

AQUARIUS

Implementing Fault Tolerant CORBA using Total Ordering

Barak Merimovich, 033147067, barakm@cs.huji.ac.il
David Rabinowitz, 033511080, dar@cs.huji.ac.il

Table of Contents

1.	Overview.....	3
2.	Operational flow.....	3
3.	Differences from the original paper	4
3.1.	Location of ordering mechanism.....	4
3.2.	Unique IDs	4
3.3.	Leader.....	5
3.4.	Notifications	5
3.5.	Garbage collection	5
4.	ORBacus modification.....	5
4.1.	The dispatch strategy	5
4.2.	Portable Interceptors	5
5.	The dissemination dispatch strategy.....	6
6.	Implementing the Total Order interface	7
7.	Technical difficulties	8
7.1.	Obfuscated code	8
7.2.	ORBacus memory leaks.....	8
7.3.	Dispatch strategies, threads and everything in the middle.....	8
7.4.	Thread pool failures	9
7.5.	The big one.....	9
8.	Appendix A – Compiling the Aquarius project	10
9.	Appendix B – Running the Aquarius project.....	10
9.1.	Support services.....	11
9.2.	Application servers	11
9.3.	Aquarius Proxy	11
9.4.	Application client.....	11
10.	Appendix C – The IDLs.....	12
10.1.	Total-order.idl.....	12
10.2.	proxy.idl	13

1. Overview

The Aquarius project aims to implement the FT-CORBA specification, using a data-centric approach. By replicating servers and allowing a client to access an object group, instead of a single object, applications can gain fault tolerance, in both the normal and Byzantine model. Clients run their operations on a standard CORBA reference, and are essentially unaware of the replication and ordering operations being done on their behalf.

2. Operational flow

A client connects to an Aquarius proxy, and builds a new object group. This is done via an administrative interface exposed by the proxy. This is the only place where the client is 'aware' of the group. Here the client chooses the application servers that will be members of his group. A new group proxy object is then created on the proxy. This object instantiates the actual implementations on the servers specified by the client. From this point on the proxy object reference is used as if it was a normal servant for the requested type.

For Example:

```
// get a reference to a proxy somehow. Can be a file,
// a Naming service, or just plain text.
org.omg.CORBA.Object obj = ???
ProxyAdmin pa = ProxyAdminHelper.narrow(obj);

// initialization parameters, including the fault tolerant
// group name. application servers that will be in the
// group, and any other application specific data
Property[] criteria = ???

// create a fault-tolerant group for the required
// application
org.omg.CORBA.Object objCounter =
    pa.create_object("IDL:ftcorba/applications/Counter:1.0",
                    criteria, factory_id_holder);

// narrow generic CORBA object to application interface
// in this case, the Counter interface
Counter c = CounterHelper.narrow(objCounter);
```

```
// Remote reference to the fault-tolerant Counter
c.inc();
int x = c.value();
System.out.println("Counter value is "+ x);
int old = c.swap(123);
System.out.println("Old value is " + old );
int newval = c.value();
System.out.println("New value is "+newval);
```

Whenever a client calls for a remote operation on the group proxy object, this request is forwarded to all of the group members – this is the dissemination step. A different thread of execution, called the Ordering Thread runs the ordering protocol. When the disseminated operation is committed and executed, its results will return to the proxy. As soon as a result returns, it is returned to the client (in the non-Byzantine model). To the client this procedure is transparent – the remote call simply blocks until a result is returned as it would on any normal CORBA remote call.

3. Differences from the original paper

In this section we describe some of the differences between the paper (“Backoff Protocols for Distributed Mutual Exclusion and Ordering”) and the implementation.

3.1. Location of ordering mechanism

The original Total Ordering paper placed the Ordering Mechanism inside each of the participating clients. Our approach places the ordering mechanism inside a dedicated proxy server. This seemed preferable for reasons of scalability – the proxy server is highly specialized to quickly perform the dissemination and ordering operations required. These proxy servers are naturally more stable than the clients, so we can assume that they will not be shut down as often as a client might be. Of-course, each client can have its own proxy, which can in fact be placed in the same process as the client using the CORBA collocated servant mechanism, thus returning us to the configuration described in the paper.

