QUORUM AND OASIS PROGRAMS:

INTRUSION TOLERANCE BY UNPREDICTABLE ADAPTATION (ITUA)
ADAPTIVE QUALITY OF SERVICE FOR AVAILABILITY (AQUA)

ITUA-AQuA Gateway

User's Guide

Dependability and Intrusion-Tolerant Middleware

For CORBA and Socket-based Applications

Gateway Release 4.4

December 2003

CONTENTS

1 Introduction

The goal of this guide is to help the reader install the ITUA-AQuA gateway (**I**ntrusion **T**olerance by **U**npredictable **A**daptation – **A**daptive **Qu**ality of Service for **A**vailability), run it, and develop applications using the gateway middleware to provide dependability to CORBA and socket-based applications. Although the purpose of this guide is to facilitate the practical use of the gateway architecture, a theoretical background is also provided.

The goal of the ITUA project is to develop an architecture for building dependable and intrusion-tolerant distributed systems. The gateway architecture allows distributed applications to request and obtain a desired level of dependability and includes a dependability manager that attempts to meet the requested availability levels by configuring the system in response to outside requests and changes in system resources due to faults.

The gateway was born in 1996 as a component of the project named AQuA to provide a framework to develop dependable CORBA applications. In 2001, the ITUA project expanded the scope of the gateway by introducing the notion of intrusion tolerance. For that purpose, the gateway has been redesigned to allow the use of different group communication systems (in particular, intrusion-tolerant systems). New communication (replication and non-replication) schemes have been introduced, and the state transfer algorithm has been modified to be intrusion-tolerant. In the context of the SAMS project, in early 2003, an extension was made to the gateway to support not only CORBA applications but also socket-based applications. Finally, an interface acting as a bridge between socket applications and the CORBA gateway has been added, and an application was developed to use it.

This document contains a description of the gateway with all of the components implemented for it during the course of these 3 projects: AQuA, ITUA, and SAMS.

Section 2 provides a summary of the basic concepts needed to understand the gateway architecture. More details can be found in the references given at the end of the section.

Section 3 describes the gateway. It presents its architecture and its different components. The configuration file that is used to initialize the gateway is also described. Finally, the way to start the gateway is explained.

Section 4 explains what a user needs to change in a standard CORBA application in order to run it in the gateway environment. These changes are minor and allow a programmer to make his/her application dependable with a few hours of work, if the application meets the requirements listed in this section. This section also presents three examples of ITUA-AQuA applications: the pinger application, the Deet server, and the castle demonstration.

Section 5 presents an overview of the different graphical user interfaces delivered with this release.

Section 6 gives the requirements that a system that uses the gateway must meet, explains how to install the gateway on a system, and details the gateway utilities.

Section 7 indicates where to find more information on the ITUA and AQuA projects.

Section 8, Appendix A, provides a description of all calls that may be made between a QoSRequestor and a dependability manager.

Section 9, Appendix B, provides a description of all calls that may be made between a host observer/controller or advisor observer and a dependability manager.

2 Concepts Used in the Gateway Architecture

The goal of the gateway architecture is to provide adaptive fault and intrusion tolerance to distributed applications by using commercial off-the-shelf hardware and operating systems. The gateway architecture allows application programmers to request a desired level of dependability during applications' runtimes. Moreover, the architecture provides adaptive fault tolerance. In distributed systems, resources change dynamically, and different types of faults can occur anywhere and anytime. Therefore, it is important to provide a mechanism for adaptive fault tolerance.

The gateway is designed to provide dependability for CORBA and socket-based applications. It provides fault-tolerance mechanisms to ensure that a client can obtain reliable services, even if the server object that provides the desired services suffers from crash failures and value faults.

2.1 Gateway Components

Figure 1 shows the different components of the gateway architecture in one particular configuration. These components can be assigned to hosts in many different ways, depending on an application's desired level of dependability and intrusion tolerance.

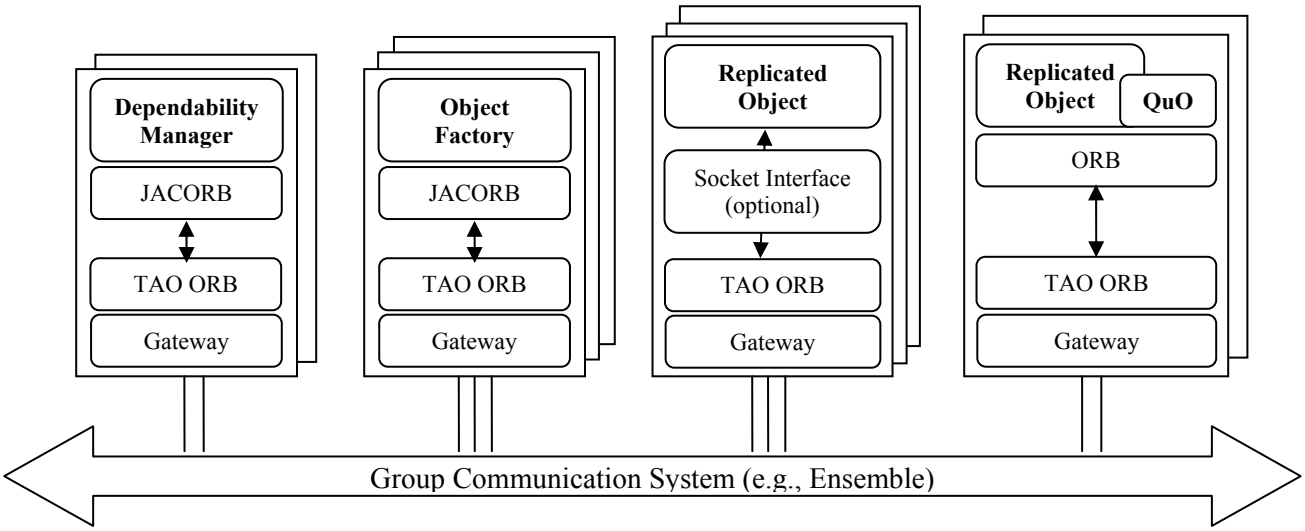

Figure 1. Gateway Architecture

In ITUA and AQuA, fault tolerance is achieved through the replication of objects. Strong data consistency among object replicas is provided. The data consistency is maintained by communication groups. All replicas of an object form a group. Messages communicated among different objects are sent through groups. A group communication system (Ensemble [Hay98, Vay98] or another), as shown at the bottom of Figure 1, is used to provide reliable message multicast and totally ordered message delivery among object replicas. In addition, the group communication system provides group membership protocols to detect process crash failures and to maintain replica consistency by transferring the state for each new group member ([Ram02]).

In order to provide a way for an application to specify its dependability requirements, a Quality Object (QuO) [Loy98, Zin97] can be used. It allows distributed applications to process and invoke dependability requests, and to receive information regarding the level of dependability that is being provided by the current system. QuO may run either in the same process as an application, or as a separate process.

In ITUA and AQuA, Proteus is one way to manage adaptive fault tolerance. It consists of a possibly replicated dependability manager, a set of object factories, and gateway handlers. The dependability manager determines a system configuration based on reports of faults and desires of application objects. An object factory that resides on each host is used to create and kill objects, as well as to provide load and other information about the host to the dependability manager. A companion intrusion-tolerant manager was developed by BBN as part of the ITUA project; it is also compatible with, and depends on, the ITUA-AQuA gateway.

Communication among all architecture components (i.e., applications, the QuO runtime, object factories, and dependability managers) is done using gateways, which translate CORBA object invocations into messages that are transmitted via Ensemble or another group communication system. Furthermore, the handlers in a gateway implement multiple replication schemes and communication mechanisms. The handlers are also used to detect application value faults, and to report value faults and group membership changes to the dependability managers. Before discussing the supported replication schemes and the gateway architecture, we review Ensemble, QuO, Proteus, and the ITUA-AQuA group structure.

2.2 Group Communication System

In order to provide adaptive fault tolerance in distributed systems, the gateway architecture requires a group communication system that provides the following properties: reliable multicast/unicast communication, totally ordered multicast, a group membership service, and virtual synchrony. Each of these concepts is described briefly in the following. The group communication system provided in this release supports crash fault tolerance, but an intrusion-tolerant group communication system (e.g., [Ram02]) could be used.

Reliable Multicast/Unicast Communication: Reliable unicast communication ensures that point-to-point messages are communicated between processes via a FIFO channel. Reliable multicast communication allows the same message to be sent to multiple processes by a single communication operation. All correct processes deliver the same set of messages. Reliable multicast is a basic element of replicated system services and is important for distributed application development.

Totally (or Atomically) Ordered Multicast: Reliable multicast imposes no restriction on the order in which messages are delivered to the members of a group, but for many applications, a consistent ordering of events is crucial. Totally ordered multicast ensures that remote method invocations are delivered in the same order to all group members. More specifically, if two correct objects A and B both deliver msg1 and msg2, A delivers msg1 before msg2 if and only if B delivers msg1 before msg2. This holds even when msg1 and msg2 are issued by different senders. Totally ordered multicast is a useful property for maintaining data consistency for all replicas.

Group Membership Service: A group is formed by a set of processes that are recipients of a multicast. The group membership service maintains consistent information at all sites about the membership of a group of processes. In order to do so, the group membership service automatically notifies the group members of group-related events, such as processes joining or leaving the group. The group membership service is also responsible for providing a process crash failure monitoring service to collect information for the operational processes. This service guarantees that processes have consistent and accurate knowledge of which other processes are operational and which are not.

Virtual Synchrony: Virtual synchrony restricts the delivery order of application and membership change messages in such a way that the group members all see the same events in the same order even though the events are actually occurring on different sites at different times [Bir96]. In virtual synchrony, all significant events, i.e., the delivery of messages, multicast view changes, and failures, appear as if each event had occurred at the same logical time in all processes. This property provides an agreed cut in the message flow, despite failures and dynamic reconfiguration of the system.

The Ensemble group communication system [Hay98], which was developed at Cornell University, is used in the gateway system to provide the above properties. Ensemble ensures reliable communication between processes in a group, ensures totally ordered delivery of multicasts in a group, provides a virtual synchrony, and detects and excludes from the group members that fail by crashing. In particular, Ensemble assumes that the process failures are crash failures, and detects process crashes through the use of "I am alive" messages. The gateway architecture uses this detection mechanism to detect crash failures and provides input to Proteus (see section 2.4) to aid in recovery. (Value faults are not detected by Ensemble, and hence must be detected at a higher level in the gateway architecture.) Moreover, Ensemble provides a leader election protocol to maintain a leader for each group. The Ensemble group leader also has a special role in the replication protocols we developed. In particular, it is responsible for forwarding replies to the sender objects, maintaining totally ordered messages among group members when they receive messages from multiple groups, and reporting group membership changes and value faults to the dependability manager(s).

Callbacks from the group underneath are transmitted to the gateway via the HOT interface sitting on top of the ML and C version of Ensemble. Upon receiving these callbacks, the replication protocols define the replication-specific reactions to the callbacks.

Note: even though the gateway can support multiple group communication systems, in this document, we mainly refer to Ensemble, which is used in the current gateway release.

2.3 Quality Objects

In order to provide a simple way for application objects to specify the levels of dependability they desire, the ITUA-AQuA architecture provides an interface for applications to specify their desires either directly or by using *Quality Objects* (QuO) [Loy98, Zin97], which allow distributed applications to specify QoS requirements at the application level using the notion of a contract. A contract specifies actions to be taken based on the state of the distributed system and desired application requirements.

The goal of QuO is to develop a common middleware framework, based on distributed object computing, that can manage and integrate nonfunctional system properties such as network resource constraints, dependability requirements, real time, and security needs. In the gateway approach, QuO is used to transmit an application's dependability requirements to Proteus, which attempts to configure the system to achieve the desired dependability. QuO also provides an adaptation mechanism that can be used when Proteus is unable to provide a specified level of dependability.

In particular, contracts in QuO can have multiple dependability requirements specified a priori to allow for multiple fallback positions, if Proteus fails to provide a requested level of dependability. To do this, QuO uses two types of regions to summarize conditions of interest. *Negotiated* regions specify the expected behavior of the local and remote objects, and are defined by predicates on the state system condition objects, which provide a view of the state of the distributed system. *Reality* regions are defined within each negotiated region and specify measured or observed conditions of interest in the distributed system. They are specified by predicates on a portion of the state of the distributed system itself, as viewed through *system condition objects*.

2.4 Proteus

Proteus [Sab99] is a flexible infrastructure for providing adaptive fault tolerance. The organization of Proteus is shown in Figure 2. Proteus makes remote objects dependable by using (1) a replicated dependability manager to make decisions regarding reconfigurations and to coordinate changes in system configurations, (2) object factories to kill and start objects and provide information to the dependability managers regarding a host, and (3) gateways that implement particular voting and replication schemes. An interface to the QuO runtime is provided to allow the application to specify the desired level of dependability.

Proteus dependability managers make decisions regarding reconfiguration based on reported faults and dependability requests from the application or QuO, and, together with the gateways, implement the chosen fault tolerance approach. The decisions include choosing the types of faults to tolerate (crash failures or value faults), the styles of replication to use (active replication or passive replication), the types of voting to use (majority voting, pass first, or leader only), the degrees of replication to use, and the location of the replicas.

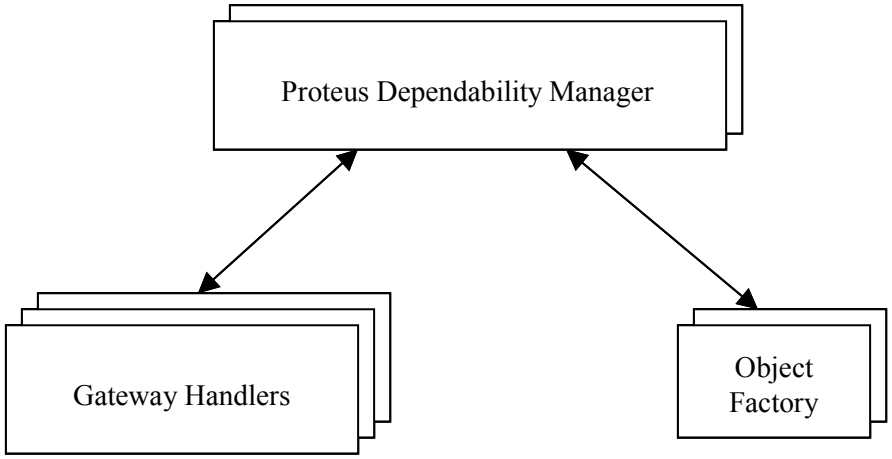

Figure 2. Proteus Architecture

Object factories are used to kill and start replicated objects (depending on decisions made by the dependability managers), and to provide information regarding the host to the dependability managers.

Gateways are used to translate between object-IIOP or socket-based and process-level (Ensemble) communication, to provide an infrastructure for implementing various replication and communication schemes, and to detect and report faults to the dependability managers.

2.5 Groups in ITUA-AQuA

In the ITUA-AQuA architecture, the basic unit of replication is the *Gateway Object*. A Gateway Object is either a two-process pair (including an application and a gateway) or a three-process pair (consisting of an application, QuO, and a gateway). QuO is included if an object contained in the application process makes a remote invocation of another object and wishes to use QuO to specify a quality of service for that object, instead of making the specification itself. Parts of the ITUA-AQuA infrastructure, including dependability managers and object factories, can be thought of as applications, and are thus also gateway objects. When we say that "an object joins a group" we mean that the gateway process of the object joins the group.

Using this terminology, we can now describe the group structure and mechanism used in the ITUA-AQuA architecture, including replication groups and connection groups. By defining multiple replication and connection groups, we can avoid the communication overhead that would occur if a single large group were used.

A *replication group* is composed of one or more replicas of a gateway object. These objects may be transient or persistent members of the group. *Persistent members* join the group when they are created, and remain in the group. *Transient members* join a replication group only when they need to multicast a message to the replicas in the group. After sending a message, these objects leave the group. A replication group has one persistent object

that is designated as its leader and may perform special functions. Each persistent object in the group has the capacity to become the object group leader, and a protocol is provided to ensure that a new leader is elected when the current leader fails. For implementation simplicity, the object whose gateway process is the Ensemble group leader is designated the leader of the replication group. This allows Proteus to use the Ensemble leader election service to elect a new leader if the object leader fails.

A *connection group* is a group consisting of the persistent members of two replication groups that wish to communicate. It provides reliable message communication from one CORBA object to a different CORBA object.

As specified above, each replicated object is located inside a replication group. The gateway provides two methods of reliable communication between objects in two different replication groups. One way is to use a connection group, which is done if the sending object is a persistent member of its replication group, and hence is in a connection group shared by the destination replicated object. In that case, a replicated object that is inside a replication group multicasts messages within a connection group to forward them to the other replicated object. Using that approach, two different objects are able to communicate using both one-way and synchronous remote method invocations. This approach requires that there be a pre-established connection group before objects send messages to each other, that the replication group sending the message communicate according to the communication scheme specified by its replication group, and that the receiving replication group communicate according to its compatible communication scheme.

In the second method of reliable communication, the sending object becomes a transient member of a replication group with which it wishes to communicate. Invocations made by transient members can only be one-way. In addition, only the leader of a sender replication group is allowed to become a transient member of another replication group, and the leader is responsible for making invocations on behalf of the sender replication group.

Communication with a transient group member includes three steps. In the first step, the leader of the sender replication group joins the receiver replication group. Each of the other nonleader replicas in the sender replication group keeps in a buffer a copy of the invocations that the leader will send to the receiver replication group, to be used if the leader fails. In the second step, when the leader joins the receiver replication group, it first multicasts the invocations to the receiver replication group. Then it multicasts notifications to the other nonleader replicas in the sender replication group to let them know that the invocations have been made. In the third step, the nonleader replicas in the sender replication group will remove the invocations from their buffers. If the leader fails before it multicasts the notifications to the other replicas, the new leader must resend those messages to the other objects by becoming a transient member, because the other nonleader replicas of the sender replication group have no knowledge of whether the messages sent by their leader were delivered successfully. The resending could cause the replicas in the receiver replication group to receive duplicate messages. Therefore, this approach is only suitable for situations in which duplicate messages are allowable.

Communication through a transient group member is useful in situations in which communication is fairly infrequent and duplicate messages are allowable. In such cases, the overhead in joining and leaving a replication group is small relative to that of maintaining a connection group between two replication groups.

