The Push Model in a Java-Based Network Management Application

Laurent Bovet

Swiss Federal Institute of Technology
Institute for computer Communications and Applications

Diploma project

Professor: Jean-Pierre Hubaux
Assistant: Jean-Philippe Martin-Flatin

March 29, 1999

Contents

List of Figures

Acknowledgments

This diploma project took place at the Institute for computer Communications and Applications (ICA) of the Swiss Federal Institute of Technology in Lausanne (EPFL). I wish to thank Professor Jean-Pierre Hubaux and Jean-Philippe Martin-Flatin for giving me the opportunity of such an interesting project. I am also grateful to Raju at AdventNet for his pieces of advice and bug fixes, Ed Jordan from Chronos.Net for helping me retrieving documentation about MIME Multipart, Peter4 on #linux IRC channel for assistance in Linux Java debugging, and to all Usenet newsgroups contributors who gracefully offered their help on various subjects.

Chapter 1

Introduction

1.1 Outline

The goal of this project is the realization of a network management architecture proposed by Jean-Philippe Martin-Flatin [1]. It aims at validating a novel approach based on Web technologies and the push model.

The scene of IP[1] network management is nowadays in constant evolution. The traditional way of managing network devices on an IP network is SNMP for the protocol and SMI for the standard data representation [18, 3]. Inspite of its poularity, SNMP suffers from several shortcomings such as high management data traffic or lack of data encryption,

As Internet grows and SMEs become aware of the added value of intranet solutions and current Network Management Platforms (NMP) need expensive Unix configuration, new techniques have been proposed in IP network management that rely on Web technologies [21, 22]. Some of the big companies playing a role in the scene of network management address these questions and propose new solutions and contributions with the technologies provided by the Java language. Sun invested a lot of efforts in this field, especially with its JDMK (Java Dynamic Management Kit) [20]. This confirms the importance of new generation network management platforms.

Our goal here is tomake a proof of concept by implementingthe main aspects of Martin-Flatin's work. The result is an application called JAMAP (JAva Management APplication).

1.2 State of the Art in Java-Based Management

We investigate here the Java-based libraries and programming frameworks publicly available on the Internet, and usable for building Network Management platforms.

1.2.1 AdventNet SNMP Release 2.1

This library proposes Java classes and beans providing access to all levels of the SNMP protocol, from beans containing a complete MIB browser to constructors for PDUs and varbinds. It supports RMI and CORBA as high-level communication layers [7].

Seven packages are provided:

snmp2 The AdventNet low-level SNMP API. The classes available in this package implement snmp communication and snmp variables defined in ASN.1, like SnmpOID, SnmpInteger, etc. There is a Java interface available in this package to implement callback, authentication etc. The usage of this low-level API is recommended in embedded applications where code size limitations apply.

[1] For acronyms, please refer to glossary on page 52.

beans The AdventNet SNMP beans package. This package consists of AdventNet SNMP Java Beans components that can be imported into any builders. Building SNMP applications using the AdventNet SNMP beans package is the easiest way to build management applications.

sas The AdventNet SNMP Applet Server package. This package provides classes to facilitate communication between network management applets and managed devices where direct communication is prohibited due to the applet security policy. This package also includes some interfaces to achieve custom extensions .

mibs The AdventNet SNMP MIB package. The classes available in this package provide all the MIB handling support like MIB loading, unloading etc.

ui The AdventNet SNMP User Interface package. This package consists of the AdventNet SNMP UI beans like the MibBrowser bean, TableBean etc. These AdventNet beans components can be imported into any builder.

rmi The AdventNet SNMP RMI package. This package consists of Java interfaces that facilitate communication using Remote Method Invocation (RMI) from remote clients to perform SNMP operations.

corba The AdventNet SNMP CORBA package.The classes and interfaces in this packages provides Common Object Broker Architecture access to AdventNet SNMP API.

1.2.2 Sun's JMAPI

JMAPI was originally an API implemented and distributed by Sun as an extension to the Java programming language [19]. At the beginning of this diploma project, version 0.8 was a well-known management API. The version 2.0, released in december 1998, turned it into an open API specification for industrials. The original JMAPI provided the following features:

- Managed Object Representation as an abstraction of underlying communications and characteristics.
- Management server between manager applet and agent.
- RMI-based communication.
- Compiler provided to create Managed-Object representation.
- Event management.
- JDBC support for persistent Managed-Objects.
- SNMP support.
- A few visual components extending the standard AWT set.

1.2.3 Sun's JDMK

At the time of writing, Java Dynamic Management Kit 3.0 is Sun's leading management product [20]. Note that it is the basis of the JMAPI 2.0 specification. Its key features are:

- A dynamic management architecture; he Java Dynamic Management runtime provides a software backplane that enables management services to be plugged in dynamically and propagated across the network.
- A library of reusable, generic management services in the form of JavaBeans components, including object repository, dynamic class loading, dynamic native library loading, relationship, basic notification, filtering, monitoring, cascading, authentication, scheduler, discovery, and launcher.
- Service creation tools, which include:

  - A Managed-Object GENerator (MOGEN), which helps developers create client applications with automatically generated remote-access modules that handle communication and ensure protocol transparency.
  - A Java SNMP MIB Compiler, which takes an SNMP MIB as input and outputs a JavaBeans component. This enables Java Dynamic Management agents to be managed by an SNMP manager, and allows Java Dynamic Management managers to access legacy SNMP devices.

- Security features:
  - Authentication (simple login / password or CRAM-MD5-based challenge).
  - Support for SSL (Secure Socket Layer).
  - JAR (Java ARchive) file signatures.

- Simple Network Management Protocol (SNMP) features:
  - SNMP Manager API.
  - SNMP MIB code generator.
  - SNMPv2 protocol support.

Chapter 2

Problem Statement

2.1 Limitations of SNMP

As we stated in the introduction, IP Network Management is evolving, and even if SNMPv3 addresses many technical issues (security issues, for example), the need for better solutions such as Web technologies can arise for several reasons exposed in [2].

As networking now concerns not only big companies, but also SMEs and even individuals, the price of NMPs becomes a real concern. Current NMPs are still expensive and usually require a Unix environment. Opening Network Management to the Web world allows virtually anyone to manage its network from home using only a standard Java-enabled Web browser. The impact on cost of management software seems clear.

SNMP provides some basic operations (get, set, getnext, ...) that are sufficient in many situations. As managed networks see their number of nodes increase, the amount of management traffic can become an important proportion of the total traffic. Moreover, the BER encoding used in SNMP is renowned as inefficient and the varbind lists are expensive because the OIDs used to name variables take much more place than the values.

Security issues are partially addressed by SNMPv3, but the encryption of data sent through untrusted network branches is still missing.

In matter of protocol, HTTP is the main candidate as it allows for persistent connections (in its 1.1 version) through which pipelined request and responses can be sent. In the field of security, HTTP supports SSL, a layer between transport and application layers that prevents eavesdropping, tampering and message forgery. HTTP is gaining from its popularity due to the Web.

In matter of information model, MIB data could be expressed in a language inherited from a flexible generic data representation such as XML, which brings extensibility to multimedia documents.

Moreover, SNMP suffers from its underlying protocol UDP. Even if TCP could be used, everyone uses UDP. This protocol is unfortunately less reliable than TCP as it does not use acknowledgements and automatic retransmission.

2.2 Push vs. Pul

Martin-Flatin gives an illustration explaining these models[2].

In software engineering, we know two approaches for exchanging data between two distant entities: the pull model and the push model. The newspaper metaphor illustrates these models: if you want to read your favorite newspaper everyday, you can either go and buy it every morning, or subscribe to it once and then receive it automatically at home. The former is an example of pull, the latter of push. The pull model is based on the request / response paradigm (called data polling, or simply polling, in traditional SNMP-based network management); the client sends a request to the server, then the server answers, either synchronously or asynchronously. This is functionally equivalent to the client “pulling” the data off the server. In this approach, the data transfer is always initiated by the client, i.e. the manager. The push model, conversely,

is based on the publish/subscribe/distribute paradigm. In this model, agents first advertise what MIBs they support, and what SNMP notifications they can send; the administrator then subscribes the manager (the NMS) to the data he/she is interested in, specifies how often the manager should receive this data, and disconnects. Later on, each agent individually takes the initiative to "push"data to the manager, either on a regular basis via a scheduler (e.g., for network monitoring) or asynchronously (e.g., to send SNMP notifications).

Chapter 3

Push techniques

3.1 Communication in Distributed Applications

Three technologies are considered for the communication [2]: sockets, Java RMI and HTTP.

3.1.1 Sockets

The agent-manager push communication through a dedicated protocol based on TCP/IP sockets may be of two kinds: one TCP connection per pushed data sent, or a persistent connection through which many pushed data can be sent. The former solution implies a network overhead caused by multiple connection setups and teardowns. The latter limits the number of agents connected to the manager. The upper bound is the maximum number of sockets open at the same time (typically several thousand on Unix systems). Both solutions demand a dedicated port number for the TCP connections.

3.1.2 Java RMI

This object-oriented technology offers a high-level approach of distributed systems. It implies multiple connections for different method calls and, like sockets, needs a dedicated port. It benefits from the advantage of the protocol abstraction and the integrability in existent frameworks. It is maybe too heavy for small devices as it requires more resources than a light-weight JVM such as Sun's EmbeddedJava JVM would.

3.1.3 HTTP

This is the standard protocol of the Web. It has many interesting features such as MIME typing, compression and security layers. In an architecture using servlets and applets, the communication through HTTP solves the problem of firewalls. Indeed, all the communication is considered like any Web browsing traffic. For SMEs not having expertise or that can not afford external consulting to setup their firewalls, this solution is very inexpensive, thus interesting. As the network devices become more sophisticated and some already have embedded Web servers, the servlet system footprint is not huge on the agent's side. HTTP is therefore the communication technology chosen for this project.

3.2 Orientation of Connections

In the push model, the agent needs a communication path to send its data to the manager. Depending on technologies and engineering considerations, two scenarios are possible for opening the connection between the peers: either the agent or the manager can open the communication path as described later.

3.2.1 Agent-initiated push

In this scenario, the agent opens the stream to the manager. This implies that it knows the address of the manager in advance, and must handle by itself the possible connection failures. In the case of HTTP, this requires the manager to have a Web server which is impossible for applets because they have not the privilegeto use the standard HTTP port.

3.2.2 Manager-initiated push

This scenario implies that the agent acts as a server and the manager connects to it to retrieve the pushed data through a persistent connection. This saves CPU cycle on the agent by as it tries to serve managers only if they are connected. It also addresses the connection failure issue as this is manager job to reconnect and generate notification events relating connection problems. Agents act passively in terms of connection handling. We chose this model for our JAMAP application.

3.3 MIME Multipart

HTTP uses MIME as a standard message format. MIME has a recursive definition [4] allowing a message to contain several nested parts that are themselves MIME messages. The Multipart type delimits parts with a boundary string. This format is suitable for a persistent stream where pushed data are sent sequentially [12]. A typical Multipart message looks like this:

```
Content-type: multipart/mixed;boundary=ThisRandomString


--ThisRandomString
Content-type: text/plain


Data for the first object.


--ThisRandomString
Content-type: text/plain


Data for the second and last object.


--ThisRandomString--
```

3.3.1 Compression

HTTP allows the use of a Content-encoding field in messages. Two compression encodings are supported: gzip and unix compress. Out interest is to compress parts containing a large amount of information to minimize management data traffic.

3.3.2 Persistent connections and timeouts

HTTP/1.0 was designed to use a separate connection for each object retrieved from servers. These connections should be open a few seconds. This allows Web servers to serve thousands of clients without being out of available sockets as they are freed when the connection is closed. In the case of IP network management, agents typically have only one manager connected to themn or at worst a few managers; so there is no need

to reclaim idle sockets. HTTP/1.1 introduces persistent connections to handle multiple request/response through the same connection, saving thus the overhead of TCP connection setups and teardown. We propose to use endless HTTP/1.1 persistent connections to implement a push communication path.

The question here is to control the life-time of the connection. No prescription is made in the specification of HTTP. We can rely only on recommendations and current implementations of servers, clients and proxies. Concerning timeouts, TCP stacks usually set the socket idle timeout to two hours for incoming data. HTTP servers are usually configurable (e.g Apache 1.3.4 and Jigsaw 2.0) generally support several hours of connection. Proxies are sensitive nodes of HTTP connections and are often configured to reset connections after a few minutes.

As a result, it seems necessary to design a reconnection mechanism to avoid loss of contact with the agent. One just has to ensure that the maximum timeout of the elements on the connection path (client, proxies, server) is longer than the push period.

Chapter 4

Java Advanced Technologies

4.1 Servlets

Java servlets are an improvement on CGI scripts. Servlets are Java classes loaded in a JVM besides the Web server. This Web server must be configured to use servlets and associate an URL with each loaded servlet. At startup time, one servlet object is instantiated for each configured servlet. When a request is performed on a servlet URL, the server invokes a method of the servlet depending on the HTTP method used in the request. The servlets implement one method per HTTP method, e.g the method `doGet` is invoked when an HTTP GET request is done on the servlet URL. As concurrent accesses are usually possible on a server, several clients may invoke concurrently the same method of the same servlet thanks to multi-threading. This allows multiple persistent connections to share the same servlet.

Servlets benefit of all technologies of Web servers like restricted access, secure transactions and virtual hosting.

At the time of writing, servlet environments are still in constant evolution. During the project, Sun's specifications changed from 2.0 to 2.1 and implementations have not yet reached stable and definitive state.

4.2 Serialization

Serialization allows any complex object to be translated into a byte stream. This form is suitable for network data exchange, and for persistence of object state (e.g in a database or file system). Objects containing references to other objects are processed recursively to serialize all the required objects. The keyword `transient` in object declaration prevents this and tells the serialization system not to serialize the referenced object. Some interface methods can be invoked for special treatments before the serialization and after the de-serialization, e.g. to recreate the transient objects.

In the field of networking, instead of defining a protocol, one can use serializable classes dedicated to communications. A `writeObject` method is called on one side, and `readObject` on the other side. Note that the latter creates new instances of objects, thus requiring the matching classes to be loadable in the JVM.

In the field of persistence, serialization is very convenient to save the state of an application or a single object: you just have to open a *file stream* and write the object to it. The restore phase may raise problems because there is no way to handle class changes. Classes must be definitive before constructing objects, because the saved objects can not be reloaded with new class versions.

We chose to use serialization for this project because of the experimental nature of the work. If the nature of data is precisely defined (by a protocol or a format), misgivings of serialization (slow and resource angry) can be avoided by writing custom parsers and generators. Thus, changing the data representation needs a partial rewrite of these components, which is not the case when using default serialization.

Chapter 5

JAMAP: High-Level Design

5.1 Architecture Overview

The overall architecture of JAMAP is depicted in fig. 5.1. Plain arrows represent interactions between components. Dashed arrows represent logic push communication paths. We will see in the detailed design (section 6.1) how they are implemented.

The management station may be any machine running a standard Java-enabled Web browser.

The management server is typically a server machine. For small networks or for rather light network management, the management server servlets could be installed on the Web server of the organization. By doing so, we can benefit from pre-existent security and access authentication schemes. Note that the network administrator will not need all access privileges on the web server. For heavy management, one can imagine a dedicated machine or even two, as it is not mandatory that both servlets run on the same machine. Note that the splitting of components of the management server allows the construction of more complex distributed systems, since the push communication path is of the same kind between all elements.

The agent is either a network device running an embedded JVM or a helper system (proxy) beside one or more network devices. This can be a low-cost PC.