3.2. Unique IDs

Each request is given a unique ID, comprising of a client (proxy) id and a monotonically increasing counter. This ensures that each request in the system does indeed have a unique ID. This ID is sent in the CORBA GIOP communications to the server in a service context – an enumerated header of each CORBA request. This behavior is compatible with the CORBA standard. To access this header, we use the CORBA Portable Interceptor mechanism – hooks that are fired when a client call is sent/returned or when a server receives/replies to a request. This mechanism is also part of the CORBA standard, so just about every modern CORBA ORB should support it. Unfortunately, this mechanism caused considerable technical difficulties, which we will describe.

3.3. Leader

The original paper has all clients run the ordering protocol. This made sense since clients can fail at will. In Aquarius the proxies can also fail (crash-fault) but they are considered more stable. Therefore one proxy is designated leader, and it alone performs the ordering protocol. All non-leader proxies have a user defined timeout. Once this timeout expires, the proxy will attempt to become the leader. Leader ‘collisions’ are handled exactly as they are in the paper.

3.4. Notifications

The original paper had the ordering thread running constantly, in anticipation of a request arriving at the servers. Reality dictates that we do not hog computer resources and bandwidth to constantly poll the servers’ state. Instead, we define a proxy leader, which sends a blocking (synchronous) get request to the servers. This request returns when an operation on the server is ready to be ordered. Once it returns, the ordering protocol is set in motion.

3.5. Garbage collection

The garbage collection procedure mentioned briefly in the paper is implemented in Aquarius as a timer task. The leader proxy is responsible for updating the stable line (which is the latest request performed by all servers), and the timer removes old IDs (which are ‘behind’ the stable line) from the server. This both releases memory and minimizes the communications overhead required to transmit the ordering state back and forth.

4. ORBacus modification

The CORBA ORB used with Aquarius is IONA’s ORBacus. While being a well-known ORB, it did not support some of the features that we required, specifically, a reactive dispatch strategy, and a flexible Portable Interceptor mechanism.

4.1. The dispatch strategy

ORBacus for Java supports 2 dispatch strategies: thread-pool and thread-per-request. The thread-pool model is obviously inappropriate for our purpose – while being ordered operations can block, and this may exhaust the pool. The only other option, thread-per-request, would not be susceptible to this issue, but this model is not very scalable. We therefore set out to modify the ORBacus source code, to add a new dispatch strategy – the dissemination dispatch strategy (And here lays the root of all evil, you might say). This strategy uses a single thread to receive client requests and queue them, and a fixed number of threads to handle all proxy operations.

4.2. Portable Interceptors

The ORBacus Portable Interceptors implementation relies heavily on indexing according to the active java thread. While this model works well with the existing dispatch strategies, it is extremely inappropriate with the dissemination dispatch strategy.

In fact, ORBacus internal assertions will fail when accessing the portable interceptor. To this end, this mechanism was also modified to fit the Aquarius proxy.

Note that these modifications have effectively hardwired the proxy to the ORBacus ORB. We have attempted to make the modifications in such a way that switching to a different ORB will be as easy as possible.

5. The dissemination dispatch strategy

Let us walk through the operations takes when a client sends a CORBA request to an Aquarius proxy:

1 – The request arrives at a proxy ORB on an ORB thread. It is then places on the incoming requests queue

2 – A proxy thread, called the Dissemination thread, removes the first request from the queue. It then sends a copy of this request to each of the participating replicas. The portable interceptor mechanism is used to add a unique ID to the requests. (All copies of one request have the same ID). The thread then waits for more requests on the incoming requests queue.

3 – The request arrives at the Aquarius application server, and its unique ID is extracted from the CORBA service context. The request is then stored with the state of ‘pending’, indexed according to its unique ID. All notification requests are then answered.

