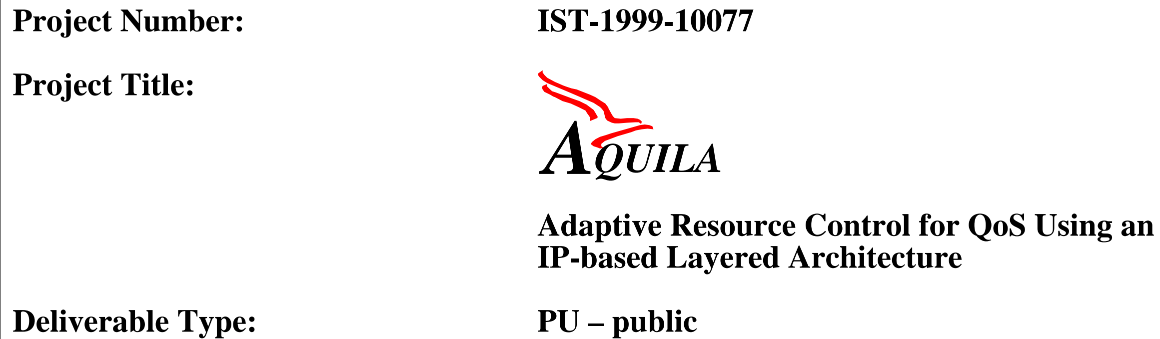
Project Number:	**IST-1999-10077**
Project Title:	 **Adaptive Resource Control for QoS Using an IP-based Layered Architecture**
Deliverable Type:	**PU – public**

Deliverable Number:	**IST-1999-10077-WP2.1-SAG-2104-PU-R/b1**
Contractual Date of Delivery to the CEC:	**June 30, 2002**
Actual Date of Delivery to the CEC:	**June 28, 2002: version b0** **December 12, 2002: version b1**
Title of Deliverable:	**Report on implementation of the resource control agent for the second trial**
Workpackage contributing to the Deliverable:	**WP 2.1**
Nature of the Deliverable:	**R – Report**
Editor:	**Martin Winter (SAG)**
Author(s):	**Lila Dimopoulou (NTU); Reinhard Frank (SAG); Falk Fünfstück (TUD); John Karadimas (QSY); Eugenia Nikolouzou (NTU); Petros Sampatakos (NTU); Martin Winter (SAG)**

Abstract:	The basic structure of the implementation as well as experience made with several software technologies like JAVA, CORBA, LDAP, XML, is reported in this document. Common mechanisms used throughout the components are described.

Keyword List:	AQUILA, IST, software technologies, implementation experience, mechanisms

Executive Summary

The main task of work-package 2.1 was the design and implementation of the resource control layer. However, the result of this work is not only the prototype software used in the trial. Additionally, many design decisions and implementation experience are worth noting and may help for similar decisions in further projects.

This deliverable shortly describes some of these decisions and reports our experience. Furthermore, some basic mechanisms used in various software components are described.

The general approach taken in the first trial has been proven to be useful and reasonable:

- The choice of JAVA as programming language fulfilled the promise: the code was relatively soon in a good status with only few errors. Especially the language helped to avoid some typical mistakes, which happen more often with other languages (e.g. memory leaks).
- The JAVA language proved its platform independence. All JAVA software developed runs on Sun Solaris as well as Windows NT without any changes.
- The rich amount of available interfaces enabled the easy use of technologies like CORBA, LDAP and XML.
- CORBA enabled us to clearly specify interfaces using IDL and minimised interfacing problems between packages implemented by different partners.
- LDAP proved its usability as an easy-to-use database concept for data, which is more often read than written.

However, there were also some problems identified:

- Using the serialisation mechanism of JNDI to store objects has two major drawbacks: It makes the data unreadable for a human, because it is stored as binary content, and it complicates upgrades of the stored objects.
- When each software component uses its own method to store and retrieve configuration data, the overall system gets hardly manageable.

This document also describes our experience with the approach to define and store configuration data using XML to represent structured data and LDAP for storage.

Additionally, it describes some commonly used mechanisms like event handling, use of pluggable algorithms and the trace facilities.

Table of Contents

1 INTRODUCTION ... 5

2 STRUCTURE ... 6

2.1 COMPONENTS ... 6

2.1.1 Executable components ... 6

2.1.2 Supporting components ... 8

2.2 NAMES AND DATABASES ... 9

2.2.1 Component names ... 9

2.2.2 Database structure ... 11

2.2.3 CORBA object naming structure ... 11

3 EXPERIENCE ... 13

3.1 CONFIGURING COMPLEX SYSTEMS WITH XML ... 13

3.1.1 Managing configuration data (QMTool) ... 13

3.1.2 External Interfaces ... 15

3.1.3 Application access to configuration data using JAXB ... 19

3.2 INTEROPERABILITY OF JAVA SOFTWARE TECHNOLOGIES ... 20

3.2.1 Description of used software technologies ... 20

3.2.2 Parallel use of multiple technologies ... 21

3.2.3 JAXB objects via CORBA ... 22

4 MECHANISMS ... 24

4.1 EVENT NOTIFICATION ... 24

4.1.1 Common event mechanism ... 24

4.1.2 Keep-alive ... 24

4.2 PLUGGABLE ALGORITHMS ... 25

4.2.1 Admission control algorithms ... 25

4.2.2 Resource control rules ... 26

4.3 TRACING ... 29

5 ABBREVIATIONS 31

Table of Figures

FIGURE 2-1: THE COMPONENTS OF THE CORE AREA 7
FIGURE 2-2: THE COMPONENTS OF THE ACCESS AREA 8
FIGURE 2-3: SUPPORTING COMPONENTS 9
FIGURE 3-1: INTERACTION OF THE BASIC MODULES OF QMTOOL 14
FIGURE 3-2: MONITORING INTERACTION 16
FIGURE 3-3: FAILURE DETECTION INTERACTION 18
FIGURE 4-1: DESIGN MODEL OF RESOURCE CONTROL RULES PACKAGE 27

Table of Tables

TABLE 3-1: INTERFACE OFFERED TO QMTOOL FOR MONITORING..17
TABLE 3-2: INTERFACE OFFERED TO QMTOOL FOR FAILURE DETECTION 19

1 Introduction

This document provides a loosely coupled collection of experience from the implementation of the AQUILA resource control layer software. The target auditory are the project partners, which might use this experience for similar design decisions in future projects.

The document is structured into three main areas:

- Chapter 2 reports the structure of the software. Associated with that is the structure of the name spaces used for JAVA packages, for the LDAP database and for the CORBA naming service.
- Chapter 3 describes our experience in to main areas: the configuration of a rather complex software system, and the interoperability of various software technologies used in the project like JAVA, LDAP, CORBA and XML.
- Chapter 4 describes some commonly used mechanisms: A global event handling has been used in the project. Another major design decision was the use of pluggable algorithms, which enable the selection of different algorithmic implementations through configuration. A common trace facility was also very valuable for both integration and trials.

A list of abbreviations ends this document.

