Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance

P. Narasimhan, L. E. Moser, P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA

and software to be replaced transparently, thereby permitting the evolution of a system with no interruption of service to the application. In addition to the Replication Manager, the Eternal system provides a Resource Manager and an Evolution Manager that handle these challenging issues.

References

[1] D. A. Agarwal, *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser and C. J. Scheiman, ‘‘Extending the operating system at the user level: The Ufo global file system,’’ *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA (January 1997), pp. 77-90.

[3] P. Felber, B. Garbinato and R. Guerraoui ‘‘Designing a CORBA group communication service,’’ *Proceedings of the IEEE 15th Symposium on Reliable Distributed Systems*, Niagara on the Lake, Canada (October 1996), pp. 150-159.

[4] B. Janssen, D. Severson and M. Spreitzer, ILU 1.8 Reference Manual, Xerox Corporation (May 1995), ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[5] S. Landis and S. Maffeis, ‘‘Building reliable distributed systems with CORBA,’’ *Theory and Practice of Object Systems*, John Wiley & Sons Publishers, New York (1997).

[6] C. A. Lingley-Papadopoulos, *The Totem Process Group Membership and Interface*, M.S. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[7] M. C. Little and S. K. Shrivastava, ‘‘Object replication in Arjuna,’’ Broadcast Technical Report 93, Esprit Basic Research Project 6360, University of Newcastle (1994).

[8] S. Maffeis, ‘‘Adding group communication and fault-tolerance to CORBA,’’ *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA (June 1995), pp. 135-146.

[9] S. Maffeis and D. C. Schmidt, ‘‘Constructing reliable distributed systems with CORBA,’’ *IEEE Communications Magazine*, vol. 35, no. 2 (February 1997), pp. 56-60.

[10] G. Minton, ‘‘IIOP specification: A closer look,’’ *UNIX Review*, vol. 15, no. 1 (January 1997), pp. 41-50.

[11] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, ‘‘Totem: A fault-tolerant multicast group communication system,’’ *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification* (1995), Revision 2.0.

[13] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, ‘‘Replica consistency of CORBA objects in partitionable distributed systems,’’ *Distributed Systems Engineering*, vol. 4 (September 1997), pp. 1-12.

[14] J. Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons Publishers, New York (1996).

[15] R. M. Soley, *Object Management Architecture Guide*, Object Management Group, OMG Document 92-11-1.

[16] D. C. Sturman and G. Agha, ‘‘Extending CORBA to customize fault-tolerance,’’ Technical Report, Department of Computer Science, University of Illinois (1996).

[17] S. Vinoski, ‘‘Distributed object computing with CORBA,’’ *C++ Report*, vol. 5, no. 6 (July/August 1993), pp. 32-38.

[18] S. Vinoski, ‘‘CORBA: Integrating diverse applications within distributed heterogeneous environments,’’ *IEEE Communications Magazine*, vol. 14, no. 2 (February 1997), 46-55.

```
while Replication Manager is running do
 obtain an intercepted system call of the object with the arguments
   case <system call intercepted>
    open() :
      execute Process_Connect() using the object's pgid
      execute Process_Join() using the object's pgid
    close() :
      execute Process_Leave() using the object's pgid
    poll() :
      execute Process_Poll() using the object's pgid
    read() :
      extract the data part of the system call
      execute Process_Receive()
    write() :
      extract the data part of the system call
      obtain the receiver's pgid using the file descriptor in the call
      execute Process_Send() using the receiver's pgid
   endcase
endwhile
```

Figure 9: Algorithm executed by the Replication Manager to communicate messages over the process group layer.

at the other end of the connection (the server object, in this case), and record the association between the connection file descriptor and the server object group identifier.

Once this association is registered, the *read()*, *write()*, and *poll()* calls made by the client object on the file descriptor can be intercepted and subsequently mapped appropriately by the Replication Manager to the known server object (process) group identifier, as shown in Figure 9.

At the server object, the Replication Manager extracts the client's object group identifier from the information packed by Totem into the client object requests that arrive. The Replication Manager then associates this information with the file descriptor of the TCP/IP connection established by the server object to communicate with the client object. Thus, intercepted system calls on the file descriptor at the server object can also be similarly mapped to the appropriate client object group identifier.

6.2 Detection of Duplicate Operations

In the course of their interactions with other objects, the replicas of an object may give rise to duplicate invocations and responses. These must be suppressed at the sender or the receiver since duplicate operations on an object can potentially corrupt its state. The Eternal system accomplishes the detection and suppression of duplicate operations by means of operation identifiers [13], which are assigned by the Replication Manager.

