Professional Thesis

XML and the Push Model in Web-Based Management

Claire LEDRICH

August 2001

Company : AT&T Labs - Research
180, Park Avenue,Florham Park, NJ 07932, USA
Advisor : Jean-Philippe Martin-Flatin
Supervisor : Refik Molva, Eurécom

Corporate Communications

Institut Eurécom

Abstract

The management of IP Networks has been in constant evolution throughout these years. SNMP is the most commmonly used protocol to manage a network device in IP Networks. SMI is used for standard data representation. And yet, with the growth of Internet technologies as well as distributed applications, the shortcomings of SNMP have become more and more obvious. In this project, we further developed the Java Management platform JAMAP, a research prototype that implements the WIMA architecture designed by J.P Martin-Flatin in his Ph.D. Thesis. JAMAP0.4, released at the end of July, fully implements the WIMA architecture and focuses on distribution aspects ad XML.

Resume

La gestion des réseaux IP a été en constante évolution ces dernières années. SNMP est le protocole le plus communément utilisé pour gérer une machine sur des réseaux IP. SMI est utilisé pour la représentation standard des données. Et pourtant, avec la croissance des technologies Web ainsi que des applications ditribuées, les manques de SNMP sont devenus de plus en plus évidents et de nouvelles techniques ont été élaborées dans le domaine des réseaux et des systèmes IP. Durant ce stage, nous avons poursuivi le développement de JAMAP, un prototype de recherche qui implémente l'architecture WIMA décrite par J.P.Martin-Flatin dans sa thèse de doctorat. JAMAP0.4, disponible depuis fin Juillet suit l'architecture de WIMA et insiste sur les aspects de distribution et sur XML.

Acknowledgments

This internship took place at AT&T Labs - Research in Florham Park, New Jersey. I simply wish to thank my advisor Jean-Philippe Martin-Flatin for giving me the opportunity to work on such an interesting project, and for all his help and advice during this first real work experience. I also wish to thank Chuck Kalmanek for his enthusiasm. I sincerely thank Gina Wright for her help.
I am also grateful to all JGuru newsgroups contributors who offered their help on various subjects.

Contents

Chapter 1

Introduction

1.1 Background

During these six months spent at AT&T Labs - Research, I got to know more about the IP world. More precisely, I worked in the field of IP networks and systems management. Up to the twenty-first century, the management of IP networks and systems has relied exclusively on SNMP, the protocol and the management architecture. SNMP was simple and allowed great interoperability. The first RFC for SNMP, SNMPv1 was isssued by the IETF in 1990 and three years later, every vendor had to support SNMP on all its devices. SNMv2 did not work that well and in 1998, SNMPv3 was issued by the IETF. But the relevance of SNMP was put into question. Actually, customers wanted to be able to manage systems as well as networks. Whereas most IP netowrks were mostly managed with SNMP-based management platforms, the management of IP systems relied on proprietary non-SNMP systems. In fact the situation at the end of the 90s had become radically different from the situation when SNMPv1 was issued by the IETF. Customers need to integrate networks, systems, application, service and policy management... which was not the case in 1990. Another problem was the object oriented paradigm and the development of distributed applications. The main points to work on are: the network overhead, data compression, manage devices across firewalls and alleviate the workload on managers. There are many alternatives, from Web technologies to mobile agents, CORBA or intelligent agents. None of them have really succeeded yet.

1.2 My Contributions

In his PhD work, Jean-Philippe Martin-Flatin, my advisor, has tackled all of the previous problems and has created a new management architecture called WIMA. JAMAP, a research prototype was developed and implemented some of the features decribed in WIMA. My task at AT&T has been to focus on the distribution aspects and the com-

munication model so as to implement all the features described in Jean-Philippe Martin-Flatin's PhD thesis. I have therefore tested JAMAP on an alost real-life environment at AT&T.

1.3 Outline of the report

In Chapter 2, we take a look at the communication model WIMA and its specifications. Since JAMAP relies heavily on Web technologies, in Chapter 3, We identify the Web technologies used in the prototype. After this background information, we actually look at JAMAP architecture. In Chapter 4, we try to get the big picture through looking at its high level design and in Chapter 5, we detail its main functionalities. Finally we conclude in Chapter 6.

Chapter 2

The Communication Model

2.1 Push Model vs.Pull Model

In Network Management the push and pull models are two different approaches for exchanging data between an agent and a manager. In the pull model, the manager(i.e.,the client) sends a request to the agent(i.e., the server) which the server then answers. This is called *polling* and is the most commonly used management-data transfer in SNMP management architectures for regular and *ad hoc* management. And the push model is usually only used for sending notifications. Martin-Flatin showed that the push model suits regular management better than the pull model[3] . WIMA actually advocates the use of the push model both for regular management and notification delivery.

2.1.1 Why not use the Pull model ?

First, using the push model saves some network bandwidth. As a matter of fact, the network overhead generated by polling is caused by the redundancy in the messages exchanged between the manager and the agent.

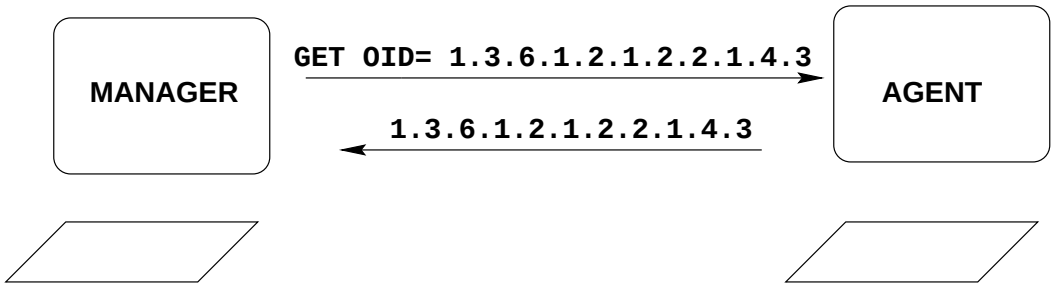

Figure 2.1: Redundancy in the Pull model

With the push model, instead of asking regularly for the value of the same variable and getting a response, the agent sends regularly the value of this variable without being asked to send it each time. It is estimated that the network overhead is roughly divided by two. Then, the agent in the push model is more intelligent. In the SNMP management architecture, the manager is usually responsible for everything. Agents are more powerful than they used to be and cost less than managers. Therefore, it is possible to transfer some of the workload to the agents. This improves the scalability of the whole management system.

