A Configurable CORBA Gateway for Providing Adaptable System Properties[1]

Mouna Seri*, Tod Courtney*, Michel Cukier**, Vishu Gupta*, Sudha Krishnamurthy*, James Lyons*,
HariGovind V. Ramasamy*, Jennifer Ren‡, and William H. Sanders*

*Coordinated Science Laboratory,
Electrical and Computer Engineering Department, and
Computer Science Department
University of Illinois at Urbana-Champaign
{seri, tod, vishu, krishnam, jlyons,
ramasamy, whs}@crhc.uiuc.edu

**Center for Reliability Engineering
Department of Materials and Nuclear Engineering
University of Maryland at College Park
mcukier@eng.umd.edu

‡ Bell Laboratories
Holmdel, NJ
reny@dnrc.bell-labs.com

1. Introduction and Motivation

Technologies for creating distributed object-oriented systems have evolved significantly over the last few years. While most early work focused on providing a functional abstraction (e.g., hiding specifics related to particular implementations of a function, or particular hardware or operating system characteristics), recent work has focused on providing higher-level, non-functional system properties, such as fault tolerance, intrusion tolerance, performance, timeliness, and consistency. Work in this area has varied widely with respect to the level at which the quality of service (QoS) is provided (e.g., network, operating system, middleware, or application) and the types of assurances that are provided to the user of the service.

At the object level, one approach that has been used is to create a proxy, called a *gateway,* that masquerades as the remote object for which the quality of service is requested. The function of the gateway depends on the nature of the quality of service provided, as illustrated in the gateways presented in [Ami95], [Nah99], [Kal00], and [Sch01]. This paper describes an adaptable gateway that can provide a wide variety of quality-of-service properties through the creation of "handlers" that can be selected at run time. Currently available handlers provide fault tolerance, intrusion tolerance, and timeliness/consistency tradeoffs. Section 2 gives an overview of the gateway architecture. Section 3 details the different handlers that provide the mentioned properties and some combinations of the properties. Finally, Section 4 concludes the paper.

2. Gateway Architecture

The gateway is a CORBA servant that acts as a proxy between two objects, usually replicated, to provide a certain quality of service. Exchanges between replicated objects include:

- state transfer information to ensure that all replicas of the same object maintain the same state at all times,
- communication and voting protocols that are specific to each replication policy and provide reliable message transmission, totally ordered messages across replicas, and automatic recovery from faults, and
- sequential or FIFO ordering guarantees.

Each replicated object has a dedicated gateway that gives every object the perception that it is talking to a single remote application. The gateway provides a standard IIOP interface to the local application and uses a group communication system to provide reliable multicast to remote objects. The group communication system used depends on the type of quality of service requested: Maestro/Ensemble [Hay98, Vay98] is used with handlers that provide fault tolerance and timeliness/consistency tradeoffs and an intrusion-tolerant group communication system is used with the handler that provides intrusion tolerance [Ram02]. A handler and a communication strategy implement each scheme in the gateway.

The handlers are responsible for sending and receiving applications' invocations to and from a remote object through the group communication system. There is one handler for each remote object with which the local application is communicating. The handler, implemented as a CORBA DSI (*Dynamic Server Interface*) servant, acts transparently underneath the application and impersonates the remote CORBA object with which the application is communicating. Therefore, the application has no knowledge of the group communication system.

Each handler in an object group is responsible for marshalling gateway messages into group communication system messages and then multicasting the messages to the other group members at the communication strategy's request. The receiving handler unmarshalls group communication system messages received through group communication system callbacks, converts them into gateway messages, and forwards them to the communication strategy for processing. In addition, the object group members execute the state transfer from old object group members to new ones by collecting and relaying the application and gateway state.

The handler's architecture is shown in Figure 1. The handler's communication strategy processes the messages received from the application and from the group communi-

[1] This research has been supported by DARPA Contract F30602-00-C-0172.

cation system. The strategy is the handler's centerpiece; it makes the decisions on how to handle every message received either from the application or from the group communication system, decides where to cast/send the message in one of the object groups, and decides whether or not the message should be buffered or removed from the buffers.

Each handler contains a reference to the DII Request Processor, which is a service object used for delivering messages to the application.

Each gateway maintains its own naming service, which provides hash-table-based transient storage for name-to-object bindings in a naming context. The reference to the application, the loaded handlers, and other useful components are stored in this naming context.

Handler and strategy factories construct the handlers and their communication strategies at initialization time. The factories register the handlers in the gateway's naming service under the handlers' names, which are provided as command line arguments.