5.2 Management Station Applets

5.2.1 Data subscription applet

This applet communicates directly with the agent. More than only providing the subscription system for regular management, it is also a tool for ad hoc management. It allows the user to perform the following tasks:

- Browse an information base in a convenient interface.
- Choose a variable or a table and get its value immediately.
- Choose some variables to monitor for a while with different GUI components (text value, time graph or table).
- Choose and subscribe to variables. These will be sent to the management server at a chosen frequency.
- Monitor some computed values such as interface utilization.

Figure 5.2 shows the graphical user interface of the MIB data subscription applet. Figure 5.3 depicts the spy frame with three different monitors.

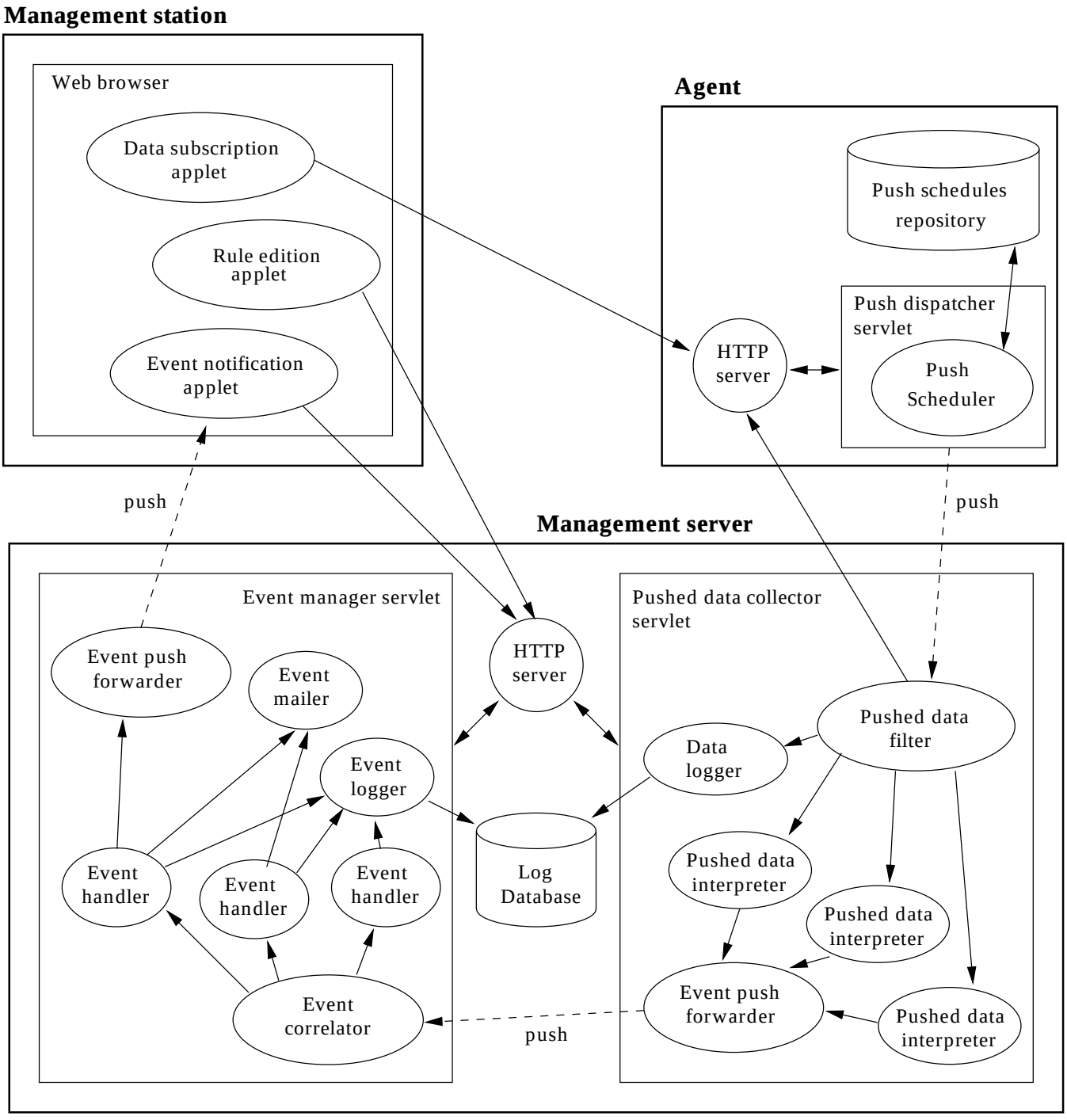

Figure 5.1: Overview of the JAMAP architecture

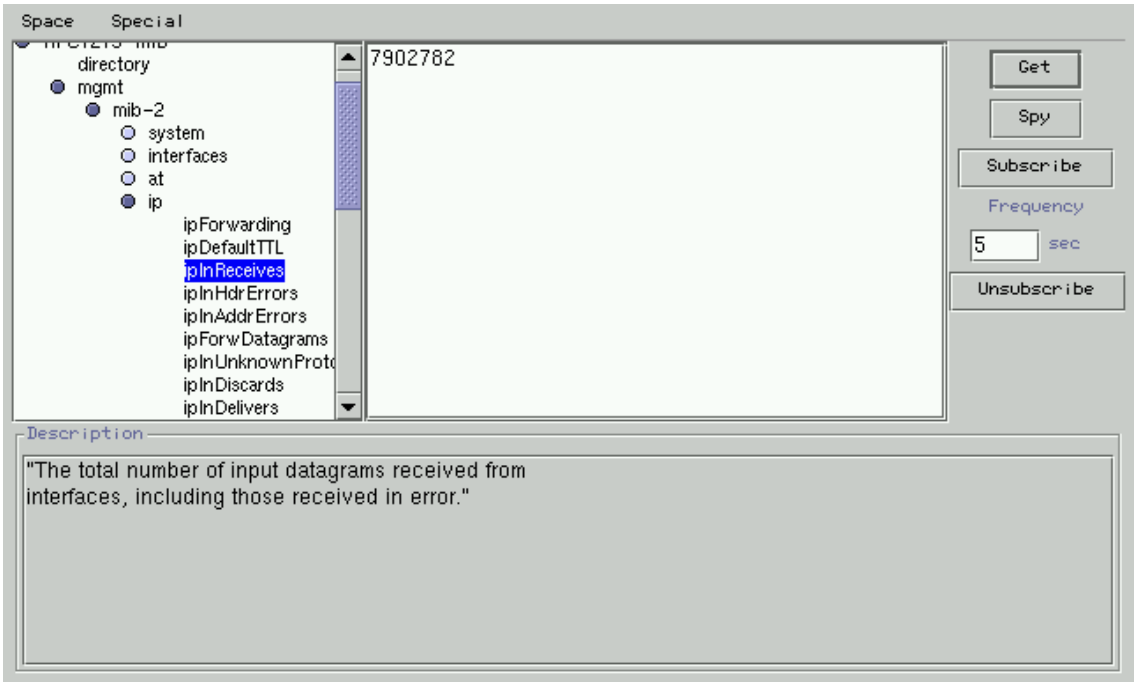

Figure 5.2: The MIB data subscription applet GUI

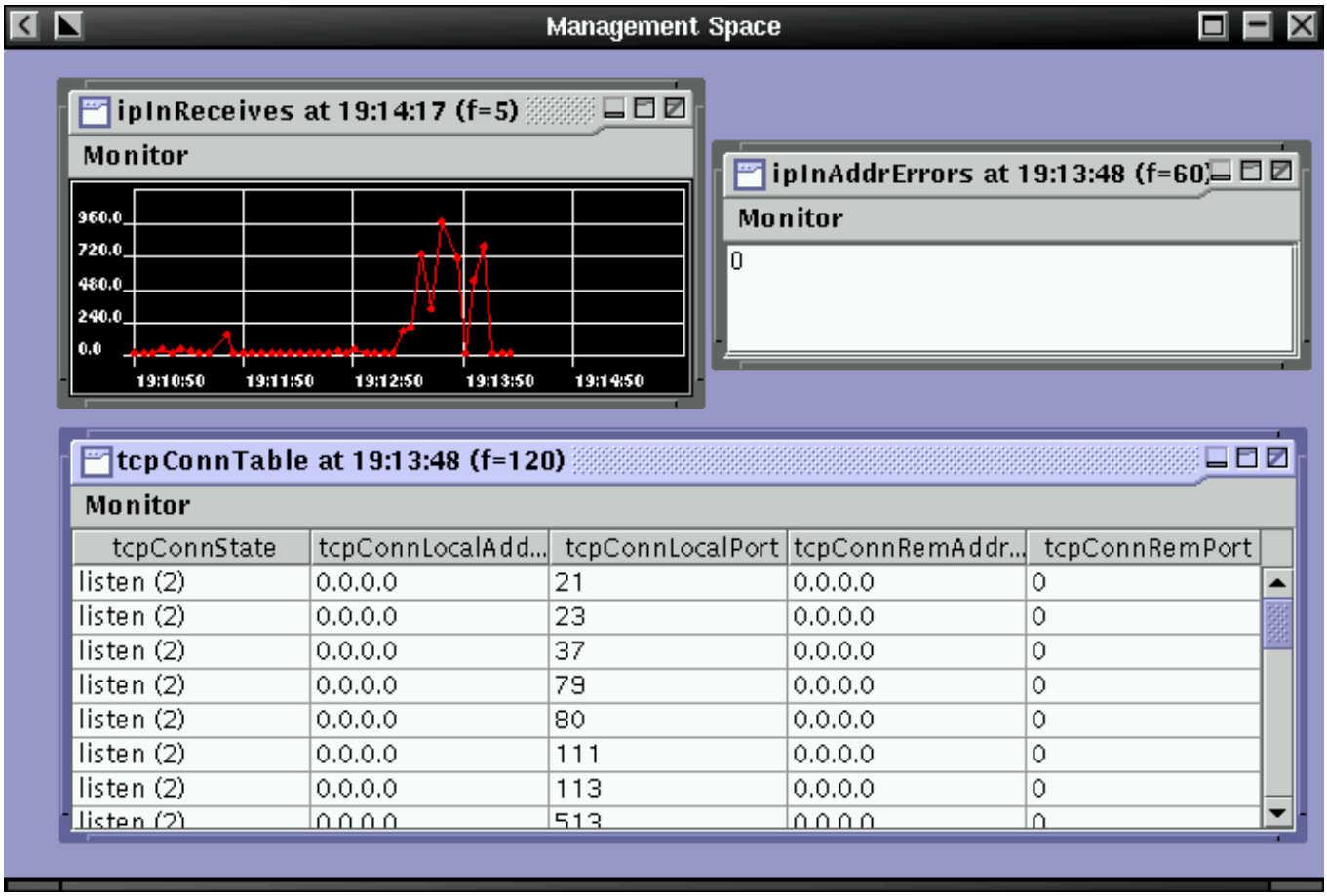

Figure 5.3: The spy frame of the MIB data subscription applet with monitors

5.2.2 Rule edition applet

This applet controls the pushed data interpreter part of the pushed data collector servlet. It allows the user to write rules applied to incoming pushed data to generate events. These rules are written in Java to offer a very wide panel of data interpretation. For example, an event can be generated if a value exceeds a threshold. More complex rules can be easily written, such as calculating if the average of the hundred last pushed values increased of 10 percent in the last hour. The figure 5.4 shows the GUI of the rule edition applet.

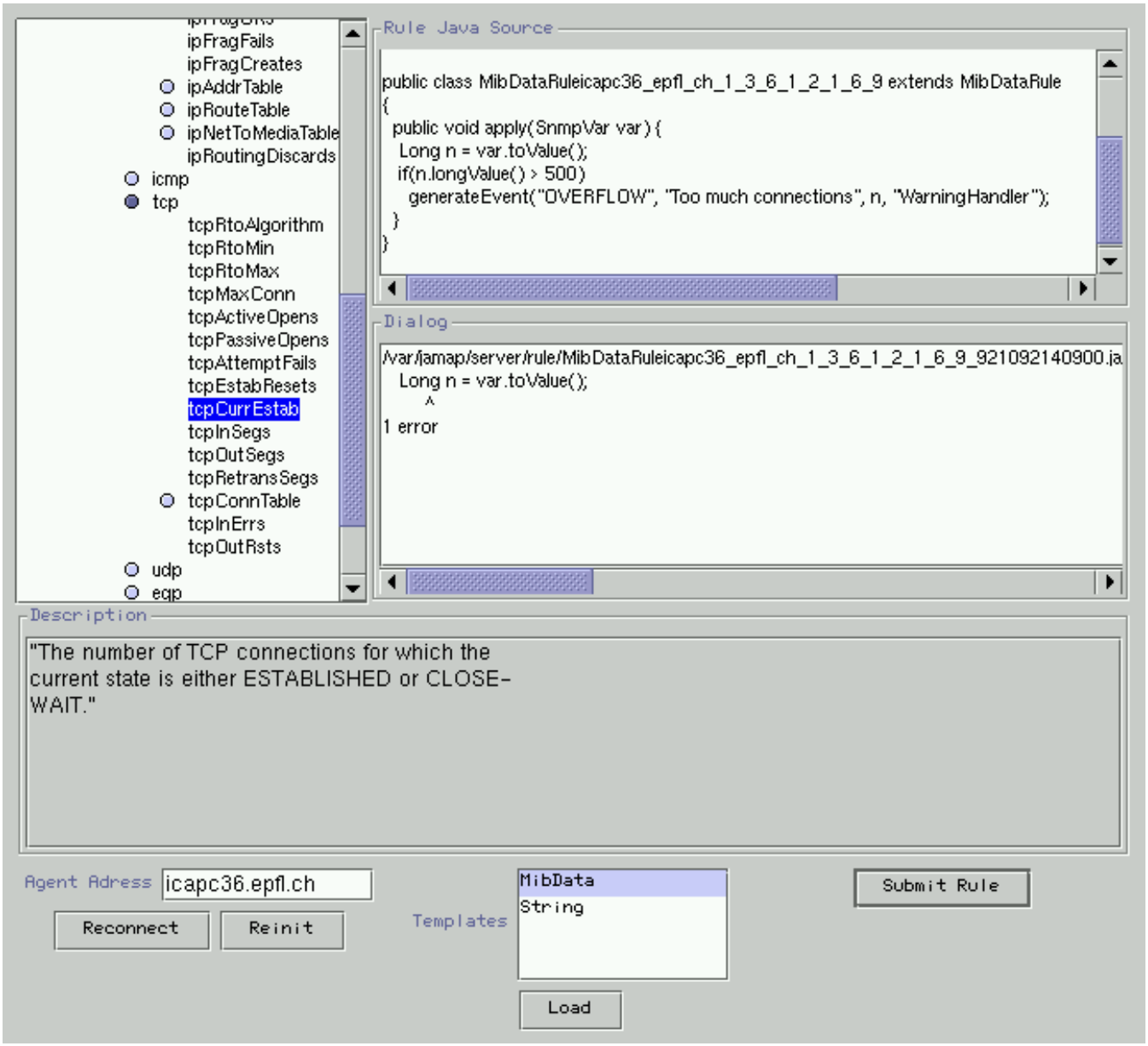

Figure 5.4: The Rule edition applet GUI

5.2.3 Event notification applet

This applet is connected to the Management Server to receive events. It contains a simple list of events and a blinking light and sound system to grab the operator's attention in case of incoming events. It is intended to stay permanently in a corner of the administrator's or operator's desktop screen.