2.1.2 The Push model

The Push model is based on publish-subscribe and consists of three major phases:

- Publication Phase
- Subscription Phase
- Distribution Phase

The Publication Phase

During this phase, the agent publishes the information models(SNMP MIBs, CIM Shemas, etc.) and the kind of notifications it can send. These data are available at fixed URLs that the administrator will visit later.

The Subscription Phase

During the Subscription phase, the administrator browses the network map, goes to the agents' URLs and loads the Subscription applets on the agent via a secure relay. Using the DataSubscriptionApplet, the administrator decides which MIB variables and at what frequency he wants to subscribe to. Using the NotificationSubscriptionApplet, the administrator decides which notifications the manager should receive.

The Distribution Phase

The Distribution phase consists in delivering MIB Data and notifications to the manager. Regarding data collection, we have a PushScheduler on the agent that tells when to send which data to which manager. Once the manager receives the data, it applies a set of rules to the values of the variables and decides whether or not to generate an event. As for delivery, the agent is supposed to have a health monitor that regularly checks the agent's health. When a problem is detected, the health monitor sends a message to a notification generator that generates a regular SNMP notification that is

sent to the management server. The management server therefore receives two types of events: notifications sent spontaneously by the agents and events generated on the management server through applying rules. An event correlator on the event manager then analyzes the events and tries to find the root cause for these events in order to reduce redundancy and drop some events. The architecture of JAMAP will be analyzed in detail in Chapter4 and Chapter5.

2.2 HTTP-Based Communication

2.2.1 Why use HTTP for the communication between agent and manager ?

Martin-Flatin chose to use a persistent connection between the manager and the agent[3]. For security reasons, if we need to go across a firewall for instance, this persistent connection needs to be initiated by the manager not the agent, which means that communication must be initiated by the client in a client-server architecture. To do this, distributed programming gives us three communication technologies:

- Sockets
  They have two major advantages: they are bidirectional and easy to program with. But they also have certain drawbacks which are time-outs and firewall crossing. Actually, if the push period is a bit longer than the time-out value, the manager will have to reconnect to the agent, which will cause CPU overhead. The second problem appears if we need to go across a firewall between managers or manager and agent. Most firewalls filter out UDP, and let through only a few ports for TCP. Large organizations can manage these last security issues but small and midsize enterprises often cannot.

- Java RMI
  Like simple sockets, Java RMI offers bidirectional communication.In addition it is fully object-oriented. But using Java RMI also has several drawbacks: it requires that the agents embed a full JVM, which most price-sensitive devices do not also, Java RMI is slow to execute and costs a lot in CPU and memory. Furthermore, the communication between RMI clients is based on firewall-sensitive sockets, hence the same issue as described above.

- HTTP
  Seeing that the previous techniques were not firewall-friendly, Martin-Flatin decided to base IP Network management on HTTP. HTTP is connection-oriented, which means that one cannot establish a connection in one direction from the client

to the server and later send data in the opposite direction. For security reasons, the manager is to initiate the communication since the manager is trusted compared to the agent which can be anywhere outside the network. The client-server model is reversed here[3] . The client is the manager and the server is the agent. It is a request-response protocol. The connection has to be established by the manager but according to the push model, the actual data transfer is initiated by the agent. So how can we have the server send an infinite number of replies to one single request fron the client ?

2.2.2 MIME Multipart

HTTP can be used for pushing data, except that you have to give an explicit response, contrary to SNMP with which you send explicit requests (get, set etc). Here Martin-Flatin used the same idea as Netscape, that is, the server will send an infinite reply to one single GET request, using the multipart type of MIME. At each Push cycle, the agent will send one MIME Part including the descriptions and values of all the MIB variables the agent had subscribed to. Notifications and Management Data use two different TCP connections.

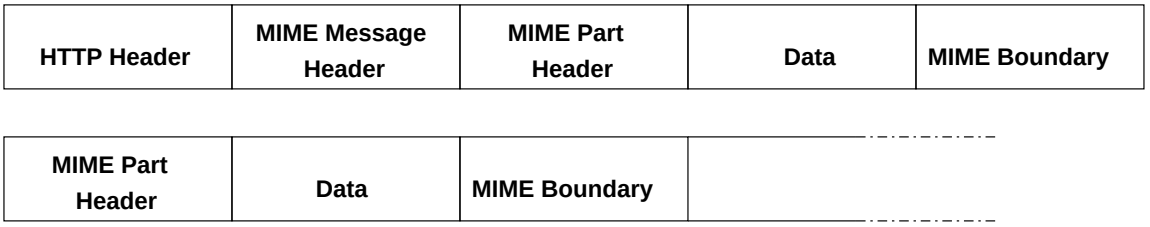

Figure 2.2: Example of HTTP and MIME headers

2.2.3 Self-Describing Data

As said before, we use MIME to separate different bulks of data. In order to describe the type of Management data transferred fron the agent to the manager, we use the Content Type field of the MIMEPart. This parameter has to define two different things: the information model: whether it is SNMP, if yes, which version, ot if it is CIM or OSI the encoding: whether it is XML, Serialized Java, BER etc. Let's have a look at examples of MIME types:

- SNMPv1-to-XML
- SNMPv2-to-SerializedJava
- SNMpv3-to-BER

Here we only studied the Serialized Java and XML encoding. In order to avoid any compliance problems, Martin-Flatin decided to use this format:

`Content-Type:application/mgmt;mapping:RFC2571-to-XML;version=1` or `Content-Type:application/mgmt;m`

With this flexible Content-Type model, any new information model or any new encoding could appear, WIMA would be able to support it. Here is an example of HTTP and MIME part headers for a response sent by the agent after receiving a request from the manager.