2 Structure

2.1 Components

The AQUILA software architectures consists of several, distributed "components" all written in Java. Note that in the following the term "component" is used to describe a separately implemented piece of software with a well-specified functionality or behaviour and with well-defined interfaces. However, there are different types of components in AQUILA: executable versus (non-executable) supporting ones, complex (e.g. the Mediazine application) versus basic ones (e.g. the subscriber component/package).

This chapter gives an overview of the most important components of the Resource Control Lay (RCL) including the inter-domain layer above the RCL (the core area) as well as the components of the access area. Not shown are the components of the measurement toolkit as well as the implemented utility components.

We use UML component diagrams for the presentation of the components and their dependencies.

2.1.1 Executable components

The following components of the AQUILA software architecture are executable ones. They communicate via CORBA with each other (except ACA ⇔ router) whereas the interfaces are specified in IDL.

Core Area (Figure 2-1):

- The BGRP (Agent) of the inter-domain layer supporting interfaces for inter-domain reservations towards the ACA and other BGRP agents.
- The Resource Control Agent (RCA) of the RCL supporting interfaces to control the resource pool limits towards itself and the ACA.
- The Admission Control Agent (ACA) supporting interfaces to establish reservations towards the BGRP agent, the EAT and other ACAs (see also Figure 2-2).
- The QoS Management Tool (QMTool) using the interfaces of the BGRP agent and the RCA for monitoring special data.
- The router (package) supporting different interfaces for router configuration, monitoring etc. towards the ACA. This component is actually not an executable one but it *represents and encapsulates* one, which is the (edge or border) router.

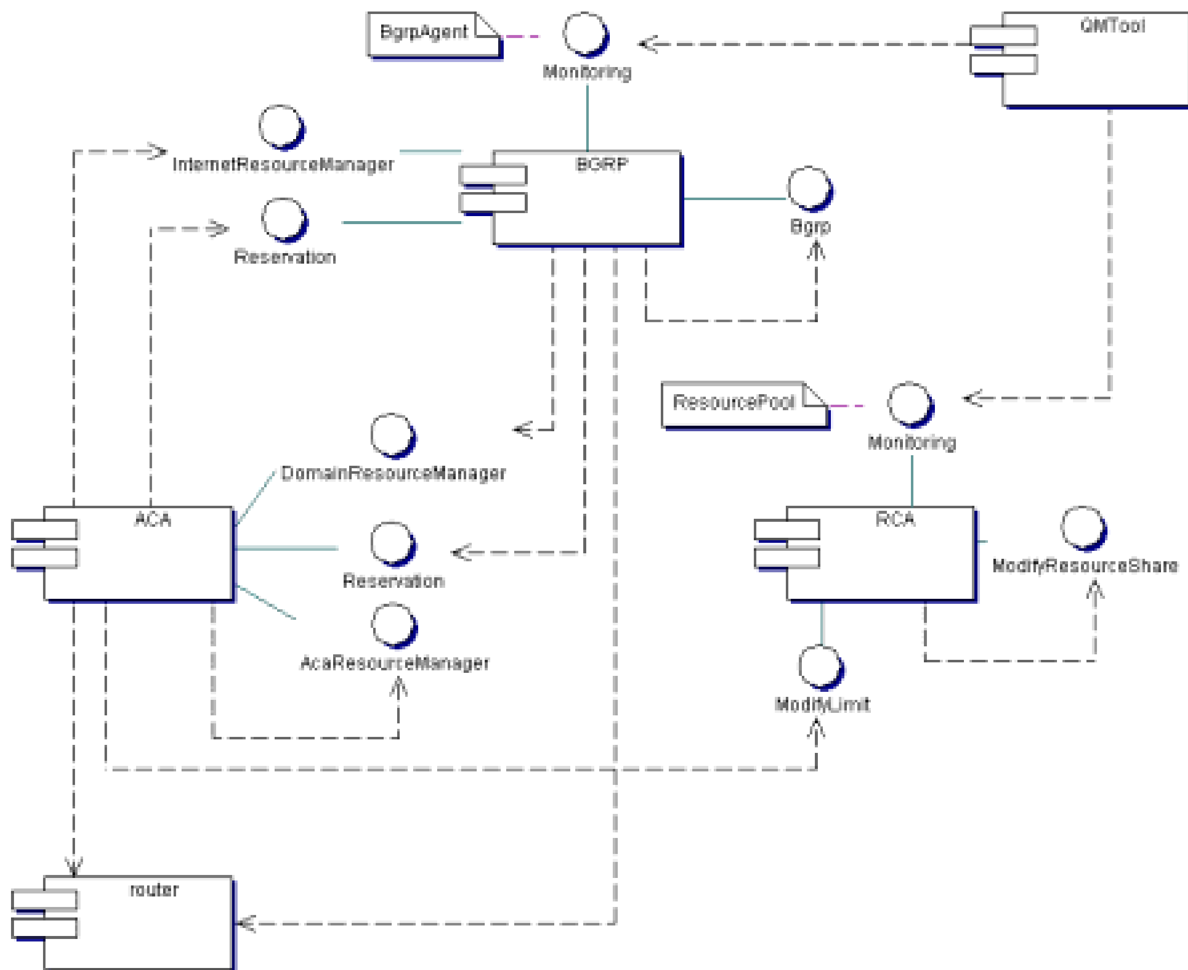


Figure 2-1: The components of the core area

Access Area (Figure 2-2):

- The End-user Application Toolkit (EAT) supporting (via the EAT API) several interfaces towards the applications and the portal, and using interfaces of the ACA.
- The Complex Internet Service Mediazine as an example for a QoS-aware application.
- The AQUILA portal which forms the GUI of the EAT using the same interfaces as Mediazine.
- The H.323 Proxy and the SIP Proxy supporting an application proxy interface towards the EAT and using the session starter interface of the EAT. (Note that although the application proxies are part of the EAT package, they run as separate components.)