5.3 Management server servlets

5.3.1 Pushed data collector servlet

The pushed data collector servlet connects to an agent and waits for pushed data. The pushed data filter controls the flow of incoming data by closing the connection if too much data is sent by the agent. This is a security feature avoiding the propagation of erroneous data, and protecting the management server from misbehaving agents. An event is sent to notify the wrong behavior of an agent.

The pushed data interpreter applies the rule for each subscribed data, and generates events that are forwarded farther through an event push forwarder.

Data may be subscribed only for logging in order to process them afterwards or perform statistics (data collecting). Thus, they are treated by a logger which interfaces with a data repository (file system or DBMS).

In the case of monitoring, the pushed data is processed immediately, either as it is or after some preprocessing; it can also be logged for further offline processing. In the case of data collection, data is always stored in a data repository (e.g file system or DBMS) and always processed offline.

5.3.2 Event manager servlet

The event manager servlet connects to one or more pushed data collector servlets and waits for pushed events. Events are processed by an event correlator which avoids redundancy in event reporting by dropping deductible events.

The correlated events are then transmitted to different event handlers according to their importance.

This handlers forward them to any kind of notification systems or logging.

5.4 Agent servlet

The agent servlet stores subscriptions sent from a data subscription applet into a local repository. The push scheduler retrieves data form the information base and sends them at chosen frequencies through open push communication paths.

5.5 An Example of Distributed NMP

Figure 5.5 shows a typical distributed Network Management Platform (NMP) showing several actors of each kind. The flexibility of the JAMAP design allows many variations in the interconnection of components. In particular, several pushed data collectors and applets can receive data from the same agent. Similarly, event managers can receive data information from several data collectors. On Figure 5.5, agent *2* sends data to management servers *a* and *b*, and agent *8* receives subscriptions from management stations *I* and *II*. Pushed data collectors *c* and *d* are stand-alone and connected with the event manager on server *b* which runs also a pushed data collector. It is even possible for several event managers to connect to the same pushed data collector (although this case would not be frequent).

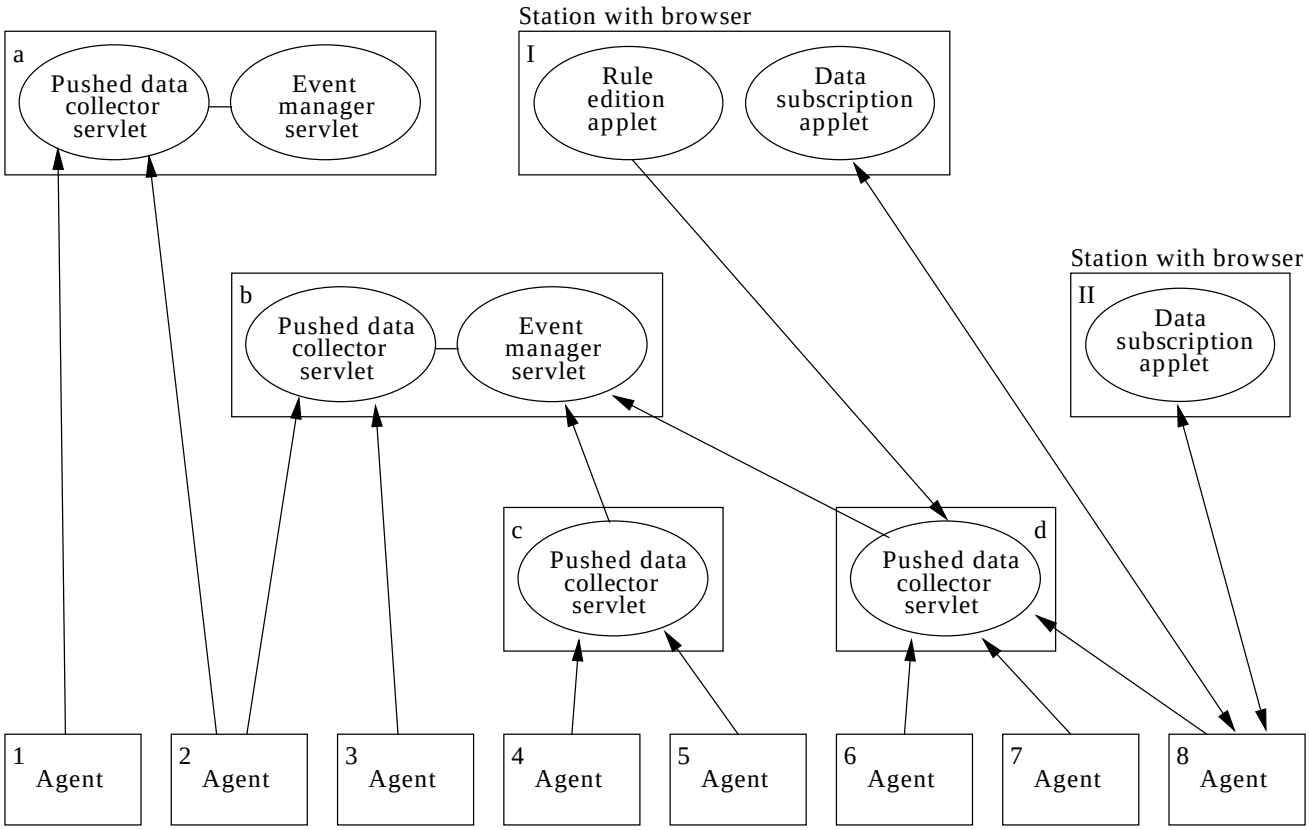

Figure 5.5: An example of distributed management platform

Chapter 6

JAMAP: Detailed design

We adopted the UML graphical notation throughout this section for our design diagrams [6].

6.1 Communication between components

As we saw in the high-level design (figure 5.1), management data and events are transmitted through push communication paths in JAMAP.

The push model assumes the publish / subscribe / distribute paradigm [2]:

1. The publication phase is simply the download of the subscription applet.
2. The subscription phase is realized by sending a subscription from the applet to the agent. In our HTTP implementation, this is done by sending a POST request to the push dispatcher servlet. The body of the POST message contains a serialized object of the class Subscription. Such an object contains an identification of the data to push, and an identification of the recipient the data should be sent to.
3. The distribution phase is the emission of data from the agent through a push communication path. These paths are implemented with HTTP / 1.1 persistent connections described in section 3.3.2. The receiver of pushed data sends an HTTP GET request to the pushing servlet. The response is an infinite stream of MIME parts. The payload of the MIME part message is a serialized object of the class Unit. The atomic pieces of information sent through the push communications path will be referred to as *units* or *push units*. Note that this phase is the same for all the push communication paths depicted in figure 5.1.

The subscribe phase implies an unsubscribe phase to stop the emission of some data. It is also implemented with an HTTP POST request, at the difference that the body of the message contains an object of the class SubscriberID. Such an object identifies a recipient component of pushed units.

The design of pushed unit creation, collection and treatment used throughout the whole JAMAP application is based on *dispatchers*, *collectors*, *distributors* and *final consumers*. Dispatchers creates the units and push them into the communication path. Collectors receive units on the other side of push connections. Distributors forward units to final consumers that treats them. Figure 6.1 shows this model. We stress out that for a given collector, only one final consumer can receive a given unit.

Note that it is a perspective common to all push communications within JAMAP. Each has its own characteristics but all use this model. We will see in next sections how other kinds of consumers are plugged between components of this model.

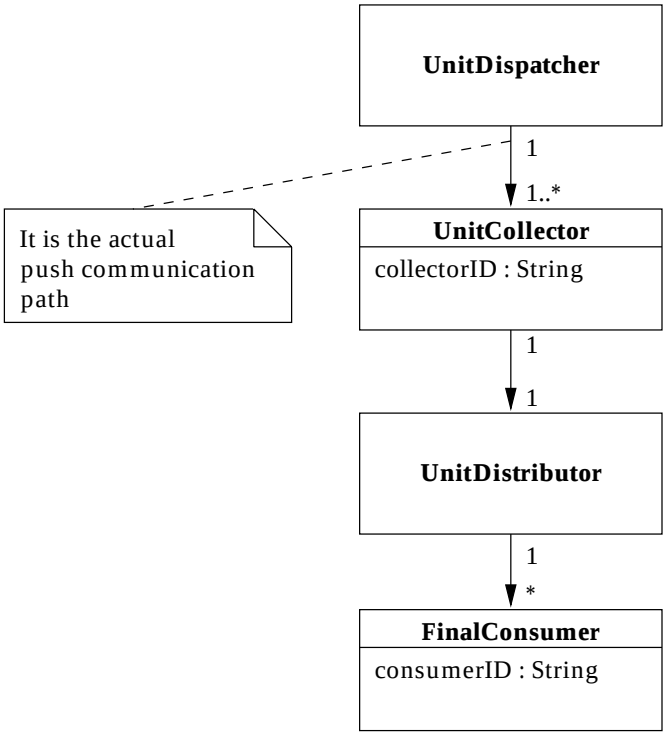

Figure 6.1: The push system

6.1.1 Unit Class

As shown on figure 6.2, it is a specialization of the Payload class which is the superclass of all classes exchanged via the network.

sourceId This identifies the origin of the unit. Note that each component of the system involved in push communication is identified by a string.

subscription A pushed unit is usually associated to a subscription. The subscription is transmitted so that the recipient can handle the unit without having to inspect its content.

object This is the actual information transmitted by the unit. It is an object of any type to allow maximum flexibility in possible extensions.

6.1.2 Subscription Class

The Subscription class objects contain all the information about a unit to push and where to push it. See it on figure 6.3.

objectType It denotes the kind of object to subscribe to. For example “MibData” would refer to an SNMP variable. “Event” would refer to a JAMAP event. Other data representation may be used.

objectID This is a unique identifier to the object to subscribe to. For example, it is an SNMP OID for a “MibData” object type.

The DataSubscription class stands for regular management push where data must be regularly sent to the collector.

frequency It is the number of seconds between each push.

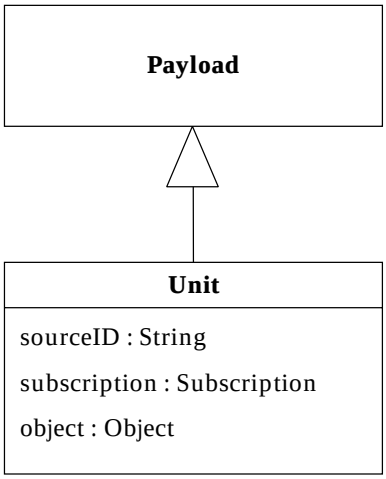

Figure 6.2: The Unit class

The EventSubscription class is used for JAMAP internal events between the pushed data collector servlet and the event manager servlet. In practice, there is no need to subscribe to an event, all generated events are automatically forwarded to any event manager. This class is anyway useful for the consistency of the system and let the possibility of distributed event management extensions open.

6.1.3 SubscriberID Class

As described before, it identifies the recipient of data defined in a subscription. See on figure 6.3 that it is part of a subscription. It is also the class of object sent to agent for unsubscription.

collectorID The ID of the collector to send the data to

consumerID The ID of the consumer to send the data to. This consumer must be bound to a distributor bound to the collector identified by collectorID.

6.2 Management Station

6.2.1 MIB Data Subscription applet

This applet is a data subscription applet for MIB variables. It does three things with MIB variables of a MIB given as parameter to the applet:

1. Provides subscription and unsubscription system for any variable. These tasks are performed with HTTP POST requests.
2. Enables the user to get the instant value of any variable and display it in a *monitor* according to its type. This task is performed with HTTP GET requests in a standard pull model. This is called the *get system*.
3. Allows the user to create a *space* (i.e. a screen area) where several variables are displayed in monitors and refreshed at configurable frequencies. This task is performed with the push system based on an HTTP GET request, as explained before. We call that the *spy system*.

The following design models present these three sets of tasks separately.

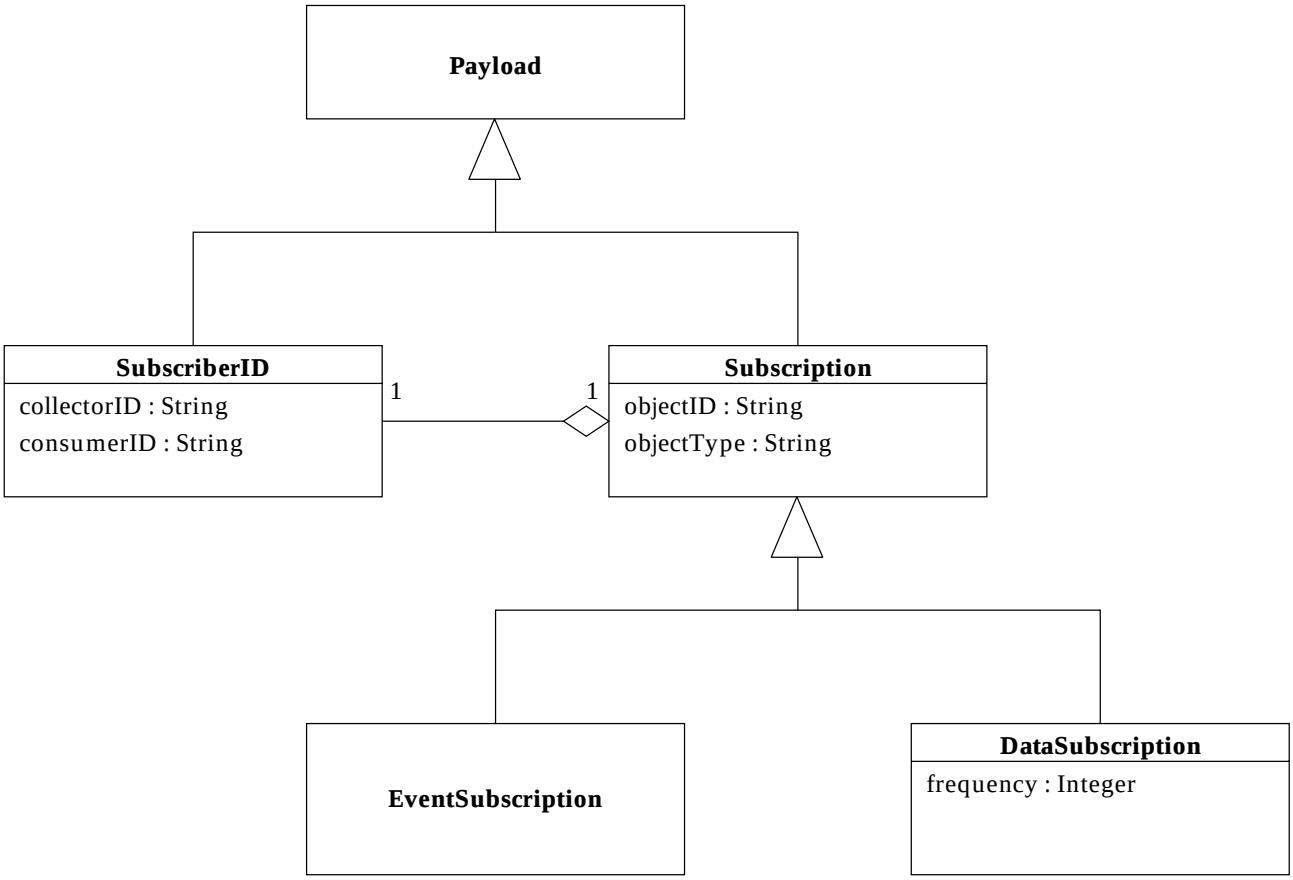

Figure 6.3: The Subscription and SubscriberID classes

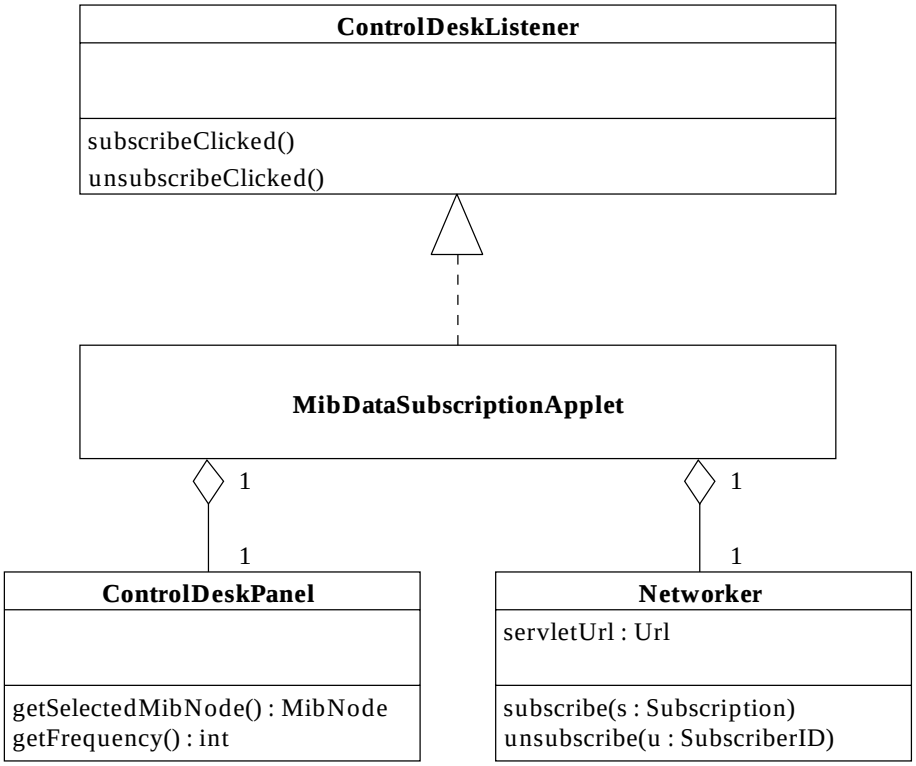

Figure 6.4: Classes of the MIB data subscription applet involved in the subscription system

6.2.1.1 Subscription system classes

All actors of the subscribe and unsubscribe operations are depicted on figure 6.4.