For an illustration of the possible use of replication groups and connection groups, consider Figure 3. Solid lines define the replication and connection groups. Dashed lines represent the occurrence of a transient member joining a replication group. We see in Figure 3 that even though a connection group is composed of two replication groups, a member of a replication group can be included in several connection groups. For example, the replicas in replication group 3 communicate with the replicas in replication group 1 through connection group 1, and they communicate with the replicas in replication group 2 through connection group 2. The leader of replication group 2 becomes a transient group member of replication group 1 in order to send messages to the replicas in replication group 2.

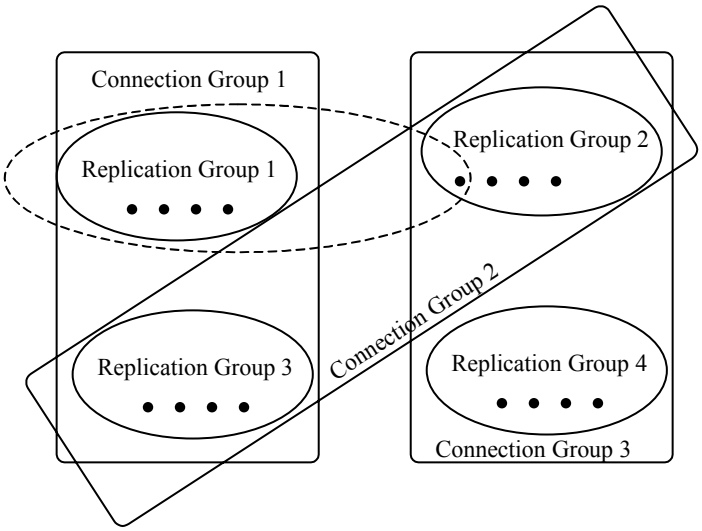

Figure 3. Example Group Structure in ITUA-AQuA

2.6 Fault Tolerance through Replication in ITUA-AQuA

In the gateway, four types of replication techniques have been developed: active replication with pass-first, active replication with Sender-Based Majority Voting (SBMV), active replication with leader-only, and passive replication. On one hand, the different replication techniques enable the tolerance of different types of faults or intrusions, provide different recovery strategies, use different communication schemes, and support different types of object replicas (deterministic or nondeterministic). On the other hand, all of the replication techniques must provide the ability to (1) ensure reliable transmission of each CORBA message from one replicated object to another, so that messages will not be lost even if replicas crash, (2) guarantee strong data consistency among all object replicas, (3) guarantee that no duplicate messages are delivered to the replicated objects, and (4) react correctly when the number of replicas in a replication group changes. In addition, all of the replication techniques require the use of the group communication system to provide reliable multicast, totally ordered message delivery, group membership services, and virtual synchrony. The types of replication techniques can be divided into two categories: active replication and passive replication.

2.6.1 Active Replication

In active replication, all members of the object group independently execute the methods invoked on the object, so that if a fault prevents one member from operating correctly, the other members will produce the required messages. This technique requires that the CORBA ORB and the object replicas be deterministic to ensure that incoming messages are dispatched from the ORB to the replicas in the same order, and that the replicas process the messages in the same order.

Three types of active replication schemes are implemented in the gateway:

Active Replication with Pass-First Scheme: In this scheme, each replica in the replication group executes each invocation independently and sends each request/reply to the leader of the group. The leader is responsible for forwarding the first received request/reply to the destination object group. This scheme can be used to tolerate process crash failures.

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 takes over for the leader (if the leader fails). Through use of this scheme, process crash failures can be tolerated.

Active Replication with Sender-Based Majority Voting Scheme: In this scheme, each replica in the replication group executes each invocation independently and multicasts its request/reply in the replication group.

Each request is digitally signed using a set of keys shared/known by all replicas. The requests/replies from the members of the source object group are voted on and the signatures verified by all members; if and only if a majority of the requests/replies are identical, the majority values are delivered to the members of the destination object group. The list of signatures used by the senders voting for this majority value is attached as a proof used by the receiving group to verify the message being received. The leader is responsible for forwarding the voting results to the destination group. Each replica executes majority-voting algorithms independently in order to take over from the leader if it fails. The SBMV scheme [Lyo03] is inspired by the Majority-Voting scheme described in [Ren01a] with added support for intrusion tolerance using cryptography.

In all three active replication schemes, in the event that one replica fails, the application can continue with results from another replica without waiting for fault detection and recovery.

2.6.2 Passive Replication

In passive replication [Rub00], during fault-free operation, only one member of the object group, the leader, executes the method invoked on the group. The gateways of the other replicas store the sequence of method invocations in a buffer but do not deliver the messages to their applications, and thus do not process the request. Periodically, the state of the leader is transferred to the other members (the backup replicas) or to stable storage. In the presence of a leader crash, a backup member becomes the new leader of the group and updates its state. The state of the new leader is made identical to the state of the old leader through application of the request messages recorded in the new leader's gateway buffer. Passive replication can be used to tolerate process crash failures.

There are two types of passive replication schemes, the *passive replication with state cast scheme* and the *passive replication with stable storage scheme*. In the first scheme, the leader multicasts its state to the backup replicas. This scheme provides a way for the backup replicas to access the state immediately during fault recovery. However, it requires that more messages be transmitted over the network. In the second scheme, the leader stores its state in stable storage. When the original leader fails, the new leader will take over from the original leader by getting the state from stable storage. Compared to the state cast scheme, this scheme reduces the number of messages transmitted over the network. However, the recovery time could be longer than for the state cast scheme, because the new leader needs time to get the state from stable storage. The extra recovery time includes both the time required for disk access and the time spent on the network.

In both of the passive replication schemes, two strategies, *every-message* and *periodic*, can be used to capture the state of the leader. *Every-message* state transfer happens whenever the leader sends a message to the outside world. For that reason, output messages will not need to be resent if the leader already sent them out before it failed. Thus, with *every-message* state transfer, replicas are not required to be deterministic. *Periodic* state transfer occurs when the leader sends out a certain number of messages. In that scenario, if the leader fails, the backup replica that takes over for it might send out duplicate messages that must be suppressed. Thus, with periodic state transfer, replicas are required to be deterministic, so that the new leader produces messages that are the same as those produced by the original leader before its failure.

3 ITUA-AQuA Gateway

3.1 Architecture

The ITUA-AQuA gateway (shown in Figure 4) is composed of:

- Handlers and the handler factories that create them,
- Replication schemes such as pass-first/leader-only/majority-voting/SBMV/state-cast/stable-storage and plain communication schemes Multicast/Dynamic-send,
- Communication strategy factories used by the handler factories to create the handlers (all strategies except the plain majority-voting and stable storage are supported in this release),
- A DII request processor to deliver requests to the application and wait for responses (if necessary),
- A simple Naming Service binding names to CORBA objects,
- A Starter Factory used to start processes (i.e. the application object) from the gateway configuration/startup file,
- A Parameter Service accessible from outside the gateway to set gateway parameters at runtime,
- A Group Factory to create group members,
- As many group members as needed for the application. Each group member is composed of:
    - A state processor handling state-transfer-related data. In the past, we used the state transfer protocol provided by the Maestro toolkit of Ensemble. We have developed our own state transfer protocol, adding a removal-of-trust capability which consists of all existing replicas voting on the state about to be installed in a new member.
    - A sending processor (processing all messages to be sent to the communication group).
    - A receiving processor (dispatching all messages received from the group to either the handlers or the state processor).
- A GCS (Group Communication System) adaptor, providing the interface to the desired GCS. In this release, 2 adaptors are provided: one for ML Ensemble, the other for the C-version of Ensemble. More adaptors can be developed as desired to support other GCSs.

The gateway is started by a simple TAO process, **aquagw**, whose sole purpose is to open a service configurator file (by default called svc.conf in the current directory) and load all the services listed in the file.

All the factories and the components listed above are implemented as dynamic link libraries loaded as CORBA services through the ACE service configurator.

3.2 Gateway components

The different components of the gateway are shown in Figure 4.

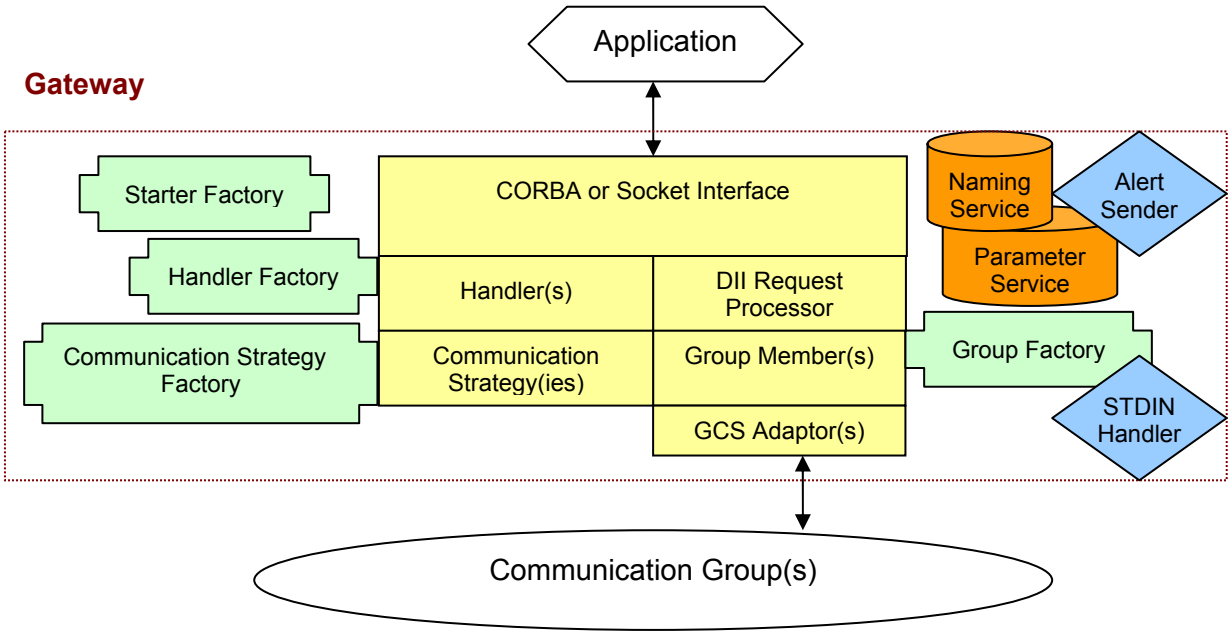

Figure 4. Gateway Components

3.2.1 GCS Adaptor

Group Communication Systems (GCSs) are used for inter-gateway communications. To support a larger number of GCSes, we introduce the notion of a GCS Adaptor, one of which is plugged underneath the gateway for each type of GCS. The goal of the adaptor is to provide an interface transparent to the gateway component for sending and receiving messages from the communication groups.

All adaptors derive from a base class and implement the same set of methods used by the gateway to access the group:

- Methods to join, leave, and block the group.
- Methods to cast or send point-to-point messages to other members of the group.
- Methods to receive heartbeats and callbacks whenever a message is received (cast or point-to-point) from another member or when the group membership view changes or a state transfer is initiated.
- Methods to suspect other members.

Some common data structures, such as the way to identify the members or represent the view, follow the same syntax for all adaptors when shared with the gateway components.

Two adaptors are provided in the release: one for ML Ensemble and another for C-Ensemble and its intrusion-tolerant addition. Factories for both kinds of adaptors are loaded automatically as services when the gateway starts up. Then the base GCS Adaptor Factory is loaded, passing parameters to create two types of adaptors through the above factories. The adaptors are named *MLEnsemble* and *CEnsemble*. Later, each handler specified in the configuration file indicates the name of the adaptor it wishes to use (MLEnsemble or CEnsemble) for its group members.

For both MLEnsemble and CEnsemble adaptors, the properties (protocol layers) used and set by default when loading the Ensemble adaptor factories are "Gmp:Sync:Heal:Switch:Frag:Suspect:Slander:Flow:Total," and the environment variables *ENS_PARAMS*, *ENS_HEARTBEAT,* and *ENS_MODES* are read to overwrite the defaults:

- "suspect_max_idle=10:int;suspect_sweep=1.000:time," for parameters
- "10" seconds, for heartbeat
- "DEERING", for multicast communication mode. To use gossip, set the *ENS_MODES* variable to UDP.

All group members created by the handlers maintain a pointer to the adaptor (a singleton of its type in each gateway) they use.

3.2.2 Group Member and Group Factory

Group Members are the components representing the group membership in a given communication group. They handle and dispatch the messages coming up and down between the gateway and the communication group. Messages sent down to the group are sent to the GCS Adaptor for marshalling via an entity called the *Sending Processor*. Messages coming up are queued up and processed by the Receiving Processor and either sent to the State Processor for state-related messages or delivered to the handlers registered to this group member.

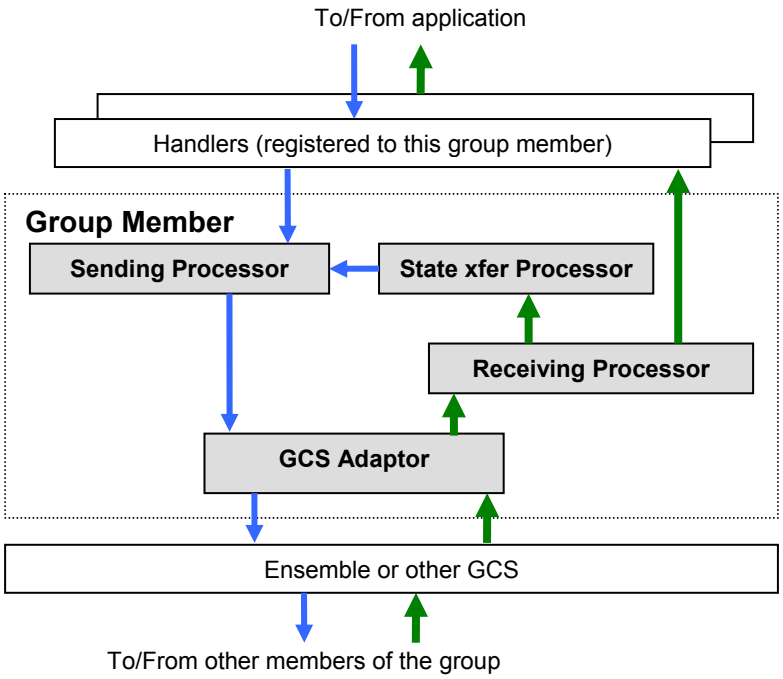

Figure 5. Group Member Components

Note: more than one handler can share a pointer to the same group member. This is the case for active or passive handlers that share the same replication group (that is, they have a singleton group member in the gateway). Whenever a message comes in through this shared group member, the message is forwarded to all handlers registered with the member. It is then up to the communication strategies to look at the message header to determine whether they are concerned about the data.

The state processor implements the algorithm described in [Gup03], which is a secure state transfer. When used with an intrusion-tolerant GCS, this state transfer completes the chain of intrusion tolerance, from top to bottom, GCS to application. In the earlier version of the gateway, tied to Maestro/Ensemble, the state transfer protocol was the Maestro state transfer (see [Vay97] and Vay98]). In that protocol, the leader's state was the state collected and installed in all new members. If the leader happened to be corrupted, all new members could get corrupted via

the state transfer. In our protocol, all members collect and cast their state when a new member joins, and a vote takes place to ensure that the state to be installed in the new member is the choice of a majority.

A group factory creating group members is loaded automatically as a service when the gateway starts up. Every handler gets an instance of this factory whenever it needs to create a group member or access a singleton group.

3.2.3 Handlers, Communication Strategies, and Factories

There are two kinds of handlers: replication handlers used to replicate objects and non-replication handlers used by objects to communicate via the group communication system with remote objects running on top of gateways.

All handlers implement specific communication strategies. 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 GCS, decides where to cast/send the message in either the replication group or connection group, and decides whether or not the message should be buffered or removed from the buffers. The following table summarizes some of the features of the strategies provided in the release.

Handler Classification	Handler Type	Communication Strategy	Tolerance	Principles
Replication Handler	Active Handler	Pass-First	Crash	All members process the application messages and send them point-to-point to their leader. The group leader sends to the connection group the first message received in the replication group. Messages are buffered for crash recovery and re-cast for total ordering.
Replication Handler	Active Handler	Leader-Only	Crash	All members process the application messages and cast them to the replication group for buffering. The leader sends out to the connection group its own messages and does not wait for the others. Messages are buffered for crash recovery and re-cast for total ordering.
Replication Handler	Active Handler	SBMV	Crash, Fault, Intrusion	All members process the application messages, using keys to sign their messages. All check the sender signatures and compare the messages sent by all. Only the messages reaching a majority are sent out by the leader to the connection group with enough proof for the recipient group to validate the messages. Messages are buffered for crash recovery and re-cast for total ordering. Corrupted messages (not reaching a majority) are ignored. All public keys or certificates must be shared among servers but also among clients so the clients can decrypt and authenticate the messages sent by the servers.
Replication Handler	Passive	State-Cast	Crash	Only the leader of the replication group processes the application messages. Messages are buffered for crash recovery and re-cast for total ordering.
Communication Handler	N/A	Multicast	N/A	Used by a permanent member of a group of objects (not necessarily replicated or identical objects) to cast and receive messages to and from the group. Only one-way messages can be cast.

Handler Classification	Handler Type	Communication Strategy	Tolerance	Principles
Communication Handler	N/A	Dynamic-Send	N/A	Used by a transient member to join a group, cast a message to the group, and leave immediately after. Some options are available to specify conditions before a cast can take place. For example, before casting any message, the member can check that a minimum number of members are seen in the group (and therefore, ready to receive). If there are not enough members, the sender will buffer the message to be cast and remain in the group long enough to see another member join and be able to cast.

Table 1. Handler and Communication Strategy Classification

Whenever a handler factory creates a new handler, it needs to create and associate with this handler the appropriate communication strategy. To provide a modular design that is capable of supporting the addition of new strategies in the future, we are using a factory to create communication strategies.

Each handler created is registered in the naming service under the handler's name, which is provided as a command line argument in the service configurator file. If the application uses state, the handler factory waits until the application is registered in the naming service before it creates the handler. This is to ensure that the application is ready, since during the handler's creation the replication group can initiate a state transfer, for which the application must be ready to provide its state.

The communication strategy factory must be loaded before the handler factory that will create handlers with this communication strategy.

Non-replication handlers like the multicast and dynamic send handlers don't need a communication strategy factory, only a handler factory.