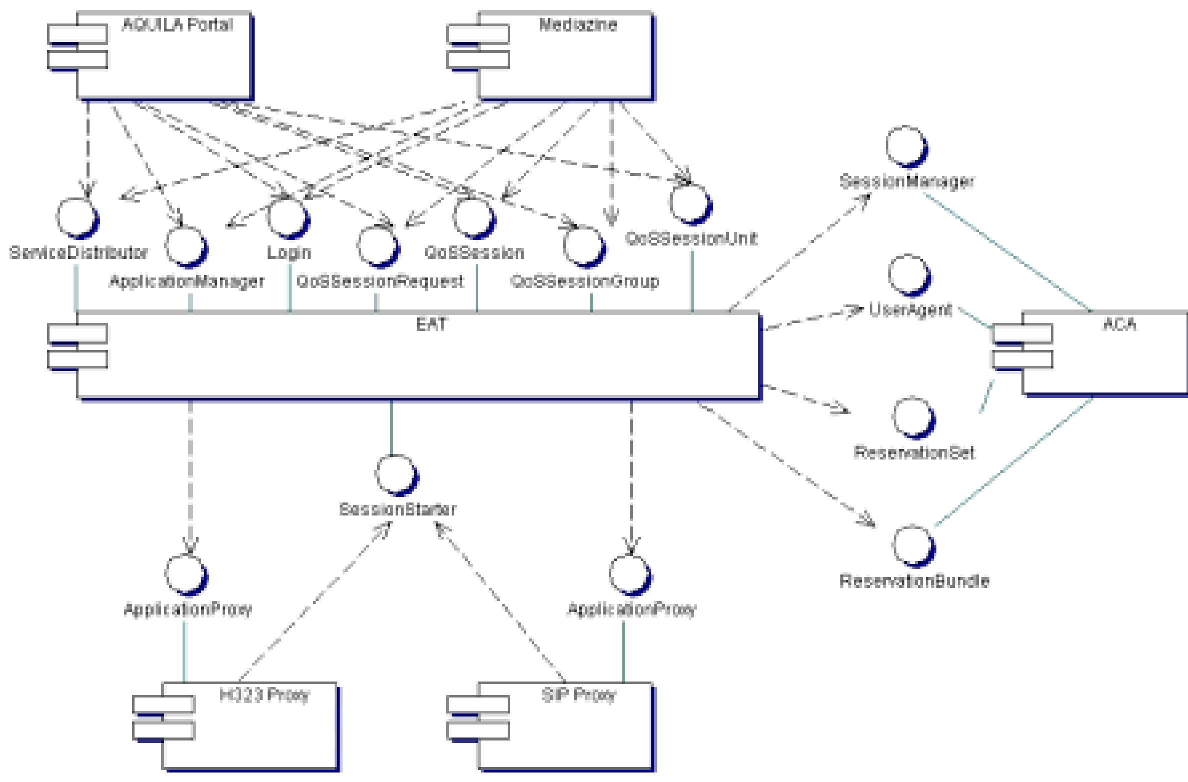


Figure 2-2: The components of the access area

2.1.2 Supporting components

There are some important components of the RCL that support others but that do not run in an own process (Figure 2-3). Instead, they are implemented by the above-mentioned components. Note that the following list does not contain the components/packages which are part of the utility package[1] or that are sub-packages[2] of RCL ones:

- The subscriber component supporting interfaces for subscriber retrieval.
- The service component supporting interfaces for network service retrieval.
- The **tc** component supporting interfaces for traffic class retrieval.

Some of these interfaces (the "manager" ones) are specified in IDL, although they are not used via CORBA. The others are specified in DTD and translated into Java classes in order to be used with JAXB.

---

[1] These are: alive, corba, event, inet, main, persist, prisig, rcrule, router, trace.

[2] Examples are: network in RCA; appProfile, converter in EAT, etc.

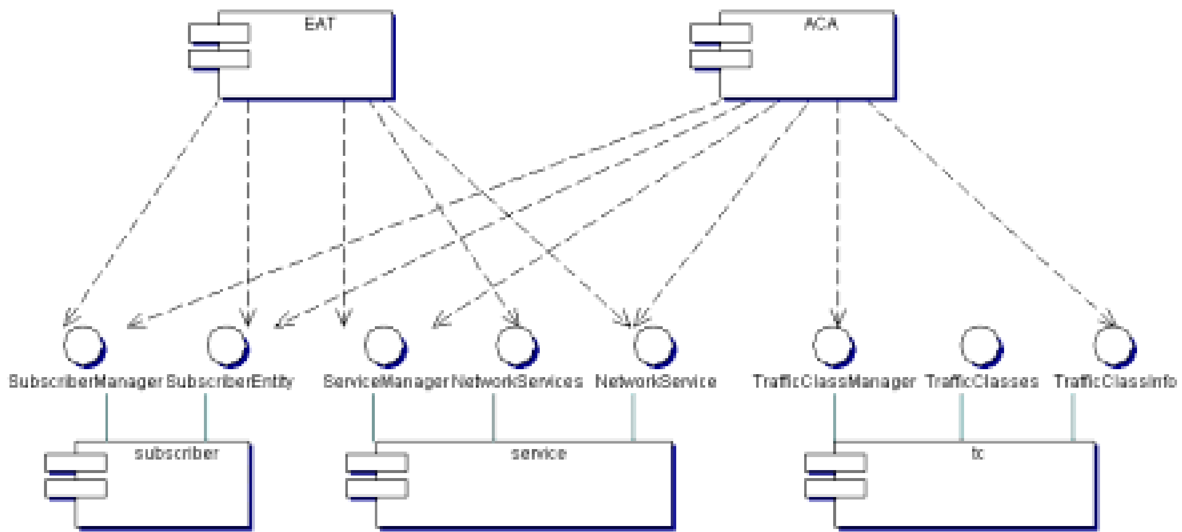


Figure 2-3: Supporting components

2.2 Names and databases

In an environment with distributed components and distributed development, it is essential to define a common and non-ambiguous naming scheme, valid for the whole project. This comprises

- JAVA package names
- names to bind CORBA server objects to the naming service
- names of objects in the LDAP database
- names of routers
- names of components associated to the routers (ACA, BGRP agent)
- names of other components

The project defined a set of rules to create these names, which are described in the following chapters.

2.2.1 Component names

As far as possible a naming schema was defined for all components, which follows the layered approach of the AQUILA software architecture:

- The root of the project's namespace was called “**aquila**”.
- Below that, four layers are defined:

- **rcl** Resource control layer
- **idom** Inter-domain layer
- **manag** Management layer
- **util** Common utilities

- In each layer, the components are used as the next level of hierarchy.
- Additional substructures are created, if necessary.

2.2.1.1 JAVA packages

JAVA packages are named according to the structure defined above:

```
aquila.<layer>.<component>[.<subcomponent>]
```

The package names consist of lowercase letters only. Additional sub-packages may be added, if appropriate.

The AQUILA software also uses JAVA technologies, which automatically create JAVA source files:

- CORBA: Creates server and client stubs and helper classes from the IDL definitions
- JAXB: Creates classes from DTD files, supported by XJS files.

These files are also generated into this package structure. The names of the IDL, DTD and XJS-Files begin with the component name, to which they belong.

2.2.1.2 Resource control layer components

Besides the package structure, which reflects the software architecture, there is a component structure, which reflects the network topology:

- ACAs and BGRP agents are associated to routers
- Resource pools reflect regional areas of the network, at the lowest level individual ACAs

For such components, names are generated according to the following schema:

```
ttttddnsss
```

with

```
tttt    = type:
        aca_    ACA
        bgrp    BGRP agent
        eat_    EAT
        lrp_    Lowest level resource pool
        rtr__   Router
```

```
dd      = device
        br      Border router
        cr      Core router
        er      Edge router

n       = number (sequential number to distinguish devices)

sss     = Trial site
        eli     Elisa Communications (Helsinki)
        spu     Salzburg Research (Salzburg)
        taa     Telekom Austria (Vienna)
        tps     Telekom Polska (Warsaw)
```

2.2.2 Database structure

The naming schema described above is also used to structure the configuration database as well as for the definition of CORBA object names.