MibDataSubscriptionApplet Main controller class.

ControlDeskPanel Graphical user interface shown on figure 5.2.

ControlDeskListener Interface for receiving events from a ControlDeskPanel object.

Networker Communication component. It is bound to the URL of the agent servlet and manages the push units paths with their MIME Multipart underlying implementation.

MibNode Represents a variable of the MIB. It identifies the data we subscribe to.

6.2.1.2 Subscription system operations

Figure 6.5 shows interactions for subscribe and unsubscribe operations. The diagram is self-explanatory. The main controller builds Subscription or SubscriberID objects and send them to the agent.

Note that the collectorID and consumerID fields of the SubscriberID involved in subscriptions obey to the following convention:

collectorID Use the ID of the *agent*. In an IP network, we use its full qualified domain name.
Example: `ro-ch-in-135.epfl.ch`.

consumerID Use the string `MibData.` followed with the complete numbered SNMP OID.
Example: `MibData.1.3.6.2.1.3.0` for the `system.sysName` variable of MIB-II.

This convention allows *public* distribution. Indeed, any collector connecting to the agent using this collectorID will receive the same units.

6.2.1.3 Get system classes

Classes involved in the get system are shown on figure 6.6.

Monitor Interface to a graphical component displaying a value or an history of values. A new value is submitted by the setValue method.

TextMonitor A MIB variable monitor. It displays a value of type SnmpVar as a text string.

SnmpVar Represents a MIB variable of any type.

TableMonitor A MIB table monitor. It displays a value of type Table in a graphical table component.

Table A two-dimensional array of values. TableMonitor expects it having String values as elements.

MonitorController An interface to a component managing a monitor. A monitor must have a controller to notify it of events. This feature is not used in the scope of the get system.

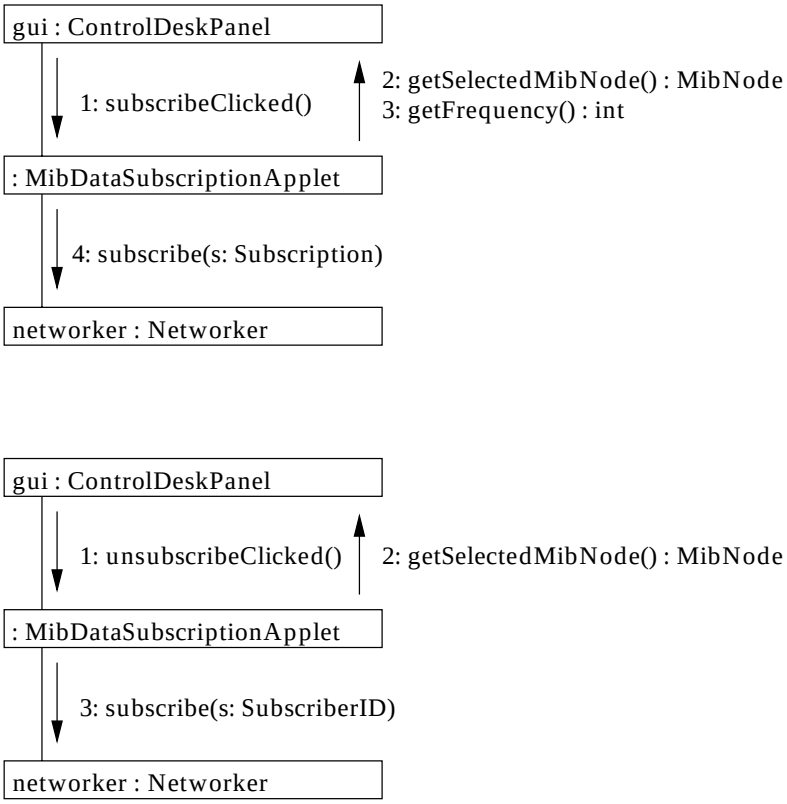

Figure 6.5: Subscribe and unsubscribe operations in the MIB data subscription applet

6.2.1.4 Get system operations

The get operation shown on figure 6.7 is described here:

1. The GUI notifies the main controller of a click on the *Get* button.
2. The main controller retrieves the selected MIB node.
3. And ask the networker to get the associated value.
4. Depending on the type of returned value, selects the right monitor for showing the value. Note that only one of those monitors is displayed on the central panel of the GUI.
5. Retrieve a reference to this monitor
6. Sends the value to this monitor

Monitors are named with strings to allow extension by creating new monitors and integrating them into the system.

6.2.1.5 Spy system classes

The spy system is also based on monitors. See the classes used on figure 6.8. The bold association lines trace the units from the networker to the monitors where they are displayed.

UnitCollector This is an active object bound to the networker to receive push units. It waits for new units and pass them to the unit distributor.

UnitDistributor Distribute the units to the final consumers. They must be registered for the unit distributor knows their ID.

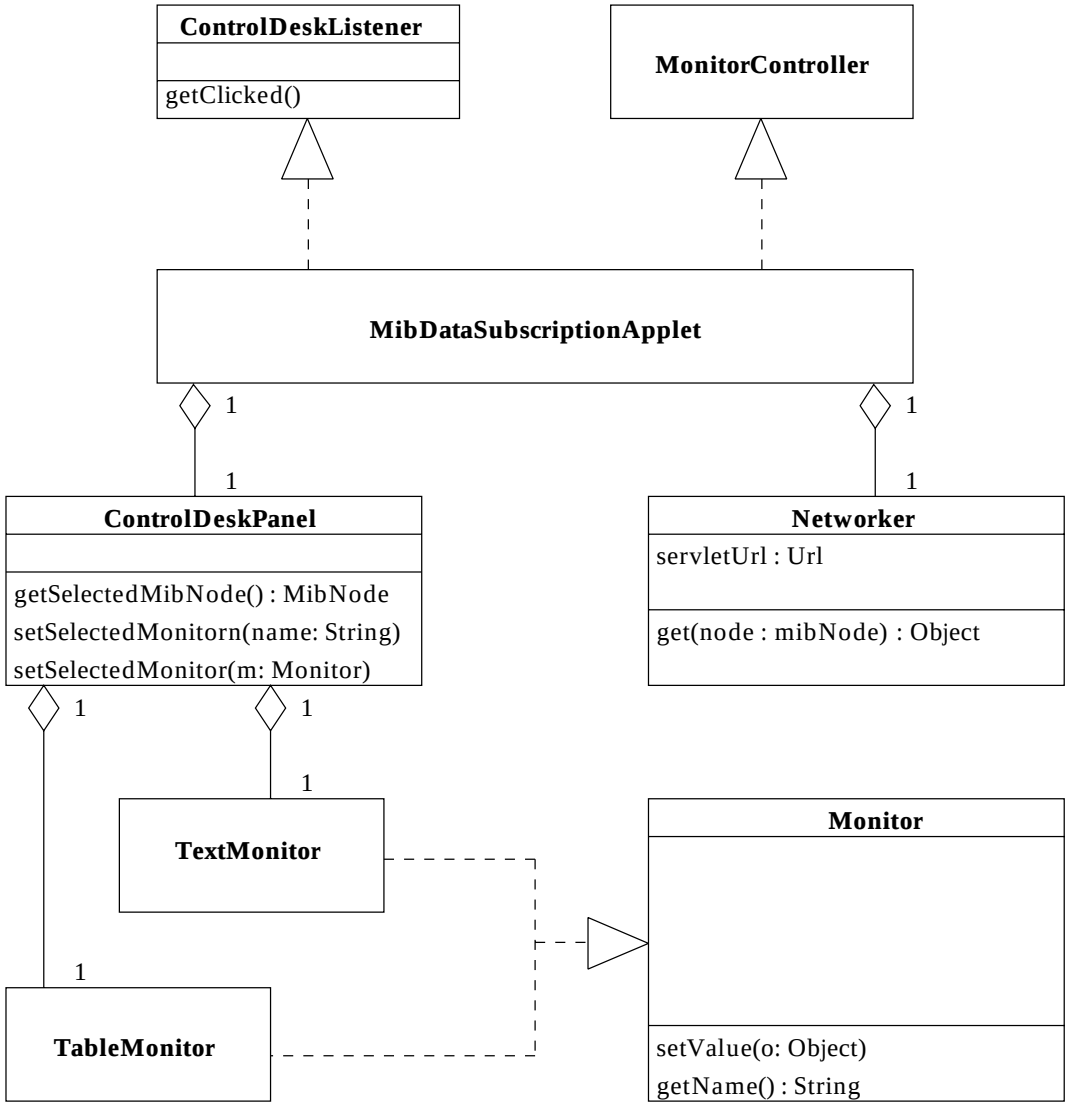

Figure 6.6: Classes of the MIB data subscription applet involved in the get system

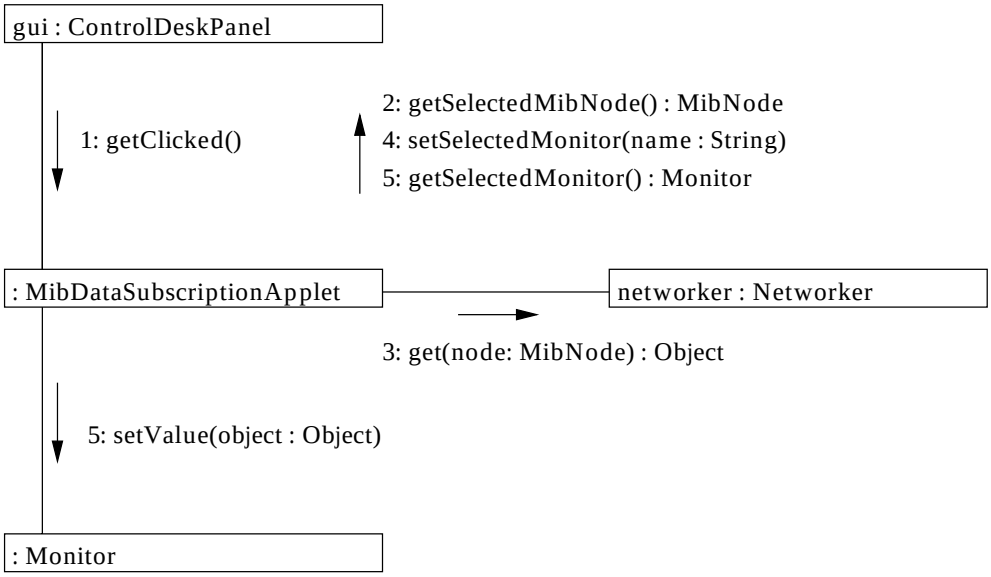

Figure 6.7: Get operation in the MIB data subscription applet

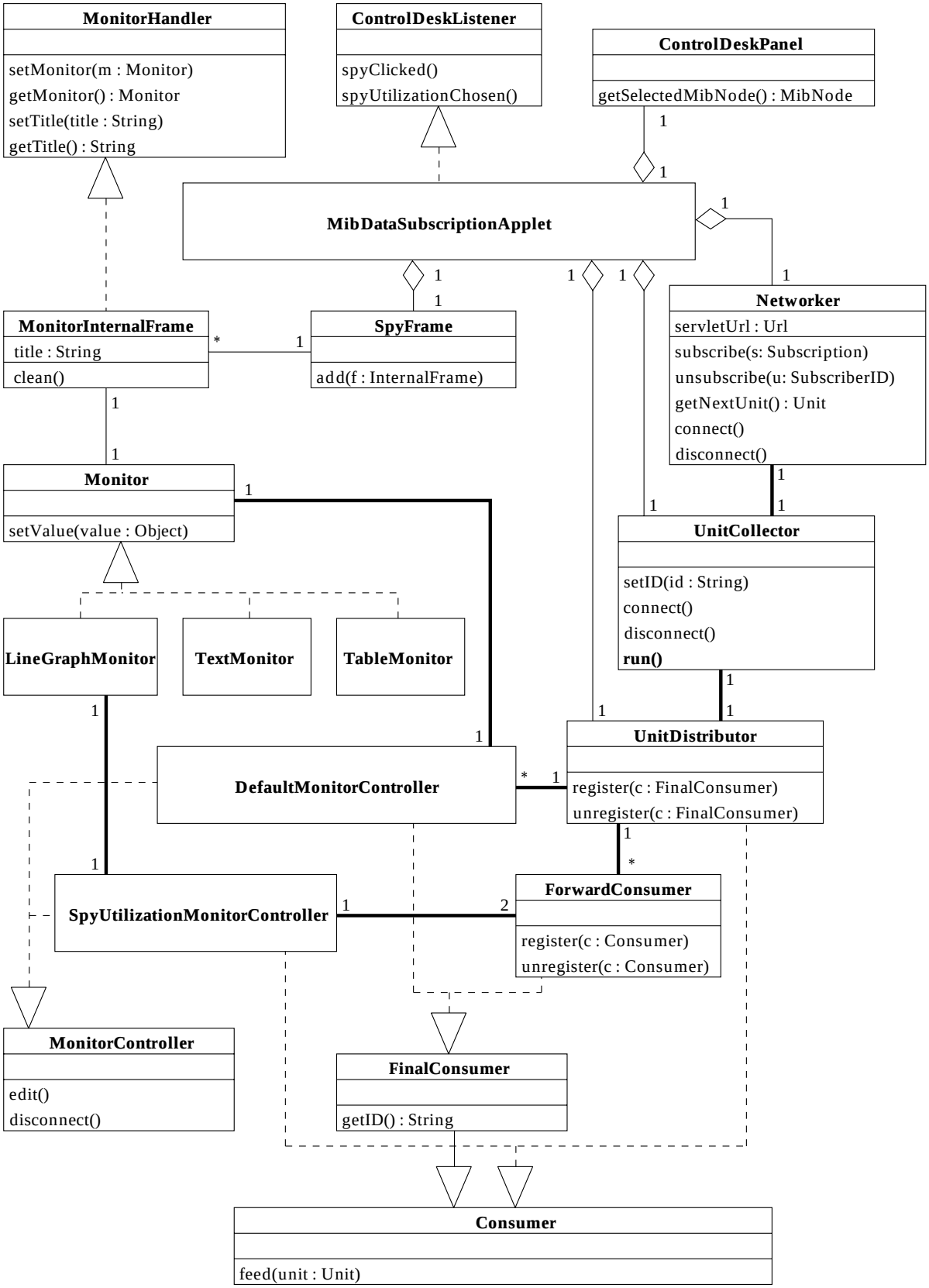

Figure 6.8: Classes of the MIB data subscription applet involved in the spy system

Consumer This is an interface to a general unit consumer. Consumer's job is to be fed with units.