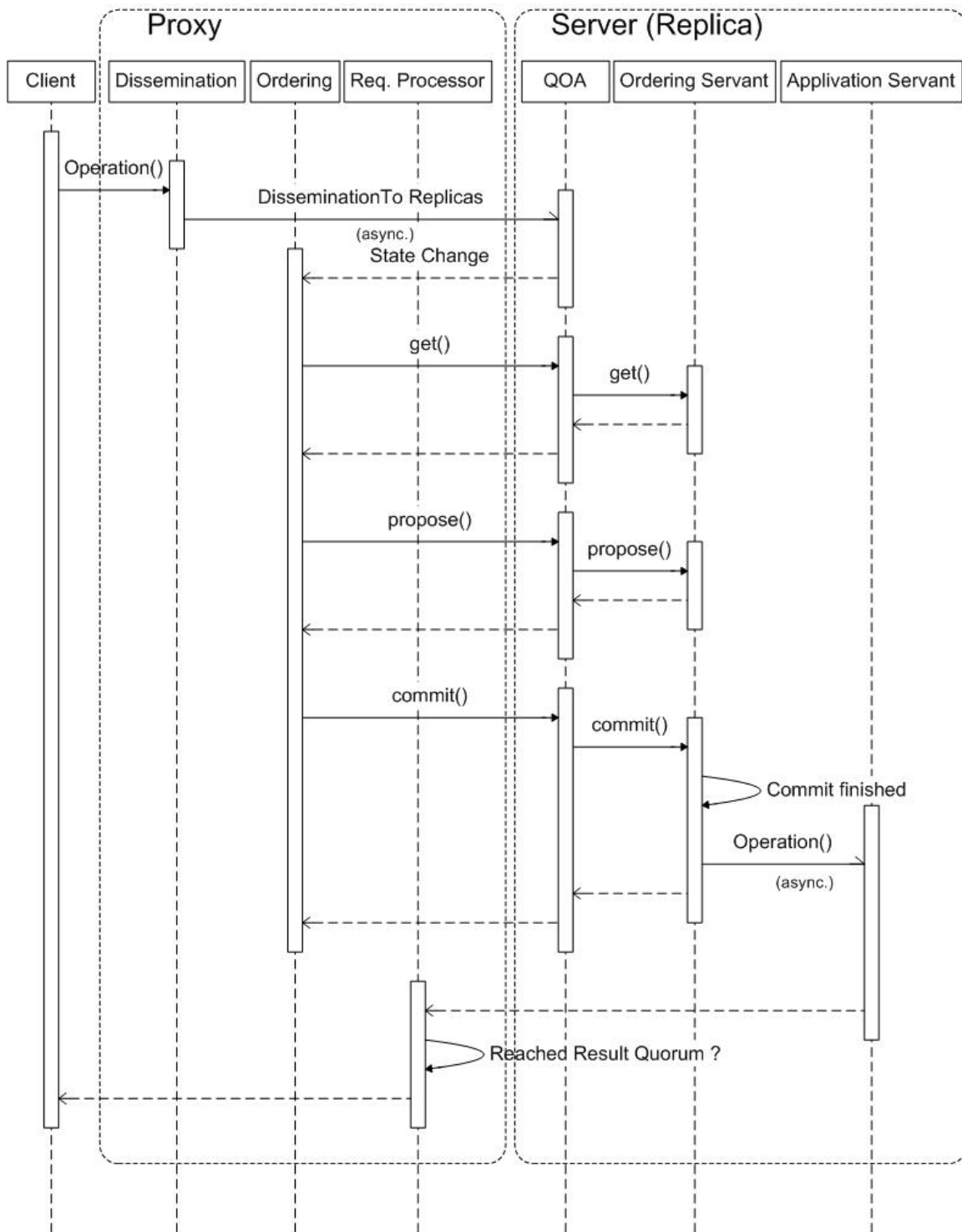
4 – The Aquarius proxy leader receives the returned notification request on a proxy thread called the Notification Thread. This thread checks the notification – old notifications are discarded, new notifications place the group on the Active Groups queue, and put the ordering thread in motion.

5 – The ordering thread, as the name implies, runs the ordering protocol. All requests sent by this thread (get, propose, commit) are also answered on this thread. In each step of the protocol the ordering thread sends the requests asynchronously, and once they are all sent, it waits for a quorum of responses.

6 – Once the ordering protocol for this group is done, new notification requests are sent to the servers. These requests are also sent asynchronously, and their answers are received on the notification thread.

7 – When an operation is committed on the Aquarius application server, it is executed on a separate thread (so as to not block the server from answering other ordering requests while running application code). Once executed, the request returns to the proxy that sent it.

8 – The proxy receives the replies on the Request Processor Thread. Once a request has returned from the application server, its result is forwarded to the client. In a Byzantine model, the proxy waits for a quorum of results before returning the agreed upon result to the client. (Aquarius gives an implementation of the quorum interface for a simple Majority Quorum)



6. Implementing the Total Order interface

When a servant is created in an Aquarius application server as part of a fault-tolerant group, it is capable of handling remote requests by the 2 different interfaces (besides the CORBA Object interface): The application interface (like the Bank interface)

and the total-order interface. Application calls are handled by the application code once they are committed by calls to the total-order interface.

