for Interoperable Object Reference (IOR), and the OA creates an IOR using it. At the client side, a Connector Factory creates a Connector using the IOR. After that, the Connector and the Acceptor instantiate their own Transport. Once a connection is made between the Transports at each side, the Transports can exchange messages using a Buffer object.

Halteren, et al. [5] applied IP Multicast to CORBA by extending the OCI. In order to meet the requirements on transport layer such as connection-oriented, reliable data transport, transported data as a stream, and notification of connection loss in the CORBA specification [19], they added the schemes for acknowledgement and retransmission of packets to the intermediate protocol. They also used modified Interoperable Object Reference (IOR) that includes IP multicast address (D class) and a sequence number. However, their approach does not provide a generic framework that can accommodate various group communication protocols in CORBA. Moreover, dynamic group membership is not supported because Java IP Multicast API is directly used as group operations.

Multicast Group Internet Inter-ORB Protocol (MGIOP) engine was designed with MGIOP specification in [11]. MGIOP consists of group ID and domain ID. Especially, it does not instantiate GIOP but encapsulate a GIOP message itself. The MGIOP engine concurrently supports different group communication protocols. It, however, requires modification of ORB because its engine must be included in ORB or connected to it.

In designing the proposed approach, we also considered a network-level interceptor - a part of portable interceptors [17] since a network-level interceptor allows flexible plug-in of transport protocols into CORBA. However, a network-level interceptor is not described in the specification of the portable interceptors [16]. We recognized two other approaches which are similar to network-level interceptor. One is the OCI and the other is MGIOP. The OCI was proposed to allow transparent plug-in of various

transport protocols in IN/CORBA specification [6, 15]. For example, Halteren and his colleagues [5] exploit OCI to support IP Multicast to ORB. On the other hand, MGIOP [11] maps the General Inter-ORB Protocol (GIOP) specifications onto a multicast group communication protocol. Its engine concurrently supports different group communication protocols. However, it needs some modification of ORB to provide an abstraction of the engine since the engine is included in ORB or connected to ORB. Consequently, OCI is more appropriate to group communication than MGIOP in terms of portability.

3. Design and Implementation

This section describes the design consideration and implementation details of the proposed scheme, extended OCI for group communication.

3.1. Design Consideration

A client application sending a message to the members of a group does not supply a member list. Instead, a membership service supplies a group address as a group identifier. It is mapped onto a current membership list, and then hides the group's internal structure from applications. Group membership service provides operations to create and change the membership of groups, and guarantees a mutually consistent view of the membership of the group. A membership list can be managed by one membership server or by exchanging a view change message among each member. Delivery ordering indicates that messages reach all of its members at the same time [1]. It guarantees a consistent state among members. Ordering types are total and casual ordering which manage concurrent messages and sequences of related messages. To provide group communication service to CORBA objects, three aspects should be considered - group address, group membership, and ordering. We assume that the underlying group communication protocol is responsible for reliable delivery of group messages.

The proposed OCI extension creates a group object reference using a group identifier provided by a

group communication protocol. A group object reference is denoted as IOR and expanded into a group identifier at OCI Transport. A client establishes a communication path to a group using the corresponding IOR. Here a concrete implementation of GIOP is needed for group communication because GIOP is an abstract protocol which does not have a group address available as an endpoint information. We therefore extend GIOP to Group Communication Inter-ORB Protocol (GCIOP). For group membership, the proposed extension provides a set of interfaces to group membership operations such as group creation, destruction, join, and leave. It is provided as a part of an Acceptor in OCI. State transfer interface is provided via an Acceptor, and state information is saved in Acceptor Info as a CORBA object type. Message ordering is a key factor for enabling the object group members to maintain a consistent state among them when more than one client interacts with the group.

3.2. System Architecture

The proposed system consists of a group communication protocol and an ORB supporting OCI. As underlying group communication protocol, we exploit the fault tolerant group communication service (FTGCS) [7], implemented in Java. FTGCS guarantees reliable group communications between the members. It has a group communication component and a group membership management component. The group communication component supports message multicasting and reliable delivery of message. The group membership management component manages dynamically a membership list and guarantees the consistency of a group view. The extended OCI supports interfaces of group membership, state transfer, ordering, group IOR, and communication components. Figure 1 shows how the underlying group communication protocol is incorporated with OCI.