When a replicated object transmits requests or responses, the Replication Manager ensures that the object's own object group identifier is included in the list of object groups that are to receive the request message. This does not imply, however, that the replicas in the object's own object group consider the incoming request message as an operation to be performed. The ''loopback'' mechanism serves only to notify the object's own object group of the transmission of the request.

Thus, every replica of the object that receives messages containing the invocations or responses of another replica in the same object group can suppress its own invocations or responses. The Replication Manager detects these duplicate operations by extracting the operation identifier from the messages that it receives from the process group layer, and then comparing the identifier with those it has already received. If the Replication Manager has already received a message containing this invocation or response, it discards the message, thereby preventing it from reaching the object and corrupting its state.

6.3 Replication Schemes

Eternal is equipped to handle both active and passive replication in a manner that is transparent to the ORB, as well as to each replicated object. Active replication, in which each operation is performed by every replica of the object, requires the detection and suppression of duplicate operations, as well as the use of the object group mechanisms.

Passive replication, in which only a designated primary replica performs each operation, requires additional mechanisms to ensure consistency of the states of the replicas. The Replication Manager performs a state transfer from the primary replica to the secondary replicas at the end of each operation. Thus, after the primary replica completes each operation, the Replication Manager of the primary replica multicasts the primary replica's updated state to the object group containing the primary replica.

7 Conclusion

The Eternal system is a CORBA 2.0-compliant system that enhances CORBA by providing replication, and thus fault tolerance, in a manner that is transparent to the application and to the ORB. The ORB can employ these replication mechanisms without having to undergo any modification to its internal structure.

We are currently implementing the Eternal system using various commercial implementations of CORBA, including the CORBA-compliant Inter-Language Unification (ILU) [4] from the Xerox Palo Alto Research Center. The techniques used in Eternal are completely generic and can interwork with any commercial CORBA implementation that is capable of communication over IIOP.

The replication of objects finds its use not only in achieving fault tolerance, but also in allowing system hardware

```
obtain interface name of object from the Interceptor
look up the table of mappings of interfaces to object groups
if interface name present in the table then
   extract the object's object group identifier ogid
else
   assign a unique object group identifier ogid
   record the interface name and its ogid in the table
endif
communicate the ogid for this interface name to the Interceptor
```

Figure 8: Algorithm executed by the Replication Manager to assign a unique process (object) group identifier for each object.

5.4 Mapping IIOP to Totem

The system calls of the CORBA objects communicating over the IIOP interface are analogous to the routines that the process group interface presents to an application process. In this context, an application process corresponds to a CORBA object in the system, and the process group identifiers of Totem correspond to the object group identifiers of Eternal.

The *open()* system call to TCP/IP corresponds to the *Process_Connect()* routine of the process group interface, since both the call and the routine are involved with the establishment of connections. The assignment of the object (process) group identifier is handled by the Replication Manager, as discussed in Section 6.

The *close()* system call on an open file descriptor corresponds to the *Process_Leave()* routine only if the file descriptor involved is the principal one, since in this case both the call and the routine correspond to the ''tearing down'' of established connections. If the *close()* system call is invoked on any file descriptor other than the principal one for a server object or the last open file descriptor for a client object, the *close()* call simply causes the Replication Manager to remove any association of the file descriptor with the object group identifier. The *Process_Leave()* routine effectively disconnects the process from the process group controller and is, thus, invoked only when the object is to be destroyed or removed from the object space.

The *read()*, *write()*, and *poll()* system calls find their counterparts in the *Process_Receive()*, *Process_Send()*, and *Process_Poll()* routines of the process group interface. The send and receive buffers of the application objects contain the information that is sent or received over Totem via the process group interface. However, the captured *read()* and *write()* system calls cannot be mapped directly onto the routines of the Totem process group layer since the Interceptor must first associate the file descriptors in the system calls with the process group identifiers in the process group interface routines.

There may be several CORBA objects to which the Interceptor ''attaches'' itself. Each such object may service multiple requests at the same time, which means multiple connections must be managed for each object. However, for the purposes of replication, an object is associated with only one object (process) group, all the members of which are identical. Thus, each replica of a replicated object is a member of an object (process) group with a unique object (process) group identifier.

The functionality of the Interceptor is implemented using the algorithm shown in Figure 7. The Interceptor does not handle all aspects of the object group mechanisms; it utilizes the services of the Replication Manager for this purpose.

6 The Eternal Replication Manager

6.1 Assignment of Object Group Identifiers

The object groups that are used for replication are handled by the Replication Manager. At creation time, when an object informs the Interceptor of its Unix process identifier, it also conveys the name of its interface. The Interceptor hands this information over to the Replication Manager, which associates a unique object (process) group identifier for each interface name. Since the interface name, rather than any ORB-specific name, is used for this association, objects implementing the same interface, but operating over different ORBs, can be members of the same object group and are treated as replicas from this viewpoint.