The state of the ordering mechanism on a specific server is called its Object State. This effectively contains a list of committed operations, a list of pending/proposed operations, and integer pointers that specify the committed line, executed line and stable line. The ordering calls run completely independent of the application (dissemination) requests that have arrived at the server. The total-order implementation is responsible for maintaining the Object State. When the first operation in the committed list that is after the current executed line is also available (i.e. the dissemination request for this ID has also arrived) the request can then be executed, and the executed line incremented.

The garbage collection procedure simply removes old elements from the beginning of the list, and removes any remaining data for this request.

Running a server as a member of a fault-tolerant group in Aquarius incurs some overhead. This includes the added time and bandwidth required to send the ordering and notification requests (as well as the added call to the proxy). Some performance metrics of the system are shown in the performance section of the “Aquarius – A Quorum Based Implementation of CORBA Fault-Tolerance” paper.

7. Technical difficulties

This part specifies some of the major problems we encountered while building Aquarius, and will hopefully explain some of our design decisions to anyone interested in the code.

The source of our problems is ORBacus itself. While this may seem like a grandiose declaration, we have more than proved several serious problems with this ORB.

7.1. *Obfuscated code*

IONA has taken delegation beyond any reasonable limit. Following the ORBacus source code proved to be a daunting task. More than just complicating our understanding of the code, we believe that the depth of the function calls has a detrimental effect on the performance of the system.

7.2. *ORBacus memory leaks*

As surprising as this may sound, ORBacus suffers from at least one severe memory leak, which only becomes apparent when a thread-per-request server uses server-side portable interceptors. We have been able to fix the problem we encountered, after considerable work with a java memory profiler.

7.3. *Dispatch strategies, threads and everything in the middle*

As mentioned above, the Aquarius proxy requires a different dispatch strategy. This strategy has several major differences from the existing strategies, but the most notable difference is the fact that a request is sent on one thread, and returns on another. For instance, dissemination requests are sent on the Dissemination Thread, and return on the Request Processor Thread. This may seem like a reasonable request, but is in fact

totally anathema to the entire design of ORBacus. In-fact, ORBacus uses the thread that the request was sent on as an index which is later used to identify the Portable Interceptor data structure (called PICurrent) in the context of the relevant request/response. Since the threads responsible for sending a request and receiving a reply are different, this entire structure crashes completely.

To compensate for this problem, we attempted several workarounds that will supply a key that can identify a request and its reply without relying on the currently active thread. Most of these failed because of the unique structure of the implementation of the PICurrent mechanism. After a thorough review of the PICurrent implementation, it is clear that it was built with the thread-pool dispatch strategy in mind. This is, in fact, the reason for the memory leak mentioned above. Eventually we did indeed use threads as an index into this data structure, but not the current thread. Instead, we map the current thread to the thread that was used to send the request. This solved MOST of the problems. Unfortunately, it did not solve all of them. Under certain load conditions, ORBacus continued to experience internal Assertion failures. We suspect that some race conditions exist that are more relevant to the ORB threads interaction with the PICurrent. As a result, certain operations used in the original ORBacus implementation were simply circumvented, so as to not raise any exceptions. We thoroughly tested this system, and are now quite convinced that our changes did no harm (as long as this modified ORB is used only for the Aquarius proxy)

7.4. Thread pool failures

Apparently, a server application using the thread pool dispatch strategy may ‘lose’ threads over time. We narrowed this problem down to certain communication failures. This was, however, a problem too complicated to address ourselves. Instead, our application servers use the thread-per-request Dispatch Strategy. This is why resolving the memory leak specified above was so crucial.

7.5. The big one

This problem had us stuck for a full month. Even after resolving the issue, we remain uncertain what the problem was, or why it occurred. Once again we encountered a memory leak that eventually left the Aquarius proxy without any available memory. Now experienced with java memory profilers, we were able to narrow the problem down to a certain type of object – `java.lang.ref.Finalizer` – that was never garbage collected by the java VM. Now, this object is created by the VM for each java object that has a `finalize()` method, and is used during the java garbage collector to call this method, before releasing the memory held by the object in question. The objects being held in this case were `com.ooc.CORBA.Request`, the ORBacus implementation of the CORBA Request interface. This led to considerable amounts of memory not being released, eventually crashing the proxy. We proved beyond any doubt that no references were being handled by any Aquarius code. More surprisingly, the `finalize()` method was never called! On one attempt to fix this issue, we rebuilt the java runtime package, `rt.jar`, after adding code we hoped would shed some light on the problem. It did indeed prove that the `Finalizer` objects for the ORBacus request were never called, and never released, but little else. The solution was simple enough, though. We simply removed the `finalize()` method from the said class. The code we removed was placed in a different location, which for the

Aquarius proxy would always be sufficient. Voila, problem solved. We considered sending this bug to Sun, since the JVM was somehow involved, but since ORBacus requires Java 1.3.1, and will simply not work with any later version, we decided against it.