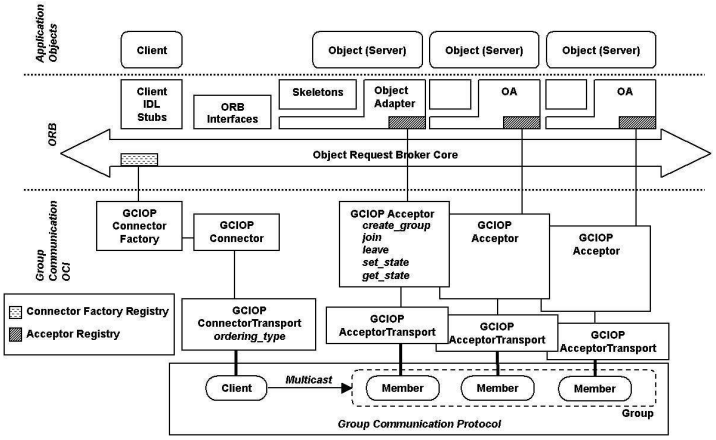

Figure 1. Extended OCI and Group Communication Protocol.

A client part consists of the ordering component that sets an ordering type for message delivery and the communication component that supports multicast communication in a group. In the extended OCI, the ordering component and the communication component are a TransportInfo and a Transport, respectively. At a server side, the group membership component allows a server object to compose a group via Acceptor operations. The state transfer component supports the state transfer mechanism and a Group IOR component constructs a group IOR as a group reference. A communication component as a part of a Transport uses FTGCS. Communication in OCI is performed between Transport objects.

A group member is instantiated in create_group() or join() function of Acceptor. Transport objects of a client side and a server side are created in connect() function of Connector, and accept() function of Acceptor, respectively. Then a group member is mapped onto Transport object. Actually Transport object can use a group communication protocol in a group communication OCI. It means that group communication is provided to application objects in CORBA. Figure 2 introduces GCIOP operations described in following subsections.

```
final public class Acceptor_impl extends CORBA.LocalObject implements OCI.GCIOP.Acceptor
{
   // Group Membership Component
   //
   // Create an object group.
   public void create_group(String group_name);
   // Add a member object to the group.
```

```
    public void join(String group_name);
    // Remove a member object from the group.
    public void leave(String group_name);
    // Transfer the state to the application object.
    public void set_state(org.omg.CORBA.Object state);
    // Return the state of the application object.
    public org.omg.CORBA.Object get_state();

    // Group IOR Component
    //
    // Add a new pofile that matches an Acceptor to an IOR.
    public void add_profile(OCI.ProfileInfo profileInfo, org.omg.IOP.IORHolder ior);
}

final public class TransportInfo_impl extends CORBA.LocalObject implements OCI.GCIOP.TransportInfo
{
    // Ordering Component
    //
    // Return the ordering value.
    public OrderingType ordering_type();
    // Set the ordering value which a client requires.
    public void ordering_type(OrderingType value);
}

final public class Transport_impl extends CORBA.LocalObject implements OCI.GCIOP.Transport
{
    // Communication Component
    //
    // Send a buffer's contents through FTGCS.
    public void send(OCI.Buffer buf, boolean block);
    // Receive a buffer's contents through FTGCS.
    public boolean receive_detect(OCI.Buffer buf, boolean block);
}
```

Figure 2. GCIOP Operations.

3.3. Group Membership Component

Group membership component provides dynamic group membership and guarantees the consistency of a group view by using group operations, view change mechanism, and state transfer. Acceptor's operations of the extended OCI support a group composition because group member objects reside at a server side and a server-part OCI is an Acceptor object. This section describes what kinds of messages are communicated between member objects and a group communication system to create a group, and then explains the procedure of a state transfer and a view change mechanism when a new comer joins.

We assume that a group creator as the first joining member is a primary member. When members receive a view-change notification message, they report the own view information to the primary member and are blocked until receiving a view-install message from the primary member which is responsible for the consistency of a group view. After confirming the same view reported from all members, the primary member multicasts a view-install message to members. In the proposed mechanism, the primary member

object can use Acceptor's operations which are mapped onto corresponding operations of a group communication system. View change mechanism is completed as receiving a view-install message. Through this procedure, an Acceptor has the reference of a group member from the underlying group communication system and server objects are connected with a given transport protocol.