There is one handler for each remote object with which the local application is communicating. The handler, implemented as a CORBA DSI (dynamic skeleton interface) servant, acts transparently underneath the application and impersonates the remote CORBA object with which the application is communicating. Therefore, the CORBA application has no knowledge of the group communication system and only cares about:

- Registering its IOR into the gateway naming service so the handler can locate the application when it needs to deliver messages to it,
- Retrieving the reference to the handler when the application (then acting as a client) needs to send requests to a remote object, and
- Implementing state transfer methods (getState()/setState) and group information callbacks (isLeader() and groupViewChange()). These methods can be empty if the application does not carry state or is not interested in group information, but they do need to be defined.

According to the selected communication strategy, each handler is a member of one or more communication groups. Active and passive strategies require two group members (one for the replication group and another for the connection group).

For these strategies, a replication group member is created only once and is shared among handlers of the same application.

Other non-replication handlers require only one group member .These handlers are often used to allow an isolated object to communicate with a group of objects either as a transient or permanent member. This is the case with the dynamic send and multicast handlers.

CORBA messages received from the application or socket interface are encapsulated without being unmarshalled into a data structure called *Gateway_Message*. When this message reaches the GCS Adaptor, it is translated into a *GCS_Message* and into the GCS data structure before being sent into the communication group. On the receiving side, the GCS adaptor takes the reverse steps: it extracts the *GCS_Message* from the data, and translates the message into a *Gateway_Message* that will be delivered to the handlers.

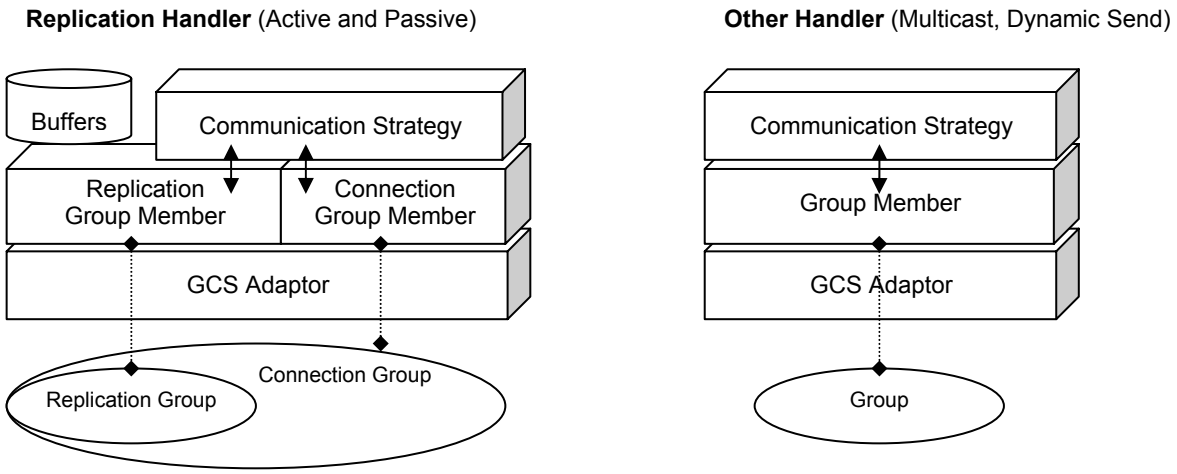

Figure 6. Handler Architecture

3.2.4 The DII Request Processor

Each handler contains a reference to the **DII Request Processor**, which is a service object used for delivering messages to the application. A single thread is available to handle all the messages arriving from the GCS in each group member.

Whenever a data message (not state-related) is received from the group member, it is sent for processing to the communication strategy. If the message is a request/reply to be delivered to the application, the strategy places the request in the DII processor message queue and informs the DII processor worker thread that there is work to do.

The DII processor thread then does the following:

1) dequeues the message at the head of the queue,

2) constructs a DII (dynamic interface invocation) request based on this message,

3) delivers this DII request to the CORBA application,

4) if a reply is expected, waits for the response and sends it back to the handler's strategy,

5) checks the queue for other messages to deliver,

6) if the queue is not empty, executes the above operations starting from step 1; otherwise, waits to be signaled by a handler when any new messages are inserted.

The components shared between handlers, such as the DII request processor and the replication group member, are separate service objects loaded through the service configurator file. Therefore, we can ensure that only one of these components is created in each gateway.

3.2.5 CORBA or Socket Interface

CORBA applications use the natural CORBA interface to interact with their gateways. Written in either C++ or java, they create an ORB and POA and use the CORBA methods to retrieve, narrow down, and invoke methods on the various gateway objects (the handler, the gateway itself, the naming service, and so forth). A minimal set of methods needs to be implemented by any CORBA application using the gateway to receive the appropriate callbacks from the gateway. The gateway uses the ACE ORB (TAO). Java applications have been successfully tested with the gateway using both VISIBROKER and JACORB.

For applications with no knowledge of CORBA, a socket interface is also available, and the way to connect to the CORBA gateway via this socket interface is described in this section. The rest of this document focuses on the pure CORBA interface with the gateway.

The socket interface is actually a program that is a mix of CORBA and socket. This program, called Sock-Proxy.cpp, is used for both sending and receiving. It has an ORB and POA and is started via a configuration file along with a gateway and a handler to communicate with a remote SockProxy application. The receiving Sock-Proxy uses an instance of a CORBA object (called SocketApp_i) that implements the interface below and for-wards to the user application sockets all data received via sendBytes().

```
// Interface forwarding bytes through sockets passed as arguments.

#include "aqua/Application.idl"
#include "aqua/OctetSeq.idl"
interface SocketApp : aqua::Application {
 // Send a message
 oneway void sendBytes( in long portNumber,
                        in long udpFlag,
                        in aqua::OctetSeq data);};
```

When used as a client, the SockProxy listens on UDP and TCP sockets and translates all data received on these sockets into an invocation of a sendBytes() method on a handler. This method has 3 arguments: the port number on which the data are expected by the receiving socket application, the type of socket to use (UDP or not), and the data to transmit as a sequence of octets. The handler, following the desired communication strategy, transmits the data through the group communication and the data are delivered via the receiving handler to another SockProxy application (set as a server).

The server SockProxy receives the sendBytes() invocation, analyzes the port number and type indicated, and writes the provided data to this port.

The SockProxy application takes the following options:

-e <remote host:remote port> to send UDP to this port
-v <local port> to receive UDP on this port
-q <remote host:remote port> to send TCP to this port
-r <local port> to receive TCP on this port

For use on top of gateways, add the following:

-a <application name>	name of the application (as defined for the handler in the configuration file)
-h <handler name>	name of the proxy handler
-o <ior output file - optional>	to write the socketApp_i IOR into a file
-f <port offset>	debugging option to offset by one the receiving and sending port, allowing the running of client and server on the same host.

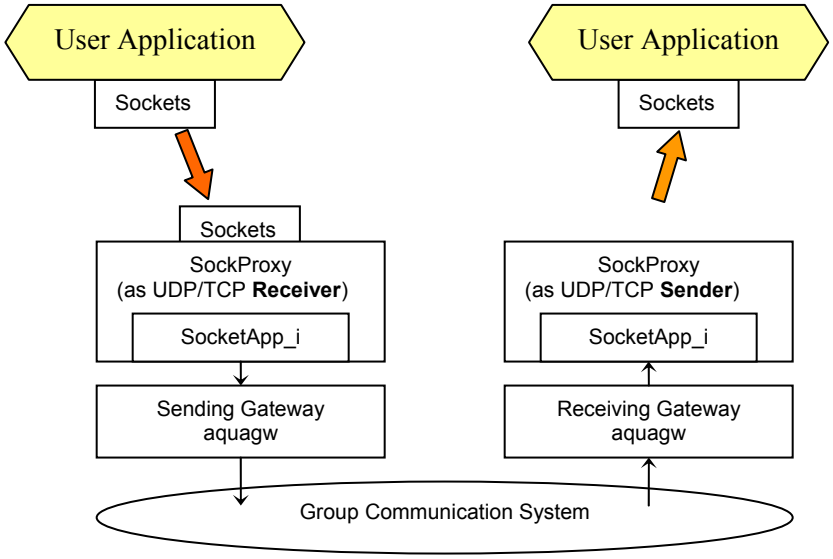

Figure 7. Socket Interface

Without gateways, the SockProxy can be run alone to verify the interface with the user socket application.

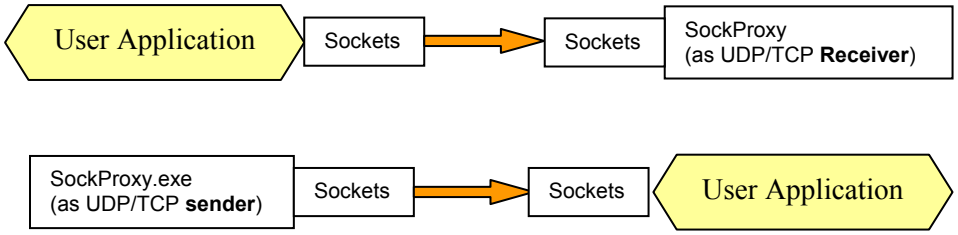

Figure 8. SockProxy Used Alone

3.2.6 Naming Service

The naming service is one of the first services to be created by the gateway. There is only one naming service per gateway, and it provides hash-table-based transient storage for name-to-object IOR bindings in a naming context.

It implements the CosNaming interface (from CORBA COSS, "Common Object Services Specification"), allowing the application to use a CORBA standardized way of accessing the gateway naming service.

The objects stored in the naming service (*specifically, the AQUA_Naming ACE dynamic service*) include, among others:

- all the handlers created in the current gateway (using the handler name indicated in the configuration file),

- the application object (under the application name specified in the configuration file, usually through the –a option of the handler factories),
- the starter factory under the name *AQUA_Starter_Factory*, and
- the gateway itself under the name *MY_GATEWAY_IOR*.

The naming service exports its IOR to the environment variable: *AQUAGW_NS_IOR*. Services associated with the gateway should use the value of this environment variable to bind with the naming service. Methods are provided in the gateway libraries to accomplish the task of retrieving the IOR from the environment variable.

3.2.7 Gateway Parameter Service

The Gateway Parameter Service (specifically the *AQUA_Gateway_Parameters* ACE dynamic service) is loaded via the service configurator file and creates a servant that can be used by an outside process to set parameters for the gateway at runtime. Some of the gateway components use this method to share parameter values with other internal components.

Parameters are defined as a pair of 2 strings:

- the parameter name, and
- the parameter value.

The parameter servant implements the following interface:

```
//! Interface for the AQUA_Gateway_Parameters, which is
//! implemented as a service object, registered into the Naming Service,
//! and used to start processes.

module aqua{

 interface Aqua_Gateway_Parameters {
  //! store the parameter value for the given keyname
  long bind_parameter(in string keyname, in string value);

  //! return the parameter value for the given keyname
  string get_parameter(in string keyname);
 };
};
```

All values passed on the aquagw process command line with the option –P are stored in the parameter service.

3.2.8 STDIN Handler

Another gateway component is the STDIN Handler. It is a service automatically loaded by the aquagw process to read all information from the process standard input stream and store it as a key-pair in the gateway parameter service.

To be successfully read and stored, the input must follow this syntax:

<PARAMETER_NAME>string_name_of_parameter**<\EOL>**
<PARAMETER_VALUE>string_value_of_parameter

Note that the <>s and end-of-line characters are required.

If the keys "<PARAMETER_NAME>" or "<PARAMETER_VALUE>" are not recognized or not provided in sequence, the reading fails and the parameter is not set.

The STDIN Handler can also be used by the application spawning the aquagw process to terminate this gateway. Keeping a handle on the aquagw stdin, writing "TERMINATE_GATEWAY" will terminate the gateway process.

3.2.9 Alert Sender

The Alert Sender is a service capable of sending CORBA and TCP alerts to a remote process. This process is usually some kind of manager monitoring the gateway and controlling it (setting parameters or terminating the gateway). We call such a process a *gateway controller*.

The controller parameters are read from the Gateway Parameter Service at initialization time but are overwritten by any option set on the command line.

For a TCP socket controller, the required parameters are:

- ALERT_HOST host where controller runs
- ALERT_TCP_PORT port where controller collects alerts

The interface with a CORBA controller has not yet been inserted in this generic class.

This service takes the following options:

-a<application_name> as used when the application or gateway is reported crashed
-c<gw type> (by default the gw type is "GW")
-p<controller_tcp_port> must be provided with option -h or ignored
-h<controller_host> must be provided with option -p or ignored
-i<controller_ior> for CORBA reports

The Alert Sender implements a method used by some of the handlers to send the groupViewChange() message to the local application via the DII Processor.

Another method is also available to the handlers to send view changes reports to the DM using the dynamic send handler named *Proteus* (if this handler has been specified and loaded).

Note: the sending of alerts from this service to a CORBA controller has not been completely implemented. The SBMV handler supports the interaction with a CORBA replication controller and makes up and sends CORBA reports directly from within the SBMV strategy code.

3.2.10 Starter Factory

The starter factory is a service available for starting processes automatically from within the application program or the service configurator file. It is used at initialization time to start the application process after loading the naming service and before creating the handlers. To use the starter factory, the following methods are available:

```
//! Interface for the AQUA_Starter_Factory, which is
//! implemented as a service object, registered into the Naming Service,
//! and used to start processes.

module aqua{
 interface Aqua_Starter_Factory
 {
```

```
  //! Kill a process created by this factory.
  long kill (in long pid);

  //! Start a process providing space-separated command line arguments.
  //! This process is not counted in the pool of started process and
  //! can die without impacting the others.
  long start_independant (in string cmd_line);

  //! Start a process providing space-separated command line arguments.
  long start (in string cmd_line);

  //! Start a process providing comma-separated command line arguments.
  oneway void start_process (in string cmd_line);
 };
};
```

The Starter Factory monitors all the processes it started (except the ones started using the *start_independant()* method). When the last monitored process has exited, the Starter Factory terminates the gateway.

3.3 Configuration

In this section, we describe how the gateway components are initialized and how the applications are started using a configuration file.

The gateway uses the ACE service configurator framework to load its components at startup time and query them at runtime. By definition, the ACE service configurator supports the configuration of applications whose services may be assembled dynamically at installation time and/or runtime. It allows for processes to be built around a framework of loadable modules that are then configured at runtime, rather than at link time.

Details for manually creating the service configuration files are provided in Section 3.3.1. Details for creating the service configuration files using a graphical editor are provided in Section 3.3.2. Example files are found in Section 3.3.3.

3.3.1 Syntax for Gateway Services

The following table lists the options available for the gateway components loaded as services through the service configurator file, and the preferred order of loading from top to bottom.

Some TAO and gateway services are automatically loaded from within the gateway code (aquagw process) and don’t need to be re-loaded in the user configuration file. These preloaded services are:

- TAO DLL Parser
- TAO File Parser
- TAO CORBALOC Parser
- TAO CORBANAME Parser
- Gateway ORB Servant
- Gateway Naming Service

- Gateway DII Processor
- Gateway Parameter Service parsing and storing all the pairs' parameter values found on the aquagw process command line (option –P)
- Gateway ML Ensemble Factory not supporting cryptography features
- Gateway C Ensemble Factory supporting cryptography features
- 2 Gateway GCS Adaptors called "MLEnsemble" and "CEnsemble"
- Gateway Group Member Factory
- Gateway Alert Sender

The following line:

dynamic AQUA_Naming_Svc Service_Object *
AQUA_Naming:_make_AQUA_Simple_Naming_Factory() "dummy"

means that we want to load *dynamically* a service object that we would like to name *AQUA_Naming_Svc*. This service is found in the dynamic link library *AQUA_Naming* (which will be translated into libAQUA_Naming.so on Unix) and is started through a call to *_make_-AQUA_Simple_Naming_Factory()* in this library. The arguments passed to this call are the arguments provided between quotes: "*dummy*".

Note: the string "dummy" replaces a program name, which is the first string provided on a command line. This program name is expected by the command line parser but not used by the service configurator file parser. When several arguments are to be provided, we chose to use commas to separate the arguments, because spaces confuse the ACE parser that is reading the service configurator file.

Gateway Services	**Syntax**	**Options**
Naming Service	dynamic AQUA_Naming Service_Object * AQUA_Naming:_make_AQUA_Simple_Naming_Factory() "dummy -o <ior_output_file>"	-o followed by the name of the file where the naming service ior is to be written.
DII Processor	dynamic DII_Processor Service_Object * AQUA_Handler_Base:_make_DII_Processor()	No arguments.
Gateway Parameter	dynamic AQUA_Gateway_Parameters Service_Object *AQUA_Base:_make_AQUA_Gateway_Parameters() "dummy –Pparam=value"	-Pparam=value Multiple -P options can be specified for multiple parameters. Attempt is made to resolve values starting with file:// by reading the ior string from the designated file name. When loading the parameter service, the gateway process passes its ior using the parameter "NAMING_IOR" so that the parameter service can register it in the naming service.
ML Ensemble Adaptor Factory	dynamic MLEnsemble Service_Object *GCS_Adaptor_MLEnsemble:_make_Ensemble_Factory()	No arguments.
C Ensemble Adaptor Factory	dynamic CEnsemble Service_Object *GCS_Adaptor_CEnsemble:_make_Ensemble_Factory()	No arguments.

Gateway Services	Syntax	Options
2 Adaptors called MLEnsemble and CEnsemble	dynamic GCS_Adaptor_Factory Service_Object *AQUA_Handler_Base:_make_GCS_Adaptor_Factory() "dummy -aMLEnsemble,-PUSE_CRYPTO=0 -aCEnsemble,PUSE_CRYPTO=1"	The only supported option is to specify adaptor default parameters: -a <adaptor type>,[comma-separated list of adaptor parameters] The list of parameters are adaptor-specific and not checked by this factory
Group Factory	dynamic Group_Factory Service_Object *AQUA_Handler_Base:_make_Group_Factory ()	No arguments.
Alert Sender	dynamic Alert_Sender Service_Object * AQUA_Handler_Base:_make_Alert_Sender()	-a<application_name> to be used when reported crashed -c<gw type> (by default the gw type is "GW") -p<controller_tcp_port> must be provided with option -h or ignored -h<controller_host> must be provided with option -p or ignored -i<controller_ior> for CORBA reports
Starter Factory	dynamic AQUA_Starter Service_Object *AQUA_Base:_make_AQUA_Starter_Factory() "dummy –d<delay> -s<time-to-sleep-after startup> -w<working-directory> [-x<comma_separated_command_line_arguments>]"	-w <working directory for all the processes to be started here> -x <process_name,[comma-separated argument list]> -d <delay before starting processes> -s <time to sleep after starting processes> To start more than one process, several –x options can be listed on the command line but all other options are common for all started processes. For example to start "process.exe –a10 -binput_file", write: -x process.exe,-a10,-binput_file
Multicast Handler	dynamic AQUA_Multicast Service_Object *AQUA_Multicast_Handler:_make_AQUA_Multicast_Handler_Factory() "dummy -a<local_appli_name> -c<GCS_adaptor_name> [-n<min_group_size>] [-r] -g<group_name>"	-a: the name of the local application this gateway belongs to -g: the name of the GCS adaptor to use (either MLEnsemble or CEnsemble) -c: the name of the group to join -n: followed by the minimum number of members in the group before the cast can take place. Default is 1. This value is applied to all handlers created. Messages cast while the actual group size is below the required minimum are dropped. -r: to indicate that sender wishes to receive its own cast. By default, sender does not receive the messages it casts.