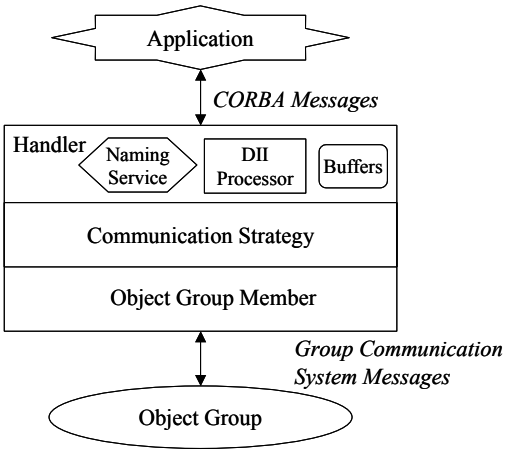

Figure 1. Handler Architecture

Finally, the starter factory is a service available for starting processes. It is used at initialization time to start the application process after the naming service is loaded and before the handlers are created.

The gateway is implemented in C++ using the Adaptive Communication Environment (ACE) framework [Sch94] and TAO, the ACE ORB [Sch98].

3. Handlers and Communication Strategies

Several handlers have been developed for the gateway architecture presented in the previous section. Some handlers provide fault tolerance through active and passive replication (tolerating crash failures and value faults). Another handler provides intrusion tolerance through the combination of replication and cryptography (tolerating arbitrary faults). Finally, other handlers provide timeliness/consistency tradeoffs combined with fault tolerance (tolerating crash and timing failures). The various handlers are shown in Figure 2 and will be detailed in this section.

3.1 Handlers that Provide Fault Tolerance

Two types of fault-tolerant replication techniques are currently: active and passive. In active replication, all members of an object group execute the method invoked on the object, whereas in passive replication, only one member of the object group, the leader, executes the method invoked on the group.

Applications that use active handlers must be deterministic, but determinism is not a requirement for passive handlers.

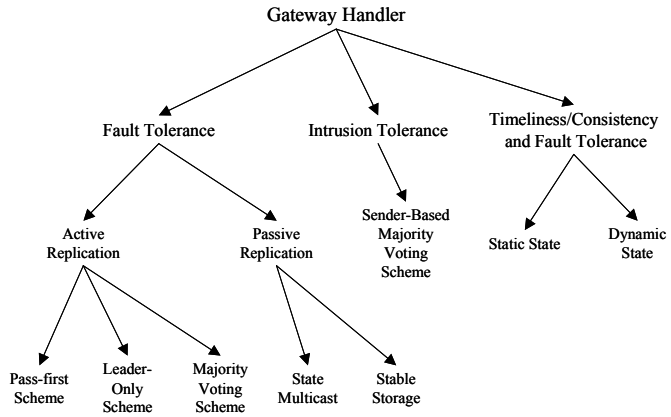

Figure 2. Overview of Handlers

We now detail the four handlers that provide active and passive replication.

3.1.1 Active Replication with Pass-First Scheme

In this scheme [Sab99], each replica in the source object group executes each invocation independently and sends each request/reply to the leader of the group. The leader is responsible for forwarding the first request/reply received to the destination object group.

Three steps are used to transmit a message from one replicated object A to another replicated object B. First, all objects in A make copies of the message and use reliable point-to-point communication to send the messages to the leader of A. The second step consists of the leader multicasting the first received request in the object group that contains A and B. After receiving the multicast, the members of A can safely delete their copies of the message, while the members of B save their copies of the message and wait for total ordering. Finally, in order to ensure that all replicas of the receiver object group see requests in the same order, the leader of B multicasts the message locally in group B. Upon receipt of that last cast, each gateway deletes its copy of the message received in step 2 and, if a reply is expected, delivers the message to the local application that is waiting for a response. Buffering is used to enable data consistency among replicas and recovery from a leader crash failure. The active replication with pass-first scheme tolerates only crash failures.

3.1.2 Active Replication with Leader-Only Scheme

In this scheme, the leader processes input messages and sends its output messages to the destination object group. The other members will process input messages and generate output messages that are suppressed unless the member becomes the new leader upon failure of the old leader.

Two steps are used to transmit a message from one replicated object A to another replicated object B. First, all objects in A send out the same message and make local copies of it, but only the leader multicasts its message to the object group that contains A and B. The objects in A delete their local copies of the message when they receive the cast notification. In the scheme's second step, which is necessary to ensure total ordering of the messages, the leader of B multicasts the received messages into group B. Upon receipt of that last multicast, each gateway deletes its copy of the message received in step 1 and, if a reply is expected, delivers the message to the local application that is waiting for a response. Buffering is used to enable data consistency among replicas and recovery from a leader crash failure. The active replication with leader-only scheme tolerates only crash failures.