Group management operations are provided in an Acceptor object, which supports dynamic group membership. However, group view and consensus of group members rely on FTGCS. When Acceptor's functions bind an object application with group members in FTGCS, they need the information of an object application and a target group. Local variables such as group_name_, host_, and gm_ are stored in an Acceptor and used as parameters of Acceptor's functions. Acceptor has the group_name_ of a group member in FTGCS. It is also the same name as that of an application object. host_ stores the local machine name and is used in constructing the profile information. gm_ indicates a group member in FTGCS, and this value is transmitted to Transport object via a function parameter. Especially one group member is mapped onto only one Transport and all group members share a group name as a static type.

An Acceptor has several functions to support a group membership, a state transfer, and communications among group members as shown the IDL specification of an Acceptor. These attributes promote an efficient execution as a common value of Acceptor functions. On the other hands, operations are allowed to support dynamic group membership and simply implemented as mapped onto the corresponding operations of FTGCS, i.e. `gm_.Create_group (group_name_)', `gm_.Join (group_name_)', and `gm_.Leave()'. After creating a group member, state transfer functions such as set_state() and get_state() call the corresponding functions in FTGCS like group membership components.

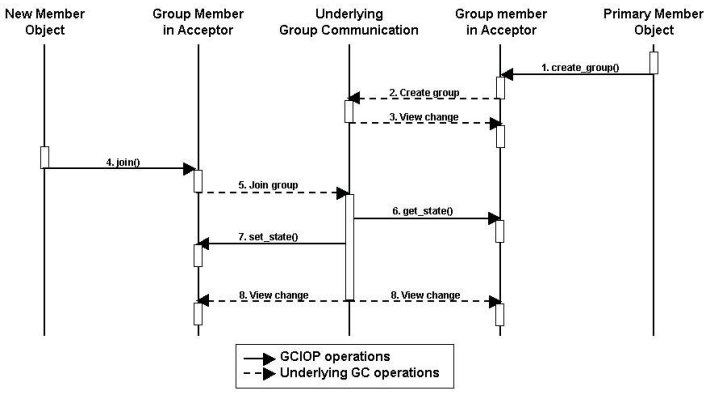

Figure 3. Interaction Diagram in Group Membership Model.

Figure 3 shows an interaction diagram in a group composition. Initially, the group is empty. The primary member object calls create_group() in an Acceptor (1). Then the create_group() function internally calls a corresponding function and joins the group in a group communication system (2). When a view-install message is arrived in the underlying group communication system, the reference of a group member is passed to an Acceptor (3) and the group creation procedure is finished. When the second member as a newcomer joins the group, the state transfer mechanism is required in addition to the composition of a group. The newcomer object calls join() in an Acceptor (4) and joins a group in the underlying group communication system (5). The state transfer is performed before a view install message is sent, and its operations are necessary because the state of a member can contain application-dependent data. The underlying group communication system gets the current member's state with a get_state() function in a primary member's Acceptor (6) and duplicates this state to a newcomer's state with a set_state() function (7). Lastly, its system multicasts a view change complete message to all members (8).

3.4. Group IOR Component

As server object is presented by IOR that ORB offers in the existing CORBA communication, the group communication OCI provides a group IOR to identify the group in CORBA group communication.

When an object exports its service, CORBA generates a corresponding IOR that includes transport information such as an IP address and a port number. ORB uses an IOR as the universal means of identifying an object. As such, a group IOR consists of a group name and a host name. The IOR is generated at run-time by the CORBA at the server, and is interpreted by a client application object. It consists of Type ID, Profile Count, and Tagged Profiles. The Type ID is based on the IDL of an object and provides the interface type of the IOR in the repository ID format. Profile Count is the number of Tagged Profiles. One or more Tagged Profiles exist in the IOR, and the Tagged Profile contains the information on the protocol supported by the target object.

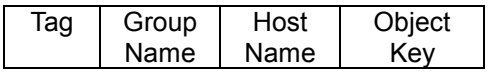

Figure 4. The format of Tagged Profile in GCIOP.