FinalConsumer Objects implementing this interface are consumers. They must provide an ID for registration to a unit distributor. These IDs are used in subscriptions.

ForwardConsumer It is a final consumer that passes received units forward to one or more other consumers. It is useful because final consumer can receive only the units they subscribed to. Forwarding allows multiplexing.

DefaultMonitorController This is a final consumer and a monitor controller. It is the core of a spy system element because it manages subscription, unsubscription and distributor registering. It's main task is to set the value of the monitor to the object encapsulated in the received units.

SpyUtilizationMonitorController This is a special consumer and monitor controller. Its purpose is to control a monitor showing the result of Leinwann's [11] formula shown on figure 6.9. This formula needs two values about an interface, the number of incoming and outgoing octets. This is why this controller is bound to two forward consumers. The result of the formula is shown on a line graph monitor since its variations are more important than its absolute value.

LineGraphMonitor This monitor shows the history of a numeric value on a graph. The x axis is a time scale.

MonitorHandler It is an interface to an object handling a monitor. It is usually a graphical component containing a monitor.

MonitorInternalFrame This monitor handler is an internal resizable frame inside a desktop frame. Its implementation is windowing system dependent. See section 7.1.1 for implementation.

SpyFrame It is an independent window containing internal frames. See it on figure 5.3.

$$Utilization = \frac{8}{ifSpeed} \cdot \frac{ifInOctets(t_1) - ifInOctets(t_0) + ifOutOctets(t_1) - ifOutOctets(t_0)}{t_1 - t_0}$$

Figure 6.9: Utilization formula

6.2.1.6 Spy system operations

The following operations are described on figures 6.10 to 6.12.

Adding a spy

1. The GUI notifies the main controller of a click on the *Spy* button.
2. The main controller retrieves the selected MIB node.
3. A monitor is created.
4. A default monitor controller is created.
5. A monitor internal frame is created.
6. This frame is added to the spy frame.
7. Let the monitor controller know the monitor it controls.
8. Let the monitor controller know which monitor handler is associated with it.
9. Give a name for the monitor that will be displayed in the frame title bar.
10. Let the monitor know who controls it.

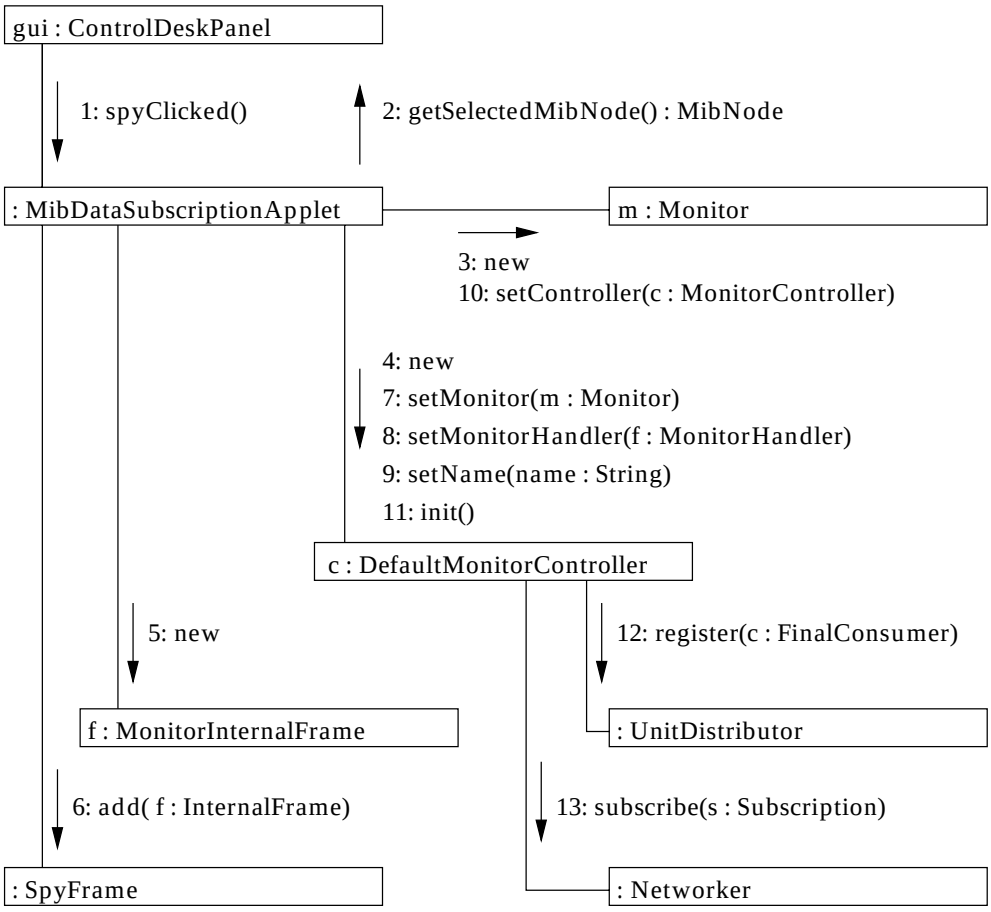

Figure 6.10: Adding a spy to the spy frame of the MIB data subscription applet

11. Tell the controller to initiate itself.
12. The controller registers itself to the unit distributor in order to receive push units.
13. It asks the networker to send a subscription to the agent for the selected MIB node.

Note that adding a utilization spy is quite similar. It implies the creation of a forward consumer.

The *edit* method of the controller is invoked by a menu of the internal frame. It pops up a dialog allowing the user to change the monitor and the frequency of the subscription. The monitor controller manages all this operation. It tells the MonitorHandler to use another monitor and resubscribe with a new frequency.

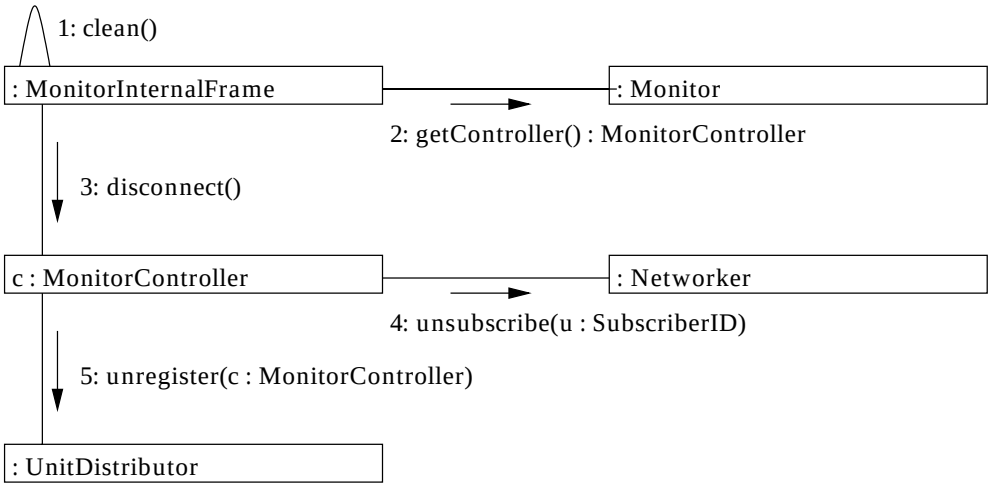

Figure 6.11: Removing a spy from the spy frame of the MIB data subscription applet

Removing a spy

1. A user event (close internal frame) launches the *clean* method of the monitor handler.
2. The monitor handler retrieves a reference to the monitor controller from the monitor.
3. It tells the monitor controller to disconnect.
4. The monitor controller tells the networker to unsubscribe.
5. The monitor controller tells the unit distributor to unregister.

Push unit treatment

1. The unit collector gets the next available unit from the networker.
2. It passes it to the unit distributor.
3. The distributor passes it to its recipient if it is registered.
4. The recipient which is a monitor controller set the value of its monitor to the received value.

6.2.2 Rule edition applet

This applet sends requests to a pushed data collector servlet for three purposes:

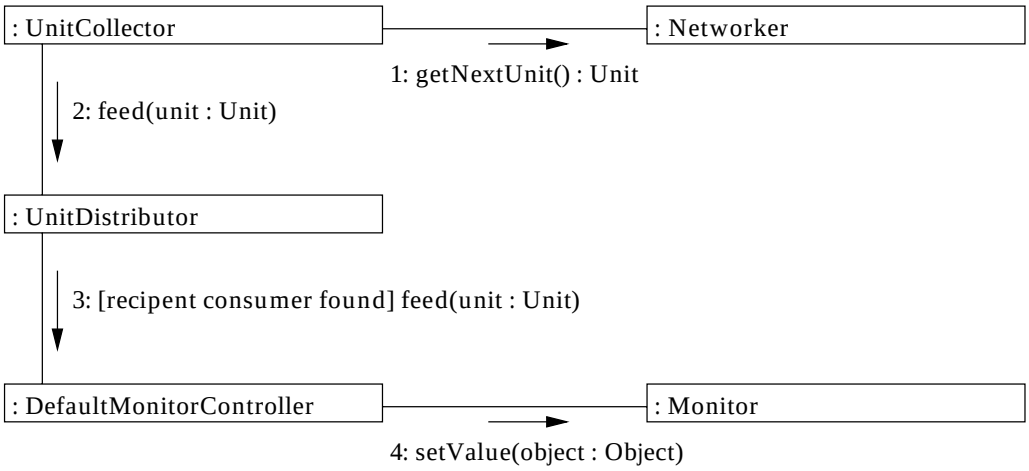

Figure 6.12: Push unit treatment in the spy system of the MIB data subscription applet

1. Post a rule. The pushed data collector servlet is able to receive rules posted by the rule edition applet. This operation creates pushed data interpreters that process values of subscribed variables. In fact, the data interpreters unmarshalls the units and apply the rule to these units. The rules are coded in Java and dynamically compiled and loaded in the pushed data collector servlet as soon as they are delivered. They extend a generic rule class providing a method called *generateEvent* that creates an event and send it to the event manager. Redactors of rules just have to implement the method *apply* in which they call the *generateEvent* method. They can optionally override the method *init* to perform some initialization tasks after object creation. We will see in section 6.4.2 how rules are applied to units.
2. Reinitialize the pushed data collector. It removes all the data interpreters.
3. Force reconnection of the pushed data collector to the applet.

All these operations are performed with HTTP post requests. The convention on URLs are described in section 6.4.2.

6.2.2.1 Classes

The design of this simple applet is depicted on figure 6.13.

RuleEditionApplet Main controller class.

RuleEditionPanel Graphical user interface shown on figure 5.4.

RuleEditionPanelListener Interface for receiving events from a RuleEditionPanel object.

Networker Communication component.

6.2.2.2 Operations

Submitting a rule

1. The GUI notifies the main controller class of a click on the *Submit Rule* button.
2. The main controller retrieves the source string of the rule from the source area.
3. It retrieves the selected MIB node.
4. It retrieves the address of the agent concerned by the rule.
5. It tells the networker to post the source.
6. It tells the GUI to display the response of the post in the dialog area.

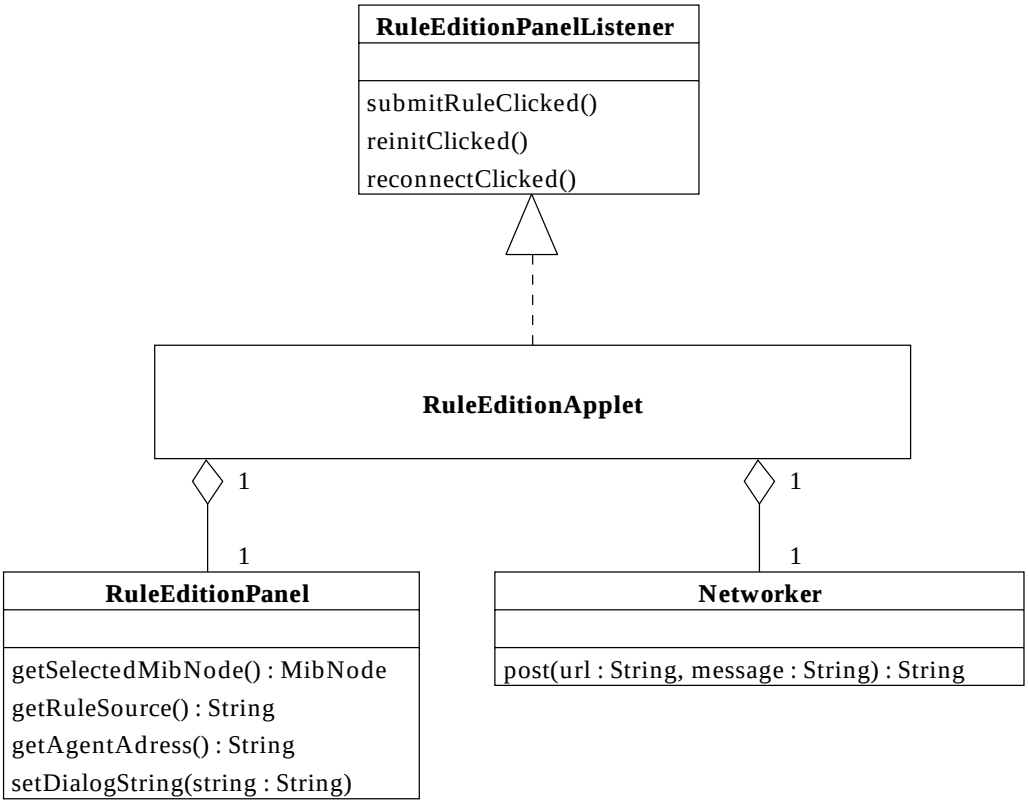

Figure 6.13: Classes of the rule edition applet

Reinitialize the pushed data collector

1. The GUI notifies the main controller class of a click on the *Reinit* button.
2. The main controller retrieves the address of the concerned agent.
3. It tells the networker to post the reinitialization order.

Reconnect the pushed data collector to the agent

1. The GUI notifies the main controller class of a click on the *Reconnect* button.
2. The main controller retrieves the adress of the concerned agent.
3. It tells the networker to post the reconnection order.

6.2.3 Event notification applet

6.2.3.1 Classes

See the classes of the event notification applet on figure 6.14.

EventNotificationApplet This is the main controller class.

EventNotifierPanel It is the graphical user interface.