Configuration information is stored in an LDAP database using the following structure:

```
c=eu
        o=aquila
                cn=rcl
                        cn=aca
                                cn=<acaname>
                        cn=eat
                        cn=rca
                        cn=service
                        cn=subscriber
                                cn=<username>
                        cn=tc
                cn=idom
                        cn=bgrp
                                cn=<bgrpname>
                        cn=gwks
                        cn=neighb
                                cn=<neighbname>
                cn=util
                        cn=alive
                        cn=prisig
                        cn=router
                                cn=<rtrname>
                                cn=message
                                        cn=<iosname>
                        cn=trace
```

Again, the hierarchical structure of the AQUILA architecture is used to structure the database. Where appropriate, multiple entries are used, each corresponding to an entity of the architecture.

2.2.3 CORBA object naming structure

To establish communication paths between the executable components of the AQUILA architecture, CORBA objects are used. Object references are stored in a CORBA naming service per domain. The namespace structure in the naming service again follows the hierarchical structure of the AQUILA architecture.

The CORBA terminology calls the full hierarchical description for an object a **name**, composed of one or more **name components**, each describing one hierarchical level. A name component again consists of an **id** and a **kind**.

Within the AQUILA project, this terminology is mapped to the hierarchical structure of the architecture as follows:

- The topmost name component id is "**aquila**".
- Hierarchical subordinated name components are used analogous to the JAVA package names.
- The lowest level name component is the AQUILA component name, as defined in chapter 2.2.1.2. If a component defines more than one CORBA object in the name service, then this name may be augmented by additional information.
- For lowest level components, the kind of a name component is the name of the interface of the CORBA object.
- For all higher-level components, the kind of a name is "directory".

As an example, the name of the inter-domain resource control interface of a BGRP agent is as follows, using the Interoperable Naming Service notation [INS]:

```
aquila.directory/
  idom.directory/
    bgrp.directory/
      <bgrpname>.InterdomainResourceManager
```

For easy access to these names, each CORBA interface defines two constant strings:

```
SERVER_PREFIX   (e.g. aquila.directory/idom.directory/bgrp.directory/)
SERVER_SUFFIX   (e.g. .InterdomainResourceManager)
```

3 Experience

3.1 Configuring complex systems with XML

3.1.1 Managing configuration data (QMTool)

3.1.1.1 Introduction

QMTool is the management tool, which will mainly serve the configuration of the components belonging to both the RCL and IDOM layers. Moreover, it will enable the detection of these components in case of failure and the monitoring of the utilisation of their resources.

For that reason, QMTool mainly entails four functionalities. The first corresponds to the **design** functionality which permits the graphical representation of the various network elements and the determination of their relations. In this way, a user-friendly interface will provide a clear view of the relation between Resource Pools, ACAs, EATs, Proxies and BGRP agents.

The second deals with the **configuration** of the AQUILA network components (visualised components) as well as some further elements that are essential for the functioning of the network. QMTool will provide a DTD enabled XML writer module that will enable the setting of the configuration data via the creation of XML files. Therefore, dependent on the DTD configuration file of each component, QMTool will be able to validate the corresponding XML configuration files. The repository of those files will be the LDAP database and at this point QMTool will communicate with the ObjectStore interface that actually performs the storage of XML files to the database.

Additionally, the **failure detection** functionality that is integrated into QMTool, will give the opportunity to the administrator of the AQUILA network to see the status of the components. The **AlivePeer** mechanism is envisaged between QMTool and the controlled components in order to view their status. In case of failure, a notification is sent (graphical notification) that will obviate the component that has failed.

Finally, the QMTool will provide **monitoring** capabilities. In particular, it will enable the monitoring of the resources of Resource Pools and BGRP Agents, which are provided to QMTool in the form of XML files. The XML files will contain data that require both graphical and textual representation and consequently, an indication of the format (attribute) will be always given to QMTool. Monitoring will be polling-based and the user will further have the chance to set the polling intervals.

3.1.1.2 Design Model

The QMTool consists of four basic modules, the GUI the XML editor, the **ControlFailure** and the **ControlMonitoring**. The first one provides a graphical representation of the whole

architecture, i.e. it depicts all the components of the RCL and the IDOM layer. Therefore, the user can be aware of the relations among those components.

The XML editor enables the user to configure the several components of the network. After the design of the network with the help of the GUI model, the several elements need to be configured. In particular, each type of component (BgrpAgent, EAT, Proxy, ACA, RCA) should be bound to a certain DTD file that identifies the structure of its configuration data. Each time the user needs to configure a certain component, an XML editor should be available so that the user fills the XML file with the appropriate configuration data. The structure of this XML file will be based on the corresponding DTD file of this component.

All this configuration data is stored at the LDAP database, at known entries, by utilising the **Xldap** object that implements the **ObjectStore** interface. The retrieval of the configuration data (XML file), in case that the user requests it (for possible modifications), is performed again through the same interface.

The **ControlFailure** and **ControlMonitoring** interfaces are only used for the start and stop of the corresponding action for a component that the user has requested. Therefore, these interfaces are only internal interfaces that serve the triggering of these actions. The whole functionality of failure detection and monitoring is performed through external interfaces with the components.

In the following figure, a rough design model of the QMTool functionality is illustrated.

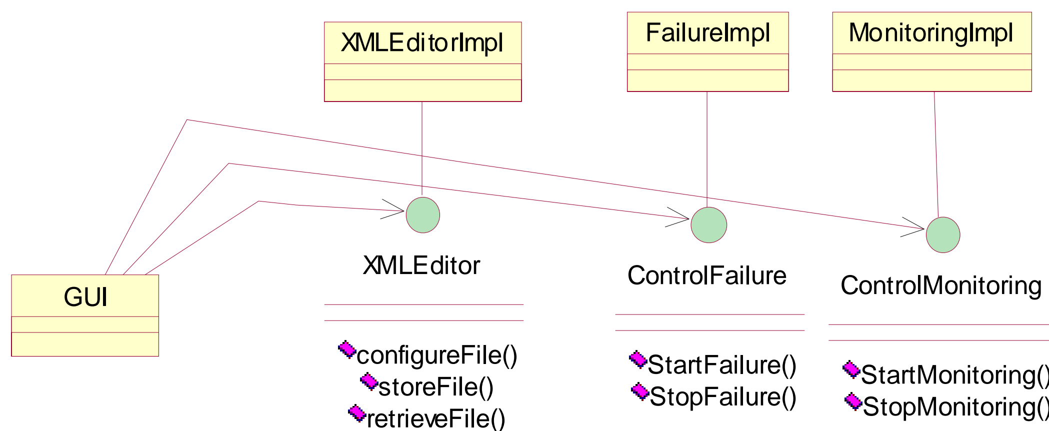


Figure 3-1: Interaction of the basic modules of QMTool

It is shown that there exists basically the **XMLEditor** interface offered to the GUI module. This interface has three main functions: "**storeFile()**" and "**retrieveFile()**" serve the storage of the configuration data to the LDAP database and their retrieval correspondingly in case that the user wants to view how the components are configured or to perform modifications. The function "**configureComp()**" is called each time the user configures for the first time a

component. This is the case where the DTD file should be taken into account in order that the XML editor provides the user with the appropriate structure of an XML file to be filled.

The **ControlFailure** and the **ControlMonitoring** interface basically provide two functions for the start and stop of Failure Detection and Monitoring of a component.