Gateway Services	Syntax	Options
Dynamic Send Handler	dynamic AQUA_Dynamic_Send Service_Object *AQUA_Dynamic_Send_Handler:_make_AQUA_Dynamic_Send_Handler_Factory() "dummy –a<local_appli_name> -g<GCS_adaptor_name> -h<remote_group_name>"	-a: the name of the local application this gateway belongs to -g: the name of the GCS adaptor to use (either MLEnsemble or CEnsemble) -h: the name of the communication group to dynamically join and scast (cast only to servers) messages to. For replicated applications, this handler should always be loaded with option -hProteus. The leader replication group members need the handler in order to send view change messages to Proteus. -c: to cast messages to group using cast (cast to all) instead of scast whenever other members (clients or servers) are present. When this option is not set, the messages will be scast to the group whenever there is at least one server in the group (in this case, the local application running on top of this handler should be defined as stateless (client) to avoid being counted as a server).
(Active) PassFirst Strategy Factory	dynamic Active_PassFirst Service_Object * AQUA_Active_PassFirst:_make_AQUA_Active_PassFirst_Factory() ""	No arguments.
(Active) LeaderOnly Strategy Factory	dynamic Active_LeaderOnly Service_Object * AQUA_Active_LeaderOnly:_make_AQUA_Active_LeaderOnly_Factory() ""	No arguments.
(Active) SBMV Strategy Factory	dynamic Active_SBMV Service_Object * AQUA_Active_SBMV:_make_AQUA_Active_SBMV_Factory() "dummy -s<send_majority_size> -r<receive_majority_size> -f -l<late_vote_timeout_in_secs> -m<majority_timeout_in_secs>	-s <send_majority_size> The initial minimum number of active replicas (in the local group) will also be set to this value (default is 2). -r <receive_majority_size> -f to use keys from files. If this option is set, keyfiles must have been generated using $AQUA_ROOT/gen-key <host_ip_address> in a subdirectory Keysets where the application resides. A key file should exist for each host involved in the communication (client-server) -l <late_vote_timeout_in_secs>. Maximum amount of time to wait for a late vote (default is 15). -c <recovery_time_in_secs>. Time given for the system to start a number of active replicas greater than or equal to the minimum required (default is 10). -m <majority_timeout_in_secs>. Maximum amount of time to reach a majority after the 1st vote (default is 10).
(Passive) StateCast Strategy Factory	dynamic Passive_StateCast Service_Object * AQUA_Passive_StateCast:_make_Passive_StateCast_Factory( )	No arguments.

Gateway Services	Syntax	Options
Active Handler Factory	dynamic AQUA_Active_Handler_Factory Service_Object *AQUA_Active_Handler:_make_AQUA_Active_Handler_Factory() "dummy –a<local_application_name> -g<GCS_adaptor_name> [-s] [-i] -h<remote_appli_name>,<handler_name>, [connection_group_name],<communication_strategy_name>"	-a: local application name -g: the name of the GCS adaptor to use (either MLEnsemble or CEnsemble) -s: to indicate that the local application uses state (and therefore implements the methods set/getState()) -i: to have the handlers following this setting (options –h that immediately follow the –i) use a fault injector. The corresponding handlers should use the SBMV to take full advantage of the injector. -h: followed by 3 or 4 handler creation arguments, in this order: • remote_application_name • handler_name (is usually same as remote application name but needs to be different from local application name) • connection_group_name (optional) • communication_strategy_name (currently Active_PassFirst is the only available active handler)
Passive Handler Factory	dynamic Passive_Handler_Factory Service_Object * AQUA_Passive_Handler:_make_Passive_Handler_Factory() "dummy –a<local_application_name> -g<GCS_adaptor_name> [-s] -h<remote_appli_name>,<handler_name>, [connection_group_name],<communication_strategy_name>"	-a: local application name -s: to indicate that the local application uses state (and therefore implements the methods set/getState()) -g: the name of the GCS adaptor to use (either MLEnsemble or CEnsemble) -h: followed by 3 or 4 handler creation arguments, in this order: • remote_application_name • handler_name (is usually same as remote application name but needs to be different from local application name) • connection_group_name (optional) • communication_strategy_name (currently StateCast is the only available passive handler)

Note: in the table above, [optional] arguments are between [].

3.3.2 Configuration File Editor

A graphical editor is included with the gateway to help with the basics of creating and maintaining the configurator files. A picture of this tool is shown in Figure 9.

This editor supports creating and saving configurator files. It prompts the user for the following pieces of information:

- **Application Type (Executable/Java Class):** Choose “Executable” if the application is a compiled binary or a custom script. Choose Java Class if it is a java program. Java programs are run using a script, included with the gateway, that is called java_exec_script on unix and java_exec_script.bat on Windows.
- **Application Class File:** This field is only enabled for Java Class applications. Enter the name of the Java class containing the “main” routine for the application.

- **Application Jar File:** This field is only enabled for Java Class applications. Enter the path to the jar file containing all the classes specific to the Java application.

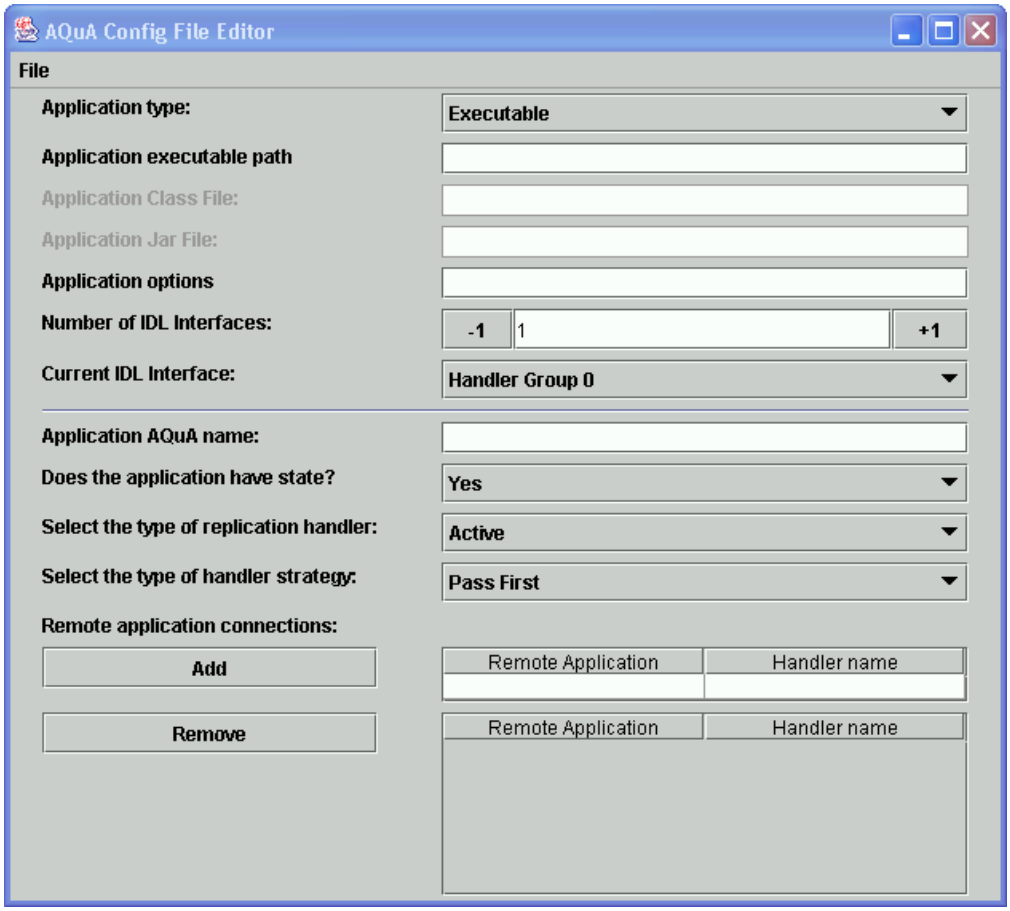

Figure 9. Gateway Configuration File Editor

- **Application Options:** If the application requires specific command-line options, they can be entered here.
- **Ensemble Debugging Messages (On/Off):** Choose "On" to enable debugging messages from the Ensemble group communication layer. Choose "Off" to disable them. These messages are especially helpful to the gateway development team when they are diagnosing a problem.
- **Number of IDL Interfaces:** The ITUA-AQuA gateway can interface with multiple interfaces within a single application. Often, it is helpful to have a single application support multiple interfaces through the same gateway. An example of this is the deetClient, which supports its interface and a QoSRequest interface.
- **Current IDL Interface:** This option menu allows the user to choose which of the interfaces he wants to display and edit. The lower half of the display (below the separator line) updates based on the interface selected by this option.
- **Application Name:** Each interface of the application needs a unique name, which is the name that is registered in the naming service. The ITUA-AQuA gateway uses this name to identify and bind to the application.
- **Does the application have state? (Yes/No):** Choose "Yes" if the application has state, and "No" if it does not. Applications with state must derive from the Application.idl file and implement the appropriate getState and setState methods.
- **Select the type of replication handler (Active/Passive/Other):** Choose "Active" for active replication, "Passive" for passive replication, and "Other" for a non-replication handler. Each choice will update the op-

tions in the next field for the strategies appropriate for the given handler type. Detailed descriptions of these handler types can be found in Chapter 2.

- **Select the type of handler strategy:** Currently, there are three choices for Active handlers, called “Pass First,” “Leader Only,” and “SBMV”; one choice for Passive handlers, called “State Cast”; and two choices for other handlers, called “Multicast” and “Dynamic Send.” Detailed descriptions of these strategies can be found in Chapter 2.
- **Remote application connection:** This allows the user to list the names of the remote applications this application needs to connect to. Typically, this list includes the server (if the application is a client) or the client (if the application is at the server). It can also include other components in the application or outside the gateway, such as a QoSRequestor or the Proteus Dependability Manager. The “Remote Application Name” is given in the first column, and must match the “Application name” for the application being connected to. The “Handler name” is the name given to the handler in the ITUA-AQuA gateway, and is used by the application to bind to the handler. There can be no duplicated names within each column. To add an entry to the connection list, enter the “Remote Application” name and “Handler name” in the upper table. Then hit “Enter” to validate the handler name and the “Add” Button to move the entries to the lower table. Repeat this process to add multiple connections to the lower table. Cells in the lower table can be edited. Remove rows in the lower table by selecting them and hitting the “Remove” button.

3.3.3 Two Examples

Described below are two examples of service configurator files to run the Deet application (4.2.2) using either active or passive replication schemes.

3.3.3.1 Active Example

```
# Load the AQUA Starter Factory and start the process
dynamic AQUA_Starter Service_Object *AQUA_Base:_make_AQUA_Starter_Factory() "dummy -
xjava_exec_script,deet.deetServer,java/Deet.jar,deetServer_lo,active,-output,ds_output.log"

# Load a dynamic send handler to communicate with the Proteus group
# Note: make sure to use as -a, the same object name used and expected
# by the DM as remote object.
dynamic AQUA_Dynamic_Send Service_Object
*AQUA_Dynamic_Send_Handler:_make_AQUA_Dynamic_Send_Handler_Factory() "dummy -
adeetClient_QoS -gMLEnsemble -hProteus"

# Load the Communication Strategy Factory.
dynamic Active_LeaderOnly Service_Object *
AQUA_Active_LeaderOnly:_make_AQUA_Active_LeaderOnly_Factory() ""

# Load the Active Handler Factory.
dynamic AQUA_Active_Handler_Factory Service_Object *
AQUA_Active_Handler:_make_AQUA_Active_Handler_Factory() "dummy -adeetServer_lo -s -gMLEnsemble -
hdeetClient,dServer_dClient,Active_LeaderOnly"
```

3.3.3.2 Passive Example

```
# Load the AQUA Starter Factory and start the process
dynamic AQUA_Starter Service_Object *AQUA_Base:_make_AQUA_Starter_Factory() "dummy -
xjava_exec_script,deet.deetServer,java/Deet.jar,deetServer_sc,passive"

# Load a dynamic send handler to communicate with the Proteus group
```

```
# Note: make sure to use as -a, the same object name used and expected
# by the DM as remote object.
dynamic AQUA_Dynamic_Send Service_Object
*AQUA_Dynamic_Send_Handler:_make_AQUA_Dynamic_Send_Handler_Factory() "dummy -
adeetClient_QoS -c -gMLEnsemble -hProteus"

# Load the Communication Strategy Factory.
dynamic Passive_StateCast Service_Object * AQUA_Passive_StateCast:_make_Passive_StateCast_Factory() ""

# Load the Passive Handler Factory.
dynamic Passive_Handler_Factory Service_Object *
AQUA_Passive_Handler:_make_Passive_Handler_Factory() "dummy -adeetServer_sc -gMLEnsemble -s -
hdeetClient,dServer_dClient,Passive_StateCast"
```

3.4 Gateway Startup

To start the gateway along with the application process, create a file named svc.conf and list the above services with arguments matching your application and environment.

Start an application along with its gateway by running:

aquagw [-d] [-f service_config_file_name] [-PISFIRST] [-P<parameter=value>

- -d to see TAO debug traces. The environment variable AQUA_VERBOSE_LEVEL is used to control the verbose level of the debug output. The verbosity level varies from 0 to 3, where 3 provides the most output.
- -f to specify a service configurator file name if different from "svc.conf".
- -PISFIRST to indicate that the replica (or standalone object) is the first to be started in its group. This flag is very important and must also be used for running non-replicated objects.
- As many –P options as desired to specify gateway parameters.

4 Gateway Applications

4.1 Building a CORBA Gateway Application

A gateway application is a CORBA servant using an ORB compatible with the TAO ORB, which is the gateway ORB. We have successfully tested the compatibility of JACORB and Visibroker for Java applications (Visibroker 4.5) with TAO. Before using another ORB, make sure it is compatible with TAO and uses Visibroker-compatible interceptors. CORBA Interceptors are used by the passive State Cast strategy in the gateway to append the state information to each CORBA message.

In this chapter, we will describe the structure of gateway applications and how they should interact with the gateway.

First, we discuss issues related to an application state. Specifically, a distinction must be made between applications that have state (and thus should have this state transferred between replicas) and those that don't have state (i.e., non-replicated client applications, or simply active replicated applications without state). An application with state must derive from an ITUA-AQuA-provided idl (Application.idl) and specify the appropriate option when loading handler factories in the service configurator file.

Second, we list the steps required to interface applications and the gateway.

4.1.1 Application State

Applications with state must implement the ITUA-AQuA Application interface described below.

```
module aqua {
 //! ITUA-AQUA Application interface.
 interface Application
 {
  //! Generic Application State
  typedef sequence<any> State;

  //! Get the application state.
  void getState(out State s);

  //! Set the application state.
  long setState(in State s);

  //! Inform the application it is now the leader replica
  oneway void isLeader();

  //! Notify the application that the communication group indicated
  //! went through a view change (list of current member ids is provided
  //! as a list of strings separated with columns)
  oneway void groupViewChange (in string groupName, in string memberIds);
 };
};
```

This interface is located in the `$AQUA_ROOT/aqua/` directory, under the file name `Application.idl`, where `$AQUA_ROOT` is the path to the ITUA-AQUA directory.

To make use of it, simply add the following at the top of the application's idl file, and derive the application's idl from it:

```
#include "aqua/Application.idl"
```

Then, in the service configurator file used by the application to start the gateway, the option `[-s]` must be set for all handler factories if the application has state (see syntax for gateway services in the table in Section 3.3.1). This option should not be set for a stateless application; if it is, the handlers will try to invoke the missing `getState()/setState()` methods to the application, causing an exception to be thrown.

The methods `isLeader()` and `groupViewChange()` need to be defined (even empty) in all applications regardless of whether they are stateful. `isLeader()` is invoked anytime the application gateway is elected the new leader of a communication group. This callback, which is associated with the `groupViewChange()`, helps to determine which groups care whether the gateway is a member of more than one group. Note that there are no callbacks to inform the application when it is no longer the leader. The correct use of the –PIS_FIRST option on the aquagw command line is essential for the successful delivery of this callback.

`groupViewChange()` is invoked anytime there is a change of membership in a group (someone joining or leaving).

4.1.2 Guidelines for Interaction with the Gateway

All ITUA-AQuA applications should follow the following steps when accessing the gateway:

1) Activate the application object (save the corresponding object reference).

2) Register the application with the gateway naming service.

3) Get a reference to the gateway handler for this application.

4) Narrow the handler reference down to the object type of the remote object.

5) Use the handler object as if it were the remote object.

Several methods are provided within the gateway framework to aid in these steps, especially steps 2 and 3. These methods are available in both Java and C++, and are found in the class AquaUtils. The Java version is located in the `components.jar` file, and the C++ version is located in library `AQUA_Handler_Base`. The Java versions are highlighted here:

- **public static String getAquaNamingService():** This method returns the gateway naming service IOR stored in the java property *aquagw.ns.ior* set by the script java_exec_script (if this script is used to start the application). It is useful if the application needs access to the naming service prior to registering itself.
- **public static NamingContextExt registerWithAquaNamingService(org.omg.CORBA.ORB orb, org.omg.PortableServer.POA poa, Servant application, String application_name):** This method registers the given "application" under the given "application_name" with the gateway naming service. This performs step 2 in the above list.
- **public static org.omg.CORBA.Object connectToAquaGateway(String handler_name):** This method returns a reference to the gateway handler specified by "handler_name". This performs step 3 from the above list.