2.3 XML

2.3.1 Why use XML preferably ?

JAMAP allows us to use any type of data representation, but there are several advantages to using XML. First, XML is easy to learn and it is not expensive, since most XML editors and parsers are available for free. Second, XML is application-domain independent and allows project managers in software development to save money. Third, XML can easily be understood by users, which simplifies debugging. Fourth, XML has a low footprint on the agent. One drawback of XML is that parsers available tfay are slow, and validation is inherently slow.

2.3.2 Management Data Representation in XML

Management Data encoded in XMl is self-describing and allows multiple information models. For example, a manager may well be in charge of two different management domains with two different information models. In the release of JAMAP that we developed during this project(JAMAP 0.4), the communication model can be based either on : HTTP/MIME/XML or HTTP/MIME/SerializedJava.

Chapter 3

Web Technologies

JAMAP relies heavily on inter-servlet communication and applet-to-servlet or servlet-to-applet communication. These servlets and applets are loaded by HTTP Servers. The HTTP Server we used during the tests was Jigsaw 2.2.0, the latest version to date of the W3C HTTP Server. Early versions of JAMAP were tested with an Apache Server but there were serious buffering issues which made us switch to Jigsaw.[7]

3.1 Servlets

3.1.1 What is a servlet?

Servlets are a Java-based improvement over CGI scripts that were typically written in a script language such as Perl. They are modules of Java code that run in a server application. They are the counterpart of *applets* on the server side. Servlets make use of the Java standard extension classes that are included in the package `javax.servlet` and the package `javax.servlet.http` in the case of HTTP requests. Since Servlets are written in Java, they provide a highly portable means to create server extensions in a server and operating-system independent way.

3.1.2 Why use servlets?

Although the traditional way of adding functionality to an HTTP Server is through the `Common Gateway Interface`, servlets have several advantages over CGI. CGI is a language-independent interface that allows a server to start an external process, which gets information about the request via different environment variables and writes the response to its standard ouputstream. Each request is answered in a separate process by a separate instance of the CGI program. Here are the main differences between the two:

- Servlets do not run in separate processes, which removes the overhead of creating a new process for each request.
- A servlet stays in memory between requests whereas a CGI program would need to be loaded and started after each request.
- A servlet can be run by a Servlet Engine in a restrictive sandbox, which allows for a secure use of untrusted servlets.

The Web server draws a link between URLs and the servlets.In JAMAP we use HTTP servlets instead of CGI servlets. In order to initialize a servlet, at start-up time, a server application loads the Servlet class and creates an instance by calling the no-args constructor. Then it calls the Servlet's init method. The latter is guaranteed to be called only once during the Servlet's lifecycle. When a client performs a request, the server invokes a method of the HTTP servlet depending on the HTTP method used in the request. For example, requesting a URL results on a HTTP GET request, and invokes the *doGet* method of the servlet. As most HTTP servers (including Jigsaw) support multithreading, several clients may invoke concurrently the same method of the same servlet, and the same servlet can be shared by multiple persistent connections.

3.2 Applets

3.2.1 Limitations of applets loaded in a browser

The fact that JAMAP uses applets allows an administrator to keep an eye on the management of his network, whatever his geographical location (at work, at home etc). As we will see in Chapter4, the administrator only needs to have a computer with a browser. The applets in JAMAP and their Graphical User Interfaces (GUI) use Sun's Swing toolkit. Even though Swing is resource-hungry, it is still one of the best toolkits today, especially with its look and feel effects. Here we are using JDK1.2 applets whose security model is much more elaborate than JDK1.1 applets.

3.2.2 The security model of JDK1.2 applets

In JDK1.2, applets running in a browser run in a restricted sandbox-like environment. The new architecture they benefit from lets users grant Java applets permission to access certain specific system resources outside their restricted environments. The new architecture is very flexible, but because the default behavior for applets running in restricted environments is no access to system resources, all access to system resources such as file systems or networking facilities is not allowed unless specifically granted. In JDK1.2 security, there are 3 keywords with which we have to deal:

- Permissions:

  They are the core of Java Security and represent access to various resources such as files, sockets etc. A security *policy* is made of numerous *permissions*:

- Policy:

  The mapping of these permissions to classes is referred to as *policy* and written in a policy file.

- Security Manager: The Security Manager then reads the policy file and grants or forbids to certain resources depending on the permissions it read.

One of the main problems of running applets in a browser context is that a user does not have easy access to the options for running the JVM. There is no simple way to deploy and use customized policy files.

3.2.3 The Java plug-in

The lack of support for the latest version of the JRE in the default JVM of Netscape is solved by using the Java plug-in. It supports the standard Java 2 SDK(i.e. JDK1.2.x) and its security model. All applets run under the standard applet security manager, which prevents potentially malicious applets from performing dangerous operations such as reading local files. RSA-signed applets can be deployed using the Java plug-in. Additionnally, the plug-in runs applets the same way in both Netscape Navigator and Internet Explorer. So we used the Java plug-in and the HTML Converter, which tells the browser to use the JRE defined in the plug-in and not its internal JVM. By default, the applets still generate a security exception due to the restrictions placed on downladed code. Here, since we were working in an Intranet, we decided not to use signed applets, but simply to manually define the installation security policy, since the applets that need special permissions are only those on the Data collector trying to acces the agent servlet or the applets on the Event Manager trying to access the data collector. Scalability is not a severe issue here and we can afford to sign applets, the process of importing signatures to the "client" machines will always be feasible. JDK1.3 brought several enhancements, including full support for RSA signatures and full interoperativity with Verisign's code-signing certficates.