3.1.2 External Interfaces

The external interfaces that are offered from the other components to QMTool serve the following functionality:

- The monitoring of network entities belonging to either the RCL layer or the IDOM layer. The monitored data varies among the different components and therefore QMTool should determine a more generic framework for supporting this kind of operation carrying the individualities of such components. The monitoring functionality will be polling-based (QMTool will request the monitoring data from the several components and there will be no notification mechanism from their part).

- The failure detection of the RCL layer entities such as RCA, ACA and EAT as well as the IDOM layer components. For this purpose, the already proposed Keep-Alive mechanism will be used and no additional interface will be required from the components.

It is intended that the external interfaces will be described and categorized based on their functionality and not on the layers they interact with.

3.1.2.1 Interfaces required for monitoring

The Management Layer and the Resource Control Layer communicate via one particular interface that should exist for the deployment of the monitoring functionality. In particular, the RCL and IDOM components that necessitate monitoring offer the Monitoring CORBA interface to QMTool as depicted below.

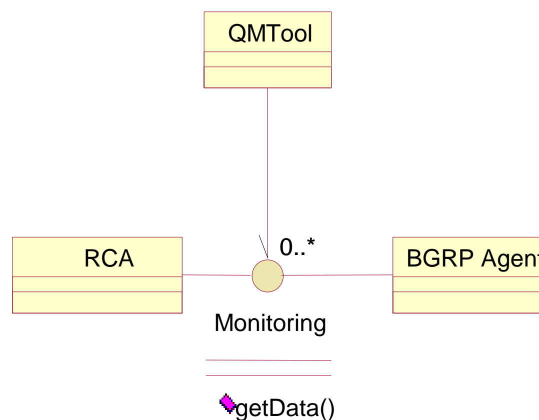


Figure 3-2: Monitoring Interaction

It is important that the monitoring data is not a fixed structure since there is the potential for being extended while additional information needs to be monitored. Therefore, it is suggested that the monitoring information that is retrieved from the several components is an XML file. In the cases where we need a graphical representation of the data, an attribute denoting the display type could be added in the XML file and serve as an indication for QMTool for the type of monitoring. For example, the monitoring data could be as follows.

```
<RCAMonitoringData>
        <usage display = "percent">0,7</usage>
        <total>200</total>
</RCAMonitoringData>
```

QMTool should display two elements, the "usage" and the "total". The "total" does not necessitate any specific monitoring type, so it will have a textual representation. However, the "usage" requires that its display type is "percent". This could have as a result the graphical representation of this value.

The monitoring interface is a CORBA interface. All the monitored components should register themselves at a specific branch of the CORBA naming service. Therefore, QMTool, when triggered by the user, should start getting the CORBA references of these monitoring objects, which will be identified by their path in the LDAP database. In this way, QMTool will be able to perform a mapping between the objects stored in the LDAP and the corresponding monitored objects.

3.1.2.1.1 Interface of the RCL and IDOM components for QMTool

The following IDL interface is offered from both the Resource Pools and the BGRP agents in order that QMTool monitors their resources. The type of data retrieved by QMTool through the "**getData**" function is of type "**Object** and represents the data to be monitored (XML file). QMTool will be aware of what type of monitoring to perform and of the type of data to

monitor by consulting the retrieved XML file. The "**getPeer**" function will serve the failure detection functionality and is therefore not described here.

```
module monitor
{
  /* Interface offered to QMTool to monitor the components*/

  interface Monitoring
  {
      void getData
      (
          out string    data    //data to be monitored
      )
      raises (InvalidDataException);

      void getPeer
      (
          out AlivePeer peer   //peer of the component
      )
  };

  exception InvalidDataException
  {
      string           reason;
  };
};
```

Table 3-1: Interface offered to QMTool for monitoring

3.1.2.2 Interfaces required for failure detection

The Management Layer should be provided with the essential interfaces for performing failure detection. Those are offered from the components of the RCL layer and IDOM layer as well and will be the ones already used for supporting the Keep Alive mechanism.

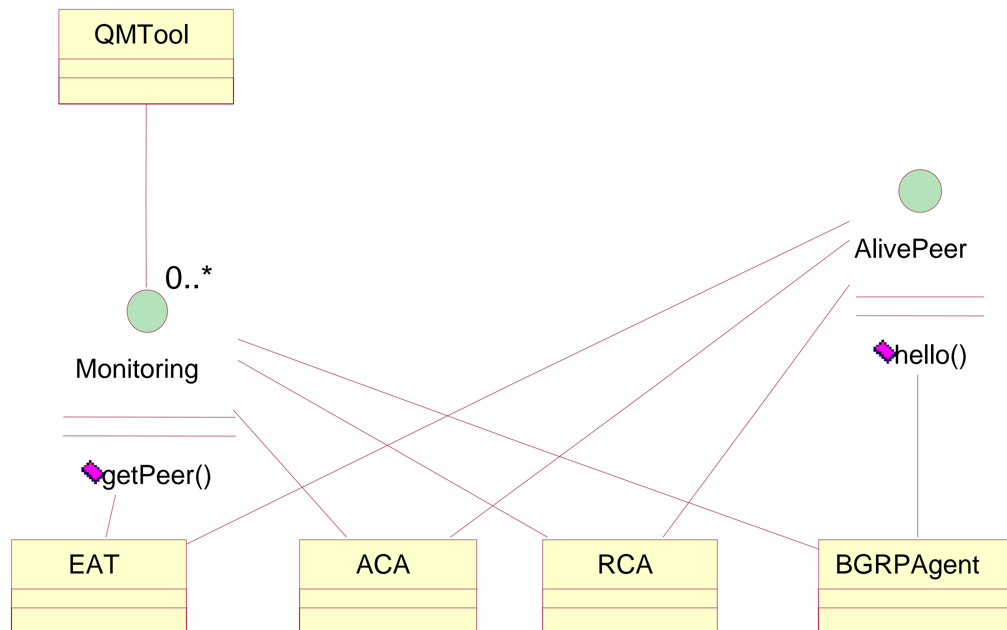


Figure 3-3: Failure Detection Interaction

The RCL elements, i.e. the RCA, ACAs and EATs, as well as the IDOM elements, i.e. the BGRP agents, implement the Alive Peer interface in order to allow QMTool being aware of their status.

QMTool retrieves a reference to the Alive Peers of these components through the "Monitoring" interface. Therefore, the "Monitoring" interface is used both for retrieving the monitored data and the Alive Peers of the components. This is necessary because the Alive Peers of each component are not registered to the CORBA Name Service and therefore cannot be resolved by QMTool.

3.1.2.2.1 Interface of the RCL and IDOM components for QMTool

The IDL specification of the interface, which is offered by the RCL and IDOM components, is shown below:

```
module alive
{
  interface AlivePeer
  {
      void hello
      (
          in AlivePeer    peer,   // sender
          in long         seq     // sequence number
      )
      raises (AliveException);
  };

  exception AliveException
  {
      string          reason;
  };
};
```

Table 3-2: Interface offered to QMTool for failure detection

3.1.3 Application access to configuration data using JAXB

Access to XML-formatted data in JAVA applications can be performed in two ways:

- Use of a XML parser to retrieve the XML elements and attributes
- Use of a binding technology to translate the XML structure into a JAVA object structure

The project has chosen to use the second approach. JAXB (JAVA API for XML Binding) is an implementation of such a technology, available as an early-access release from SUN Microsystems.