Figure 4 shows the format of a Tagged Profile used in GCIOP. **Tag** indicates the protocol used for object communications, which is represented by a constant value. Since `01' is already assigned for IIOP, we choose a different number for the proposed scheme. **Group Name** is used as a group identifier not only in ORB, but also in the underlying group communication protocol. **Host Name** represents a group member. **Object Key** identifies a particular object instance. Practically Acceptor fills up the profile information to construct the IOR at a server side. ProfileBody of a group IOR generated from IDL specification. Actually add_profile function makes the information of ProfileBody which is filled up based on GCIOP.

3.5. Group Multicast Component

The proposed system's communication model is one-to-many, and thus one client transmits messages to multiple servers, that is, a group. A connection between a client and a group is accomplished by client's Connector and group's Acceptor. After connection establishment, one Connector and several Acceptors

instantiate a Transport object and these Transport objects can use the multicast communication within them. For this, a group multicast in the proposed scheme includes the prepared procedure except object invocation procedure due to the OCI communication mechanism.

3.5.1. Connecting a Client to Members

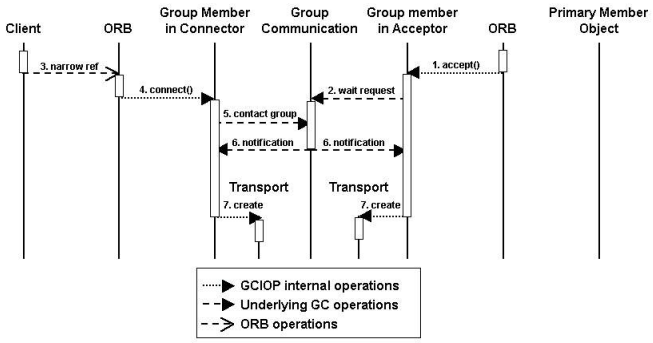

Figure 5. Procedure of connecting a client to members.

When the OA is activated, an Acceptor instantiates a process group member of the underlying group communication service and a server object atop the Acceptor joins a group through an Acceptor method. After that, the server object waits the request from a client. A member can dynamically join or leave a group using the methods of a GCIOP Acceptor. When the connection between a client and a group is established, an Acceptor instantiates a Transport object in the accept method and assigns a process group member into a Transport of a server object. After the initialization of a client, a Connector Factory Registry in ORB creates a Connector Factory. To get a GCIOP Connector Factory, the Connector Factory reference needs to be narrowed to a reference to a GCIOP Connector Factory. The Connector Factory creates a Connector based on the IOR, which the OA has published. Then a Connector creates a Transport.

Figure 5 illustrates the procedure of connecting a client to a group. The server side's ORB calls first an accept() function in an Acceptor (1). The Acceptor is desired to receive the notification for a connect request in a given transport protocol (2). On the other hand, a client maps the local proxy onto the remote object reference included in IOR (3) and the client side's ORB calls a connect() function in a Connector

(4). A Connector contacts a group by means of the group reference (5). Thereafter a group communication system returns the notification as a connection establishment to a Connector and Acceptors (6). Corresponding Transport objects are created by each Connector and Acceptor and transmitted each group member reference which a Connector and an Acceptor contain (9). Communications between a client and a group is performed with Transport objects.

3.5.2. Object Invocation

Operations of the communication component are automatically requested by the ORB as an internal operations in CORBA. After Transport objects make a connection between client and server, they exchange messages by means of send and receive functions via Buffer object. Transported data is a byte stream. Since the default communication between Transport objects is the IIOP and interfaces for a group communication uses the same name as an interface, we need to implement IIOP-based operations, i.e. send_detect, receive_detect, send_timeout, and receive_timeout functions which are similar to send and receive. These functions call the corresponding functions in FTGCS, i.e. void SendMessage(byte[] data, OrderingType ordering, int atomic) and McastMsg Receive(). Notice that only contents of McastMsg such as a byte stream must be taken a copy to Buffer object. Following operations in a Transport are used when a client and server have blocking mode and threaded mode.

- public void send(OCI.Buffer buf, boolean block)

Buffer object used for the message exchange between a client and a group is the intermediate object. Block option is simply transferred to the underlying protocol. In this function, SendMessage(buf.data(), info_.ordering_type(), atomic) is called. buf.data() is a message itself as the byte array. info_.ordering_value() is the ordering value in the previous section and atomic option is the default value of FTGCS.