3.3 Java Serialization

Serialization allows any complex Java object to be translated into a byte stream. The goal is to represent the state of this object in a serialized form sufficient to reconstruct the object as it is read. It can be used in two different ways:

- Remote Method Invocation (RMI) i.e. communication between objects via sockets.

- Lightweight persistence, i.e. a way to archive an object for use in a later invocation.

Two streams *ObjectInputStream* and *ObjectOutputStream* are used to write and read objects. Objects containing references to other objects are processed to serialize (and later deserialize) all required objects. The keyword *transient* in object declaration is used to prevent this object to be serialized. This is useful when you protect sensitive information or functions. For example a file descriptor contains a handle that provides access to an operating system resource and being able to forge a file descriptor would allow some form of illegal access, since restoring state is done from a stream. In JAMAP, Serialization was used for both reasons. It provided a simple way to communicate via sockets, before XML was used, and the lightweight persistence provided by serialization allowed us to store rules and the agents' configuration.

3.4 XML

3.4.1 XML Overview

XML is a technology for the Web. HTML is a *Markup Language*, and was designed for hypertext, not for information in general. XML marks up content with tags to convey information. The tags delimit the content and XML syntax allows us to define structures of arbitrary complexity. Allof this is done with ordinary text, not binary data formats. XML allows general meta-information, has built-in internationalization and platform-independence, and is on its way to be the format for structuring information. An XML document is simply a text with markup tags and other meta-information.

3.4.2 XML DTDs and XML Schemas

When working with XML, you have two different options:

- XML Document Type Definitions (DTDs)
- XML Schemas

XML DTDs represent a simple way to specify the rules by which XML documents are written. A DTD is a set of information that explains the rules used by a designer to extend the core rules of XML syntax and be able to describe an application domain. In order to cope with an increasing complexity of XML documents, one may refer to several DTDs in one application; That resulted in name collisions and ambiguity. To address this issue, the W3C decided to develop XML Schemas and namespaces. According to the W3C's Recommendation 'Namespaces in XML' , a namespace is :

A collection of names, identified by a URI reference, which are used in XML Documents as element types and attribute names.

XML Schemas, conversely, are written in XML, allow the use of multiple namespace, and provide for strong typing of content. We will show a very short example of an XML DTD and its translation in XML Schema:
from R.Anderson,M.Birbeck *Professional XML*, Wrox Press 2000.

```
<!ELEMENT Name (Honorific?, First, MI?, Last, Suffix?)>
<!ELEMENT Honorific (PCDATA)>
<!ELEMENT First (PCDATA)>
<!ELEMENT MI (PCDATA)>
<!ELEMENT Last (PCDATA)>
<!ELEMENT Suffix (PCDATA)>
```

'?' stands for optional. The corresponding XMLschema:

```
<Schema ...>
<element name=''Name''>
<type>
<element name=''Honorific''
type=''string'' minOccurs=''0'' maxOccurs=''1''/>
<element name=''First''
type=''String''/>
<element name=''MI''
type=''String''minOccurs=''0'' maxOccurs=''1''/>
<element name=''Last''
type=''String''/>
<element name=''Suffix''
type=''String''minOccurs=''0'' maxOccurs=''1''/>
</Schema>
```

3.4.3 The choices made for JAMAP

In JAMAP, we used XML Schema and the event-based Simple API for XML to process XML messages with the XML parser from jclark , from the package `com.jclark.xml.sax`

as advised by R.Anderson in *Professional XML*

Chapter 4

JAMAP High-Level Architecture

4.1 Three - Tier architecture

In a client-server model, we can have two different architectures:

- Two-Tier architecture It is the simplest of all client-server architectures. In this model, the client talks directly to the database, without any intervening server. It is typically used in small environments. The server is a more powerful machine that services the client with a database management system. It accesses databases either on the same machine or behind it. The client then has both presentation logic and application logic responsibilities.This model has the advantage of being very simple but it is not scalable. When yo uhave more than 100 users, performance starts to deteriorate. Below is an example of a two-tier architecture.

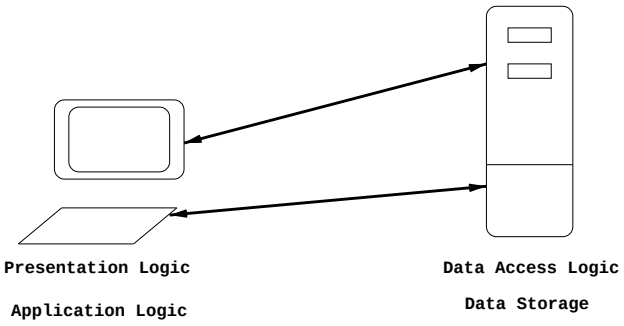

Figure 4.1: Two-Tier Architecture

- Three-Tier architecture This one is more complicated. A three-tier architecture addds a middle-tier between the client and the database management server. This middle layer can be used for a number of things, including application servers. In JAMAP, the three-tier architecture is not completely implemented.For now the Management Server and the Data Server are on the same machine. We have not used a formal data server yet. Briefly, the regular manager is split in three(see Figure 3.2):

  1. The Management Station The machine responsible for the presentation logic, any machine with an embedded browser.
  2. The Management Server The machine resposible for application logic, a fixed powerful dedicated machine.
  3. The Data Server The machine responsible for the data access logic.