The Replication Manager maintains a globally accessible table of the mapping between object (process) group identifiers and interface names. Each time an object is created, if it is the first replica of the object in the system, an entry is created in this table for the object's interface and a unique object (process) group identifier is assigned to it. When further replicas of the object are created and distributed across the system, this table is referenced to ensure that all of the replicas of the object are assigned to the same object group. The object group identifier is assigned or discovered by the Replication Manager, on behalf of the object, using the algorithm shown in Figure 8. In complete implementations of CORBA, the Interface Repository, which stores the interface definitions, can be used to register the object group identifier associated with each interface name.

When a server replica opens its ''listening'' connection, it discovers its object group identifier and joins its object group. When a client replica wishes to establish a connection to this server object, it must first discover its own object group identifier and join its object group. The client replica then must discover the object group identifier of the object

5.3 The Process Group Interface

The intercepted *read()*, *write()*, and *poll()* system calls are also mapped onto their corresponding calls in the Totem system interface. It is crucial that the underlying multicast group communication system, in our case Totem, possesses an interface to which the Interceptor and the Replication Manager can map these intercepted system calls.

In order that the services be provided to the application transparently, the group communication system must provide a simple interface to enable the objects that constitute the application to invoke its services. The interface that Totem provides to an application above it is designed to hide the implementation details of the underlying protocols by presenting only a small number of essential primitives that the application needs to use. The interface is intended to be simple and elegant and yet to allow the application to exploit fully the process group mechanisms of Totem.

On each processor, a process group controller manages all of the process groups on that machine. For each process group on that processor, the process group controller maintains information about the member processes (both local and remote) and provides membership services for joining the group, leaving the group and updating the membership. It also maintains a list of the process groups hosted by the machine.

To establish a connection with the process group controller of Totem, the application calls the *Process_Connect()* routine, supplying the identifier of the process group to which the application process wishes to connect. If the process group does not exist, the process connects to a process group of which it is the only member. The routine returns the identifier of the communication socket that connects the process to the process group controller.

To join a process group with which it has established a connection, a process calls the *Process_Join()* routine, supplying the identifier of the process group that it wishes to join, as well as the identifier of the socket between the process and the process group controller. The *Process_Leave()* routine, which takes the same arguments, initiates the removal of a process from the specified process group.

A process can send messages to a process group using the *Process_Send()* routine with the receiving process group identifier and the message to be sent as arguments. A process can receive messages from another process using the *Process_Receive()* routine with the receiving buffer as an argument. The received message is disassembled and the identifier of the sending process is extracted from the message header, along with information about the process groups to which the message is addressed.

The socket between the application process and the process group controller can be polled for any messages pending delivery, using the *Process_Poll()* routine. Routines are also supplied to close the communication socket, once the process disconnects from the process group controller.

The calls on the IIOP interface are mapped, through the Interceptor and the Replication Manager, to the process group interface of the Totem system. The implementation of Eternal makes it possible to use any multicast group communication system, as long as it provides the same fault tolerance guarantees and a similar process group interface as Totem. The set of routines that the Totem process group interface uses facilitates the mapping of the IIOP calls onto Totem. The routines can be provided as a library that is used by the Interceptor and the Replication Manager.

```
while Interceptor is running do
 listen for any newly created CORBA objects
 for each CORBA object created do
  obtain process identifier pid and interface name of the object
  obtain object group identifier ogid from the Replication Manager
  specify the system calls (those used by IIOP) to intercept
  while the object is operational do
   wait to intercept the specified system calls when they occur
   case <system call intercepted>
    open() :
      if first open() on /dev/tcp then
       record this as the primary file descriptor for this ogid
       invoke Replication Manager to handle this system call
      endif
      if subsequent open() on /dev/tcp then
       add file descriptor to the list of descriptors for this ogid
       obtain ogid of the object at the other end of the connection
      endif
    close() :
      if server and close() on the primary file descriptor then
       invoke Replication Manager to handle this system call
      endif
      if client and close() on the last open file descriptor then
       invoke Replication Manager to handle this system call
      endif
    poll() :
      if poll() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
    read() :
      if read() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
    write() :
      if write() on the previously recorded file descriptor then
       invoke Replication Manager to handle this system call
      endif
   endcase
   resume the operation of the object
  endwhile
 endfor
endwhile
```

Figure 7: Algorithm executed by the Interceptor.