- public boolean receive_detect(OCI.Buffer buf, boolean block)

receive_detect function is repeatedly performed to confirm whether the message sent from a client is arrived to a receive queue in FTGCS. If a received message do not exist, this function returns FALSE. Otherwise this function internally calls gm_.Receive() method and returns TRUE. Then the received message is transmitted into Buffer object.

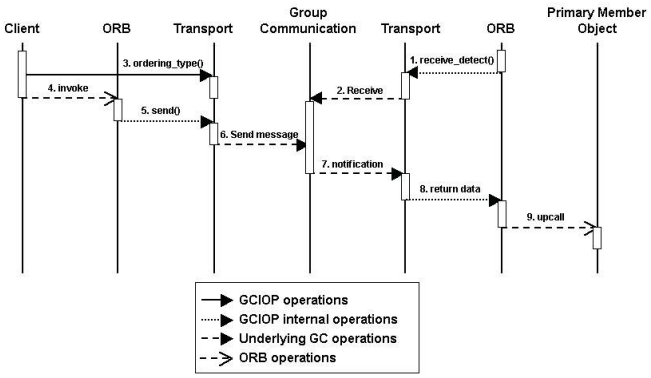

Figure 6. Procedure for Object Invocation.

Figure 6 shows the procedure for object invocation. When a client invokes a remote object via ORB, low level communication is basically a message passing. In group communication, an ordering type of data transmission is changeable at run time. The proposed system allows a client to set the ordering type by ordering_type function in a TransportInfo class (3). In the viewpoint of implementation, the main difference between ConnectorTransportInfo and AcceptorTransportInfo is whether ordering operations in GCIOP TransportInfo object is allowed or not. Only the ConnectorTransportInfo object allows an application programmer to use an ordering operation such as setting an ordering type. A ConnectorTransportInfo has an ordering variable as an OrderingType which are enumerated as casual and total ordering. We can get the ordering value as the return value of ordering_type(). On the other hand, ordering_type(OrderingType value) stores a value parameter to value_ which is actually applied to message delivery in a Transport. A send operation in a Transport calls uses an ordering value as the parameter of a send() function, i.e. SendMessage(buf.data(), info_.ordering_type(), atomic).

After selecting the ordering type, a client multicasts messages to a group in earnest. If a client requires an object invocation (4) at a server side, the client side's ORB calls a send() function in a Transport (5) and a message is transmitted to a group via a SendMessage() function of the group communication system (6). While the server side's ORB continuously examines a message arrival by using a receive_detect() function of a Transport object (1). A receive_detect() function has a boolean return type; if a return value is true, a server gets a message by using a Buffer object as the parameter of a receive_detect() function; otherwise the procedure of (1), (2), (7), and (8) is repeated. Lastly the ORB makes a call to a server object and the object invocation is complete (9).

3.6. Group Object Key

Multicast messages require the receivers to use a common object key for the operation that is invoked when a message arrives. However an object key contained in the proposed group IOR does not identify an object implementation in each member because of reusing an existent object key. An existing BOA and POA models does not allow all object implementations of a group to be presented by a single object key [20]. A new object adapter (OA), which generates a group object key, needs to be redesigned for a group object model in CORBA. Since we intend not to change ORB, we, instead, substitute each member's object key for an object key included in the client's request message. We assume that all members know representative object key of a group, which indicates a primary member's object key, by means of some extra mechanism such as a naming service. Every member's ORB creates an object key of own object implementation. Each member's GCIOP Transport keeps its object key in the local variable. When the request message from a client arrives at GCIOP Transport, a member determines whether the received message is sent to the joined group or not. If the received object key is equal to the representative object key, it replaces the key with the local object key. Then the operations requested by a client will be invoked.

4. Performance Evaluation

This section describes the experimental results, comparing the proposed scheme with one-to-one remote invocations using IIOP. We use ORBacus Java 4.01 supporting OCI [15]. The proposed scheme is implemented as libraries, i.e., the Java Archive (JAR) file format. Its library consumes 213K bytes, including the files generated from GCIOP IDL. And the FTGCS library consumes 73K bytes. We run the experiments using ten hosts, one Dell PowerEdge 2300 (dual 500Mhz processors, 512MB of RAM), one Compaq Proliant (dual 6/800Mhz processors, 512MB of RAM), one Armada notebook (450Mhz processor, 192MB of RAM), and 7 PCs (Pentium III processor, 384MB or 256MB of RAM) running Windows 2000 connected by a local 10Mbit Ethernet network. Each machine runs a single server object. A client runs on Compaq and makes an invocation to a server object group. We run the experiments 50 times.