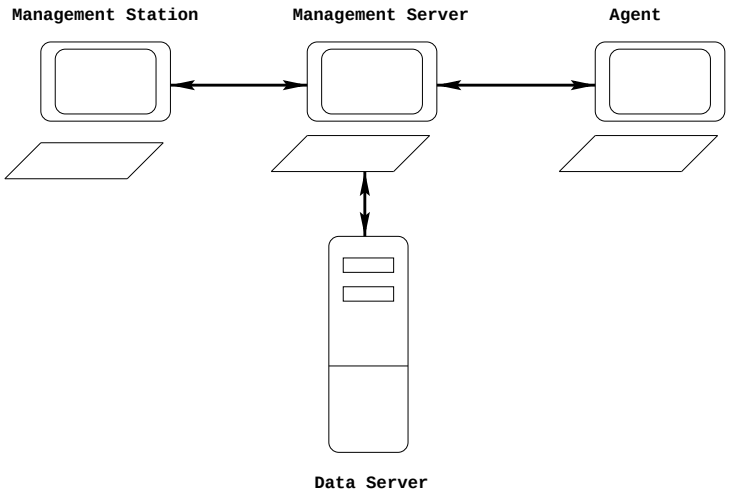

Figure 4.2: Three-Tier Architecture

We have three kinds of interactions:

1. Between the Management Station through the Management Server, and the Agent

   This is the case when the manager on the Management Station wants to subscribe to MIB variables. The Management Server is used as a proxy to authenticate the manager. Such a three-tier architecture allows us to control the connections to the agent in case of a firewall between the Management Station and the Management Server and betweem the Managent Server and the Agent.

2. Between the Management Station, through the Management Server, and the Data Repository

During the subscription phase, the Agent should communicate its configuration, ie its SubscriptionTable to the Data Server.

3. Between the Agent and the data repository

   This is the case during the distribution phase, the data leaves form the agent, is analyzed within the Management Server and is logged in the Data Server.

4.2 JAMAP Overview

The *big picture* of JAMAP is depicted in Figure 3.3. The links with arrows indicate interactions between components in one or more directions. As we said before the Management Station can be any machine as long as is has an embedded Web browser. The Management Server is a server machine, in our tests, the Management Server was in two parts, a Data Collector and an Event Manager, each of them on a different machine. The agent is a network device which could well be low-cost.

4.3 Management Station Applets

4.3.1 DataSubscription Applet

Proxy

To access the Data Subscription Applet, the manager browses the Network Map and chooses the agent(s) he wants to manage. This applet communicates with the agent through a proxy run on the Management Server. First the manager has to go through a login servlet and authenticate himself then he is redirected to the page he wanted to reach originally. This way, there is never a direct potentially unsafe connection to the agent. This proxy allows managers to control what information goes to the applet. The proxy servlet handles URL of that form:

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

This will make the browser think that the request originally came from: `www.myPC.com/appletpage.html`. The only problem with proxies is that they can become bottlenecks in case too many clients use them.

The DataSubscription Applet

After choosing the Information Base he wants to browse (for example MIBs and which MIB), the manager access the web page with the DataSuscriptionApplet. The Data Subscription Applet has two different sides. On one side, the applet provides the sub-

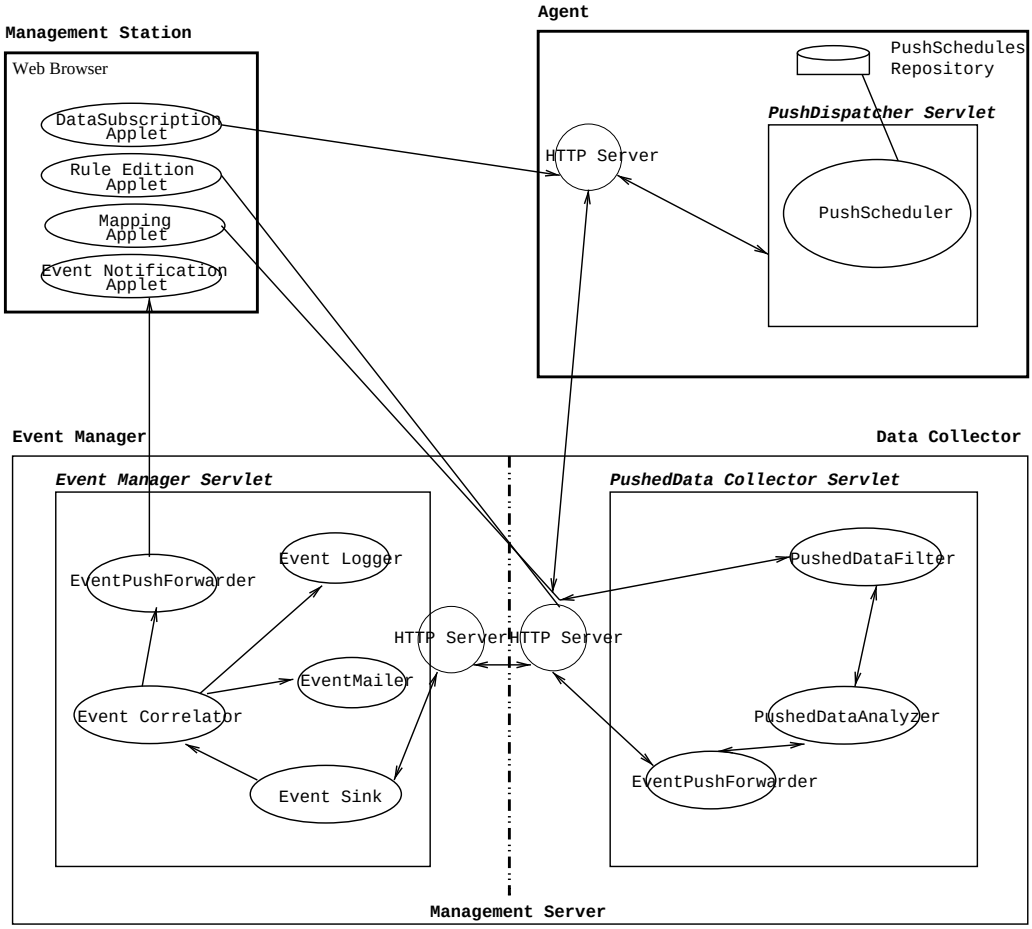

Figure 4.3: JAMAP Architectural Overview

scription system to configure regular management. To do so, the manager must choose the following parameters:

- The Management Data Id of the variable he wants to subscribe to.
- The frequency at which he wants the agent to send the subscribed data.
- The representation with which he wants the data to be sent.