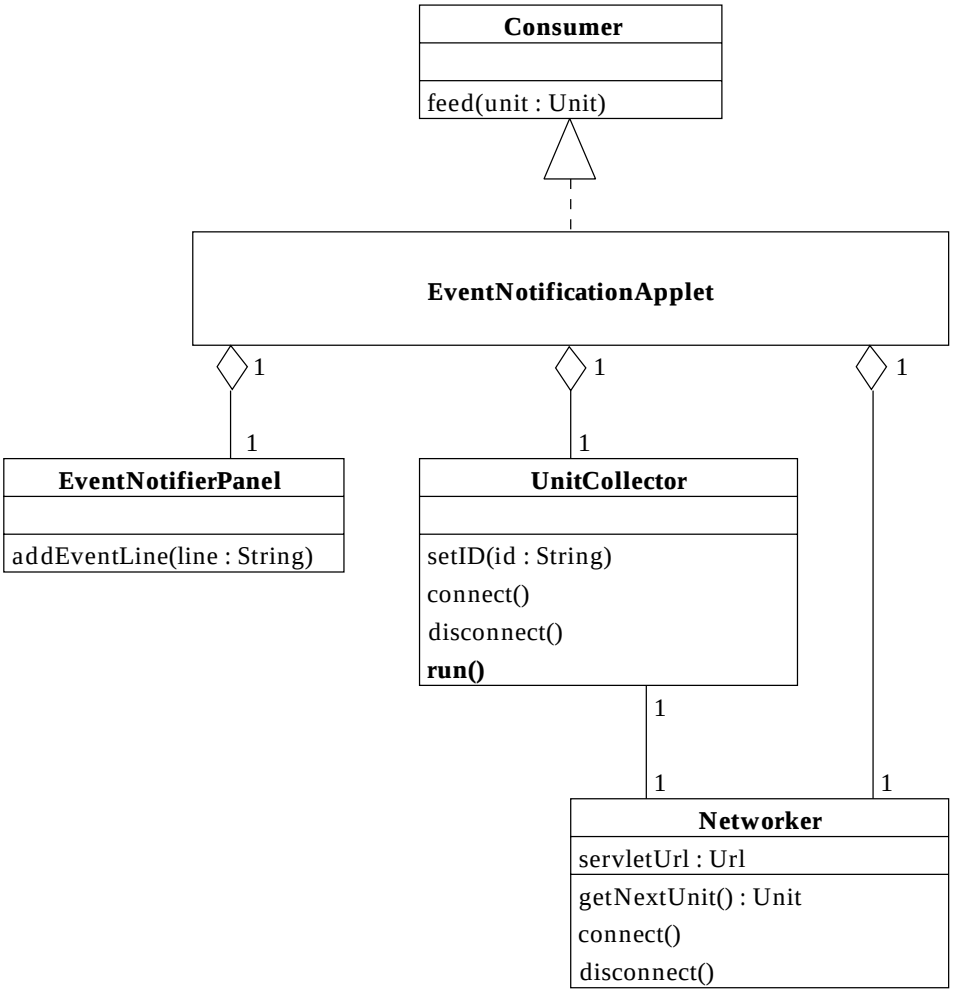

Figure 6.14: Classes of the event notification applet

6.2.3.2 Operations

The only operation is the treatment of incoming units:

1. The unit collector gets the next available unit from the networker.
2. It passes is to the main controller.
3. The main controller unmarshalls it to extract the event and tells the GUI to display it.

At time of writing, this applet is only at specification stage. It will match the following requirements:

- Show a list of events ordered as they occurred. The list should display all pertinent information about events.
- The user must be able to delete some event lines.
- A blinking light and sound signal should notify the user of occurrence of events.

6.3 Agent

6.3.1 Get servlet

This is a very simple pull-based applet. It retrieves data given their objectID and objectType as URL query string parameters.

6.3.1.1 Classes

The figure 6.15 shows the classes of the get servlet.

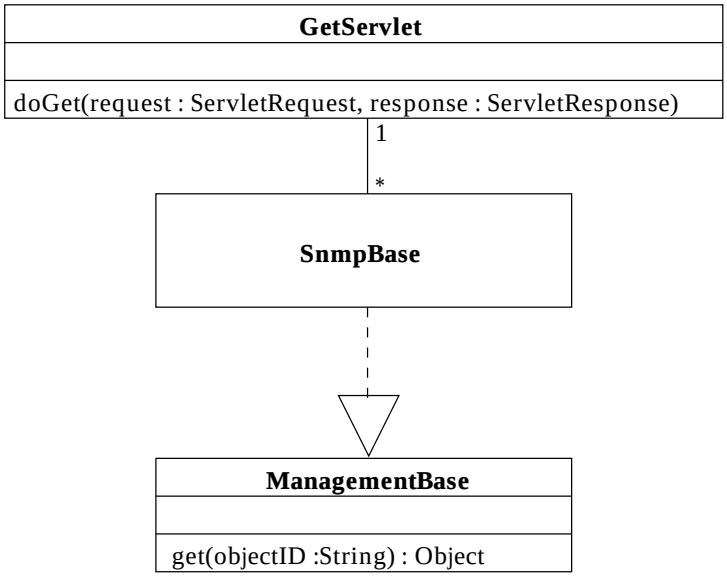

Figure 6.15: Classes of the agent get servlet

SnmpBase This is an object implementing the underlying SNMP protocol for retrieving requested data.

ManagementBase Interface to any class handling management information base.

6.3.1.2 Operations

The only operation is fired by a GET request on the servlet. Currently, the only supported information base is that of the SNMP world, so only "MibData" objectType parameters are treated. The response contains a serialized object retrieved from the SNMP base through the *get* method. This object can be of two types:

SnmpVar Represents a single MIB variable. If the objectID refers to a scalar variable.

Table Two-dimensional array of strings. If the objectID refers to a table.

6.3.2 PushDispatcherServlet

This servlet provides the push mechanism.

6.3.2.1 Classes

Classes of the push dispatcher servlet are depicted on figure 6.16.

PushScheduler Core class of the dispatcher system. It retrieves information from the base and push them to the response output stream accordingly to the subscription table.

MultiPartObjectOutputStream This class handles a MIME Multipart stream. Objects written to it are serialized and sent into the stream as different parts. Such a stream is usually the response output stream of the servlet.

SubscriptionTable Hash table containing sub-tables as value and collectorIDs as keys.

SubscriptionSubTable Hash table containing subscriptions as values and consumerIDs as keys.

6.3.2.2 Operations

Adding a received subscription to the subscription table When the servlet is posted a message with *subscribe* as URL query string, it calls the *put* method of the subscription table with the posted Subscription object.

Removing a subscription from the subscription table When the servlet is posted a message with *unsubscribe* as URL query string, it calls the *remove* method of the subscription table with the posted SubscriberID object.

Push scheduling The scheduling process is depicted on figure 6.17.

1. A GET request is received by the servlet with a collectorID as URL query string.
2. A push scheduler is created.
3. The push scheduler creates management base objects (currently only SNMP is supported)
4. The push scheduler creates a MIME multipart object output stream bounded with servlet response output stream.
5. The servlet tells the scheduler to begin activity.
6. The scheduler retrieves a list of all subscription associated with the collectorID.
7. It gets each object scheduled for the current time slice from the management base.
8. It sends them into the stream.

The steps 6 to 8 are repeated until the collector closes the connection.

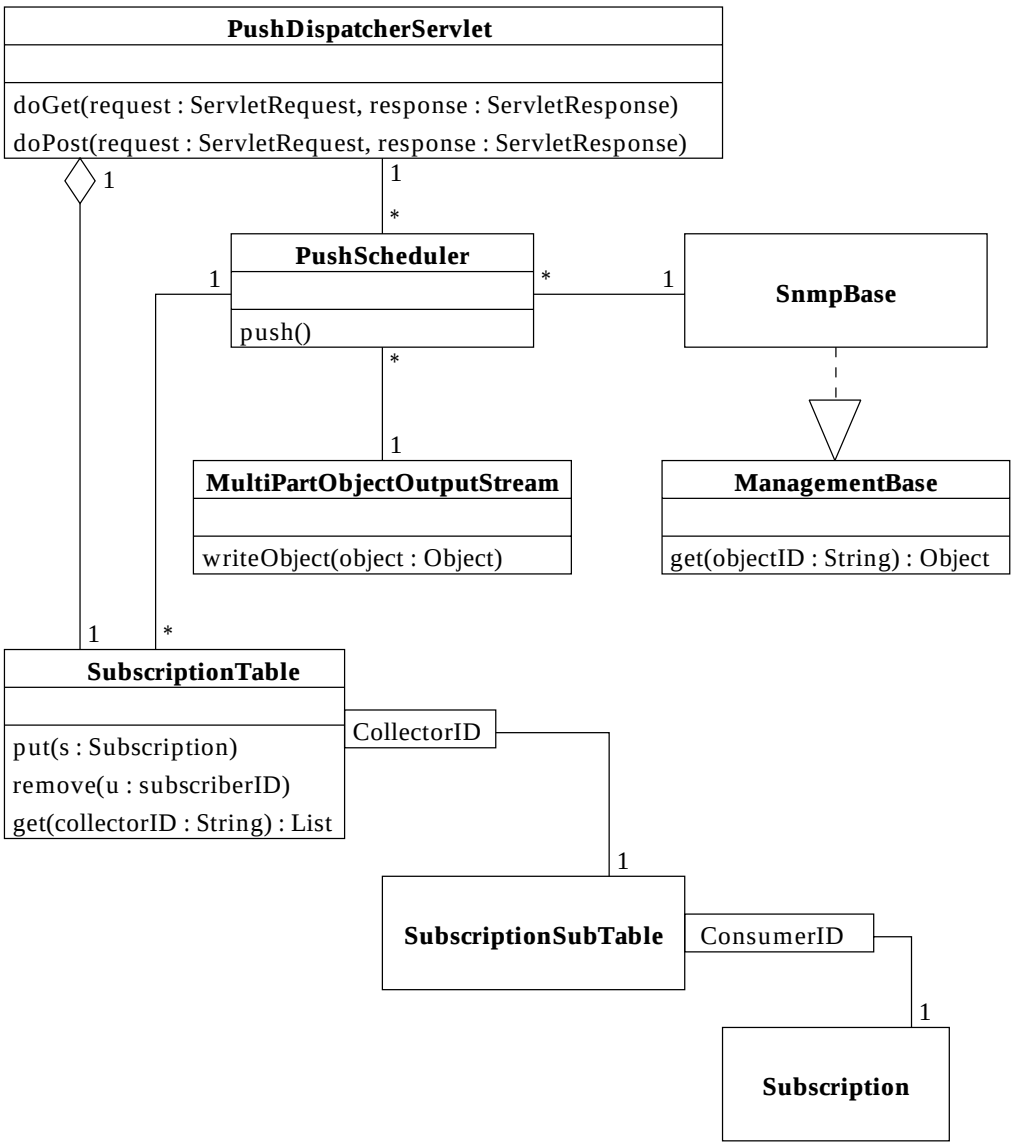

Figure 6.16: Classes of the push dispatcher servlet

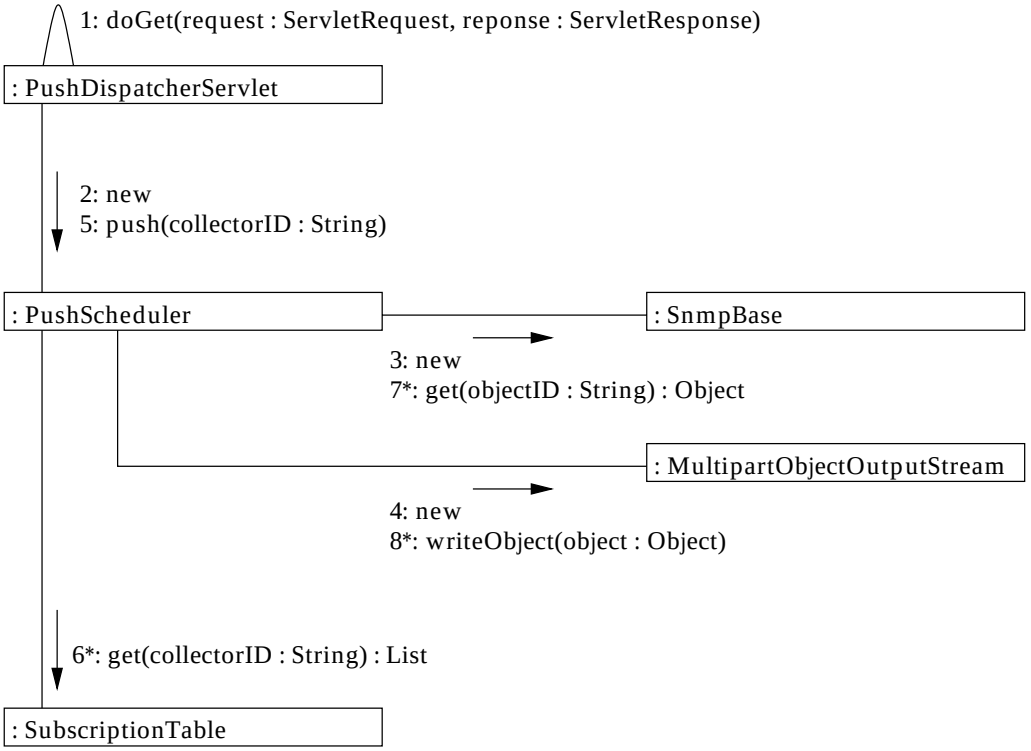

Figure 6.17: Push scheduling in the push dispatcher servlet.

6.4 Management Server

6.4.1 Events

The management server uses an internal event type encapsulated in units. See this class on figure 6.18.

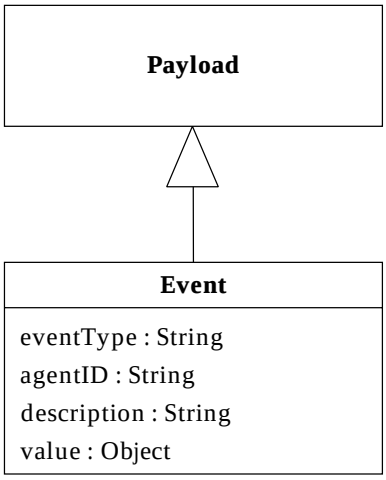

Figure 6.18: The Event class

eventType Characterizes the event. There is still no convention.

agentID Identifies the agent concerned by this event.

description Text explaining the event.

value An opaque object containing any value associated with the event. It must have a string representation method for human reading. The type of this value should be deductible from the event type.

6.4.2 Pushed data collector servlet

6.4.2.1 Classes

Figure 6.19 shows classes of the pushed data collector servlet. Note the bold lines showing push connections with the agents and event managers.

CollectorTable A table of all unit collectors in the servlet. They are indexed on their collectorID.

PushedDataFilter This relay consumer controls the flow of incoming units. If the number of units received by minute exceed a threshold, the push connection with the agent is automatically closed and an event is generated. This prevent excessive reaction to a wrong agent behavior.

PushedDataInterpreter Just apply a rule to the object encapsulated in received units.

InterpreterRule This is a generic rule. Actual dynamically loaded rule classes extend child of this class. See figure 6.20 for an example: MibDataRule is the generic rule for MIB data and MibDataRule_1 is a dynamically loaded rule class.

RuleClassHandler This class is the core system of the dynamic rule compilation and loading. It uses Sun JDK's compiler class and the class loading capabilities of Java. See section 7.1.4.

PushForwardConsumer It is a special consumer that waits for units and forwards them to an output stream. It uses the MultiPartObjectOutputStream class discussed in section 6.3.