Table 1. Multiple object invocation using GCS.

# of Server Obj.	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10
Invocation latency (ms)	172.62	174.98	171.36	171.13	174.70	171.60	174.75	173.60	173.85
FTGCS (ms)	124.34	124.33	124.46	123.04	124.25	124.35	124.80	123.55	124.30
Group interface time (ms)	48.28	50.65	46.9	48.03	50.45	47.25	49.95	50.05	49.55

The proposed scheme executes multiple object invocations as only one invocation by means of FTGCS. When a client makes an invocation to server objects at first time, the Transport objects of a client and server objects are instantiated because an OCI-based communication requires the communication between a client and server objects via a Transport object. After creating a process group member in an Acceptor, a server object transfers it into a Transport at the connection establishment. On the other hand, a client instantiates a process member in a Transport object. Table 1 shows the invocation latency including request/reply time, Transport object creation time, and the process member instantiation time at a client side. The latency is almost constant in the proposed scheme even if the number of server objects increases

since a single request performs multiple object invocations. Invocation time using GCS includes the cost consumed by GCS when the proposed scheme makes multiple object invocations. FTGCS is implemented atop IP multicast and maintains the group view and delivery queue to support group communication properties such as group membership and reliable communication. Additionally transmitted messages in FTGCS are converted into the proprietary message format including sender ID, message sequence number, ordering type, and so on. These required mechanisms for group semantics bring about a little overhead.

Table 2. Multiple object invocation using IIOP.

# of Server Obj.	# 1	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10
Invocation latency (ms)	74.06	167.38	248.39	332.66	440.82	536.04	649.55	736.67	816.28	912.40

Multiple object invocations in the standard CORBA using IIOP are supported by multiple one-to-one remote invocations because the standard CORBA does not support multicast communication. Table 2 shows the request/reply latency at a client as the number of server objects increases. In the ORBacus with the OCI, the connection establishment between a client and server object requires the Transport object instantiation. The values include the Transport creation time. The latency constantly becomes larger as the number of server objects increases. This is because a new connection between a client and a server object should be established whenever a server is added.

Figure 7 shows the comparison between multiple object invocation latency using IIOP and that using GCS. The experimental result shows that the invocation latency using IIOP constantly increases while the proposed scheme does not incur the extra overhead even if the number of group members increases.

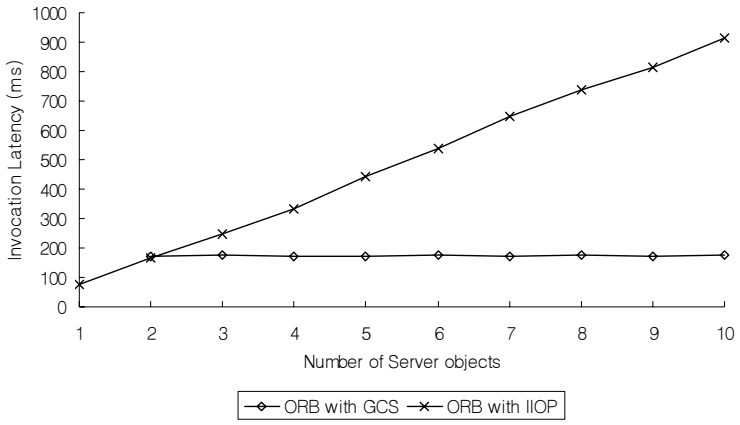

Figure 7. Object Invocation (milliseconds): Using IIOP and GCS.