System Calls of the IIOP Interface	Routines of the Process Group Interface
open(*fd*)	**Process_Connect**(*pgid*), **Process_Join**(*pgid*)
close(*fd*)	**Process_Leave**(*pgid*)
read(*fd*, <buffer to read data into>)	**Process_Receive**(<buffer to receive data>)
write(*fd*, <data>)	**Process_Send**(*pgid*, <data>)
poll(<list of *fd*s >)	**Process_Poll**(<socket to the process group contoller>)

Figure 5: Correspondence between the IIOP system calls and the process group layer routines. Here, *fd* refers to the file descriptor returned from opening $/dev/tcp$, and *pgid* refers to the process group identifier. Only the arguments that are relevant to the mapping are shown.

connection with the process group interface of a reliable group communication system, such as Totem, in anticipation of any communication that might follow. Thus, a given object, via the file descriptor associated with this first *open()* call, is associated with a particular connection to the Totem system interface. Subsequent *open()* calls to TCP/IP, which represent client-server communication, are recorded by means of their file descriptors, which are then monitored for any activity. All client-server interactions on these file descriptors can be channelled through the respective connections of the client and the server to Totem.

5.2.2 *poll()* System Call

A server object, on establishing its principal ''listening'' connection, polls its associated TCP/IP file descriptor, and blocks till it hears from a client object that requires its services. A *poll()* call may also be executed in the middle of a series of client-server interactions, when either object is waiting in anticipation of communication from the object at the other end of the TCP/IP connection. It is also possible for an object to poll several file descriptors simultaneously for activity.

5.2.3 *read()* and *write()* System Calls

Typical communication between objects in the Eternal system consists of a sequence of *read()* and *write()* system calls that operate over IIOP. For each object, these system calls are associated with a file descriptor on which they are invoked. The Interceptor records and monitors all of the active file descriptors associated with an object, and the Replication Manager maps these file descriptors onto the underlying multicast group communication system, in our case Totem. Thus, any system call that uses one of these file descriptors can be mapped to the Totem system interface.

The *read()* and *write()* system calls are associated with receive and send buffers, respectively, that store the information that is received or is to be sent. The contents of these buffers represent the user-level abstractions of the messages that are communicated between client and server objects.

The *read()* and *write()* system calls used by IIOP contain, in the first few bytes, the GIOP header. The IIOP *read()*s and *write()*s are distinguished from other *read()*s and *write()*s by the first four bytes of the data, which represent the **magic** field of the GIOP header, as shown in Figure 6. This field, along with the list of file descriptors associated with the TCP/IP connections, helps in discarding any *read()*s and *write()*s that might not require the IIOP interface and, thus, are not of interest.

5.2.4 *close()* System Call

Since the *open()* system call is intercepted, the *close()* system call for each associated file descriptor must also be intercepted. This call is typically invoked when a client object wishes to close a TCP/IP connection once it has completed communication with a server object. It can also be used to ''tear down'' the principal connection of a server object, thereby removing it from the CORBA object space.

The *close()* system call, like the *open()* system call, must be handled at both server and client objects. At both client and server objects, Eternal deletes any reference to the file descriptor associated with the connection (that is now being closed). Thus, if the object reuses the same file descriptor for future connections, a new association will be registered.

```
struct MessageHeader {
    char    magic[4];
    Version    GIOP_Version;
    boolean    byte_order;
    octet    message_type;
    unsigned long    message_size;
};
```

Figure 6: Structure of the header of a GIOP message.