JAXB starts from a DTD (Document Type Definition), describing the data structure of the XML configuration data. An XML-to-JAVA compiler (**xjc**) is the used to automatically generate a set of JAVA source files out of the DTD. Details of this compilation process, such as mapping of attributes to JAVA types, can be described in an XML-to-JAVA schema file (XJS file).

The generated JAVA files contain code to create an XML representation of the data contained in JAVA objects (marshalling) or to read the XML representation and instantiate the JAVA objects (unmarshalling).

AQUILA stores configuration data in XML format as a binary attribute in an LDAP database. An AQUILA utility package has been created to wrap the LDAP database access and the JAXB unmarshalling procedure into an easy-to-use object store, which uses an LDAP path (DN = distinguished name) to retrieve the XML data and to unmarshal the JAVA objects from that. Applications can the easily use the objects to access the data. Internals of the underlying technologies are not exposed.

Additionally, applications can choose to be informed, when data in the LDAP database changes. They can the retrieve the new data and reconfigure themselves during runtime.

As the LDAP database does not know about the XML structure of the content, notifications from the LDAP server cannot tell, what part of the data has changed. The JAXB generated classes however contain an "equals"-method, which can be used to compare XML trees based on their content and find out, what has changed. If the XML tree contains references generated from an IDREF attribute in the DTD, then also the referred sub-tree is compared during this process.

In general, the early-access release of JAXB used in the project worked well for our purposes. Some already known errors and problems could be circumvented.

3.2 Interoperability of JAVA software technologies

3.2.1 Description of used software technologies

Within the project, a set of technologies was used:

- JNDI (JAVA Naming and Directory Interface) to access the LDAP database and the CORBA naming service
- CORBA for communication between the AQUILA executable components
- JAXB for translation of XML data into JAVA objects

3.2.1.1 JNDI

JNDI is a common interface to multiple naming and directory services. So-called service providers are used to perform the mapping from the common interface to the specific directory technology. Currently service providers are available for:

- LDAP
- COS Naming
- RMI Registry
- NIS
- DSML
- DNS
- File System
- Novel NDS

- Windows Registry

The AQUILA project uses the first two of them, which are already contained in JDK1.3.1.

For LDAP, JNDI is used to create, modify and delete objects in the LDAP database. For COS Naming (CORBA Common Object Services), JNDI is used to bind and lookup CORBA objects in the name server.

3.2.1.2 CORBA

The CORBA technology is used for communication between distributed objects, possibly written in different programming languages. CORBA objects are defined in a specific interface definition language (IDL). An IDL compiler maps this definition into the language-specific server and client stubs used to transport object method invocations.

AQUILA used the CORBA technology also to describe interfaces between objects in the same process. This has been done, because the use of IDL as an interface definition language allows the specification of interfaces in a very early project stage, where deployment and distribution of the entities is still not decided. Later on, the objects can be grouped into processes without affecting the interface definition.

3.2.1.3 JAXB

As described before, AQUILA stores configuration data in XML format. To access this data, JAXB unmarshalling is used, which translates the XML data into a tree of object instances.

3.2.2 Parallel use of multiple technologies

The use of multiple technologies in parallel can cause compatibility problems. The main source of such problems was the different views of what an “object” is:

- In JAVA, an object is simply the instantiation of a JAVA class.
- In LDAP, an object is an entry in the LDAP database, based on the LDAP schema definition.
- JNDI already allows translating this into JAVA objects and to store and retrieve serialised objects. However, not all LDAP objects are accessible in this way, and so an additional view is offered, which allows individual access to all attributes of an LDAP object.
- In CORBA, interfaces must be described using the IDL interface definition language. A CORBA object is the instantiation of such an interface.
- JAXB objects are pure data objects, corresponding to a DTD element and its attributes and content.

3.2.3 JAXB objects via CORBA

Especially problematic is the interoperation of JAXB and CORBA. Both technologies require, that their objects are described in a language different from JAVA: CORBA uses IDL as description language, while JAXB generates objects out of a DTD specification. The generated files are not compatible with each other.

For the development of the AQUILA software, we have tried to avoid such problems by separating the application of the technologies:

- CORBA and IDL is used only for communication
- JAXB and XML/DTD is used only for configuration

In some special cases, JAXB generated objects have to be transported via CORBA, e.g. the application profiles towards the clients of the EAT API. Due to the above-mentioned incompatibilities, this is not a simple task.

Usually, all data entities (structures, enumerations, value types) that are foreseen for CORBA communication are specified in IDL. The IDL-to-Java compiler then generates Java classes that automatically implement the interface **org.omg.CORBA.portable.IDLEntity**, which is an extension of **java.io.Serializable**, in order to mark them for CORBA marshalling/transport/unmarshalling.

However, data classes that are generated via JAXB from a DTD do not automatically implement this interface not even **java.io.Serializable**. Also the CORBA type “any” cannot directly be used, since it also expects the implementation of **java.io.Serializable**.

The first solution was to create new subclasses of the JAXB root class that additionally implement **java.io.Serializable**. In this case, it is possible to use the IDL type “any” locally at Java-level, i.e. the interface for retrieving the JAXB data is specified in IDL but used without CORBA. This approach is used in the packages service and **tc**, for instance.

When CORBA is used to communicate the JAXB data in this manner, only the root object of the JAXB tree (that implements **java.io.Serializable**) is transmitted. The underlying objects are null, because they do not automatically implement this interface.

Thus, a second solution had to be found: Since the JAXB objects are created from XML data stored in the LDAP database, the XML byte stream can directly be forwarded to the client. The client is now able to create the JAXB objects by itself by simply calling the method ”unmarshal” of the JAXB root class and using the retrieved byte array as input stream.

The first solution provides rather good performance. However, it breaks with the CORBA idea of having programming-language independent interfaces, as the objects wrapped in the CORBA Any are native JAVA objects. Additionally, it requires, that all objects in the data structure implement **java.io.Serializable**. For that reason we used this solution only, when CORBA interfaces were used for communication within a single process and with simple data structures.

The advantage of the second solution is a clear design. This comes to the cost of a longer runtime, as marshalling and unmarshalling are time-expensive operations. For that reason, we used this solution for inter-process communication in situations, where runtime is not so critical (e.g. start-up).

4 Mechanisms

4.1 Event notification

In distributed software systems, event notification is a key feature for inter-process communication. The AQUILA project therefore designed and implemented two basic event mechanisms, which are used in all components:

- The common event mechanism allows for the definition of individual events, which can be sent by an event source to a set of event observers.

- The keep-alive mechanism checks for the availability of remote objects and informs so-called **AliveObjects**, when the communication partner has died.

Both mechanisms cover different needs and are implemented independently of each other.

4.1.1 Common event mechanism

The common event mechanism implemented in the **util.event** package allows for the definition of individual events, which can be sent by an event source (“**Subject”**) to a set of event observers (“**EventObserver”**). This specification will not contain a detailed description of this mechanism. However, some experience will be given.