3.1.3 Active Replication with Majority Voting Scheme

In this scheme [Ren01], each replica in the source object group executes each invocation independently and multicasts its request/reply in the group. The requests/replies from the members of the source object group are voted on; if and only if a majority of the requests/replies are bit-wise identical, the majority values are delivered to the members of the destination object group. The leader is responsible for forwarding the voting results to the destination group. Each replica executes majority-voting algorithms independently in order to be able to take over from the leader if it fails. All replicas maintain information regarding the outcomes of votes, so any replica can become the leader in the event that the leader crashes.

Three steps are used to transmit a message from one replicated object A to another replicated object B. First, all replicas in A send out the same message and multicast it to the source object group A. Upon receipt of this multicast, all replicas record the message in a buffer and check if a majority of values has been reached for the message. In step 2, once a majority has been reached for the message, only the leader will forward the majority value to the object that contains A and B. The third step, which is necessary for total ordering, consists of the leader of B re-multicasting the message to group B. As before, buffering is used to enable data consistency among replicas and recovery from a leader crash failure. The active replication with majority voting scheme tolerates both value faults and crash failures.

3.1.4 Passive Replication

In this scheme [Rub00], only the leader of the object group executes each invocation and forwards the request/reply to the destination object group.

Sending a request or reply is a three-step process for the leader. The first step consists of collecting the application state and either associating it with the message or saving it in stable storage. Once the state has been collected, the message, with or without the state attached, is multicast in the source object group, allowing non-leaders to buffer the leader's state. The second step starts with the leader multicasting the message without state to the object group that contains A and B. The third step, which is necessary for total ordering, consists of the leader of the receiving group B re-multicasting the message in its object group.

Because state is transferred after every message, all replicas always have the most recent state of their leader. If there is a leader crash failure, the newly elected leader does not need to replay a sequence of messages to get a state consistent with the old leader, and the application does not need to be deterministic.

The passive replication scheme tolerates only crash failures.

3.2 Handler that Provides Intrusion Tolerance

The handler that provides intrusion tolerance is called the *Sender-Based Majority Voting* handler (SBMV) because the voting takes place in the source and destination groups when the request/reply is being sent to the other group. The algorithm assumes the presence of public key digital signatures that are unforgivable and collision-resistant. In the first step, members of the source object group multicast the request and signature to all members of the group. The leader of the group then collects the messages, and, once a majority value for the request is reached, sends the request to the object group that includes the source and destination object groups. At that stage the leader also appends all the signatures that it received with the messages it used to determine a majority, to prove that the vote did occur. The receiving members then check the proof to ensure that the vote occurred correctly in the sending group. Then, to ensure total ordering, the leader of that group multicasts the request to all members of the destination object group. The SBMV scheme tolerates arbitrary faults.

3.3 Handlers that Provide Timeliness/Consistency Tradeoffs Combined with Fault Tolerance

Handlers that provide timeliness and fault tolerance target two kinds of applications: 1) stateless applications in which the state of the replicas is static [Kri01], and 2) applications in which the state of the replicas is dynamic and may be modified as a result of requests from the clients [Kri02]. In the case of dynamic state, concurrent operations have the potential to introduce replica inconsistency. Therefore, if there is a dynamic state, our gateway handlers not only select replicas to service the clients, but also maintain replica consistency using a combination of immediate and lazy update propagation.

The replica selection is driven by the QoS requested by the clients. We target clients that have specific temporal and consistency requirements and allow them to express their requirements in the form of QoS specifications. A client expresses its timeliness requirement by specifying its expected response time and the probability with which it expects its temporal constraint to be met. Failure to meet a client's deadline results in a timing failure for the client. For replicas with dynamic state, the QoS model includes the consistency

requirement in addition to the timeliness requirement. Our QoS model regards consistency as a two-dimensional attribute that consists of an ordering guarantee and a staleness threshold. The ordering guarantee is a service-specific attribute for all the service's clients. It applies to the order in which the servers will process their requests in order to prevent conflicts between operations. We have developed gateway handlers that support sequential and FIFO ordering. The staleness threshold, which is specified by the client, is a measure of the maximum degree of staleness a client is willing to tolerate in the response it receives.

Our selection approach attempts to prevent the occurrence of timing failures for a client by selecting replicas from the available replica pool based on an understanding of the client's QoS requirements and the responsiveness of the replicas. The client-side gateway handlers do the replica selection. Each client handler has three main modules for this purpose: a replica selector, an information repository, and a timing failure detector. The client-side handler transparently intercepts a request from the client, and hands over the request to its selector module. The selector retrieves the replica list for the service from its information repository, and chooses the replicas to service the request based on the client's QoS requirements and the performance history of the replicas stored in the information repository. The handler then multicasts the request to the selected set of replicas through the group communication layer. Although multiple replicas may respond to a request, a client handler delivers to its client only the earliest response. These handlers tolerate both crash and timing failures.