For C++ applications, taking the above steps results in the following code (taken from the pinger client example application).

```
#include "handlers/base/AquaUtils.h"
...
// Activate the application object (type Pinger_var)
Pinger_var pinger = pinger_impl._this (ACE_ENV_SINGLE_ARG_PARAMETER);
ACE_TRY_CHECK;

CORBA::String_var ior =orb->object_to_string (pinger.in() ACE_ENV_ARG_PARAMETER);
ACE_TRY_CHECK;

Pinger_var server;

// register this application with the naming service. this
// allows the aquagw process to connect to its application
AquaUtils::RegisterWithAquaNamingService(orb,pinger.in(),appli_name);

// connect to the gateway of the server
CORBA::Object_ptr obj = AquaUtils::ConnectToAquaGateway(orb, handler_name);

// narrow the handler to the remote application
server = Pinger::_narrow (obj ACE_ENV_ARG_PARAMETER);
```

For Java applications, following the above steps results in code similar to the following (inspired by the Deet example application):

```
...

// Activate the application object (type deetClient)
deetClient client = new deetClient( key );

// Activate the servant with the ID on myPOA
try {
 poa.activate_object_with_id( key.getBytes(), client );
}
catch (Exception e) {
 System.out.println("Failed to activate object" + e);
 System.exit(1);
}

// register the client application with the gateway naming service
AquaUtils.registerWithAquaNamingService(orb_,poa,client,key);

POA rootPOA = AquaUtils.rootPOA( orb_ );
try {
 rootPOA.the_POAManager().activate();
}
catch (org.omg.PortableServer.POAManagerPackage.AdapterInactive e) {
 System.err.println("error with the poa " + e);
 System.exit(1);
```

```
}

// get reference to gateway handler
obj=AquaUtils.connectToAquaGateway(handler_name_);

// narrow reference down to remote application
deet_server d = deet_serverHelper.narrow( obj );
```

4.1.3 Replicating Servers Using the Dependability Manager Object Factories

The Dependability Manager (DM) has an associated process called the Object Factory. The Object Factory is tasked by the DM to start the processes of replicated objects. There should be an object factory on each host where replicas are desired. Object factories are started by this command:

```
cd $AQUA_ROOT/proteus/java
aquagw –d –f factoryX.conf  (where X=1 to 12)
```

Each factory should be started using a unique value of *X* in the `factoryX.conf` filename. When the DM receives a QoS request (using a QoS requestor like the one described in Section 4.1.4.1), it sends commands to the necessary number of object factories on the remote hosts. Each object factory then starts the appropriate process.

The list of processes the object factory may start is defined in a ".procs" file. The default file is located at $AQUA_ROOT/proteus/java/common.procs. The default file contains instructions for starting all the processes required for the example applications included with the gateway.

The basic format for the ".procs" file is pairs of lines called *command sets*. The first line is the ITUA-AQuA application name for the process to start. The second line is the full command to start the process.

The path to the procs file (or a subset of the procs file) is given to the QoS requester to populate the pull-down list of available processes over which a QoS request can be issued. This is also discussed in Section 4.1.4.1.

Here is an example from common.procs of the command set for the "deetServer" application.

```
deetServer_lo
aquagw_launcher aquagw -d -f apps/deet/gw-lo-server.conf
```

4.1.4 Calls Between Application and Dependability Manager

Proteus supports the development of objects that can make QoS requests from the dependability manager and also observe its actions. One type of these objects, *QoS observer/requesters*, can be used to make QoS requests to the manager, and can receive callbacks regarding the ability of the manager to satisfy the requester's requests. (An example of an application that may contain a QoS observer/requester is QuO itself.) Furthermore, since the dependability manager supports a standard, well-defined interface, an application object can also make QoS requests directly to the dependability manager.

The second type of objects the dependability manager supports is that of *advisor observers*. Advisor observers can "subscribe" to a variety of information used by the manager to make decisions, including information about faults

detected and fine-grained information regarding actions taken by the manager. In addition, advisor observers can receive information regarding the status of hosts that may be used to execute object replicas, and can be used to request that a particular host be added or deleted from the set of active hosts. In particular, as will be seen in the following, they can be used by an application or QuO to specify hosts that should not be used to execute replicas, if the application or QuO has information that leads it to believe that the host should not be used.

4.1.4.1 QoS Observer/Requester

QoS observer/requester objects specify the level of dependability desired of a remote object, and receive information regarding the ability of an ITUA-AQuA-based system to meet that level of dependability. These objects can be implemented in a QuO system condition object, or as part of an application object that makes use of the remote object. Six methods are used in the interface between the dependability manager and the QoS observer/requester. Three of the methods are implemented in the dependability manager, and receive information regarding the desired dependability of ITUA-AQuA-managed objects. The other three methods are implemented in the QoS observer/requester, and receive information about the dependability manager's ability to meet a request.

The three methods implemented in the dependability manager support the definition, modification, and removal of QoS requests. The first method, `registerQoSRequest()`, is called to register a new QoS request (one that does not replace or supercede any pre-existing request). The argument *QoSRequest_info* is passed with this call and contains the specifics of the QoS request. This information includes 1) a reference to the QoS observer/requester that should be notified when events concerning this QoS request occur, 2) the remote object whose dependability is to be managed, 3) the number of crash failures, value faults, and time faults of this object that should be tolerated, and 4) the length of time that the dependability level of the remote object can remain below the requested value before a callback must be made to the specified QoS observer/requester. Upon a successful request, the method returns the QoSRequest ID that should be used in the future whenever the caller wants to refer again to this request.

The `updateQoSRequest()` method substitutes a new QoS request for one that was previously registered. The parameters of that call are the QoSRequest ID (identifying the QoS that will be updated) and the QoSRequest_info (containing the new information concerning the QoS request). Finally, the `removeQoSRequest()` method eliminates a previously registered QoS request without replacing it with a new request. This method is used only when a requesting object no longer needs a remote object. When the dependability manager receives a `removeQoSrequest()`, it adjusts the number of replicas of the referenced object to satisfy any other requests that have been made for this object, killing replicas if appropriate.

The dependability manager calls three methods on QoS observers/requesters. Two of them serve to indicate that a QoS request has become unsatisfied, and that a QoS request that had become unsatisfied has once again become satisfied. Both of these method calls have the QoSRequest ID, the current number of replicas, and the current number of "active hosts" as parameters. More specifically, the `QoSRequestNotSatisfied()` method is called:

1) When a new QoS request requests that more faults be tolerated than is possible given the current set of active hosts.

2) When a QoS request that was previously satisfied by the number of hosts that were then active can no longer be satisfied by the current number of active hosts.

3) When a QoS request that was satisfied is no longer satisfied, due to a change in the number of replicas of the managed object. (Note that the callback occurs after the dependability manager has attempted to obtain the QoS request for a period of time defined by the recovery time specified in the QoSRequest_info.)

Similarly, the `QoSRequestSatisfied()` method is called when a QoS request that previously could not be satisfied (as indicated by the `QoSRequestNotSatisfied()` call) can once again be satisfied.

The `NoMajority()` method is called when the dependability manager receives the noMajority message from the leader of the replication group when tolerating value faults.

Six calls exist between a QoS observer/requester and a dependability manager. The three calls from a QoS observer/requester to a dependability manager are:

- `registerQoSRequest()`
- `updateQoSRequest()`
- `removeQoSRequest()`

The three calls from a dependability manager to a QoS observer/requester are:

- `QoSRequestNotSatisfied()`
- `QoSRequestSatisfied()`
- `NoMajority()`

Each of these methods is described in more detail in Appendix A.

A simple QoSRequester is provided as part of the ITUA-AQuA framework, within the `proteus.jar` file. To incorporate this object into an existing application, the following steps should be taken:

1. Add the constructor for the QoSRequester to the application. When the client application is started, a popup window will appear so that the user can specify the QoS requirements for the server. The constructor looks like this:
   QoSRequester(String server_ name, String procs_filename, String aquagw_naming_ior, org.omg.CORBA.ORB orb): "server_name" is the ITUA-AQuA name for the server interface. "procs_filename" is the name of the file containing the commands to start the servers (discussion of the ".procs" file is found in Section 4.1.2). "aquagw_name_ior" is the ior string for the ITUA-AQuA gateway naming service. "orb" is the CORBA orb created within the client.

2. Update the configuration file for the application to add another IDL interface for the QoSRequester. An example of this is shown in `Figure 10`. In the new IDL interface, the application name must be <Main-Application>:QoSRequester. The QoSRequester does not have state, does not need to be marked as the first replica, and should use the active replication handler. There should be one remote connection to the application "Proteus", using a handler name of <Main-Application>:QosRequester:Handler. In this case <Main-Application> refers to the Application name for the primary IDL interface of this application, in this case "deetClient".

3. Update the Configuration file for the Proteus Dependability Manager. An additional Remote Application Connection needs to be added to connect the application to the newly added QoSRequester. This new connection should have a Remote Application name that matches the Application name used for the QoSRequester added in Step 2, <Main-Application>:QoSRequester. For simplicity, the same name should be used as the Handler name.

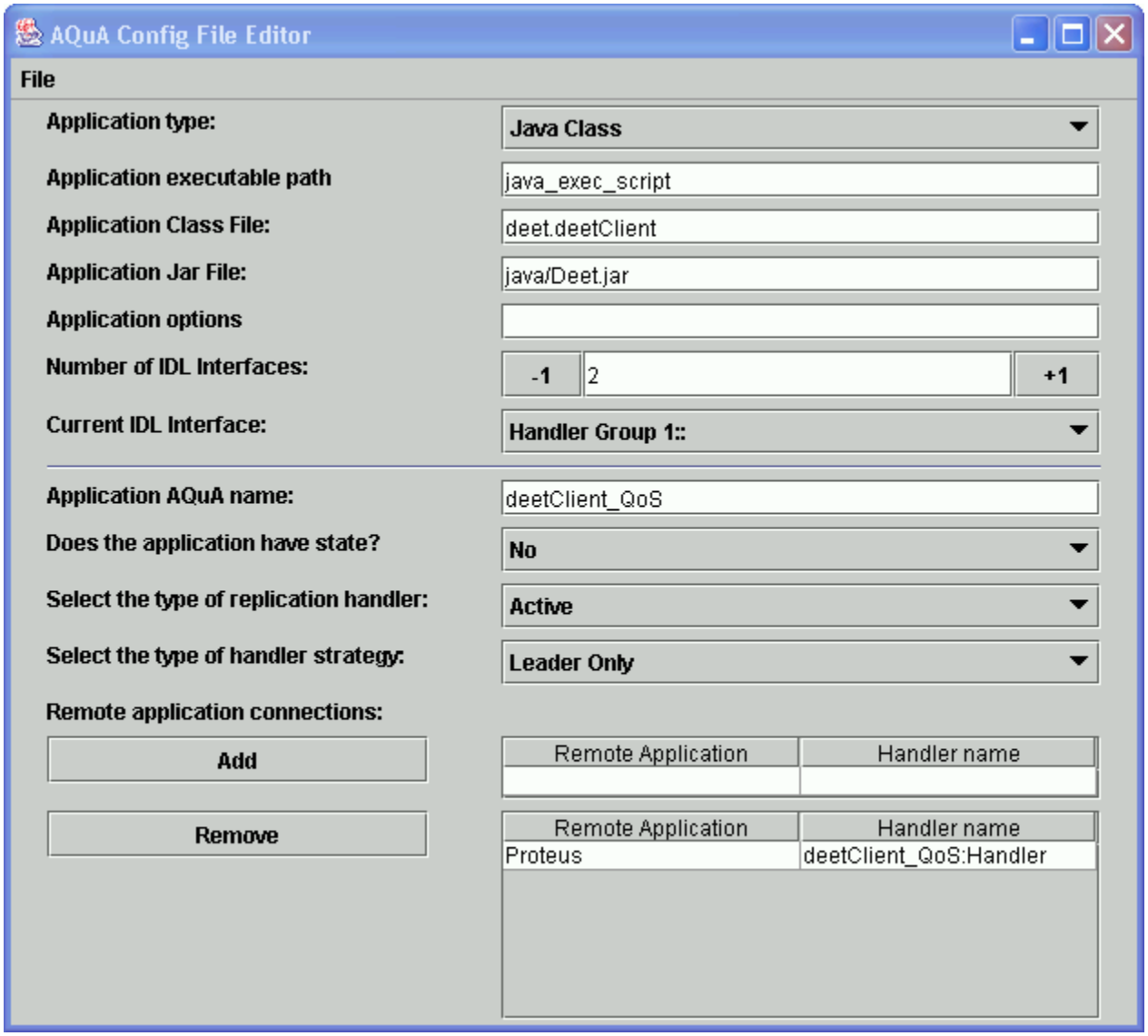

Figure 10. Adding QoSRequestor to Configuration File

4.1.4.2 Advisor Observer

An advisor observer can "subscribe" to a variety of information maintained by the dependability manager to make decisions, including information about faults detected and fine-grained information regarding actions taken by the manager. It can also receive information focusing on the status of hosts that may be used to execute object replicas or make requests regarding which hosts can be used to execute replicas. We now describe these two functions of an advisor observer, which are called the *dependability manager observer* and the *host status manager*.

The dependability manager supports multiple advisor observers, which can dynamically register and de-register with the dependability manager at runtime. More specifically, the `registerAdvisorObserver()` method, called on the manager by advisor observers, registers an advisor observer with the dependability manager. To register, an advisor observer passes 1) a reference to the advisor specifying the methods that should be called when events concerning this advisor observer occur, and 2) a specification of the information it desires from the dependability manager. Upon a successful return, the register call returns an advisor observer identification ID that can be used to de-register the advisor observer, using the `removeAdvisorObserver()` method.

An example implementation of the Advisor Observer interface can be found in the `observers` directory inside the gateway distribution.

4.1.4.2.1 Dependability Manager Observer

As a dependability manager observer, an advisor observer object can be used by QuO or an application object that wishes to receive more detailed information concerning fault notifications and decisions of the dependability manager advisor than is provided by the `QoSRequestNotSatisfied()` and `QoSRequestSatisfied()` method callbacks. An object may want this information, for example, to make decisions on how to adapt in a particular situation. To receive this information, each advisor observer implements several methods that may be called by the dependability manager. The types of events and actions of which an advisor observer is notified depend on the type of information an advisor observer requests when it registers with the dependability manager.

Each advisor observer implements several methods, as shown in the following table, to receive information from the dependability manager and to specify the action that is to be taken upon receiving that information. In particular, the `faultOccurred()` method is called on each advisor observer when the dependability manager detects a fault. This method provides, as arguments to the call, the type of fault detected, the host where the fault was detected, and the dependable object associated with the fault. All advisor observers receive this information, regardless of what other information they requested when they registered with the dependability manager. The seven other methods that must be implemented by an advisor observer are called on those advisor observers that have requested information related to a particular call. Specifically, the `notifyNumReplicas()` method provides the name of the dependable object to which the call refers and the number of replicas of this object in the current system configuration. The remaining calls provide this information, as well as the name and status information (e.g., load) for the host to which the call refers.

Method	Function
`FaultOccurred()`	Called when the dependability manager detects a crash failure or value or time fault.
`NotifyNumReplicas()`	Called when a new QoS request is registered and whenever the number of replicas in the replication group changes.
`ReplicaStartAttempted()`	Called when the dependability manager attempts to start a replica.
`ReplicaKillAttempted()`	Called when the dependability manager attempts to kill a replica.
`ReplicaStartFailed()`	Called when a new replica either could not be started by the object factory or could be started but could not join the replication group.
`ReplicaKillFailed()`	Called when the replica could not be killed by the object factory, and the kill failure was reported to the dependability manager.
`ReplicaStartSuccessful()`	Called when a replica was started successfully by the object factory, joined the replication group, and was reported to the dependability manager.
`ReplicaKillSuccessful()`	Called when a replica was killed successfully by the object factory and was removed from the replication group, and the removal was reported to the dependability manager.

Table 2 Dependability Advisor Observer Callbacks

4.1.4.2.2 Host Status Manager

As a host status manager, an advisor observer can receive status information concerning hosts that are being used to execute dependable objects, and give instructions regarding hosts that should or should not have replicas placed on them by the dependability manager. An advisor observer gives these instructions by suggesting changes in the status of hosts. Depending on a host's status, it is placed in a certain set by the dependability manager.

The dependability manager has three sets of hosts: an active host set, an inactive host set, and a removed host set. When an object factory registers a host to the dependability manager, the host is placed into the active host set. When an advisor observer requests that a host be deactivated, the host is placed into the inactive host set. When the dependability manager detects the failure of an object factory or a host, the host is placed into the removed host set. Replicas running on removed hosts are assumed to have failed. If a host in the inactive host set is reactivated by an advisor observer, the host is moved back to the active host set. If a failed object factory or a failed host is restarted, the host is moved from the removed host set to the active host set. The newly started object factory communicates with the dependability manager advisor to initiate its state. The dependability manager will not create replicas on a host that is in the inactive host set or in the removed host set. It will also migrate the replicas on a host in the inactive host set to hosts in the active host set.

Two methods are called by an advisor observer on the dependability manager to activate and deactivate hosts. The method `deactivateHost()` is used to request that the dependability manager deactivate the host specified by the call. When this call is made, the dependability manager will move the host from the active host set to the inactive host set, and also migrate the replicas from this inactive host to hosts in the active host set. The method `activateHost()` is used to request that the dependability manager move an inactive host from the inactive host set back to the active host set. Each advisor observer must implement four methods to receive information concerning hosts from the dependability manager. Whether these methods are called depends on the information that the advisor observer requested when it registered with the dependability managers. The method `hostActivated()` may be called 1) when a host that is either in the removed host set or in no host set registers with the dependability manager, and 2) when a host that is in the inactive host set is reactivated. The method `hostRemoved` may be called when a host or an object factory failure is detected by the dependability manager, and the host is moved from the inactive or the active host set to the removed host set. This method will be activated when the dependability manager detects host and object factory failures. The method `hostDeactivated()` may be called when a host is deactivated by an advisor observer. The method `hostInfo()` is called by the dependability manager for all hosts in the active set if the `hostInfo()` method is enabled.

Among the sixteen calls between an advisor observer and the dependability manager, four calls come from an advisor observer to a dependability manager:

- `deactivateHost()`
- `activateHost()`
- `registerAdvisorObserver()`
- `removeAdvisorObserver()`

The twelve other calls, which come from a dependability manager to an advisor observer, are:

- `faultOccurred()`
- `hostActivated()`
- `hostRemoved()`
- `hostDeactivated()`
- `notifyNumReplicas()`
- `replicaStartAttempted()`

- `replicaKillAttempted()`
- `replicaStartFailed()`
- `replicaKillFailed()`
- `replicaStartSuccessful()`
- `replicaKillSuccessful()`
- `hostInfo()`

All sixteen calls are detailed in Appendix B.