8. Appendix A – Compiling the Aquarius project

The main packages which are part of Aquarius are:

- fcorba.qoa – implements the server-side logic of an Aquarius application server.
- fcorba.proxy – implements the Aquarius proxy
- fcorba.orb – additional files required for the modified proxy orb (required for the additional dispatch strategy). This package is part of the MOB.jar archive
- com.ooc.*, org.omg.* - ORBacus implementation
- SyncUtil, fcorba.util – utility classes used by various packages
- aquarius.bank.* - A sample application with a GUI, that enables viewing the server Object State.

In addition, several jar files are required:

- threadPool.jar
- OB.jar, OBNaming.jar – The standard ORBacus CORBA ORB, and the naming service supplied with it
- SOB.jar – The standard ORBacus ORB with the fixed memory leak. It is used with the Aquarius application servers.

All relevant files are supplied in a zip archive called aquarius.zip. In addition, 2 makefiles are also supplied:

- makefile.orb – The makefile supplied by ORBacus. Used to build the standard OB.jar.
- makefile - The Aquarius makefile, used to build the Aquarius packages, and run test servers and clients.

The build options are:

- make -f makefile.orb : builds the ORBacus ORB.
- make proxy : builds the Aquarius proxy
- make qoa : builds the Aquarius proxy and qoa

9. Appendix B – Running the Aquarius project

To run the Aquarius project, several processes must be active at the same time.

9.1. Support services

First, the CORBA Interface Repository and Naming Service must be run. These can be activated using the supplied makefile, by calling 'make irserv' and 'make namesrv', respectively. The host and port where these 2 services were run must be specified in the Aquarius makefile.

9.2. Application servers

Next, the application servers must be activated. The test application servers can be activated by calling 'make QOAServer'. Any number of these can be executed. Each server will register itself in the Naming Service, so it will be available to clients. Alternatively, call 'make QOAServerClear' to remove all existing application servers from the Naming Service before registering this one.

The Server supports the following command line parameters:

- -clearNames – deletes previous application server references in the Naming Service. Note that the name with which the server registers itself is determined by the application itself.

9.3. Aquarius Proxy

Next, activate an Aquarius proxy, by calling 'make QOAProxy' or 'make QOAProxyLeader' to mark this proxy as the leader on startup. Remember, if no proxy is marked as leader and operations are pending, then the non-leader proxies will timeout, and contend for the leadership position, so that eventually a leader will be elected, even if no leader was initially selected. Any number of proxies can be run at the same time. Each will register itself at the Naming Service on startup. Calling 'make QOAProxyLeader' automatically removes all previous proxies from the Naming Service.

The Proxy supports the following command line parameters:

- -leader – indicates that this proxy is the leader proxy
 - -clearNames - deletes previous proxy references in the Naming Service.
- Note that the name with which the proxy registers itself is determined by the application itself.

9.4. Application client

Finally, the application client can be run. This is done by calling 'make QOATester'. The supplied test application runs a simple counter object: the interface supports increment, decrement and swap of an integer value.

The Proxy supports the following command line parameters:

- -groupName <Group Name> - indicates the name of the group object that this client will use. If the group does not exist, the client will request a new group be created with this name. This group will consist of all application servers currently registered in the Naming Service.
- -proxyName <Proxy Name> - indicates which proxy the client will connect to. This is the name of the proxy as it appears in the Naming Service.
- -proxyFileName <File Name> - indicates a text file that contains a CORBA IOR reference. This IOR will be used to connect the client to an existing group object on an active Aquarius proxy.