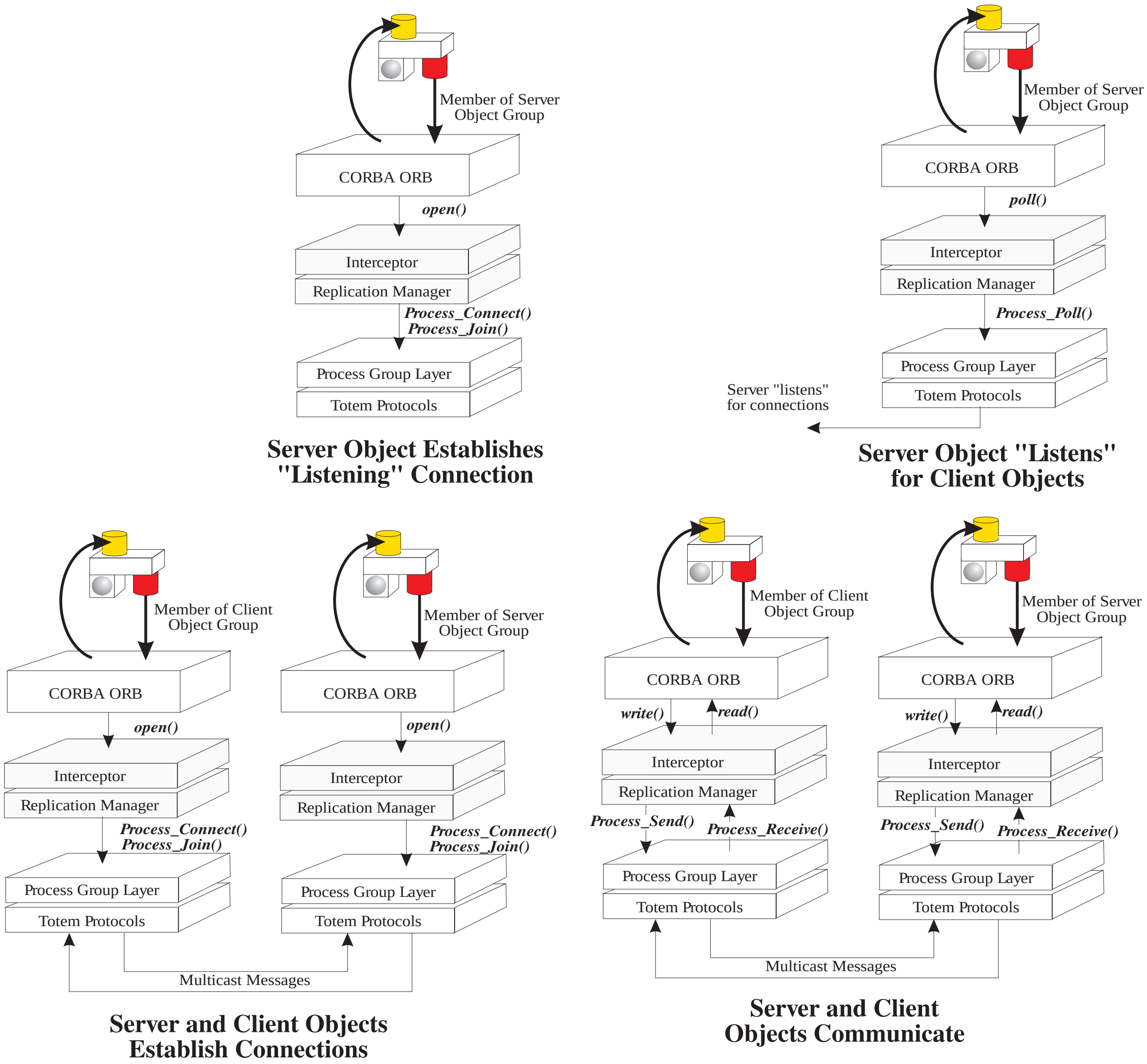

Figure 4: Mapping the IIOP interface onto the process group interface.

5.2 IIOP-Specific System Calls

Each time a server object publishes its identity or each time an object interacts with any other object in the system, the IIOP interface is used. If the ORB's native protocol is IIOP itself, this use is unnecessary.

5.2.1 *open()* System Call

Since the IIOP interface uses TCP/IP, the *open()* system call to TCP/IP is among those intercepted. The file descriptor returned from this call is recorded so that it can be monitored for activity by the system. There are two cases in which an *open()* call may be invoked. In the first case, a connection over TCP/IP is established by a server object that publishes its Interoperable Object Reference (IOR) across the network and ‘‘listens’’ for any client object that requires its services. The second case occurs when a client object requests service from a server object and the two objects establish separate connections to TCP/IP in order to communicate.

Thus, each server object has a principal TCP/IP connection, on which it ‘‘listens’’ for clients, and establishes additional TCP/IP connections when the client objects desire to communicate with the server object. The additional TCP/IP connections are typically open for the lifetime of the client objects, while the principal ‘‘listening’’ TCP/IP connection is open for the lifetime of the server object.

The first *open()* call to TCP/IP, in turn, triggers the Replication Manager, via the Interceptor, to establish a

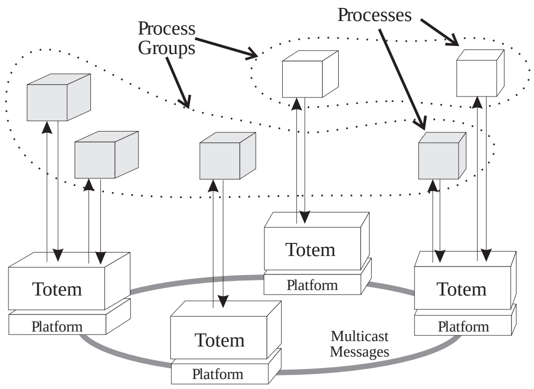

Figure 2: Process groups in Totem.