RelayConsumer It is an extension to the Consumer interface for objects relaying units to another consumer.

6.4.2.2 Operations

Creating an interpreter rule from a received rule source This operation is described on figure 6.21. The URL query string must contain the following string parameters:

command For this operation, it must be set to "postrule"

className The name of the rule class. To be unique it must be a concatenation of the parent class name, the collectorID and the consumerID it will refer.

agentAddress The address of the agent the unit collector should connect if it is not yet connected.

collectorID Identifies the collector the new data interpreter will be bound to.

consumerID The identifier of the new interpreter consumer

1. The servlet runner calls doPost with *command=postrule* in query string parameters.
2. Retrieves the unit collector from the given collectorID.
3. Saves the rule source code contained in the posted message into a file.
4. Tries to compile it. The compiler output is written in the output stream of the response.
5. If the compilation was successful, creates an objects of this rule class.
6. Initializes it.
7. Creates a pushed data collector associated with the rule.
8. Register it to the unit distributor in order to receive units.

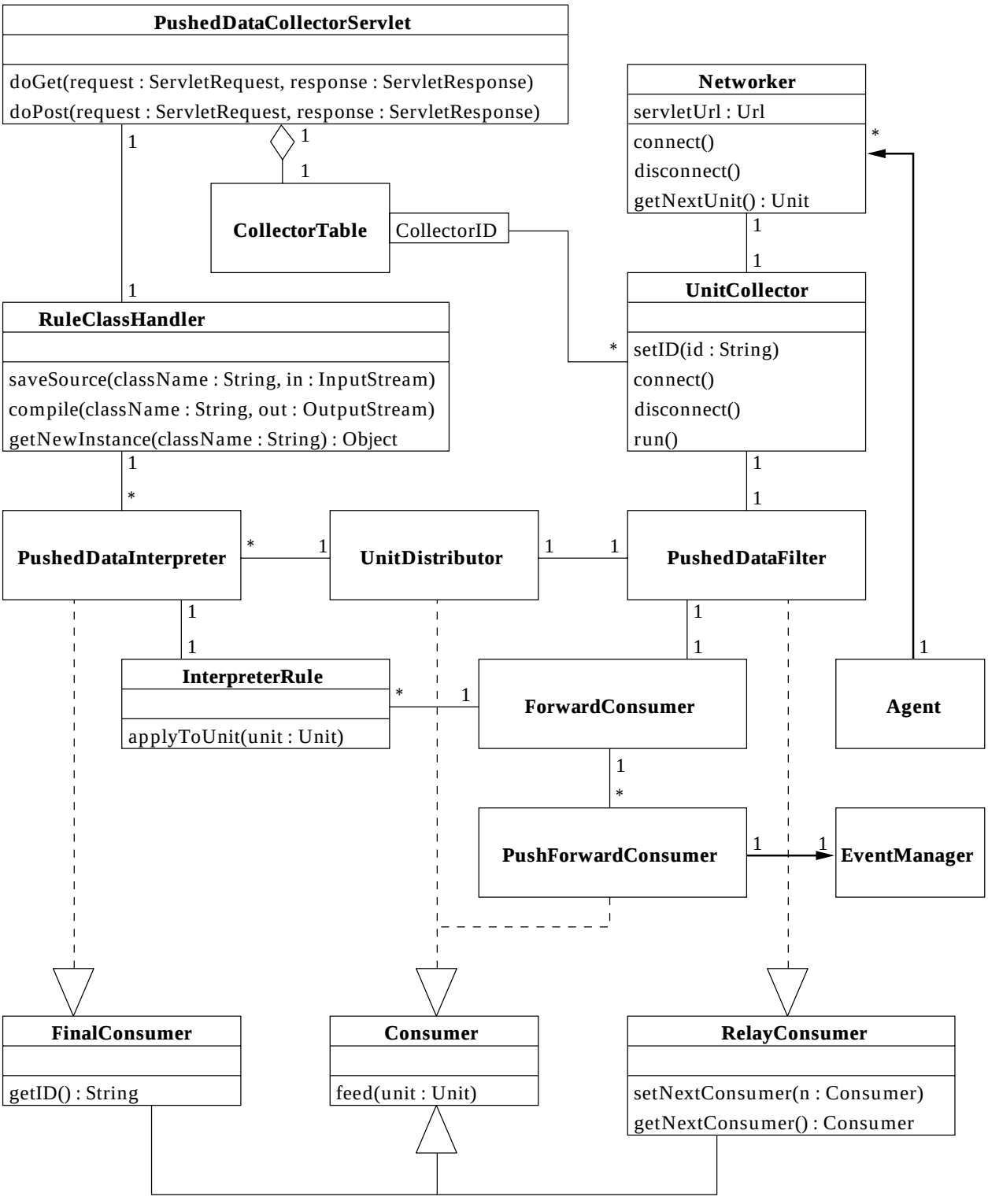

Figure 6.19: Classes of the push data collector servlet

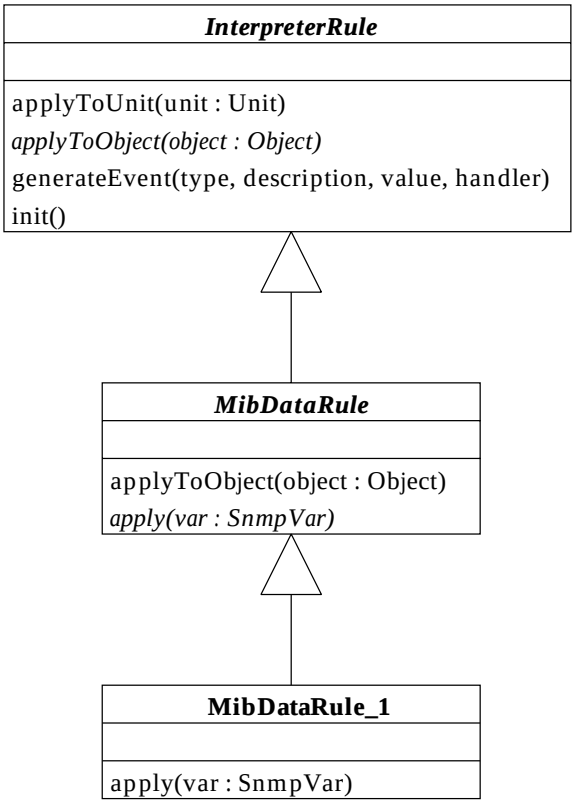

Figure 6.20: Rule classes of the pushed data collector servlet

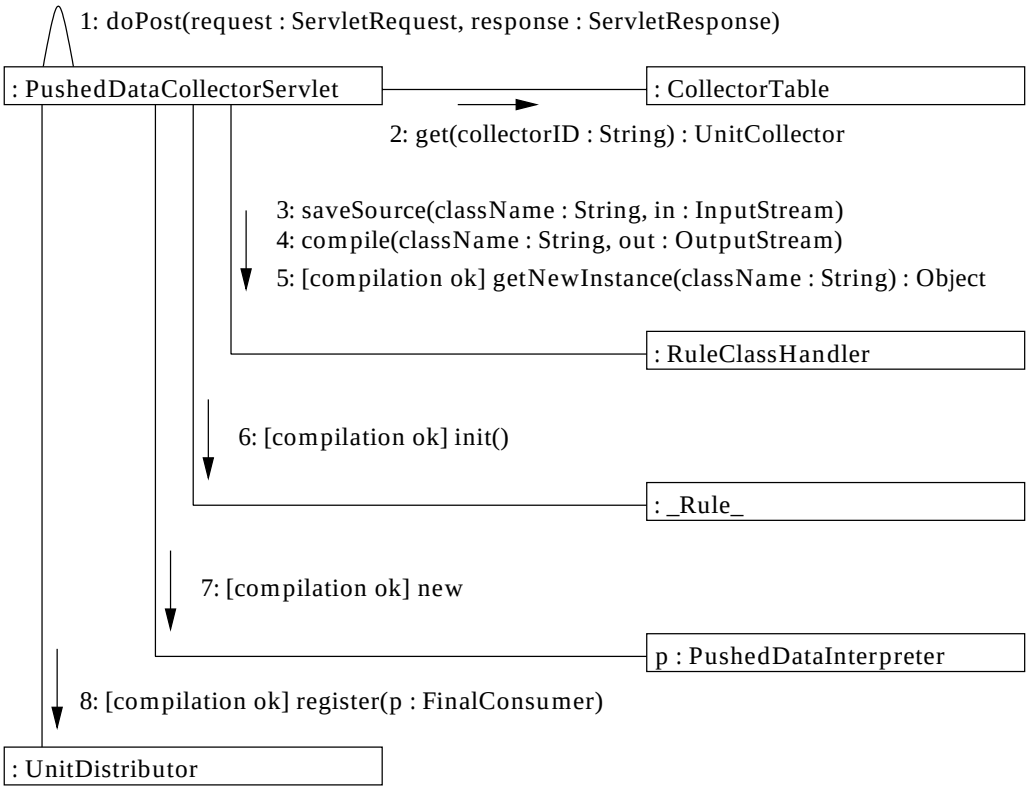

Figure 6.21: Treatment of rule posted to pushed data collector servlet

Reconnecting to agent The URL query string must contain the following string parameters:

command For this operation, it must be set to “reconnect”.

collectorID Identifies which unit collector must reconnect.

Calls the *connect* method of the collector identified by the given collectorID.

Reinitializing a unit collector The URL query string must contain the following string parameters:

command For this operation, it must be set to “reinit”.

collectorID Identifies which unit collector must be reinitialized.

agentAddress The address of the agent to connect.

Unregisters all pushed data interpreters from the unit distributor and reconnects to the agent.

Treatment of a unit received by the pushed data collector applet Figure 6.22 shows how units are treated.

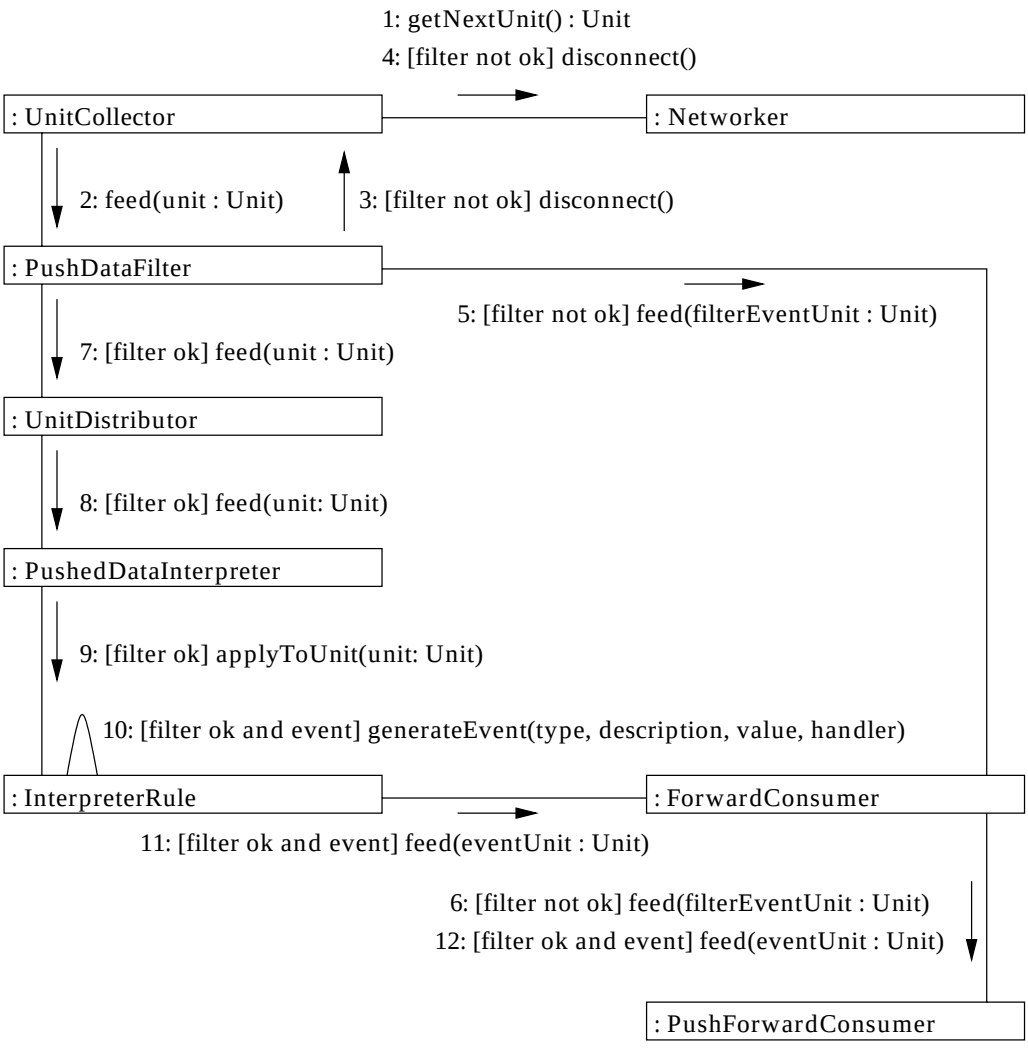

Figure 6.22: Treatment of unit in the pushed data collector servlet

1. The unit collector gets the next available unit from the networker.
2. It passes it to the data filter.
3. If the filter considers too much units were received in the last minute, it tells the unit collector to disconnect.
4. The unit collector tells the networker to disconnect.
5. The data filter notifies the event manager of the failure by sending an unit containing an event to the forward consumer.
6. The forward consumer sends the event unit to registered push forward consumers.
7. If all is OK for the data filter, the incoming unit is passed to the unit distributor.
8. The unit distributor sends it to its recipient data interpreter.
9. The pushed data interpreter applies the rule to the unit.
10. Depending on the rule, an event can be generated.
11. The rule passes it to the forward consumer.
12. The forward consumer sends the event unit to registered push forward consumers.

Handling of event manager connection Event managers can connect to the pushed data collector servlet to open a push communication path for generated events. They send a GET request to the servlet which fires the *doGet* method. This one creates a push forward consumer that registers to the forward consumer and waits for events. The connection is hold open and events are dispatched to the event manager.

6.4.3 Event Manager servlet

6.4.3.1 Classes

The classes of the event manager servlet are shown on figure 6.23.

EventCorrelator It should deduce correlation between incoming events and drop events that are consequences of others. As this function depends on network topology and characteristics, it is a project in itself. We let this part to future development.

EventLogger It is a client of the log servlet described in section 6.4.4.3. It just writes a line for each event in a remote log file.

EventMailer It is an SMTP client able to send events by e-mail to operators and administrators.

6.4.3.2 Operations

As the architecture of the event manager servlet is rather similar to the pushed data collector, we let the reader extrapolate without providing superfluous interactions diagrams.