4.2 Gateway Application Examples

4.2.1 The Pinger Application

The pinger application is written in C++ and is the simplest example included with the gateway. It is a client-server application. The client transmits a short message to the server (a "ping") and the server replies. This message is repeated 10 times by default. This can be adjusted by editing the `gw-client1-svc.conf` file and changing the `-i10` argument on the client application StarterFactory line to `-iX`, where `X` is the desired number of pings.

It is recommended that the pinger be the first application run after the gateway is installed, to verify that the installation is configured correctly. Here are the steps for running the pinger application:

Configuring the Pinger Application

1. Notes:
    a. Before configuring and running the pinger demonstration, you need to configure the gateway for your system. See Section 6.2 for information on how to do this.
    b. In order to run a gateway-based program, you need to make sure that the Ensemble gossip server is running. The command to start it is $AQUA_ROOT/bin/`gossip`.
2. Open four terminals and `cd` to `$AQUA_ROOT/handlers/tests/Pinger` in each of them.
3. In the first window, run the logger utility:

    ```
    >server_loggerd
    ```

4. In the second window, run the first server:

    ```
    >aquagw -f gw-server1-svc.conf –PIS_FIRST
    ```

5. In the third window, run the second server:

    ```
    >aquagw -f gw-server1-svc.conf
    ```

6. In the fourth window, run the client:

```
aquagw -f gw-client1-svc.conf –PIS_FIRST
```

7. After the client completes ten pings, the client sends a "shutdown" command to the servers, and the client and two servers should exit.

   In the server terminals there should be output similar to the following:

```
(5404|1) Server> recv `Hello #10'

(5404|1) Server> send `Bonjour #10'
```

   In the `server_loggerd` terminal there should be output similar to this:

```
>server_loggerd
(5390|1) starting up server logging daemon
(5390|1) connected with localhost
(5390|1) server 1> ready to handle requests...
(5390|1) connected with localhost
(5390|1) server 1> ready to handle requests...
(5390|1) connected with localhost
(5390|1) client 1> starting now...
(5390|1) client 1> finished! Got 10 replies over 10 requests sent
in 0460 ms
(5390|1) closing log daemon at host localhost (fd = 8)
(5390|1) server 1> finished! Got 10 requests and sent 10 replies.
(5390|1) closing log daemon at host localhost (fd = 6)
(5390|1) server 1> finished! Got 10 requests and sent 10 replies.
(5390|1) closing log daemon at host localhost (fd = 7)
```

8. This test can be repeated using "`gw-client1-1000-svc.conf`" in step 6 for a longer test of 1000 pings.

4.2.2 The Deet Server

The Deet application is the simpler of the two Java applications provided with the gateway. It is used by the gateway development team to test various properties of the gateway . (Its name is a reference to a popular insect repellent.) Deet is a client-server application. The server maintains state constituted of a string and a single-integer counter. Each time the client sends an "echo" message to the server, the server state is incremented. The Deet client and server are shown in `Figure 11`.

Deet can be run two different ways, either manually or with the dependability manager. Running it manually is a simpler test to execute and allows the user to start and stop the server replications directly. Running with the dependability manager is a more advanced test that covers the entire system. In that mode, the dependability manager is responsible for starting replicas to maintain the desired QoS level. Instructions for running Deet in both modes are provided.

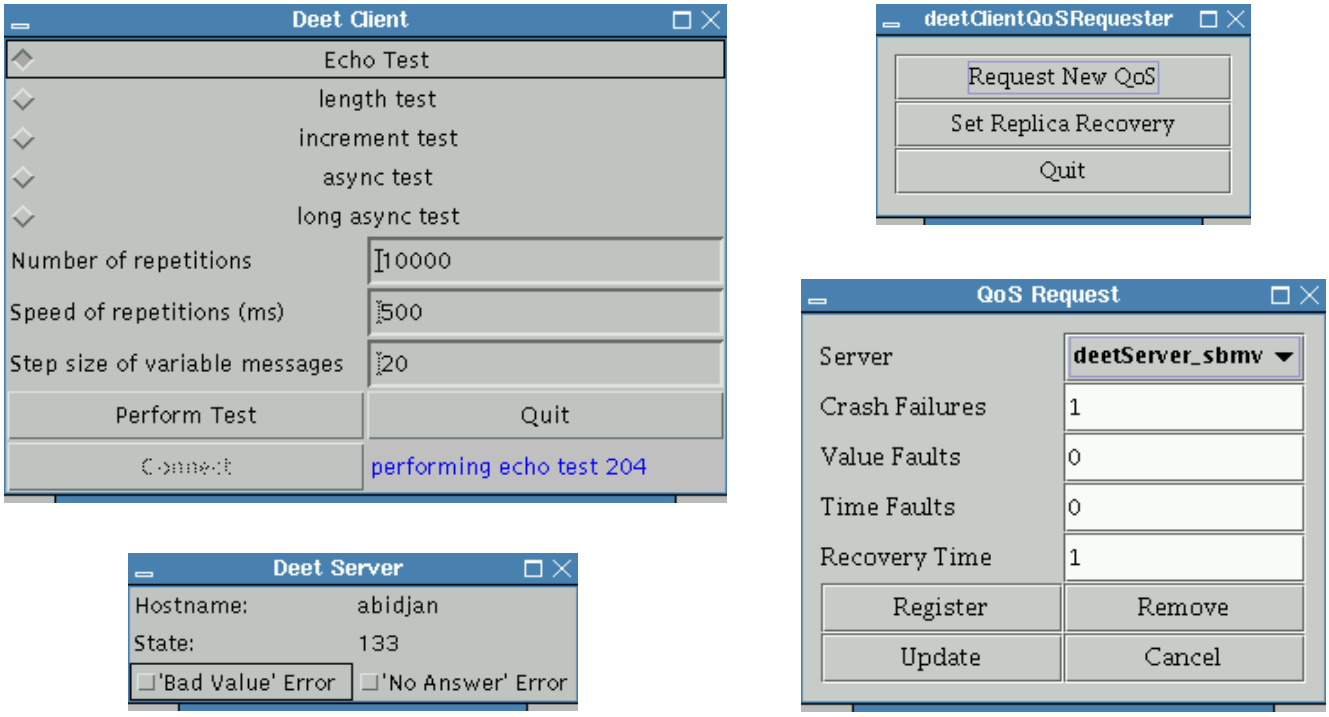

Figure 11. Deet Client, QoS Requester, and Deet Server Applications

Configuring and Running the Deet Application Manually

1. Notes:

   a. Before configuring and running the Deet demonstration, you need to configure the gateway for your system. See Section 6.2 for information on how to do this.

   b. If you do not wish to use multicast, in order to run a gateway-based program, you need to make sure that the Ensemble gossip server is running. The command to start it is: $AQUA_ROOT/bin/`gossip`.

2. Open three terminal windows, and cd to `$AQUA_ROOT/apps/deet`. Make sure the DISPLAY variable is set correctly if either terminal is logged into a remote host.

3. In the first window, start the first server (for example the one using the leader-only strategy):

   ```
   aquagw -f gw-lo-server.conf -PIS_FIRST
   ```

4. In the next window, start the second server:

   ```
   aquagw -f gw-lo-server.conf
   ```

5. In the next window, start the client:

   ```
   aquagw -f noqos-lo-client.conf -PIS_FIRST
   ```

6. After the client and servers are up, connect the client to the server using the “Connect” button on the client.

7. Now the client is ready to send commands to the servers. The client has a choice of five different tests, but only the first two are of interest to the user community:

    a. echo test: Send simple "ping" messages to the server.

    b. length test: Send a variable-length message to the server. The message length grows with each successive repetition.

    The client can also adjust the number of messages sent in the test, the delay between sendings of messages, and the amount by which the messages increase in size for the length test.

    For the first test, select the "Echo Test" with 10 messages. Leave the other two values as the defaults. Hit the "Perform Test" button. The client will send messages to the servers. The servers should both increment their state values from 0 to 10.

8. Repeat the above test, using 1000 echoes instead of 10. Now, terminate the first server using either <Ctrl-C> or kill. There should be a small pause; then the second server will be promoted to be the leader, and the handling of the echo messages will resume. Now, restart the server that was killed using the same command from Step 3. After it stabilizes it will have the same state as the existing server replica and should resume handling the echo requests.

Configuring and Running the Deet Application with the Dependability Manager (DM)

Using the DM allows you to test the intrusion-tolerant capabilities of the gateway by using the SBMV handler instead of the leader-only as above.

1. Before you begin:

    a. Configure the gateway for your system. See Section 6.2 for information on how to do this.

    b. If using gossip, start the gossip server now.

    c. Generate keys in a Keysets subdirectory. For each host on which you intend to run a replica (therefore wherever you will have a factory started) or the client, run:

        gen-key <host ip address>

2. Open five terminal windows: three on one machine, referred to as <hostA>, and one each on <hostB> and <host C>. If using Unix, make sure your DISPLAY environment variable is set correctly.

3. In the first window on <hostA>, start the dependability manager:

```
cd $AQUA_ROOT/proteus/java
aquagw -f gw-active-dm.conf -PIS_FIRST
```

4. In the second window on <hostA>, start the object factory for this host:

```
cd $AQUA_ROOT/proteus/java
aquagw -f factory1.conf -PIS_FIRST
```

5. In the window on <hostB>, start the object factory for this host:

```
cd $AQUA_ROOT/proteus/java
aquagw -f factory2.conf
```

6. In the window on <hostC>, start the object factory for this host:

   ```
   cd $AQUA_ROOT/proteus/java
   aquagw -f factory3.conf
   ```

7. In the fourth window on <hostA>, start the client:

   ```
   cd $AQUA_ROOT/apps/deet
   aquagw -f qos-sbmv-client.conf –PIS_FIRST
   ```

8. Start the deetServers using the "Request New QoS" button. Pushing this button brings up an interface that allows you to specify the type of deetServer and the level of QoS.

9. The Deet server can be of one of two types, either active or passive. With 3 factories, you will be able to create 3 servers and therefore tolerate 1 value fault or intrusion. You would normally select the type of deetServer by selecting one of these strings:

   a. "deetServer_pf": active replication with pass first policy

   b. "deetServer_lo": active replication with leader only policy

   c. "deetServer_sbmv": active replication with sbmv policy

   d. "deetServer_passive": passive replication

   For this test (intrusion-tolerant), select "deetServer_sbmv".

10. After the QoS confirmation dialog appears, use the client to start a test of 1000 pings (see the manual configuration instructions for details on how to do this). During this test, you can kill the Deet servers with a <Ctrl-C> in the server x-terminal window or with kill. Kill one of the servers. The dependability manager will acknowledge that the server has failed and start a new one, thus maintaining the level of QoS that was originally requested. Use the Handler Fault Injector GUI to inject handler-level faults or the Deet server GUI to inject application faults. After injecting faults, observe how the DM is notified of the faults and kills and restarts the faulty replica without the client noticing anything.

4.2.3 The Castle Demonstration

Overview

The "castle demonstration" (see `Figure 12`) consists of a single, undependable client (called the "game board" in the following) and multiple dependable remote objects, called "guards," that interact with the game board. There are currently two types of guards: active, which tolerate crash failures with active replication and the pass-first voting policy, and passive, which tolerate crash failures with passive replication. Three different guards may be active at any one time and are given different colors: black, red, and purple.

The game board represents a castle depicted in the center of the screen; its perimeter is defined by the square surrounding it and "raiders" (shown as dots outside the perimeter of the castle) move toward the castle in an attempt to reach it. The castle guards (shown as dots on the perimeter of the castle) are implemented as dependable remote gateway objects, and move to attempt to intercept raiders as they move toward the castle. To enable them to do this, the game board makes remote CORBA calls to each guard in succession, informing it of the current position of the raiders. The guard replies, giving its new position, and attempts to move toward the closest raider. If a guard intercepts a raider, the raider disappears, scoring a point for the guards. If a raider is not intercepted, it eventually reaches the castle, scoring a point for the raiders. Of course, the point of the application is not to build a better guard, but to illustrate how remote objects can be made dependable using the gateway architecture.

Selecting "Request new QoS" in the QoS Requester application (top right of `Figure 12`) brings up a QoS request window, which allows the user to specify the desired QoS for the guard. After the request is submitted to the dependability manager, a QoS observer window will appear in order to receive information about the request (middle right side of `Figure 12`). Each host on which a replication of a guard is running contains a window like the one on the lower right side of `Figure 12`. This window contains a progression bar that moves from left to right, indicating that the guard is processing requests from the board.

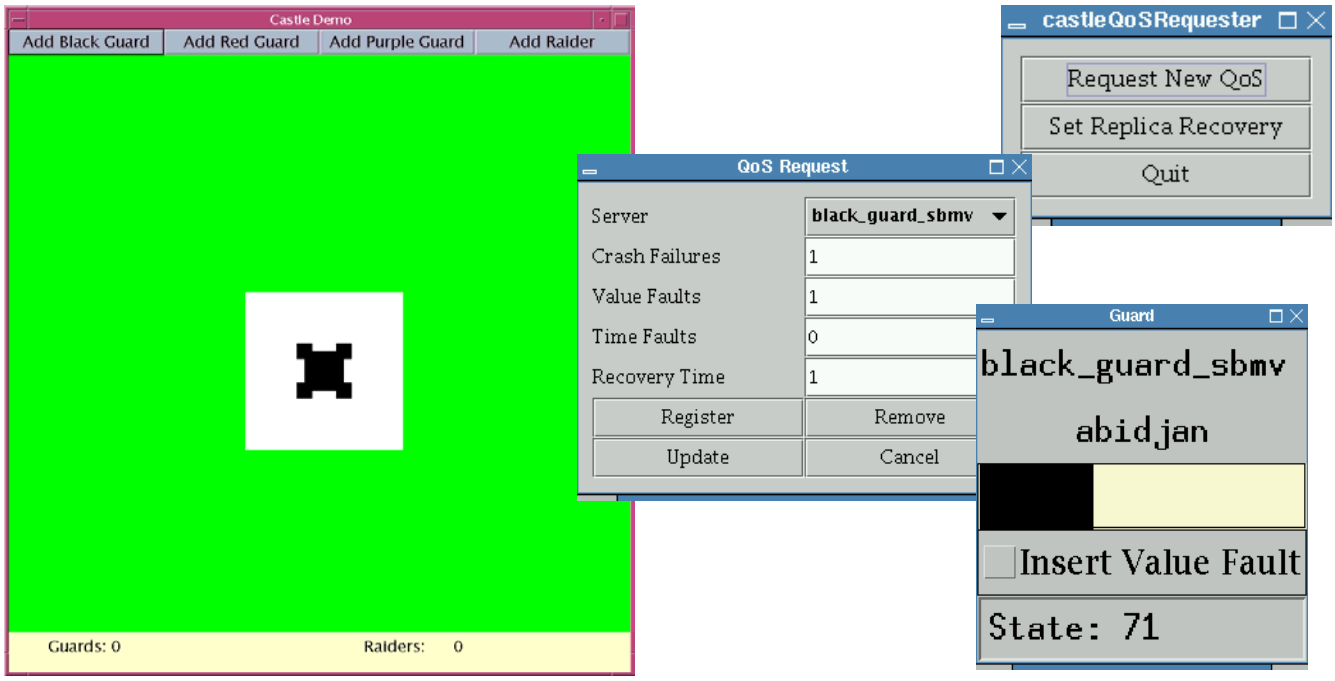

Figure 12. Castle Demonstration

Configuring and Running the Castle Demonstration

1. Before you begin:

   a. Configure the gateway for your system. See Section 6.2 for information on how to do this.

   b. In order to run a gateway-based program, you need to make sure that $AQUA_ROOT/bin/`gossip`, the Ensemble name server, is running.

2. Open four terminal windows: three on one machine, referred to as <hostA>, and the fourth on a second machine, referred to as <hostB>. If using Unix, make sure your DISPLAY environment variable is set correctly.

3. Start the Proteus dependability manager on <hostA>:

```
cd $AQUA_ROOT/proteus/java
aquagw -d -f gw_active-dm.conf -PIS_FIRST
```

4. Start the factory on <hostA>:

```
cd $AQUA_ROOT/proteus/java
aquagw -f factory1.conf -PIS_FIRST
```

5. Start the factory on <hostB>:

```
cd $AQUA_ROOT/proteus/java
aquagw -f factory2.conf
```

6. Start the game board on <hostA>:

```
cd $AQUA_ROOT/apps/castle-demo
aquagw -d -f board-lo-client.conf –PIS_FIRST
```

7. Wait until all programs started in the previous steps are active.

8. Start the demonstration.

    a. Add raiders to the board by pressing the “Add Raider” button several times.

    b. Start one or more dependable remote guards by using the “Request New QoS” button. When you push this button, the application will bring up an interface that allows you to specify the QoS for each guard you create. Allowable guards are:

        i. Active replication: “black_guard_<pf/lo/sbmv>”, “red_guard_<pf/lo/sbmv>”, “purple_guard_<pf/lo/sbmv>”

        ii. Passive replication: “black_guard_passive”, “red_guard_passive”, “purple_guard_passive”

    Since, in this example, we chose to start a leader-only client (by using the file board-lo-client.conf), make sure to start servers of the same type, with the suffix ‘_lo’.

    c. A popup dialog will appear in order to state that all replicated guards of a given color are started. Then hit the “Add Guard” button for the color. This will activate the guard in the board, and the guard will start moving.

    d. Watch the dependability manager action screen to see how the guard replicas are created. You can now proceed to crash various guards (or inject faults if using SBMV), and see how the system recovers.

4.2.4 Simulated Replication Controller

A simple replication controller has been developed to simulate the interaction between the BBN intrusion-tolerant manager and SBMV gateways.

The SBMV handler can use keys stored in files or be more secure by getting a certificate personally attributed at startup. We call the latter the “Dynamic key,” and it requires the existence of a manager (called here *replication controller* or *repcon*) to start up each gateway and share the new gateway key/certificate with all other replicas.

Here is how the exchange of dynamic keys works.

On the manager's side:

The manager starts a replica using the program “aquagw” and the command line options

-PMANAGER_IOR=<mgr_ior> and -PGATEWAY_NAME=<appli_name>.

The gateway name is usually the application name used by the application (-a option of the handler factory in the application configuration file). The manager keeps a handle to the replica's standard input stream and sends a session key through it. Shortly after, the newly started replica is expected to invoke the CORBA method replicaKey() on the manager in order to provide a public key encrypted using the session key described above for authentication.