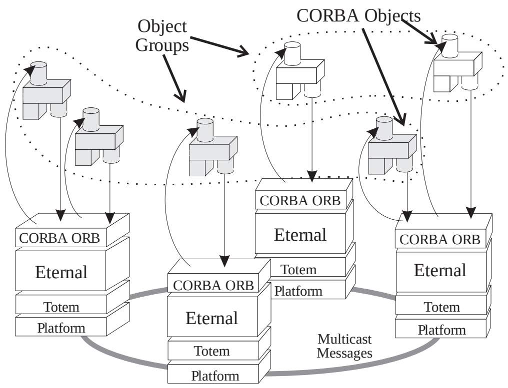

Figure 3: Object groups in Eternal.

operating system can be extended at the user level, without requiring modifications to the kernel or to the standard system libraries. One way of doing this is by intercepting system calls from specified processes before these calls reach the kernel, and then modifying these system calls to implement the desired functionality. The mechanisms are entirely transparent to an application process whose system calls are intercepted.

Such an approach is useful for the development of global file systems [2] and for the testing of kernel extensions. The Eternal system employs the same approach to ''attach'' itself transparently, via the Eternal Interceptor, to all objects that operate over a CORBA 2.0-compliant ORB.

5.1 Intercepting System Calls

The Eternal system can ''attach'' itself to any CORBA object and can ''catch'' a specified set of system calls that are made by the ORB during the object's interactions with the system. To do this, the Interceptor, given the process identifier pid assigned by Unix to the object being intercepted, locates and performs a continual trace on the file $/proc/pid$, which is part of the $/proc$ file system of the Unix system.

The system calls to be captured at either the entry to, or the exit from, the system call can be specified $apriori$. In the normal course of events, these system calls would reach the kernel and be executed. However, in Eternal, the tracing facilities provided under the $/proc$ interface are exploited to enable the specified system calls to be intercepted before they reach the kernel. The arguments, and possibly the return values, of these system calls can be extracted and examined, and the system calls can themselves be modified before they are forwarded to the operating system. Furthermore, all of these mechanisms can be implemented without the intercepted object being aware of their existence.

The obvious advantage of such an approach is that the operation of the Interceptor is transparent to both the CORBA objects and the ORB itself. This functionality can be implemented entirely at the user level, with no modification to the operating system. The application objects and the ORB need not be recompiled to take advantage of the intercepting capability. Once the Interceptor is started, it waits to receive a message from any newly created CORBA object. As a part of its initialization phase, every object supplies its Unix process identifier pid to the Eternal system. The Interceptor then monitors $/proc/pid$ for the entire lifetime of the object.

A typical CORBA object invokes many system calls during its lifetime. These include calls for memory allocation, runtime library access, file operations and network operations. While some of these calls may be local to the machine, any system call that constitutes communication with another object, whether local or remote, must take place using the ORB. The system calls of interest are those that are used by the objects to communicate over IIOP.

All CORBA objects in Eternal use the IIOP interface. The IIOP interface is a simple generic interface to TCP/IP, which makes capture of its calls easy. Since we are only interested in system calls that are IIOP-specific (communication-specific) and not object-specific, the system calls that need to be intercepted are the same for all of the objects operating over the CORBA ORB. Since interception of the calls is transparent to the ORB, any off-the-shelf commercial CORBA ORB, that is capable of communicating over IIOP, can be used unmodified.

Once the system calls of IIOP are intercepted by Eternal, the relevant arguments are extracted from the system calls and passed to the process group layer for communication over Totem. However, the ORB is unaware that its messages are delivered by Totem, since it ''believes'' that it is using only the IIOP interface, the calls of which were originally intended for TCP/IP.

cation. Server objects, that use IIOP to interact with their client objects in an environment of heterogeneous ORBs, publish their references in the form of IIOP IOR profiles.

The primary motivation for the use of IIOP is that all CORBA 2.0-compliant implementations can use this simple generic interface, irrespective of the internal details of the vendor's ORB, and the platform on which the ORB operates. A number of commercial ORBs now provide IIOP as their native protocol, since an increasing number of CORBA applications require interoperability over different platforms and the ability to operate over the Internet.

3 The Eternal System

The Eternal system is designed to work with any commercial off-the-shelf CORBA 2.0-compliant ORB with no modification whatsoever to the ORB. Moreover, the fault tolerance is provided in a manner that is transparent to the application objects.

Since the underlying fault tolerance capabilities are hidden from the application, the application programmer does not need to worry about the difficult issues of asynchrony, replica consistency, concurrency, and the handling of faults. The Eternal system replicates and distributes the application objects across the system, and allows the programmers to write the application as if it were a sequential program to be run on a single machine.

Fault tolerance is provided by replication [7] of both client and server objects across the distributed system. As shown in Figure 1, Eternal exploits the reliable totally ordered message delivery of the underlying Totem system to ensure replica consistency in the presence of faults. In addition, mechanisms are provided to detect and suppress duplicate operations and to support nested operations [13].