The data collector the data will be sent to is not chosen by the manager. A data collector has an agent table that tells it to which agents to connect to. A subscription is then characterized by these different parameters. and sent to the agent. On the other side, this applet can also be considered as a tool for ad hoc management. The user can perform the following tasks :

Figure 4.4: Secure Relay

- Browse an Information base in a convenient interface.
- Choose a table or a variable and immediately *Get* its value.
- Monitor some computed values such as interface utilization with a *Spy*

Figure 3.5 depicts the Data Subscription Applet Graphical User Interface.

4.3.2 The RuleEditor Applet

From the home page, the manager chooses the data collector he wants to edit the rules for and loads the corresponding applet. This applet is just here to allow the manager to either write rules from scratch or load template files for InstantaneousRule or TemporalRule. The rules are then saved and compiled dynamically to make sure there are no syntaxic errors. The manager can also choose to edit a rule that was previously compiled by clicking on*Modify Rule* The Dialog window is there to check if the compilation encoutered problems or not. The rule is then saved after the name of the template chosen.

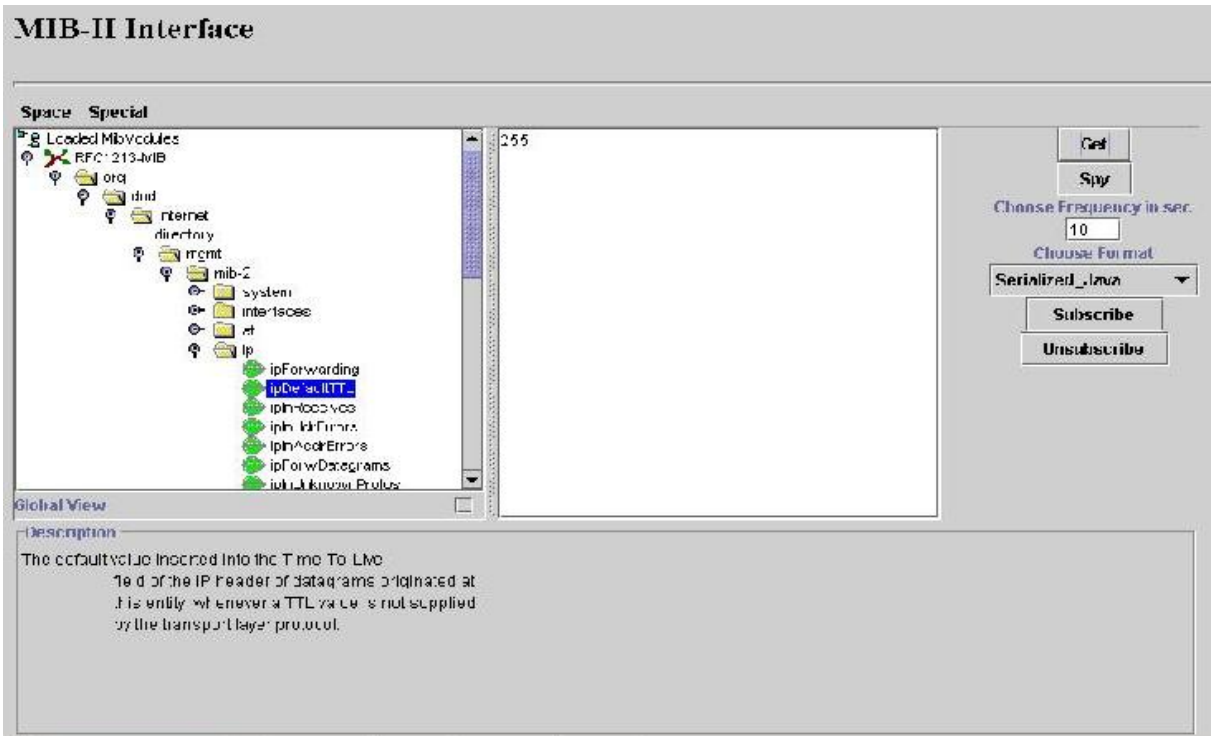

Figure 4.5: mib-II.html/DataSubscription Applet GUI

4.3.3 The Mapping Applet

After going through the Rule Editor Applet, the manager has saved a rule *draft*. No rule is activated yet. The manager still has to go through the Mapping Applet before he can apply or activate a rule. Since the manager may have subscribed to the same OID but with different frequencies each time. The mapping applet allows the manager to actually *map* an OID and the frequency at which the variable is sent with a rule that will be applied to this OID, arriving at this frequency, and from this agent only. The manager first chooses the agent, then the rule and the oid he wants to apply the rule to. Then he loads the subscriptions for this oid on this agent and maps the rule with the latter. This applet also allows the manager to temporarily deactivate or recativate a rule that was previously compiled. The manager can map a rule either before he launches the collectors and the event manager or afterwards while the agents are already sending data to the collectors. In that case, the Dialog window would say that the rule was put into production.