5. Conclusion

Group communication is one of key components supporting object replication. CORBA provides little support for fault tolerance and high availability that can be supported by means of object replication. We have extended the OCI in order to fulfill the need on a generic group communication framework for CORBA. We have first defined GCIOP that has end-point information such as group name. Then we designed the group communication Info Object and the OCI to use group semantics. The proposed system architecture consists of group membership component, group IOR component, and group multicast component. Group membership component provides operations for dynamic group membership and guarantees the consistency of a group view through Acceptor's operations. To identify the group in CORBA group communication, group IOR component constructs a group IOR that is filled up based on the GCIOP information in an Acceptor. And group multicast component provides multicast communication within a group via each Transport object under a client and server objects. We described the internal procedure for each component. All the group semantics are exposed to the CORBA objects as mapping group operations in the proposed scheme into the corresponding operations in the underlying group communication protocol. We conduct multiple remote invocations using IIOP and using the proposed scheme among ten server objects. Experiment results show that the proposed scheme does not

induce the performance degradation even if the number of server objects increases.

We assume that the group communication model is one-to-many at present. Here a sender is located atop a Connector with multiple receivers atop an Acceptor. If application objects are constructed as a reactive client/server, which plays a role of both a client and a server, the proposed design can also be applied to a peer-to-peer model. Furthermore, it supports not only group communication in CORBA, but also existing group communication without any modification of ORB and dependency on the OS. We plan to apply other group communication protocols to the ORB using the proposed approach and evaluate the performance of systems based on the proposed scheme, comparing with existing group communication system in CORBA.

References

[1] K.P. Birman, ``The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37-53, December 1993.

[2] P. Felber, B. Garbinato, and R. Guerraoui, ``The Design of a CORBA Group Communication Service," *15th IEEE Symposium on Reliable Distributed Systems*, pp.150-159, October 1996.

[3] P. Felber and R. Guerraoui, ``Programming with Object Groups in CORBA," *IEEE Concurrency, 8 (1)*, pp. 48-58, Jan.-March 2000.

[4] R. Guerraoui and A. Schiper, ``Software-Based Replication for Fault Tolerance," *IEEE Computer*, *30 (4)*, pp. 68-74, April 1997.

[5] A.T. van Halteren, A. Noutash, L.J.M. Nieuwenhuis, and M. Wegdam, ``Extending CORBA with Specialised Protocols for QoS Provisioning," *International Symposium on Distributed Objects and Applications*, pp. 318-327, September 1999.

[6] *IN/CORBA Initial v1.1*, Object Management Group, OMG Document telecom/98-06-03, June 1998.

[7] D. Lee et al., ``Development of reliable group communication module for object group," *Project Report, CDS&N Lab., Information and Communications University (ICU)*, December 1999.

[8] S. Maffeis, ``Adding Group Communication and Fault-Tolerance to CORBA," *The 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.

[9] S. Maffeis, ``The Object Group Design Pattern," *The 1996 USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996.

[10] L.E. Moser, P.M. Mellar-Smith, P. Narasimhan, L.A. Tewksbury and V. Kalogeraki, ``The Eternal System : An Architecture for Enterprise Application," *International Enterprise Distributed Object Computing*, pp. 214-222, September 1999.

[11] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, R.R. Koch and K. Berket, ``Multicast Group Communication for CORBA," *International Symposium on Distributed Objects and Applications*, pp. 98-107, September 1999.

[12] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, ``Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance," *Third USENIX Conference on Object-Oriented Technologies and Systems*, pp. 81-90, June 1997.

[13] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, ``The Interception Approach to Reliable Distriubted CORBA Objects," *Third USENIX Conference on Object-Oriented Technologies and Systems*, pp. 245-248, June 1997.

[14] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, ``Using Interceptors to Enhance CORBA," *IEEE Computer*, pp. 62-68, July 1999.

[15] *ORBacus Web Site*: http://www.ooc.com/ob/, Object-Oriented Concepts, Inc.

[16] *Portable Interceptors: Joint Revised Submission*, Object Management Group, OMG Document orbos/99-12-02, December 1999.

[17] *Portable Interceptors: Request for Proposal*, Object Management Group, OMG Document orbos/98-09-11, September 1998.

[18] D.C. Schmidt, ``Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services," In: R. Martin, F. Buschman, and D. Riehle, *Pattern Languages of Program Design 3*, (Addison-Wesley, 1997).

[19] *The Common Object Request Broker: Architecture and Specification, Rev. 2.3*, Object Management Group, OMG Document formal/98-12-01, June 1999.

[20] *Unreliable Multicast Inter-ORB Protocol: Initial Submission*, Eternal Systems, Inc. et al., OMG Document orbos/2000-02-03, February 2000.