Events in the common event mechanism are specified as CORBA **valuetypes** in an IDL file. All events are derived from a root called “**GeneralEvent”**. Similar to the Exception mechanism in JAVA, a hierarchy of events can be defined.

The interfaces used to register an **EventObserver** and to trigger an event are also defined as IDL. This allows using the event mechanism within a process as well as between processes and even platforms.

An event distribution mechanism allows a single event to be sent to several **EventObservers**. Events can also carry accompanying information, which may be used by the **EventObserver** to get more detailed information about the event.

4.1.2 Keep-alive

Actions like login and reservation set-up establish associations between objects in different processes: For example, if the EAT establishes a reservation at an ACA, the ACA creates an object representing this reservation and returns the reference to this object for later use. In the following time, the EAT assumes, that this reservation object is alive and still exists.

If however the ACA fails later on, this assumption is no longer true. The keep-alive mechanism is used to inform the EAT of that case.

The keep-alive periodically sends hello messages to all servers, to which such associations exist (here: from the EAT to the ACA). If there is no response to a hello message within a certain time, then the server is considered dead and an information is sent to all affected objects within the EAT.

4.2 Pluggable algorithms

The set of algorithms proposed in the AQUILA context, serves two cases:

- For resource control: several algorithms may be used to calculate the amount of additional bandwidth to be requested or the size of the resource cushion to be kept while releasing resources. These are called "**rcrules**".
- For admission control: AC algorithms and the way parameters used for AC calculations are retrieved, may vary. Especially, a declaration based admission control (DBAC) and/or a measurement-based variant (MBAC) may be used.

Both are regarded as pluggable algorithms, which can be configured through the network management platform (QMTool). Each pluggable algorithm (for resource control and admission control) is actually consisted of a number of different implementations/variations. Concerning the configuration of the pluggable algorithms, a parameter is involved, which names the actual instance of the variation of the algorithm used. The JAVA based implementation makes it easy to instantiate the named classes from the configured names during start-up.

The corresponding packages (**rcrule** and ACA) include a class (**RcRuleImpl** and **InterfaceCtrl** respectively), which acts as a proxy. These classes instantiate the objects, which actually implement the different algorithms, at runtime calling the **Class.forName** method. The arguments for the instantiation of the object are retrieved from the configuration data (an XML file in our case). Having a uniform and well-defined interface, new algorithms could be implemented, namely "plugged in". Their relation with the components that are going to use them is succeeded with the configuration files. For instance, the configuration file for the **objectX** includes a parameter **rcruleName** = *RcRuleSet1*; *RcRuleSet1* is the name of the class that implements the desired algorithm.

This approach facilitates the usage of a wide range of algorithms with minor modifications to the existing software.

4.2.1 Admission control algorithms

An important aim for the implementation of the admission control was to support so much flexibility for the trials as possible. In this context, pluggable algorithms have been required, and the implementation has been done as described above. Therefore, an easy mechanism is offered to test different algorithms in different scenarios. The switching between the algorithms can be done only by a modification of the configuration data. In addition, the adding of new algorithms is easy and can be supported without modification of any existing source code.

The admission control algorithms need a lot of parameter in order to calculate the new needed bandwidth amount. These parameters depend on the interface information and have to be specified for each router interface on that the admission control algorithms have been activated. These also give the operator a great flexibility to tune the algorithm. But it should not be hidden the disadvantage of this benefit, because a detailed knowledge of the algorithms is needed and a lot of different configuration data have to be configured at first for each admission controlled interface.

All admission control algorithms have to implement a small interface that only offers the possibility to retrieve the calculated values

Ti: the injected traffic by this traffic class,

Bi: the bandwidth requirements of this traffic class,

Ri: the traffic that is generated by this traffic class.

Therefore, the functionality of each algorithm is capsulated into one class and an easy plug in of new algorithms is possible.

The three retrievable values are then the basis for the next step in the admission control procedure, the QoS constraints and the Policy constraints check. Thus, a well-defined interface is designed to bind the algorithms to this admission control component.

The class names of the algorithm are fixed listed in the related DTD file and not freely editable. The benefit is that no misconfiguration is possible, but the DTD file has to be changed if the class names have to be renamed or new algorithms are additionally offered.

The inter-working between the algorithms and the traffic conditioning settings (that calculates the token bucket rate) is difficult to understand and improvable, because the algorithms use as unit [bit/sec] and the conditioning works with the unit [packets]. In addition, no clear transformation matrix between these calculation areas has been defined. Also often specific traffic parameter are used like round trip time, that depends on the destination address, and specifiable by the user for the requested network service. This seems to be difficult, because the user and or the used application profiles don't know this value and the impreciseness is difficult to calculate.

4.2.2 Resource control rules

The purpose of this section is to present the resource control rules used in the second trial focusing on the details concerning the implementation and integration of those algorithms in the implementation context.

The resource control rules are related to the handling of network resources. In particular, different rules are implemented for different components and more specifically for the ACA Resource Component (RC), the Resource Control Agent (RCA) and the BGRP Agent.

Those **Rules** should be able to answer the following questions:

1. ACA: Should I ask/release resources from/to the Resource Pool structure?
2. ACA: How much bandwidth has to be released to the Resource Pool structure?
3. RCA: Should I ask/release resources from/to the above Resource Pool?
4. RCA: How much bandwidth has to be released to the above Resource Pool?
5. RCA: How much additional bandwidth should I give to the requester? The requester could be either the ACA or a lower Resource Pool.
6. BGRP: Should I release resources?
7. BGRP: How much bandwidth should be released?

In order to provide a design model which could be re-usable and extendable with new recourse control rules, the following approach was followed, where 3 different rules are defined, one for each component. Specifically RcRuleSet1 is defined for the RCA, RcRuleSet2 for the ACA and the RcRuleSet3 for the BGRP.