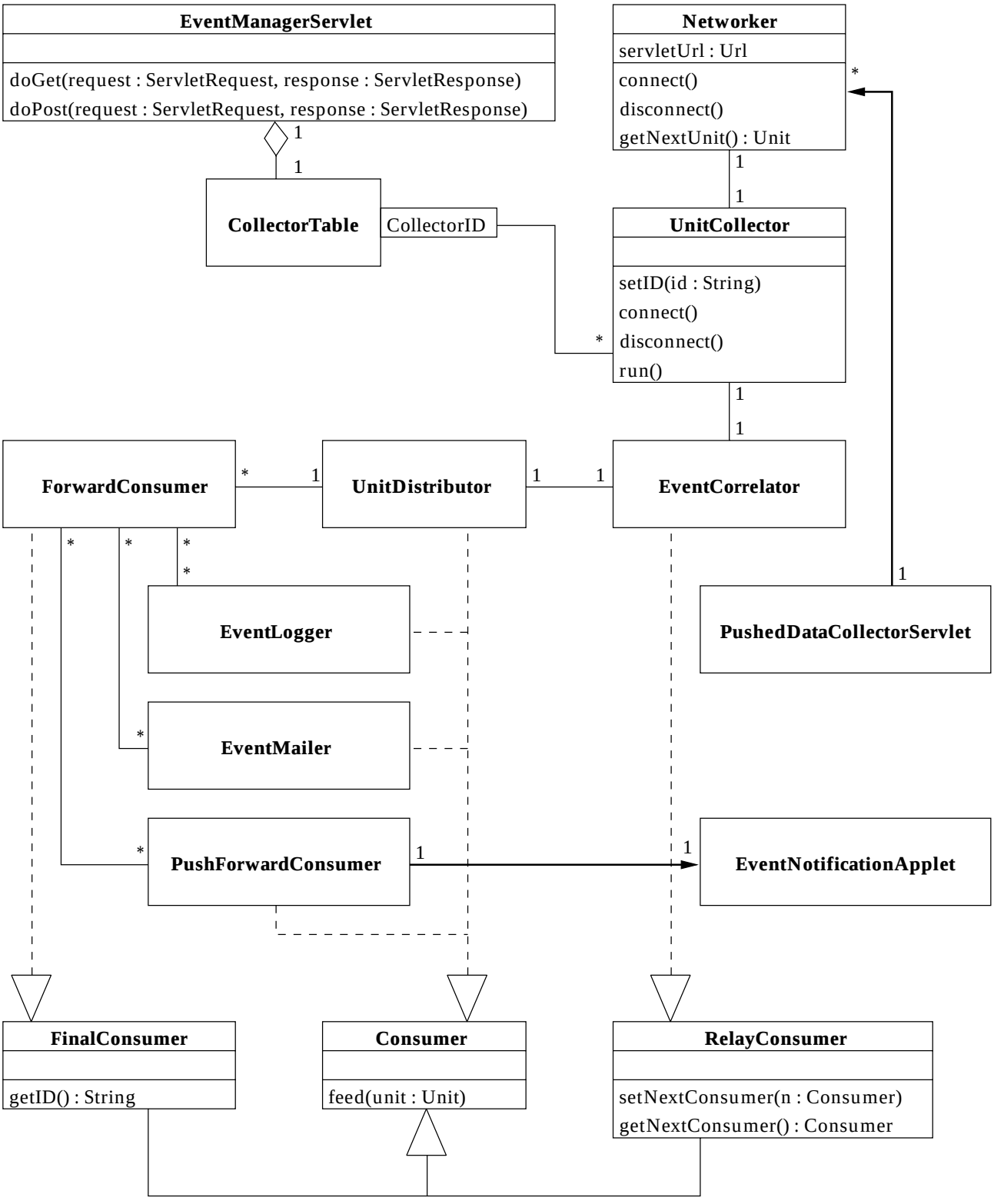

Figure 6.23: Classes of the event manager servlet

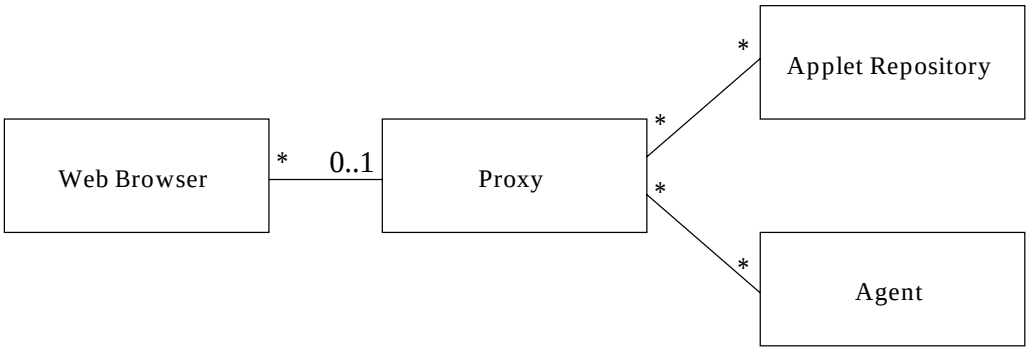

Figure 6.24: Proxy: high-level class diagram

6.4.4 Utility servlets

6.4.4.1 Proxy

The proxy servlet is useful to avoid security policy restrictions of web browsers. Indeed, they authorize applets to connect only to the host they are downloaded from. This way, the data subscription applets should be always stored on the agent. To use applets coming from any web site with any agent, the trick is to make the browser "think" the applet and the agent servlet are on the same host, the proxy host as seen on figure 6.24.

The proxy servlet can handle URLs of the form:

```
http://proxyhost/ProxyServlet/www.vendor.com/appletpage.html
```

This will retrieve the object at URL:

```
http://www.vendor.com/appletpage.html
```

This trick is powerful because the following prefix will be considered as base URL of the object:

```
http://proxyhost/ProxyServlet/www.vendor.com/
```

Thus, relative references to other objects are resolved from this base. The proxy is then invisible.

Moreover, multi-threading capabilities of the Java language and the servlet system allow persistent connections and HTTP push through the proxy. The misgiving of proxies is that they can become bottlenecks if too much clients use them. This can be avoided by distributing proxy servlets on different hosts of the network.

6.4.4.2 Persistent storage

Two servlets have been developed to allow remote storage:

StoreObjectServlet Stores the content of POST messages into a repository. The query string of the URL must be an identifier of the stored object.

RestoreObjectServlet Restores an object and give it as response of a GET request. The query string of the URL must be an identifier of the stored object.

6.4.4.3 Log

This servlet write contents of POST messages into a text log. The query string of the URL must be an identifier of the log to use. This servlet is accessed by the EventLogger seen in event manager servlet design.

Chapter 7

JAMAP: Implementation

7.1 Technologies

7.1.1 Java & Swing

All our software is written in Java using the Linux port of JDK 1.1.6 by Blackdown [13].

The applets and graphical interface components use Sun's Swing GUI toolkit. In particular, the spy frame GUI with its internal monitor frames is realized with Swing's desktop panes. In spite of a resource -hungry implementation, Swing proposes one of the best toolkits of the moment. It is very complete and supports several different *look and feel*s. It is the standard graphical interface toolkit of Java version 2 where it is also called Java Foundation Classes.

We used NetBeans Developer 2.0 [17] as development environment for graphical interfaces. This environment supports beans and Swing and is entirely written in Java.

7.1.2 Web servers and servlet runners

At the beginning of this project, we used the Apache Web Server [14] with its servlet module Jserv [15] because this server is very popular. At this time, the Jserv version 0.8 did not support concurrent accesses to servlets and the response stream was buffered, thus delivering units in bursts instead of one by one at the time they are pushed.

At the time of writing, Jserv 1.0 (released in december 1998) matches the requirements and remains to be tested with JAMAP.

For the development, we used the Jigsaw Web server from the W3 Consortium [16]. This server supports all recent technology improvements as it is an experimental testbed product. It has the big advantageof being entirely written in Java, thus allowing fine control over servlet experimentation. Even though this server is configurable with a graphical interface, it is much more diffcult to get used than the Apache server. Several concepts are not obvious at first, such as frames and indexers.

7.1.3 Web browsers and JVMs

The applets have been tested with Netscape Communicator 4.5 under Linux 2.1.125 (glibc), Microsoft Internet Explorer under Windows NT 4.0 and the applet viewer of Sun's JDK Linux port.

Throughout this project, Swing was not officially supported by current Web browsers, Netscape Communicator 4.5 and Microsoft Internet Explorer 4.0 supported a good deal of Sun's JFC, but some trouble were experienced with serialization of Swing components with Netscape Communicator 4.5. Unfortunately, debugging was impossible because Communicator hanged without any error message or core-dumped.The same problems appeared with MSIE 4.0 but we did not spend much time debugging on Microsoft platforms. As the applet viewer works very well with our Swing applets, we are confident about future browsers versions that will fully support Sun's JFC.

7.1.4 Dynamic class compilation and loading

To compile the interpreter rules of the pushed data collector servlet, we used the sun.tools.javac.Main class provided with Sun's JDK. This is the core class of the `javac` compiler. Unfortunately, this class is not documented and we were forced to decompile it to identify which method had to be called to perform a simple compilation. We used Jad 1.5.6e (referenced by Blackdown Web page [13]).

The dynamic class loading is a feature of the Java language. The core API provides a method to instantiate objects from a class by giving its name as a string. The class loader of the JVM searches the class file in the filesystem, and loads it in the JVM's memory. This allows to load a class at run time without knowing its name in advance.

Once the class is loaded, it behaves like other classes. We are only limited by the fact that a class can not be modified at runtime. That means that if a posted rule class name already exists, another name must be used, e.g. by adding a suffix number to it.

Unfortunately, memory used by loaded classes is freed only when the JVM restarts. The administrators must be aware of this and must separate rule debugging phases from exploitation.

7.2 External Libraries

7.2.1 AdventNet SNMP

We used the following classes of the AdventNet SNMP suite [7]:

- The MibTree class that is a UI bean showing the MIB tree.
- The MibNode class. It is a node of a MIB tree.
- The SnmpVar class representing an SNMP variable.
- The SnmpTarget class that is a bean abstracting an SNMP device.
- The SnmpTable class that is a bean representing an SNMP table.

7.2.2 HTTPClient

We used the Util class of this package written by Ronald Tschalär [8] that provides more features than JDK 1.1.6 network classes, in particular for HTTP header parsing.

7.2.3 IBM AlphaWorks SMTP

We used the SMTPConnection class in the event mailer consumer to send mail to operators and administrators [9].

7.2.4 JDK's Java compiler class

We used the sun.tools.javac.Main class of Sun's JDK Java compiler [10] to implement dynamic compilation.

7.3 Packages

Our JAMAP framework is organized in 9 packages:

jamap The root package.

jamap.applet Applets.

jamap.gui Applet GUIs and dialogs.

jamap.monitor Classes concerning monitors.

jamap.push All general classes involved in the push system.

jamap.server Classes only on the management server side.

jamap.servlet Servlets.

jamap.snmp All classes specific to SNMP and SMI.

jamap.spy Undocumented classes used in a former pull-based prototype (obsolete).

Chapter 8

Conclusion

8.1 Summary

This document presented the JAMAP project and the technologies used for the realization of a Java-based network management application. A framework implementing HTTP-based push communication and providing components managing data units has been studied and developed. The focus was put on the object-oriented design of elements of the application using applets and servlets.

This project implemented some of the proposals that Martin-Flatin exposed in [1], and introduced original ideas such as dynamic rule compilation and unit-based data management.

The resulting application meets the expectations as the push system works well and the framework design is consistent and extensible.

8.2 Benefits

First, I became familiar with IP network management, and I now feel at ease with the design and implementation of distributed Java applications. But most of all, I realized how exciting it was to work with cutting-edge technologies. My time was shared between application design, prototypes programming and Internet tools to get fresh information about technologies. I followed closely the evolution of servlet specification, Web servers and Web browsers. During my work, I had a lot of contact with various people on the Internet, from external libraries bug reports to use of IRC for help on some tricky debugging.

Another interesting aspect of the work was the continuous object-oriented design that I wished to be always open to future extensions. This led to a very flexible result that can even be used for other applications than network management. I am now convinced that Java is a major step toward better programming, as it keeps the implementation close to the design and facilitates fast prototyping.

8.3 Future work

In the short term, it would be a good idea to integrate notifications handling and a network map apple to the JAMAP frameworkt. It will also be necessary to treat in depth the persistence of subscription tables, pushed data interpreters and spy frame monitors. They have been partially implemented but not documented.

In the longer term, one could separate the framework and the application to provide a JavaBean-based API. The application could be a project in itself by completing the event correlator and improving the configurability in order to make a network management application usable on a real network.

Another improvement would be to replace serialized objects for units with a generic data representation such as XML that is not bound to a programming language. This could lead to a new specification for network management exchanged data.

Finally, further work could be done on interfacing JAMAP with existing applications and management environment. Integrating JAMAP as a sub-system of JDMK, for example, could prove valuable.

Acronyms

API	Application Programming Interface
ASN	Abstract Syntax Notation
AWT	Abstract Windowing Toolkit
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DBMS	DataBase Management System
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IRC	Internet Relay Chat
JAMAP	JAva Management APplication
JAR	Java ARchive
JDBC	Java DataBase Connectivity
JDMK	Java Dynamic Management Kit
JDK	Java Development Kit
JFC	Java Foundation Classes
JMAPI	Java Management API
JVM	Java Virtual Machine
MIB	Management Information Base
MIME	Multipurpose Internet Mail Extensions
MSIE	MicroSoft Internet Explorer
NMP	Network Management Platform
NMS	Network Management Station
NT	New Technology
PC	Personal Computer
RFC	Request For Comment
RMI	Remote Method Invocation
SME	Small or Medium-sized Enterprise
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator

Bibliography

[1] J.P. Martin-Flatin. *The Push Model in Web-Based Network Management*. Technical Report SSC / 1998 / 023, version 3, SSC, EPFL, Lausanne, Switzerland, November 1998.

[2] J.P. Martin-Flatin. *Push vs. Pull in Web-Based Network Management*. Technical Report SSC / 1998 / 022, version 3, SSC, EPFL, Lausanne, Switzerland, November 1998.

[3] J.P. Martin-Flatin. *IP Network Management Platforms Before the Web*. Technical Report SSC / 1998 / 021, version 2, SSC, EPFL, Lausanne, Switzerland, December 1998.

[4] R. Fielding, J. Gettys, H. Frystyk and T. Berners-Lee (Eds.). *RFC 1945. Hypertext Transfer Protocol – HTTP/1.0*. IETF, May 1996.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee (Eds.). *RFC 2068. Hypertext Transfer Protocol – HTTP/1.1*. IETF, January 1997.

[6] M. Fowler, K. Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, USA, 1997.

[7] AdventNet SNMP suite < `http://www.adventnet.com/products.html` >

[8] Ronald Tschalär, HTTPClient < `http://www.innovation.ch/java/HTTPClient/` >

[9] IBM AlphaWorks < `http://www.alphaWorks.ibm.com/` >

[10] Sun Microsystems Java Developper Kit 1.1 < `http://www.javasoft.com/products/jdk/1.1/` >

[11] A. Leinwand. “Accomplishing Performance Management with SNMP”. In *The Simple Times*, 1(5):11-12, 1995.

[12] Netscape. An Exploration of Dynamic Documents < `http://home.mcom.com/assist/net_sites/pushpull.html` >

[13] Java Linux < `http://www.blackdown.org` >

[14] Apache HTTP server < `http://www.apache.org` >

[15] Java Apache Project < `http://java.apache.org` >

[16] Jigsaw < `http://www.w3.org/Jigsaw/` >

[17] NetBeans < `http://www.netbeans.com` >

[18] M. T. Rose. *The Simple Book: an Introduction to Networking Management*. Revised 2nd edition. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[19] Java Management API < `http://www.javasoft.com/products/JavaManagement/index.html` >

[20] Java Dynamic Management Kit < `http://www.sun.com/software/java-dynamic/` >

[21] C. Wellens and K. Auerbach. "Towards Useful Management". In *The Simple Times*, 4(3):1-6, 1996.

[22] B. Bruins. "Some Experiences with Emerging Management Technologies". In *The Simple Times*, 4(3):6-8, 1996.

Index