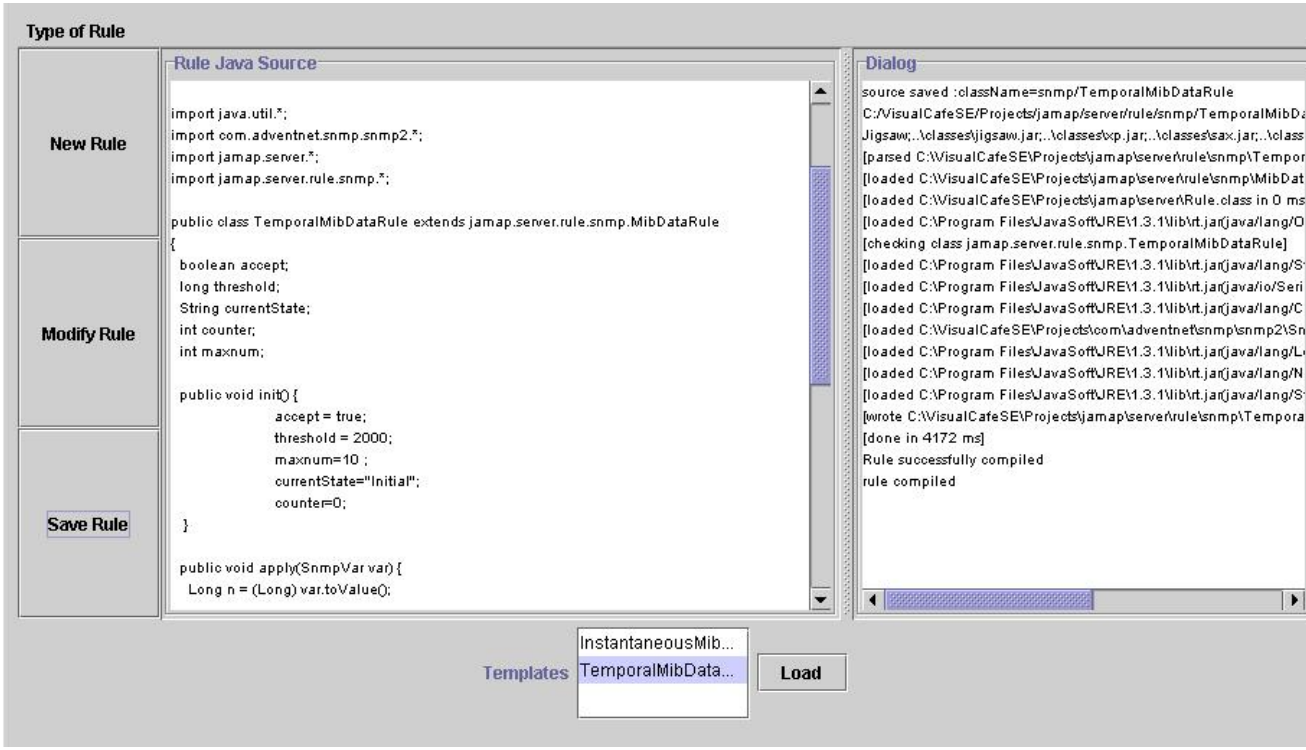

Figure 4.6: RuleEditor Applet GUI

4.3.4 EventNotification Applet

As we will see later, when events are generated,depending on their severity they are either just logged through an EventLogger or they are logged and sent to an EventMailer that will warn the administrator by sending him/her an email.. This applet communicates with the EventManager to receive all incoming events. The EventLogger keeps the list events in its memory and transmits them to the EventNotificationApplet. For each incoming event, a line is added to the list of previous events and the frame blinks and rings ten times.

4.4 Agent Servlet

The Agent servlet is called the PushDispatcher servlet. We coud have separated the agent servlet in two as it realizes two different tasks. First, we can distinguish the configuration part in which the agent stores its subscriptions received from the Data-Subscription Applet in a local repository. Then, to map the rules to a precise oid, we already resorted to the agent servlet that gives the subscriptions associated with each oid. A subscription sheet can be accessed from the web page to check the subscrip-