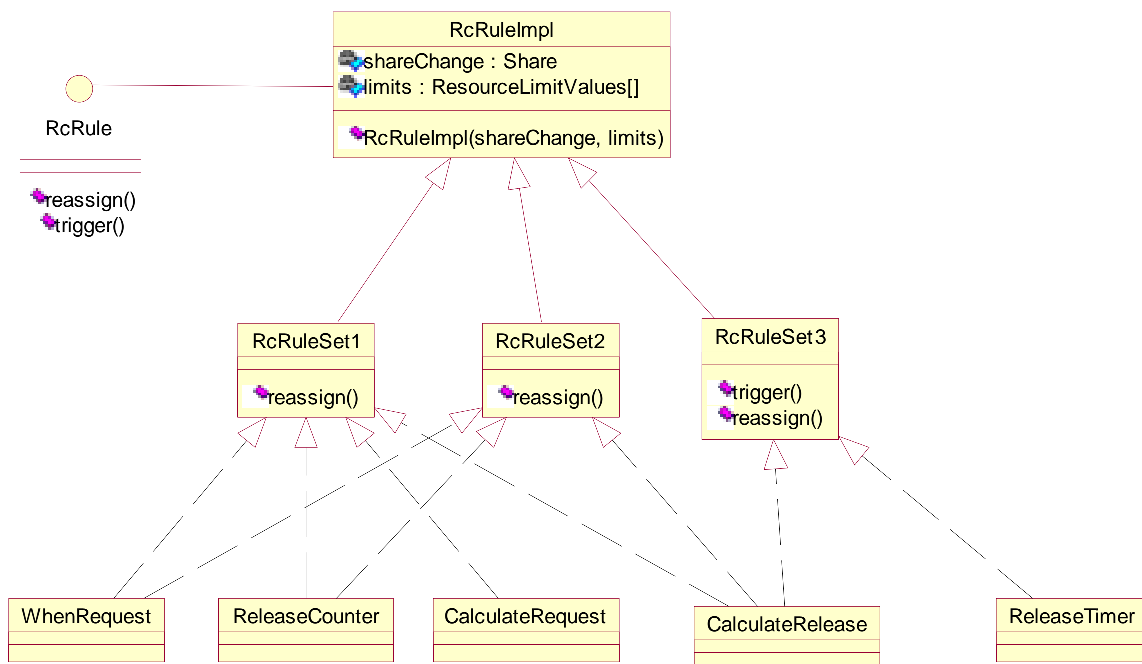


Figure 4-1: Design model of Resource Control Rules package

Even if three different rules are created, the questions 2, 4 and 7 are realised by the same function (***CalculateRelease***). From the other hand, the questions 1 and 3 are realised by the same function concerning the release part (***ReleaseCounter***). The same applies for the request

function (***WhenRequest***). Moreover, question 6 is realised by the ***CalculateRequest***, while the question 6 by the ***ReleaseTimer***.

In the sequence, the realisation of the different functions is explained.

4.2.2.1 Parameters to be configured for each function

For the ***ReleaseCounter*** and the ***CalculateRelease*** the following parameters should be defined:

> ***blockSize***: amount of free resources
> ***counter***: number of requests

For the ***CalculateRequest*** function the following parameters should be defined:

> ***Amed***: a multiple for the additional resources to be assigned
> ***wLow***: a low watermark for the free resources

4.2.2.2 RCA component – RcRuleSet1

Concerning the RCA the following rules apply:

WhenRequest: if ***(TotBW*** < ***currentReserved*** + ***new_Request)*** then request from the above level to assign at least a new ***TotBw*** = ***currentReserved*** + ***new_Request***.

ReleaseCounter: In order resources to be released to the upper level two conditions should apply: the number of so far made reservation requests should be more than or equal to the ***counter*** parameter (***number_of_resource_requests*** <= ***counter)*** and the amount of current free resources should be more than or equal to the amount of resources determined by the ***blockSize*** parameter (***free_resources***< = ***blockSize).***

CalculateRequest: In case a Resource Pool should assign a new ***TotBW/ACLimit*** to the lower level RP/ ACA, it should calculate this new ***TotBW/ACLimit***. Therefore the following actions are performed:

AdditionalBW = newTotBW – TotBw;
*If((Amed)*additionalBw)<= freeBw*(wLow))*
    *newBw = (Amed)*additionalBw;*
else
    newBw = additionalBw;

In this way, a multiple of the requested resources may be assigned, under the conditions that an appropriate amount of free resources is available. The objective is not to assign the total amount of free resources, but to keep them in order to serve future requests.

CalculateRelease: This function calculates the bandwidth to be released, which is equal to the corresponding ***blockSize***.

4.2.2.3 ACA component – RcRuleSet2

Concerning the ACA, the ***WhenRequest*** is based on the same concept with the one that has been described in the above paragraph. If a new request cannot be accommodated under the current assigned ***ACLimit***, the ACA specifies the new appropriate ***ACLimit*** and requests it from the above Resource Pool.

Moreover, the ***ReleaseCounter*** and ***CalculateRelease*** functions follow the same concept as for the RCA.

4.2.2.4 BGRP component – RcRuleSet3

Concerning the ***CalculateRelease*** function of the BGRP is the same with the one defined above. The only difference is the question "when to release resources". This is actually answered by the ***ReleaseTimer*** function. The BGRP agent calls the ***trigger*()** function of the RcRuleSet3 every *Release Period, RP*. If between sequential calls of this function an amount of resources equal to ***blockSize*** is free, then the ***CalculateRelease*** function is triggered.

4.3 Tracing

During all the phases from offline tests, through integration until the trials, trace information produced by the software components provide valuable means to find errors and to evaluate results. In order to produce useful information, a trace facility should fulfil the following conditions:

- Trace information should clearly state their origin
- Trace information should be accompanied by a severity level (informative, warning, error, fatal)
- Trace information should be time-stamped
- Time stamps should be consistent over multiple computers
- For selective tests, trace information should be able to be filtered
- Tracing should not affect runtime too much

Not all conditions can be fulfilled at the same time. Especially runtime measurements could be affected by very detailed trace information. So the user should be allowed to select between the full range of trace information and a small, specific set.

Within the AQUILA project, a trace utility was developed, which offers a uniform interface for all applications. Trace information is divided into two classes:

- log messages, accompanied by a severity level

- debug messages, which are used to trace specific operations (e.g. start of a thread). A special debug message is also used for time measurements.

All AQUILA components contact a common trace server. This server is used for two purposes:

- It provides a graphical user interface, which controls the detail of the trace messages. For log messages, a severity threshold can be specified. Debug messages can be switched on and off individually for each type.

- It collects trace information from all attached AQUILA components and writes them into a global trace file.

Global trace information is especially valuable in a distributed environment. Time synchronisation between the hosts running the AQUILA software guarantees consistent timestamps. In the global trace file, it is easy to follow the sequence of operations performed in each component.

When writing the source code, care should be taken on two points:

- Where to insert trace calls. Important functions should write a trace message at the start and at each possible end (error cases and non-error cases)

- What information to provide. The information must be sufficient to allow the reader of the trace file to understand the sequence of operations. For many procedure calls, it is valuable not only to state, that this procedure is now entered, but also to log the most important parameter values. For errors, as much information as possible should be provided.

The AQUILA trace facility allowed us to perform already most parts of the integration without additional debuggers. The trace information was in almost all cases sufficient, to identify and correct any problems.

5 Abbreviations

A

ACA	Admission Control Agent
API	Application Programming Interface

B

BGRP	Border Gateway Reservation Protocol

C

CORBA	Common Object Request Broker Architecture

D

DBAC	Declaration Based Admission Control
DTD	Document Type Definition

E

EAT	End-user Application Toolkit

G

GUI	Graphical User Interface

I

IDL	Interface Definition Language
IDOM	Inter-domain layer

J

JAXB	JAVA API for XML Binding
JNDI	JAVA Naming and Directory Interface

L

LDAP	Lightweight Directory Access Protocol

M

MANAG	Management Layer
MBAC	Measurement Based Admission Control

Q

QMTool	QoS Management Tool
QoS	Quality of Service

R

RCA	Resource Control Agent
RCL	Resource Control Layer

S

SIP	Session Initiation Protocol

U

UML	Unified Modelling Language

X

XJS	XML to JAVA Schema
XML	Extensible Mark-up Language