4 Group Communication Models

4.1 Process Groups

An increasing number of distributed applications are structured as collections of processes that interact or cooperate to accomplish a particular task. Such a collection of processes is called a process group and can be considered abstractly as a single unit, as shown in Figure 2. A process group may reside entirely within a single processor, or may span several processors.

A process group is characterized by its membership, and processes can be added and removed from the process group by the execution of a group membership protocol. A process is permitted to be a member of more than one process group, thereby resulting in intersecting process groups.

The services of a process group can be invoked transparently, with no knowledge of its exact membership or the location of its member processes. Thus, a process in the system can address all of the members of a process group (including its own) as a whole, using a multicast group communication system, such as Totem. A process can send messages to one or more process groups, of which it may or may not be a member. These messages are totally ordered within and across all receiving process groups.

The Totem system provides reliable totally ordered multicasting of messages to processes in process groups. Each message is assigned a unique timestamp, and these timestamps establish the total order of delivery of messages to the application. For messages multicast and delivered within the same configuration of processors, Totem provides these message delivery guarantees despite communication and processor faults, message loss, and network partitioning. The process group layer takes advantage of these services and guarantees of the underlying Totem protocols to provide reliable totally ordered multicasts within and across process groups.

4.2 Object Groups

Analogous to the notion of a process group, an object group is a collection of objects that cooperate to provide some useful service, as shown in Figure 3. This abstraction enables a client object to invoke the services of a server object group transparently, as if it were a single object. The server object group can also return the results to a client object group transparently, as if it were a single object.

An object group may consist of similar or dissimilar objects. In the Eternal system, a replicated object is represented by an object group, the members of which are identical and are the replicas of the object. Both client and server objects can be replicated and thus can be represented as object groups. The reliable totally ordered multicasts of Totem are used to communicate the invocations to, and the responses from, the object group. The replicas of an object receive the same operations in the same order, thereby ensuring consistency of the states of the object replicas. The exact location of the replicas of the object, the degree of replication, and the type of replication (active or passive) is transparent to an object that invokes the services of a replicated object.

5 The Eternal Interceptor

The Eternal Interceptor is a user-level layer between the ORB and the operating system. The principle underlying the design of the Interceptor is that the functionality of an

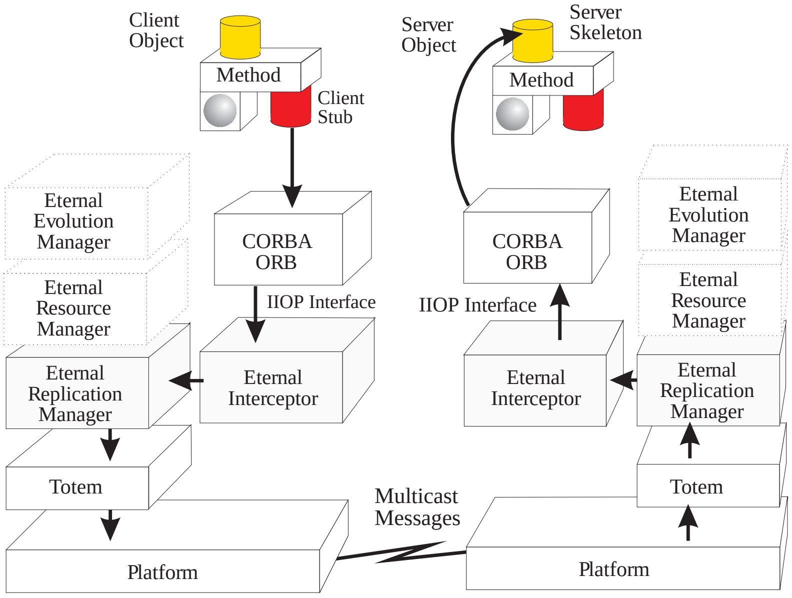

Figure 1: Structure of the Eternal system.

the system calls made by the ORB to TCP/IP, and also on the relevant part of the Replication Manager, which diverts the calls to Totem. In addition, Eternal supports the evolution of a system by exploiting the replication of objects to perform live upgrades of objects and their interfaces. Resource management is also provided for the creation, placement, and distribution of objects.

2 CORBA and the IIOP Interface

The CORBA standard specifies an interface for each distributed object. This interface is written in the declarative syntax of the OMG Interface Definition Language (IDL). The language-specific implementation of a server object is hidden from client objects that require the service provided by the server object; the server object can be invoked only through its interface.