Inside this method replicaKey():

- the manager authenticates the public key, then
- casts the message replicaCert() to the other managers in the same multicast groups, then
- invokes set_parameter() on the replica's gateway in order to provide the new certificate plus as many calls to set_parameter() as there were certificates received in the multicast group from other managers before the replica called replicaKey(),
- and finally the manager returns success.

Whenever a certificate is received from the multicast group (from other managers), either store the certificate in a cache if the replica is not up yet or invoke set_parameter() on the replica passing the certificate along.

The certificate name, used in set_parameter() should be:
<replica_IP_address>_<appli_name>_cert

The methods supported by the manager as defined in the repcon idl are:

```
 /* gw_name: the gateway name passed to the replica at startup
  * real_key_size: the size of 'certificate' before it was zero
  *   padded at the end to be encrypted
  * key: the replica's public key certificate encrypted with DES
  *   using the manager's session key
  */
 long replicaKey (in string gw_name, // matching the appli name
        in long real_key_size,
        in aqua::OctetSeq encrypted_certificate);

 /* The method used by managers to exchange the certificate attributed
  * to their replicas.
  */
 oneway void replicaCert (in string replica_name, // appli_name
           in string cert);
```

On the replica's side:

The SBMV Handler Factory takes 3 options:
-r: the receive majority size
-s: the send majority size
-f: to use keys stored in file

When the handler is created, the flag set by the option above is passed along.

If key files are used, the key used by this handler is supposed to have been previously generated using the program gen-key, and the key is supposed to be stored in <current directory>/Keysets/<local_ip_address>.p15.

Therefore, all gateway applications running on the same host (clients and servers) will use this same key.

If the option -f is not set, upon creation, the SBMV handler checks if the 2 conditions to use dynamic keys are set:

1. reference to a replication manager located in the Gateway Parameter Servant under the parameter name "MANAGER_IOR".
2. a gateway name under the parameter name "GATEWAY_NAME".

If these 2 conditions are not met, the handler defaults to using key files; otherwise, the handler enters an infinite wait on the parameter "MANAGER_SESSION_KEY", checking for it every 2 seconds. This session key is supposed to come via stdin from the manager that started this object.

Once the session key is obtained, the parameter "USE_CRYPTO" is set to 1 so other gateway components are aware that some steps will require the existence of security keys.

A new cryptographic object (class CryptObj) is created. During this creation, a child process is spawned to generate a public/private key pair.

The public key is encrypted using the manager's session key and sent via a CORBA call "replicaKey()" to the manager waiting for a reply.

If the invocation comes back successful (return value null), the handler continues its initialization and eventually attempts to join the replication group. At this point, the attempt is blocked until a parameter named <local_ip_address>_<local_object_name>_cert is found in the GPS.

This parameter is the public key authenticated by the manager base on the certificate sent earlier. This certificate is the one the handler is supposed to use from now on for every message it sends out.

Note: the manager is supposed to share the authenticated certificate with all other managers (sent thru multicast group, method "replicaCert()"), each of which forwards or caches the info to/for its own replica. This last step is made by invoking set_parameter() on the replica's gateway.

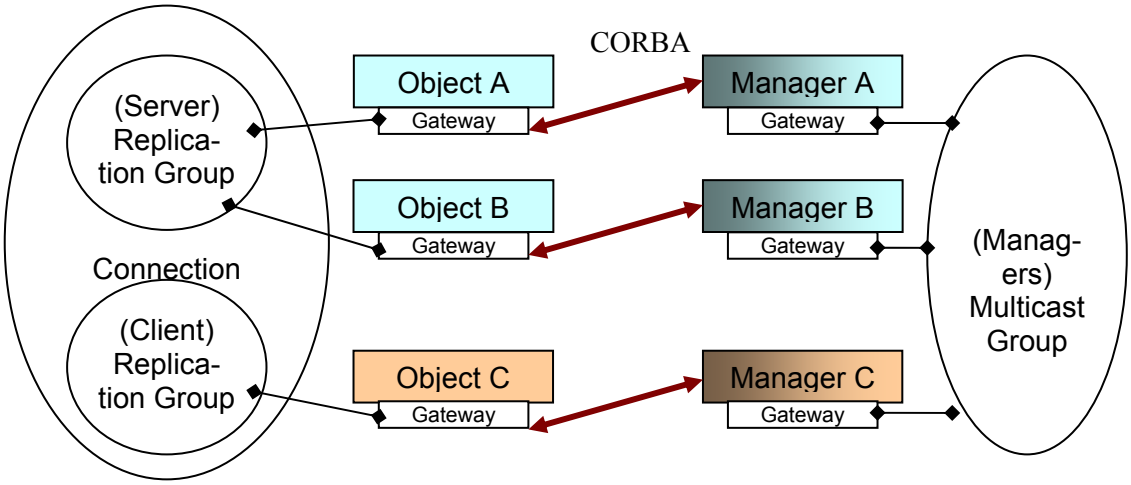

Figure 13. Interface Between Managers and SBMV Gateways

The code for this simulated managers is located under $AQUA_ROOT/handlers/tests/active/itua_repcon. A README file in this directory explains how to start up the managers.

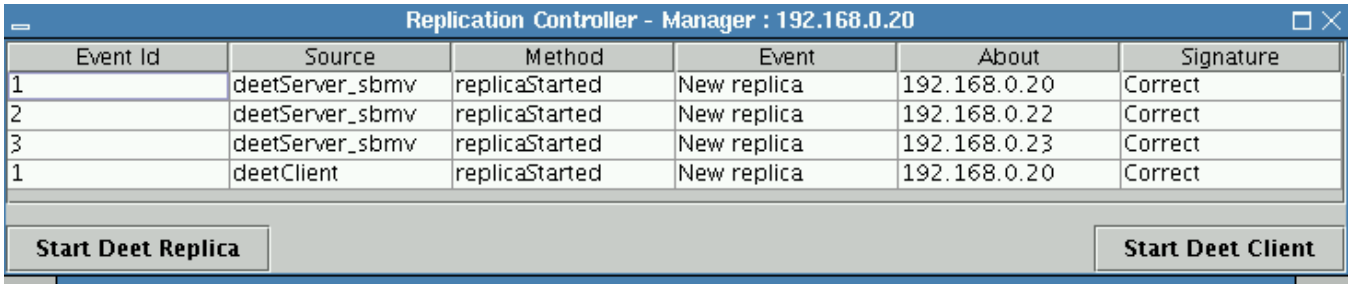

Figure 14. Simulated Manager GUI

5 Graphical User Interfaces

The dependability manager also provides several graphical interfaces to allow monitoring of the status of applications built using the gateway architecture.

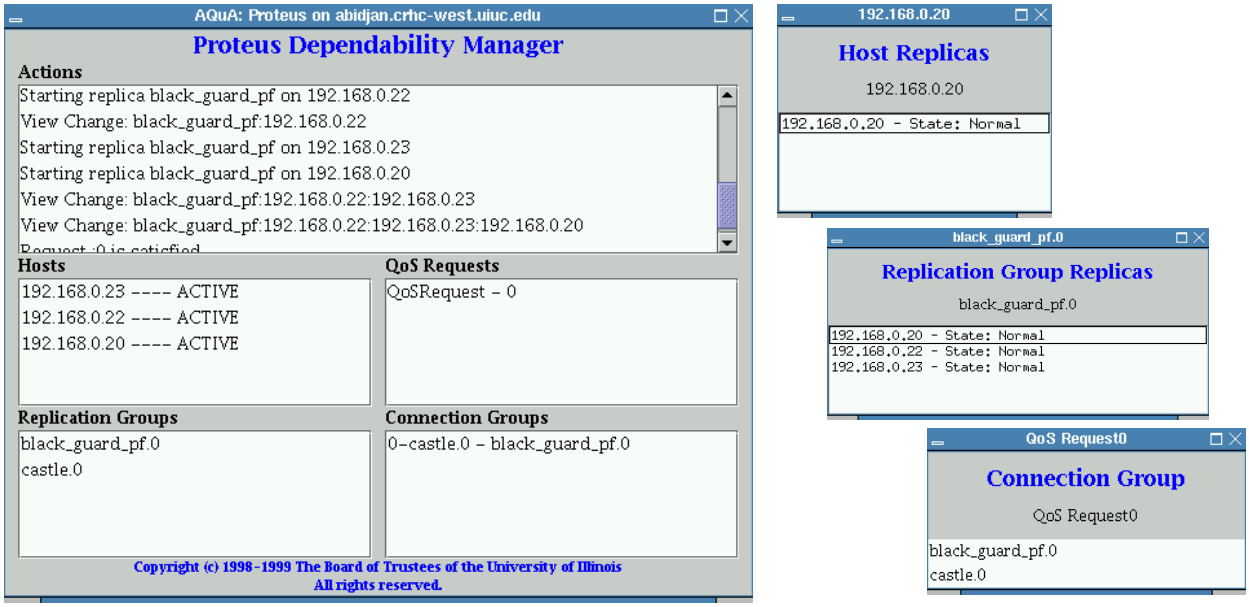

Figure 15. Dependability Manager Graphical Interface

The actions taken by the dependability manager to create replicas can be seen on the dependability manager graphical interface, shown on the left side of `Figure 15`. This interface displays a textual record of the actions that the dependability manager takes, and provides facilities for monitoring the status of hosts and replicas. In particular, the set of hosts on which object factories are running, the list of QoS requests, the replication groups, and the connection groups can be monitored. To do so, a user opens a window corresponding to the host, replication group, or connection group he/she is interested in.

Examples of these windows are shown on the right side of `Figure 15`. When a user double-clicks on one of the hosts listed in the window shown on the left side of `Figure 15`, a window like the top right one appears. This window indicates, for the corresponding host, the list of objects running on that host and their states (normal, start pending, kill pending, join pending, remove pending, or dead). When the user double-clicks on one of the replication groups listed in the window shown on the left side of `Figure 15`, a window like the center right one appears. This window shows the list of objects included in the corresponding replication group, and their states. Double-clicking on one of the connection groups listed in the window shown on the left side of `Figure 15` causes a window like the lower right one to appear. This window indicates, for each connection group, the replication groups included in the connection group and their types (sender or receiver). These windows are updated whenever a change in requested QoS occurs (via a `registerQoSRequest()` from a QoS observer/requester) or the dependability manager is notified that a fault has occurred (via a `viewChange()` method call from a gateway).

Finally, `Figure 16` shows an object factory. Each object factory window lists the object replicas running on the given host and indicates the host load.

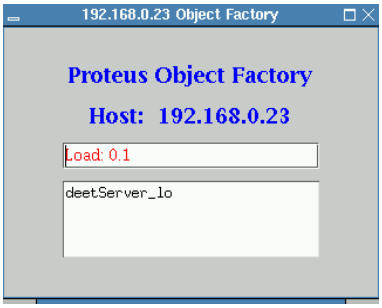

Figure 16. Proteus Object Factory

6 Installing the gateway and using gateway utilities

6.1 System Requirements

The gateway environment runs on Windows, Solaris 2.7 for Sparc processors and Red Hat Linux 7.x for x86 processors. A typical configuration would include multiple machines connected via an Ethernet. The required configuration for this version of the gateway is:

- Solaris 2.7 for Sun Sparc, RedHat Linux 7.x, Windows 2000
- ACE - TAO 5.3.1
- Ensemble version 1.42
- JACORB 1.4
- Java version 1.4
- gcc version 2.96

6.2 Gateway Installation

To configure the gateway for your system, follow the steps below for each operating system:

1) Download and install ACE and TAO (version 1.3.1), defining the variable ACE_ENABLE_SWAP_ON_WRITE in your ACE_wrappers/ace/config.h

2) Untar the gateway release files:

   a) Choose a place to install the gateway for the given OS, perhaps `/usr/local/ITUA-AQuA-solaris` or `/opt/ITUA-AQuA-linux`.

   b) `cd` into this new directory and untar the three release files using the command "`tar -zxvf <file.tgz>`" or a Zip tool such as `winzip` on Windows. You should then see one directory: `ITUA-AQuA`.

3) Follow the remaining instructions contained in the README file inside the ITUA-AQuA directory.

6.3 Gateway Utilities

There are several utilities packaged with the gateway that aid in the development of gateway applications.

6.3.1 Setup Environment Tool

This tool, which is shown in `Figure 17`, is located at `$AQUA_ROOT/bin/aqua-setup-env`. It should be run immediately after the ITUA-AQuA tar file and associated components are installed. This tool prompts the user for various paths and parameters necessary for running the gateway, and then creates an environment file for

the desired shell. Users can then source this file to set up their environments to run the gateway. Additional details about running this editor can be found in `$AQUA_ROOT/README`

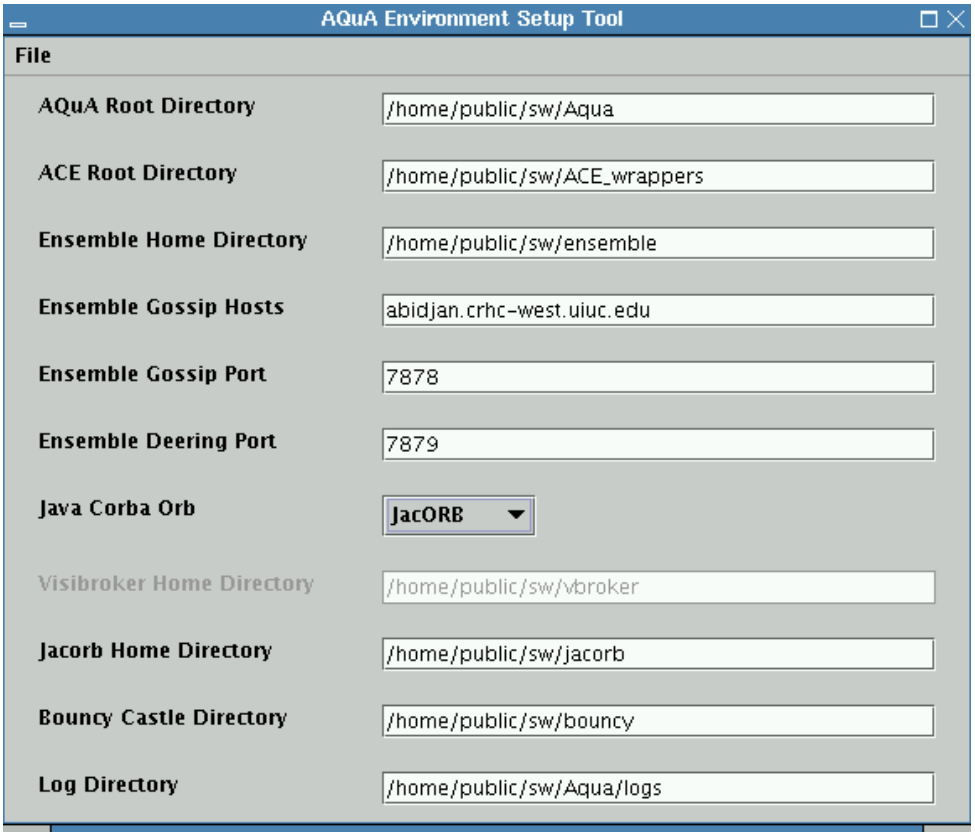

Figure 17. Gateway Environment Setup Tool

6.3.2 Logger Server

The *logger server* is used to capture specific debugging information during ITUA-AQuA gateway application execution. Applications can connect to the logger server and send output messages. If the server is present, it receives the messages and outputs them to a terminal window. This is useful during application debugging. Because multiple applications can send logging output to the same logger server, the logger server is very helpful in resolving inter-process timing issues.

To output data to the logger server, the user should start the logger server, using this command:

```
server_loggerd
```

The application then connects to the server like this, where `logger_host` is the machine on which `server_loggerd` is running:

```
AQUA_Logging_Client *logger = new AQUA_Logging_Client (logger_host);
```

The application sends log messages using a command like this:

```
logger->send_log("This is a log message.");
```

6.3.3 Factory Launcher Script

The "factory_launcher" script can be found in $AQUA_ROOT/bin. It is called automatically by the object factories to start the replicated processes. By default it displays the console output of the started processes in an x-terminal. Sometimes it is helpful to archive the application trace information to a file for later analysis. The factory_launcher script can be modified to send the output to a file instead of the x-terminal. To do so, edit the factory_launcher script and find these three lines:

```
# override the command line and force file output
#XTERM="";
#FILE="1";
```

Remove the leading "#" from the second and third lines to force the output to a file. The name of the file will be "/tmp/<application name>.<machine name>.out.<num>", where <num> is an integer that is incremented to guarantee unique filenames.

6.3.4 Replica Starter and Test Scripts

The replica starter is a program used to send QoS requests to the DMs using a dynamic send handler and start requests to the simulated replication controllers (called "repcon") via a multicast handler. The purpose of this java application is to automate the startup of replicas.
The replica starter is implemented as an aqua application and takes the following options:

- -p[object_name],[crash_faults],[value_faults] for QoS requests
- -r[replica_name],[replica_conf_file],[client_name],[client_conf] for starting replica via RepCon.
  Note: when using dynamic keys, the client must be started along with replicas so the replicas can also get the client's key at startup.

Other parameters can be passed outside the config file via the gw command-line option -P<PARAMETER_NAME=PARAMETER_VALUE>.

The following parameter names are supported:

- to specify 1 QoS request for the DM:
  QOS_CRASH_FAULTS => number of crashes to tolerate
  QOS_VALUE_FAULTS => number of value faults to tolerate
  QOS_REPLICA_NAME => object name

- to specify 1 start request for the RepCons:
  RC_REPLICA_NAME => object name
  RC_REPLICA_CONF => config file for replica
  RC_CLIENT_NAME => client object name
  RC_CLIENT_CONF => config file for client

  Note: only the leader of the RepCon multicast group will start the client. The others will ignore the client info parameters.

  RC_MIN_NUMBER => minimum number of repcons to be present before sending the start request.

  Up to 2 clients can be specified via gw parameters variables
  RC_CLIENT_NAME/RC_CLIENT_NAME2 and RC_CLIENT_CONF/RC_CLIENT_CONF2

The code for the replica started is in $AQUA_ROOT/tests/java and several test scripts are provided in the directory $AQUA_ROOT/tests/scripts. The list of hosts to be used is in the file hosts.prp and must be edited and adapted to one's configuration prior to use of any of the scripts.

7 To Learn More About the ITUA and AQuA Projects

Readers interested in more details on the ITUA and AQuA projects can read the papers that have been written about them and see the web page given below.

7.1 References

[Bir96] K. P. Birman, *Building Secure and Reliable Network Applications*. Greenwich, CT: Manning Publications, 1996.