4. Conclusions

This paper presents an overview of a CORBA gateway for providing adaptable system properties. The different components of the gateway architecture are described. The various handlers that provide fault tolerance, intrusion tolerance, and timeliness/consistency tradeoffs are detailed, and the relationships among them are explained. All the handlers are developed independently from each other. New handlers that provide other levels of quality of service can be easily developed as dynamic libraries and interfaced to the gateway without recompiling it. All handlers are implemented as objects derived from a common base class, which is the only interface required by the gateway. The gateway is configurable in the sense that only the handlers required by an application are loaded at startup time. The handlers are loaded as services using the ACE service configuration file.

In addition, the framework provides an object called the *QoS requester* and a dependability manager that can be used by the client application to request and update quality of service at runtime [Ren02]. The QoS requester allows the adjustment of attributes such as the number of crash failures and value faults to tolerate or the majority size to achieve when a vote occurs. Some requests will lead to the adjustment of the number of replicas to provide the quality of service requested.

Acknowledgements

We thank the other members of the AQuA and ITUA projects, past and present, for their contributions that led to this gateway. We also thank Jenny Applequist for her editorial comments.

5. References

[Ami95] E. Amir, S. McCanne, and H. Zhang, "An Application Level Video Gateway," *Proc. of ACM Multimedia '95*, San Francisco, pp. 255-265, November 1995.

[Hay98] M. G. Hayden, *The Ensemble System*, Ph.D. thesis, Cornell University, 1998.

[Kal00] W. Kalter, B. Li, W. Jeon, K. Nahrstedt, and J.-H. Seo, "A Gateway-Assisted Approach Toward QoS Adaptations," *Proc. of the IEEE International Conference on Multimedia and Expo 2000 (ICME 2000)*, New York, pp. 855-858, July-August 2000.

[Kri01] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "A Dynamic Replica Selection Algorithm for Tolerating Timing Faults," *Proc. of the International Conference on Dependable Systems and Networks (DSN-2001)*, Göteborg, Sweden, July 1-4, 2001, pp. 107-116.

[Kri02] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An Adaptive Framework for Tunable Consistency and Timeliness Using Replication," Proc. of the 2002 International Conference on Dependable Systems and Networks (DSN-2002), Washington, DC, June 23-26, 2002.

[Loy98] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, "Specifying and Measuring Quality of Service in Distributed Object Systems," *Proc. of the First International Symposium on Object-oriented Real-time Distributed Computing (ISORC '98)*, Kyoto, Japan, pp. 43-52, April 1998.

[Nah99] K. Nahrstedt and D. Wichadakul, "QoS-Aware Active Gateway for Multimedia Communication," *Proc. of 6th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'99)*, Toulouse, France, pp. 31-44, October 1999.

[Ram02] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," *Proc. of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 23-26, 2002.

[Ren01] Y. Ren, M. Cukier, and W. H. Sanders, "An Adaptive Algorithm for Tolerating Value Faults and Crash Failures," *IEEE Transactions on Parallel and Distributed* Systems, vol. 12, no. 2, February 2001, pp. 173-192.

[Ren02] Y. Ren, T. Courtney, M. Cukier, C. Sabnis, W. H. Sanders, M. Seri, D. A. Karr, P. Rubel, R. E. Schantz, and D. E. Bakken, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *IEEE Transactions on Computers*, to appear.

[Rub00] P. G. Rubel, "Passive Replication in the AQuA System," Master's Thesis, University of Illinois, 2000.

[Sab99] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA," *C. B. Weinstock and J. Rushby (Eds.), Dependable*

Computing for Critical Applications 7, vol. 12 in series Dependable Computing and Fault-Tolerant Systems (A. Avizienis, H. Kopetz, and J. C. Laprie, Eds.), pp. 149-168. Los Alamitos, CA: IEEE Computer Society, 1999.

[Sch94] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," *Proc. of the 6th USENIX C++ Technical Conference,* Cambridge, Massachusetts, April 1994.

[Sch98] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp 294-324, April 1998.

[Sch01] R. Schantz, J. Zinky, J. Megquier, J. Loyall, D. Karr, and D. Bakken, "An Object-level Gateway Supporting Quality of Service," *Computer Systems Science and Engineering,* vol. 16, no. 2, pp. 97-107, March 2001.

[Vay98] A. Vaysburd and K. P. Birman, "The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 73-80, 1998.

[Zin97] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55-73, April 1997.