Invocations of objects and responses from invoked objects are handled through the ORB, which acts as the intermediary or ‘‘communication bus’’ for all of the interactions between the distributed objects in the system. At a client object, a stub, generated by the IDL compiler, receives the request, marshals the call into the format appropriate to the request, and passes it to the ORB. At the server object, a language-specific mapping of the IDL specification, a skeleton, unmarshals the parameters of the call and performs any additional processing to invoke the appropriate method. The results of the operation are returned to the client object via the ORB.

CORBA provides location transparency, meaning that the client objects convey their requests only to their ORBs, which then undertake the task of locating a suitable server object and then dispatching the request to it. Thus, a client object need not be aware of the location of a server object since the ORB has access to this information. Every CORBA object is identified by an object reference, which is assigned to it by the ORB at the time the object is created. Client objects associate object references with their requests to enable the ORB to route their requests to the appropriate destinations.

The interoperability of CORBA arises in the context of communication between heterogeneous ORBs. Every CORBA 2.0-compliant ORB is equipped with the ability to communicate using the Internet Inter-ORB Protocol (IIOP) [10, 12], which ensures that objects running over different ORBs can interwork when they use the IIOP interface. Only the ORB hosting an object needs to know the details of the object, while other ORBs that wish to interact with the object need only be able to address it. Every object is assigned an Interoperable Object Reference (IOR) for this purpose.

The General Inter-ORB Protocol (GIOP) is a general set of specifications that enable the messages of the ORB to be mapped onto any connection-oriented medium that meets a minimal set of assumptions (reliable, byte stream-oriented, loss-of-connection notification). The Internet Inter-Orb Protocol (IIOP) is GIOP with the messages transported by TCP/IP. By sending IIOP messages over TCP/IP, the ORBs can use the Internet as the backbone for their communi-

Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance *

P. Narasimhan, L. E. Moser, P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
priya@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system is a CORBA 2.0-compliant system that provides, in addition to the location transparency and the interoperability inherent in the CORBA standard, support for replicated objects and thus fault tolerance. Eternal exploits the Internet Inter-ORB Protocol (IIOP) interface to ''attach'' itself transparently to objects operating over a commercial CORBA Object Request Broker (ORB). The Eternal Interceptor captures the IIOP system calls of the objects, and the Eternal Replication Manager maps these system calls onto a reliable totally ordered multicast group communication system. No modification to the internal structure of the ORB is necessary, and fault tolerance is provided in a manner that is transparent to both the application and the ORB.

1 Introduction

Distributed systems consist of clusters of computers that are capable of both functioning autonomously and cooperating harmoniously to achieve a particular task. The integration of an object-oriented paradigm with a distributed computing platform yields a framework in which objects are distributed across the system. Objects invoke other objects, or are themselves invoked, to provide services to the application.

The Object Management Group (OMG) has established the Common Object Request Broker Architecture (CORBA) [12, 14, 15, 17, 18], which is a standard for communications middleware that defines interfaces to distributed objects and that provides mechanisms for communicating operations to objects by means of messages. The key component of this architecture is the Object Request Broker (ORB), which handles requests to, and responses from, the objects in the distributed system.

*Research supported in part by DARPA grant N00174-95-K-0083 and by Sun Microsystems and Rockwell International Science Center through the State of California MICRO Program grants 96-051 and 96-052.

Unfortunately, the current CORBA standard makes no provision for fault tolerance, which has led to research aimed at making CORBA-based applications reliable. One approach has been to build the fault-tolerance capabilities into the ORB itself [16], as in Electra [8, 9] and in Orbix+Isis [5]. Another approach, adopted in the Open-DREAMS project [3], advocates that reliability be provided as part of the suite of object services available to the ORB. While the former approach makes the fault tolerance transparent to the application, it also involves considerable modification to the CORBA implementation to enable the ORB to take advantage of a multicast group communication system underneath it. On the other hand, the latter approach simply adds an object group service on top of an unmodified ORB, and uses no underlying multicast group communication system, thereby making the system interoperable and portable, but with the fault tolerance visible to the application programmer.

The Eternal system that we are developing provides fault tolerance transparently to the application using CORBA, without modification to the ORB. The mechanisms for achieving reliability are hidden from the application programmmer, and concern only the system developer. The Eternal system can utilize any commercial implementation of the CORBA 2.0 standard. Although Eternal is layered over a multicast group communication system, the vendor's ORB does not need to be altered to utilize the fault tolerance that Eternal provides. Furthermore, the system is designed to enable objects running over different ORBs to interact with each other.

The Eternal system exploits the services provided by the Totem multicast group communication system [1, 6, 11] to maintain the consistency of the replicas that are employed for fault tolerance. However, since Eternal only deals with interfaces of objects and of the ORB, any multicast group communication system, with an interface and guarantees similar to those of Totem, can alternatively be used.

The structure of the Eternal system is shown in Figure 1. In this paper, we focus on the Interceptor, which ''catches''