[Cou02] T. Courtney, J. Lyons, H. V. Ramasamy, W. H. Sanders, M. Seri, M. Atighetchi, P. Rubel, C. Jones, F. Webber, P. Pal, R. Watro, M. Cukier, and J. Gossett, "Providing Intrusion Tolerance with ITUA." *Supplemental Volume of the 2002 International Conference on Dependable Systems & Networks (DSN-2002)*, Washington, DC, June 23-26, 2002, pp. C-5-1 to C-5-3.

[Cuk01] M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett, "Intrusion Tolerance Approaches in ITUA." FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, Göteborg, Sweden, July 1-4, 2001, pp. B-64 to B-65.

[Cuk98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, "AQuA: An Adaptive Architecture That Provides Dependable Distributed Objects." *Proc. 17th IEEE Symposium on Reliable Distributed Systems (SRDS-98)*, West Lafayette, IN, USA, pp. 245-253, October 1998.

[Cuk99] M. Cukier, J. Ren, P. Rubel, D. E. Bakken, and D. A. Karr, "Building Dependable Distributed Objects with the AQuA Architecture," in *Digest of FastAbstracts presented at the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, WI, USA, pp. 17-18, June 1999.

[Gup03a] V. Gupta, *Intrusion-Tolerant State Transfer for Group Communication Systems.* Master's Thesis, University of Illinois, 2003.

[Gup03b] V. Gupta, V. Lam, H. V. Ramasamy, W. H. Sanders, and S. Singh, "Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures." *Dependable Computing: Proceedings of the First Latin-American Symposium (LADC 2003)*, São Paulo, Brazil, October 21-24, 2003, *Lecture Notes in Computer Science* vol. 2847 (Rogério de Lemos, Taisy Silva Weber, and João Batista Camargo Jr., eds), Berlin: Springer, 2003, pp. 81-101.

[Gup03c] V. Gupta, V. Lam, H. V. Ramasamy, W. H. Sanders, and S. Singh, "Stochastic Modeling of Intrusion-Tolerant Server Architectures for Dependability and Performance Evaluation." University of Illinois at Urbana-Champaign Coordinated Science Laboratory technical report UILU-ENG-03-2227 (CRHC-03-13), December 2003.

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

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

[Lyo03] J. P. Lyons, *A Replication Protocol for an Intrusion-Tolerant System Design*, Master's Thesis, University of Illinois, 2003.

[Pal01] P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, W. Sanders, M. Cukier, and J. Gossett, "Survival by Defense-Enabling." *Proceedings of the New Security Paradigms Workshop 2001*, Cloudcroft, New Mexico, September 11-13, 2001, pp. 71-78.

[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." *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 23-26, 2002, pp. 229-238.

[Ren01a] J. Ren, M. Cukier, and W. H. Sanders, "An Adaptive Algorithm for Tolerating Value Faults and Crash Failures." *IEEE Transactions on Parallel and Distributed Systems*, Special Issue on Dependable Network Computing, vol. 12, no. 2, February 2001, pp. 173-192.

[Ren01b] Y. Ren, *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2001.

[Ren99] J. Ren, M. Cukier, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Building Dependable Distributed Applications Using AQuA." *Proc. 4th IEEE Symposium on High Assurance Systems Engineering (HASE'99)*, pp. 189-196, Washington D.C., USA, November 1999.

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

[Sab98] B. S. Sabnis, *Proteus: A Software Infrastructure Providing Dependability for CORBA Applications*, Master's Thesis, University of Illinois, 1998.

[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." *Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-7)*, pp. 137-156, San Jose, CA, USA, January 1999.

[San02] W. H. Sanders, M. Cukier, F. Webber, P. Pal, and R. Watro, "Probabilistic Validation of Intrusion Tolerance." Fast Abstract in the *Supplemental Volume of the 2002 International Conference on Dependable Systems & Networks (DSN-2002)*, Washington, DC, June 23-26, 2002, pp. B-78 to B-79.

[Ser01] M. Seri, T. Courtney, M. Cukier, V. Gupta, S. Krishnamurthy, J. Lyons, H. Ramasamy, J. Ren, and W. H. Sanders, "A Configurable CORBA Gateway for Providing Adaptable System Properties." *Supplemental Volume of the 2002 International Conference on Dependable Systems & Networks (DSN-2002)*, Washington, DC, June 23-26, 2002, pp. G-26 to G-30.

[Vay97] A. Vaysburd and K. P. Birman, "Building Reliable Adaptive Distributed Objects with the Maestro Tools." *Proc. of the Workshop on Dependable Distributed Object Systems, OOPSLA*, pp. 213-215, Atlanta, GA, USA, October 1997.

[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.

7.2 Web Page

The web page addresses for the ITUA and AQuA projects are <http://www.crhc.uiuc.edu/PERFORM> (select the "ITUA" or "AQUA" projects from the menu to the left) and <http://itua.bbn.com/>.

8 Appendix A

This appendix provides a description of all calls that may be made between a QoSRequestor and a dependability manager. Prototypes of the calls are given in bold, and a description of the required parameters and their functions follows.

`long registerQoSRequest(in QoSRequestInfo QoSRequest_info)`

Return value

On success, the return value is the QoS request ID that should be used whenever the caller wants to refer again to this request; on failure, -1 is returned.

Parameters

QoSRequest_info

Information concerning the QoS request using the QoSRequestInfo structure.

Function

The `registerQoSRequest` method is called by a QoS requestor/observer to register a new QoS request (one that does not replace or supercede any pre-existing request) with the dependability manager. Its argument includes information about the QoS requested as well as a reference to a QoS observer that will receive information about the request.

Data structure

```
struct  QoSRequestInfo {
       QoSObserver observerReference;    // object to be notified when
                                         // events relevant to this QoS request occur
       string clientName                 // name of the object that issues the QoS request
       string objectName;                // name of replicated object to manage
       short crashFailuresToTolerate;    // the number of crash failures to tolerate
       short valueFaultsToTolerate;      // the number of value faults to tolerate
       short timeFaultsToTolerate;       // the number of time faults to tolerate
       long recoveryTime;                // the time for the Dependability Manager to attempt
                                         // to achieve the QoS request. (Unit: minutes)
   };
```

`crashFailuresToTolerate`, `valueFaultsToTolerate`, and `timeFaultsToTolerate` can be specified to be zero. If the number of faults to tolerate is set to 0 for a type, faults of that type will not be tolerated by Proteus. Otherwise, Proteus will tolerate faults of that type. For example, if only crash failures need to be tolerated, the `crashFailuresToTolerate` must be set to be greater than 0, and both the `valueFaultsToTolerate` and the `timeFaultsToTolerate` are set to 0. In the current release, crash failures and value faults can be tolerated.

short updateQoSRequest(in long QoSRequestID, in QoSRequestInfo QoSRequest_info)

Return value

Return value 0 on success, -1 on failure.

Parameters

QoSRequestID

ID number that was assigned when the QoS request was registered.

QoSRequest_info

Information concerning the QoS request using the QoSRequestInfo structure.

Function

The `updateQoSRequest` method substitutes a new QoS request for one that was previously registered.

Data structure

QoSRequestInfo contains the information concerning the QoS request by the application.

```
struct  QoSRequestInfo {
       QoSObserver observerReference;    // object to be notified when
                                         // events relevant to this QoS request occur
       string objectName;                // name of replicated object to manage
       short crashFailuresToTolerate;    // the number of crash failures to tolerate
       short valueFaultsToTolerate;      // the number of value faults to tolerate
       short timeFaultsToTolerate;       // the number of time faults to tolerate
       long recoveryTime;                // the time for the Dependability Manager to attempt
                                         // to achieve the QoS request. (Unit: minutes)
   };
```

`crashFailuresToTolerate`, `valueFaultsToTolerate`, and `timeFaultsToTolerate` can be specified to be zero. If the number of faults to tolerate is set to 0 for a type, faults of that type will not be tolerated by Proteus. Otherwise, Proteus will tolerate faults of that type. For example, if only crash failures need to be tolerated, the `crashFailuresToTolerate` must be set to be greater than 0, and both the `valueFaultsToTolerate` and the `timeFaultsToTolerate` are set to 0. In the current release, crash failures and value faults can be tolerated.

```
short removeQoSRequest(in long QoSRequestID)
```

Return value

Return value 0 on success, -1 on failure.

Parameters

QoSRequestID

ID number that was assigned when the QoS request was registered.

Function

The `removeQoSRequest` method eliminates a previously registered QoS request without replacing it with a new request. This method is used only when an application no longer needs a dependable remote object.

```
oneway void QoSRequestNotSatisfied(in long QoSRequestID, in long curren-
tReplicas, in long currentActiveHosts)
```

Return value

void

Parameters

QoSRequestID

ID number that was assigned when the QoS request was registered.

currentReplicas

Current number of object replicas in the replication group.

currentActiveHosts

Current number of hosts in the active host set (i.e., active hosts).

Function

The callback `QoSRequestNotSatisfied` is called:

1) When a new QoS request requests that too many faults be tolerated.

2) When a QoS request that was satisfied by a former number of active hosts is no longer satisfied by the current number of active hosts.

3) When a QoS request was satisfied but is no longer satisfied due to a change in the number of replicas. In this case, the callback occurs after the dependability manager has attempted to obtain the QoS request for a period of time defined by the recovery time.

```
oneway void QoSRequestSatisfied(in long QoSRequestID, in long currentRepli-
cas, in long currentActiveHosts)
```

Return value

void

Parameters

QoSRequestID

ID number that was assigned when the QoS request was registered.

currentReplicas

Current number of object replicas in the replication group.

currentActiveHosts

Current number of hosts in the active host set (i.e., active hosts).

Function

The callback `QoSRequestSatisfied` is called when a QoS request that could not be satisfied (indicated by the `QoSRequestNotSatisfied` call) can once again be satisfied.

```
oneway void noMajority(in long QoSRequestID, in long currentReplicas, in
long currentActiveHosts)
```

Return value

void

Parameters

QoSRequestID

ID number that was assigned when the QoS request was registered.

currentReplicas

Current number of object replicas in the replication group.

currentActiveHosts

Current number of hosts in the active host set (i.e., active hosts).

Function

The `NoMajority` method is called when the dependability manager receives the noMajority message from the leader of the replication group when tolerating value faults.

9 Appendix B

This appendix provides a description of all calls that may be made between a host observer/controller or advisor observer and a dependability manager. Prototypes of the calls are given in bold, and a description of the parameters of each call and its function follow.

`short deactivateHost(in string hostName)`

Return value

Return value 0 on success, -1 on failure.

Parameters

hostName

Name of the host that should be deactivated.

Function

This method is used to request that the dependability manager deactivate a host. The dependability manager will move the host from the active host set to the inactive host set, and will also migrate the replicas from this inactive host to hosts in the active host set.

```
short activateHost(in string hostName)
```

Return value

Return value 0 on success, -1 on failure.

Parameters

hostName

Name of the host that should be activated.

Function

This method is used to request that the dependability manager move an inactive host from the inactive host set back to the active host set. Upon receiving this call, the dependability manager will check the QoS request table to determine if enough replicas have been created for each QoS request. For each object whose QoS was not satisfied, the dependability manager will create one replica on the activated host.

`long registerAdvisorObserver(in string advisorObserverName, in AdvisorObserverFunction advisorObserverfunction)`

Return value

Return value on success is the observer ID that can be used to remove this advisor observer; the return value is -1 on failure.

Parameters

advisorObserverName

Name of the advisor observer that will receive information about the QoS request.

advisorObserverfunction

Specification of the type of information to provide to the observer. See data structure below.

Function

This method is used to register an advisor observer with the dependability manager. The dependability manager supports multiple advisor observers. All advisor observers receive information regarding faults that occur. Other information is provided if requested.

Data structure

```
struct AdvisorObserverFunction {
    sequence<short> method_active;
    long hostInfo_rate; // host load update rate (Unit: minutes)
};
```

Each item in the method_active sequence is a number identifying a callback method that should be called when the corresponding event occurs. The numerical codes of the methods are:

- hostActivated = 1
- hostRemoved = 2
- hostDeactivated =3
- notifyNumReplicas = 4
- replicaStartAttempted = 5
- replicaKillAttempted = 6
- replicaStartFailed = 7
- replicaKillFailed = 8
- replicaStartSuccessful = 9
- replicaKillSuccessful = 10
- hostInfo = 11

```
short removeAdvisorObserver(in long observerID)
```

Return value

Return value 0 on success, -1 on failure.

Parameters

observerID

ID of the advisor observer that should be removed.

Function

This method deregisters an advisor observer from the dependability manager.

```
oneway void faultOccurred(in FaultInfo fault_info)
```

Return value

void

Parameters

fault_info

Structure containing information about a fault that has occurred.

Function

The `faultOccurred` method is called by the dependability manager on all advisor observers when it detects a fault.

Data structure

The fault_info data structure provides each advisor observer with information concerning each fault that occurs.

```
typedef short FaultType;

const FaultType REPLICA_CRASH = 0;

const FaultType VALUE_FAULT = 1;

const FaultType TIME_FAULT = 2;

const FaultType HOST_CRASH = 3;

const FaultType NO_MAJORITY = 4;


struct FaultInfo {
      FaultType typeOfFault;
      string  hostName;
      string  objectName;
      string  objectID;
    };
```

```
oneway void hostActivated(in string hostName)
```

Return value

void

Parameters

hostName

Name of the host that was activated.

Function

The method `hostActivated` is called by the dependability manager on all subscribing advisor observers:

1. When a host that is either in the removed host set or in no host sets registers with a dependability manager;
2. When a host that is in the inactive host set is reactivated.

```
oneway void hostRemoved(in string hostName)
```

Return value

void

Parameters

hostName

Name of the host that was removed.

Function

The method `hostRemoved` is called by the dependability manager on all subscribing advisor observers when a host or object factory failure is detected by the dependability manager. This method will be implemented when the dependability manager is able to detect host and object factory failures.

```
oneway void hostDeactivated(in string hostName)
```

Return value

void

Parameters

hostName

Name of the host that was deactivated.

Function

The method `hostDeactivated` is called by the dependability manager on all subscribing advisor observers when a host is deactivated.

```
oneway void notifyNumReplicas(in string objectName, in string objectID, in
properties objectProps, in long currentReplicas)
```

Return value

void

Parameters

objectName

Name of a replicated object.

objectID

ID of the replication group for the replicated object.

objectProps

Structure specifying properties of the replicated object.

currentReplicas

Number of object replicas in the replication group for the replicated object.

Function

The method `notifyNumReplicas` is called by the dependability manager on all subscribing advisor observers:

1. When a new QoS request is registered.
2. Whenever the number of replicas of *objectName* changes.

objectID is the ID number that was assigned by the dependability manager to the replicated object. All object replicas in the same replication group have the same *objectID*.

Data structure

```
struct TaggedItem {
     string tag;
     string value;
   };

typedef sequence<TaggedItem> properties;
```

No object properties are currently supported.

```
oneway void replicaStartAttempted(in string objectName, in string objectID,
in string hostName, in properties objectProps, in properties hostInfo)
```

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager attempted to start.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the dependability manager attempted to start the object replica.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the attempt occurred.

Function

The method `replicaStartAttempted` is called by the dependability manager on all subscribing advisor observers when the dependability manager attempts to start a replica.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };
```

typedef sequence<TaggedItem> properties;

Currently, one host property type is supported. It has tag "load," and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

```
oneway void replicaKillAttempted (in string objectName, in string objectID,
in string hostName, in properties objectProps, in properties hostInfo)
```

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager attempted to kill.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the dependability manager attempted to kill the object replica.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the attempt occurred.

Function

The method `replicaKillAttempted` is called by the dependability manager on all subscribing advisor observers when the dependability manager attempts to kill a replica.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };
```

typedef sequence<TaggedItem> properties;

Currently, one host property type is supported. It has tag “load,” and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

`oneway void replicaStartFailed (in string objectName, in string objectID, in string hostName, in properties objectProps, in properties hostInfo)`

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager failed to start.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the dependability manager failed to start the object replica.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the failure occurred.

Function

The method `replicaStartFailed` is called by the dependability manager on all subscribing observers when a new replica either could not be started by an object factory or could be started but could not join the replication group.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };

typedef sequence<TaggedItem> properties;
```

Currently, one host property type is supported. It has tag "load," and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

`oneway void replicaKillFailed (in string objectName, in string objectID, in string hostName, in properties objectProps, in properties hostInfo)`

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager failed to kill.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the dependability manager failed to kill the object replica.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the failure occurred.

Function

The method `replicaKillFailed` is called by the dependability manager on all subscribing observers when a replica could not be killed by an object factory and the kill failure was reported to the dependability manager.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };
```

typedef sequence<TaggedItem> properties;

Currently, one host property type is supported. It has tag “load,” and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

```
oneway void replicaStartSuccessful (in string objectName, in string objec-
tID, in string hostName, in properties objectProps, in properties hostInfo)
```

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager started successfully.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the object replica was started successfully.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the success occurred.

Function

The method `replicaStartSuccessful` is called by the dependability manager on all subscribing observers when a replica was started successfully by an object factory, joined the replication group, and was reported to the dependability manager.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };

typedef sequence<TaggedItem> properties;
```

Currently, one host property type is supported. It has tag "load," and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

```
oneway void replicaKillSuccessful (in string objectName, in string objectID,
in string hostName, in properties objectProps, in properties hostInfo)
```

Return value

void

Parameters

objectName

Name of an object replica that the dependability manager successfully killed.

objectID

ID of the replication group of which the object replica should be a member.

hostName

Name of the host on which the object replica was successfully killed.

objectProps

Structure specifying properties of the replicated object.

hostInfo

List of information concerning the host on which the success occurred.

Function

The method `replicaKillSuccessful` is called by the dependability manager on all subscribing observers when a replica was killed successfully by an object factory and removed from the replication group, and the removal was reported to a dependability manager.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };

typedef sequence<TaggedItem> properties;
```

Currently, one host property type is supported. It has tag “load,” and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.

No object properties are currently supported.

```
oneway void hostInfo(in string hostName, in properties hostInfo)
```

Return value

void

Parameters

hostname

Name of the host whose information the dependability manager provides to an advisor observer.

hostInfo

List of information concerning the host.

Function

This function is called by the dependability manager on all subscribing advisor observers to provide them with host information for all active and inactive hosts at the rate specified in the `registerAdvisorObserver` call.

Data structure

```
struct TaggedItem {
      string tag;
      string value;
    };

typedef sequence<TaggedItem> properties;
```

Currently, one host property type is supported. It has tag "load," and its value is equal to the second load value reported by the uptime command. This value is the 5-minute load average of the host.