10. Appendix C – The IDLs

10.1. *Total-order.idl*

```
module ftcorba {  
  
    module qoa {  
  
        const long REQUEST_ID_SERVICE_ID = 1001;  
  
        typedef long ClientID;  
  
        struct UniqueID {  
            ClientID cid;  
            long counter;  
        };  
  
        typedef UniqueID Rank;  
        typedef UniqueID RequestID;  
  
        typedef sequence<RequestID> RequestIDseq;  
  
        struct ObjectState {  
            RequestIDseq    committed;  
            RequestIDseq    proposed;  
            RequestIDseq    pending;  
            Rank            proposer;  
            long            committed_line;  
        };  
  
        exception RankException {  
            Rank r;  
        };  
  
        exception ObjectNotExist {  
            sequence<octet> objID;  
        };  
  
        exception StateTransfer {  
            long committed_line;  
        };  
  
        interface TotalOrder {
```

```

ObjectState getNotify(in Rank r)
    raises (ObjectNotExist, RankException);

ObjectState get(in Rank r)
    raises (ObjectNotExist, RankException);

void propose(in Rank r,
             in long committed_line,
             in long stable_line,
             in RequestIDseq committed,
             in RequestIDseq proposed)
    raises (ObjectNotExist,
           RankException,
           StateTransfer);

void commit(in Rank r,
            in long committed_line,
            in long stable_line,
            in RequestIDseq committed)
    raises (ObjectNotExist,
           RankException,
           StateTransfer);

Rank getRank();

};

};

};

```

10.2. *proxy.idl*

```

#include <OB/orb.idl>

module proxy {

    //////////////// Property Manager definitions
    ////////////////
    typedef string Name;
    typedef any Value;
    typedef CORBA::RepositoryId TypeId; // this is typedef'd
                                        // to a string

    struct Property {
        Name nam;
        Value val;
    };
};

```

```

typedef sequence<Property> Properties;
exception UnsupportedProperty {
    Name nam;
    Value val;
};

interface PropertyManager {
    void set_type_properties(in TypeId type_id,
                            in Properties props)
        raises (UnsupportedProperty);

    Properties get_type_properties(in TypeId type_id);
};

typedef sequence<PropertyManager> PropertyManagers;

////////// End of Property Manager definitions //////////

////////// Generic Factory Definitions //////////////////////////////////////

typedef Name Location;
typedef Properties Criteria;

exception NoFactory {
    Location the_location;
    TypeId type_id;
};

exception CannotMeetCriteria {
    Criteria unmet_criteria;
};

exception InvalidCriteria {
    Criteria invalid_criteria;
};

exception InterfaceNotFound {};

exception ObjectNotFound {};

exception ObjectNotCreated {};

```

```

interface GenericFactory {
    typedef any FactoryCreationId;

    Object create_object(
        in TypeId type_id,
        in Criteria the_criteria,
        out FactoryCreationId factory_creation_id)
        raises(NoFactory,
            CannotMeetCriteria,
            InvalidCriteria);

    void delete_object(
        in FactoryCreationId factory_creation_id)
        raises(ObjectNotFound);
};

////////// End of Generic Factory Definitions //////////

////////////////////////////////////// QOAAdmin ////////////////////////////////////////
// This is the interface that QOA Servers must //
// implement to be accessible via the Proxy //
//////////////////////////////////////
const Name QOA_GROUP_NAME = "qoadmin.GroupName";
const Name QOAADMIN_REF = "qoadmin.ref";
const Name QOA_FACTORY = "qoadmin.Factory";

// Factory info property defs
const Name FACTORIES = "cacheservice.Factories";
struct FactoryInfo {
    GenericFactory fact;
    Location the_location;
    Criteria the_criteria;
};
typedef sequence<FactoryInfo> FactoryInfos;

interface QOAAdmin:PropertyManager, GenericFactory {
    boolean supportsType(in TypeId type);
};

typedef sequence<QOAAdmin> QOAAdmins;

```

```

//////////////////////////////////// End of QOAdmin definitions //////////////////////////////////////

//////////////////////////////////// ProxyAdmin definitions //////////////////////////////////////
typedef sequence<Object> Objects;
exception ObjectGroupNotFound {};

//////////////////////////////////// ProxyAdmin //////////////////////////////////////
// This is the inteface that clients access to reach //
// the proxy. //
////////////////////////////////////
const Name PROXY_GROUP_NAME = "proxyadmin.GroupName";
const Name PROXY_GROUP_MEMBERS =
    "propxyadmin.GroupMembers";
const Name PROXY_QOA_ADMIN = "proxyadmin.QOAdmin";
const Name PROXY_FACTORIES = "proxyadmin.Factories";
const Name PROXY_OBJECTS = "proxyadmin.Objects";

struct GroupMembers {
    Name groupName;
    PropertyManagers val;
};

interface ProxyAdmin : PropertyManager, GenericFactory {

    PropertyManagers getServersForType(in TypeId type);
    Object get_object(in string groupName)
        raises (ObjectNotFound);
    Objects getGroupMembers(in string groupName)
        raises (ObjectNotFound);

};

//////////////////////////////////// End of ProxyAdmin definitions //////////////////////////////////////
};

```