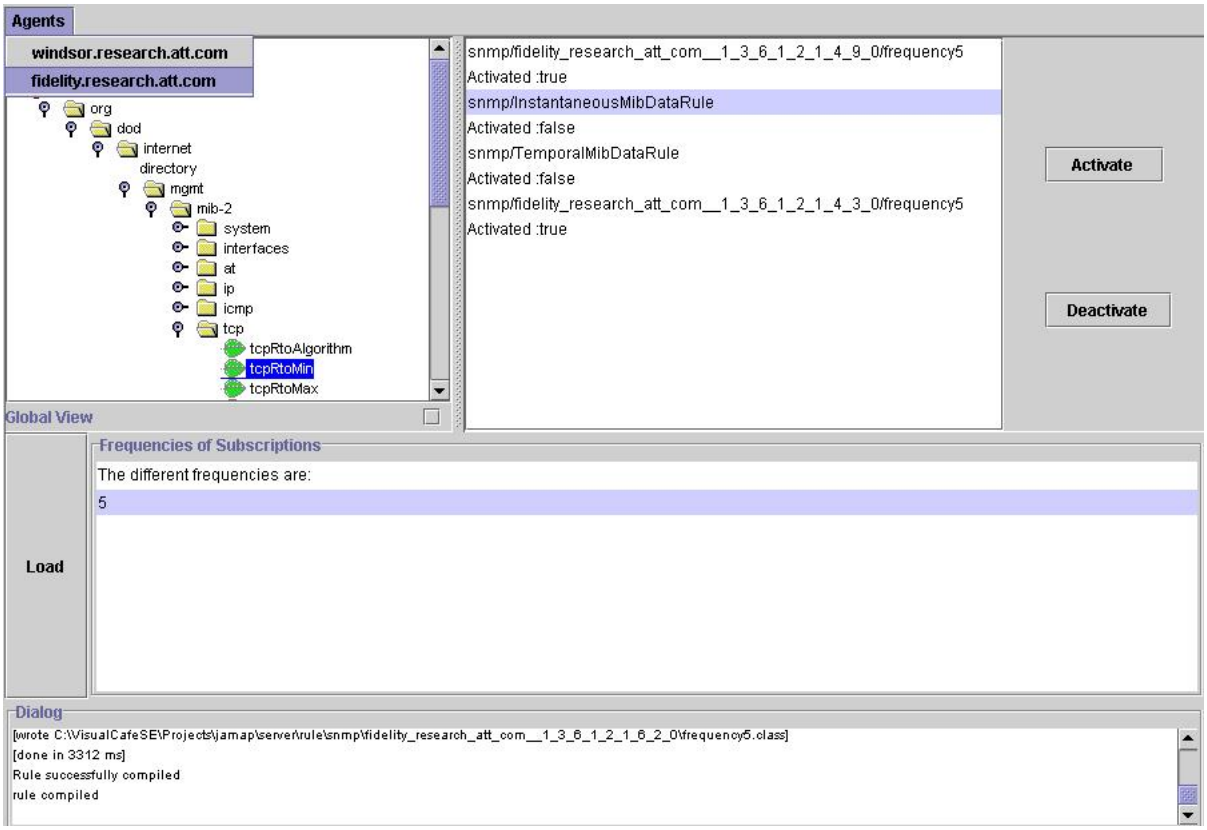

Figure 4.7: Mapping Applet GUI

tions on the agent. Secondly, the agent servlet is the core of the push system and the Management Server has to connect to the agent servlet to start any push cycle.

4.5 ManagementServer Servlets

The Management Server is separated in three parts : The PushedData Collector and Notification Collector and the EventManager. The PushedDataCollector servlet could be separated in two servlets itself, one servlet would realize the configuration part and the other one would take care of the execution of the push system. The configuration part of the PushedData Collector has already been studied within the Management station applets paragraph so we will detail here the execution of the push system. From the home page, the administrator launches the Data Collectors and Notification Collectors that will connect to all the agents they have in their AgentTable. The PushedData Collector and Notification Collector then open an HTTP Connection to their agents and wait for pusheddata or notifications. The PushedDataCollector has a PushedDataFilter

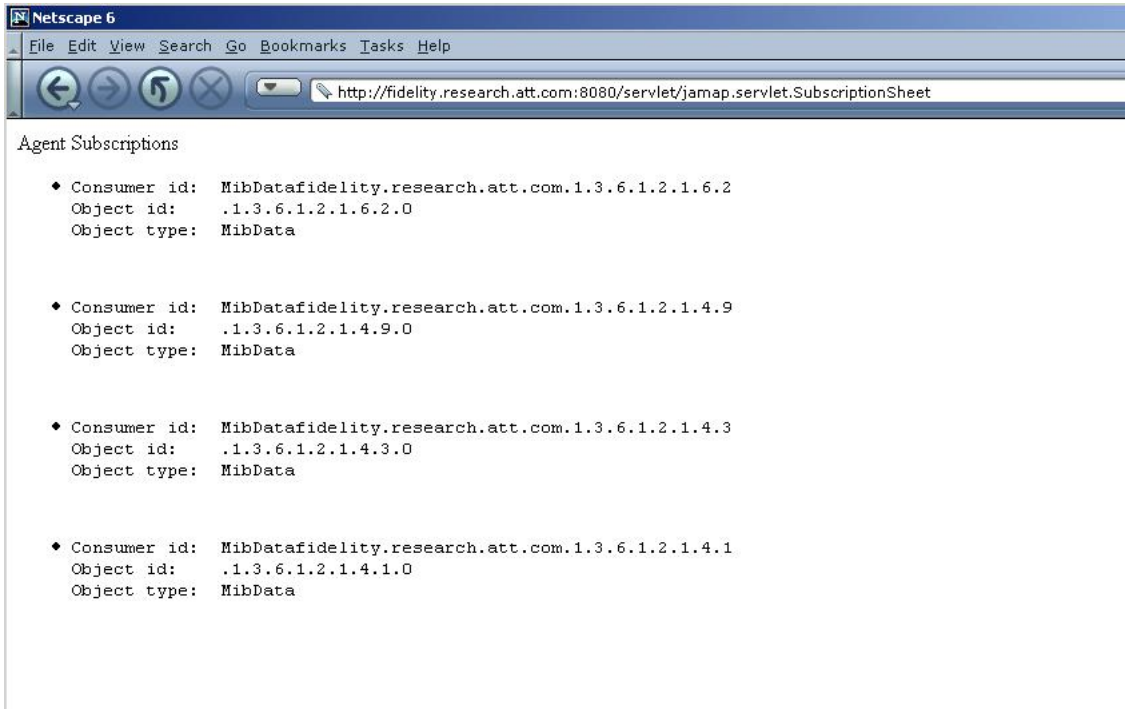

Figure 4.8: Subscription Sheet GUI

and the Notification Collector has a NotificationFilter that will close the connection if too many data per minute get to the PushedData Collector and Notification Collector. As data my be subscribed to only for logging, in order to process them afterwards or perform statistics, all data are logged in a Data Repository. Then the data are analyzed in the PushedDataAnalyzer that has a hashtable of all activated rules. The PushedDataAnalyzer checks if it has rules in its table that corresponds to the data it has just received and if so, and if so, forwards the data to the rule class concerned. The rule class generates events that are forwarded farther through a PushForwardConsumer.

Also from the web page, the manager launches the Event Manager that will connect to its Data Collectors and open an HTTP connection to each of them. Then it waits for pushed events. All events arrive on an EventSink that checks whether too many events were recived or not and if yes, disconnects the Event Manager from the Data Collector. Then events go through an event correlator that processes them and should correlate them. For now, the Event Correlator only has 4 input queues,depending on the severity of the event, and processes them in an order according to their severity. Still depending on their severity, events are sent to event handlers, event mailers or simple event loggers. The event logger will communicate the events to the manager on the Management Station through the Event Notification Applet. Figure 3.9 shows the home

page from which the system is launched:

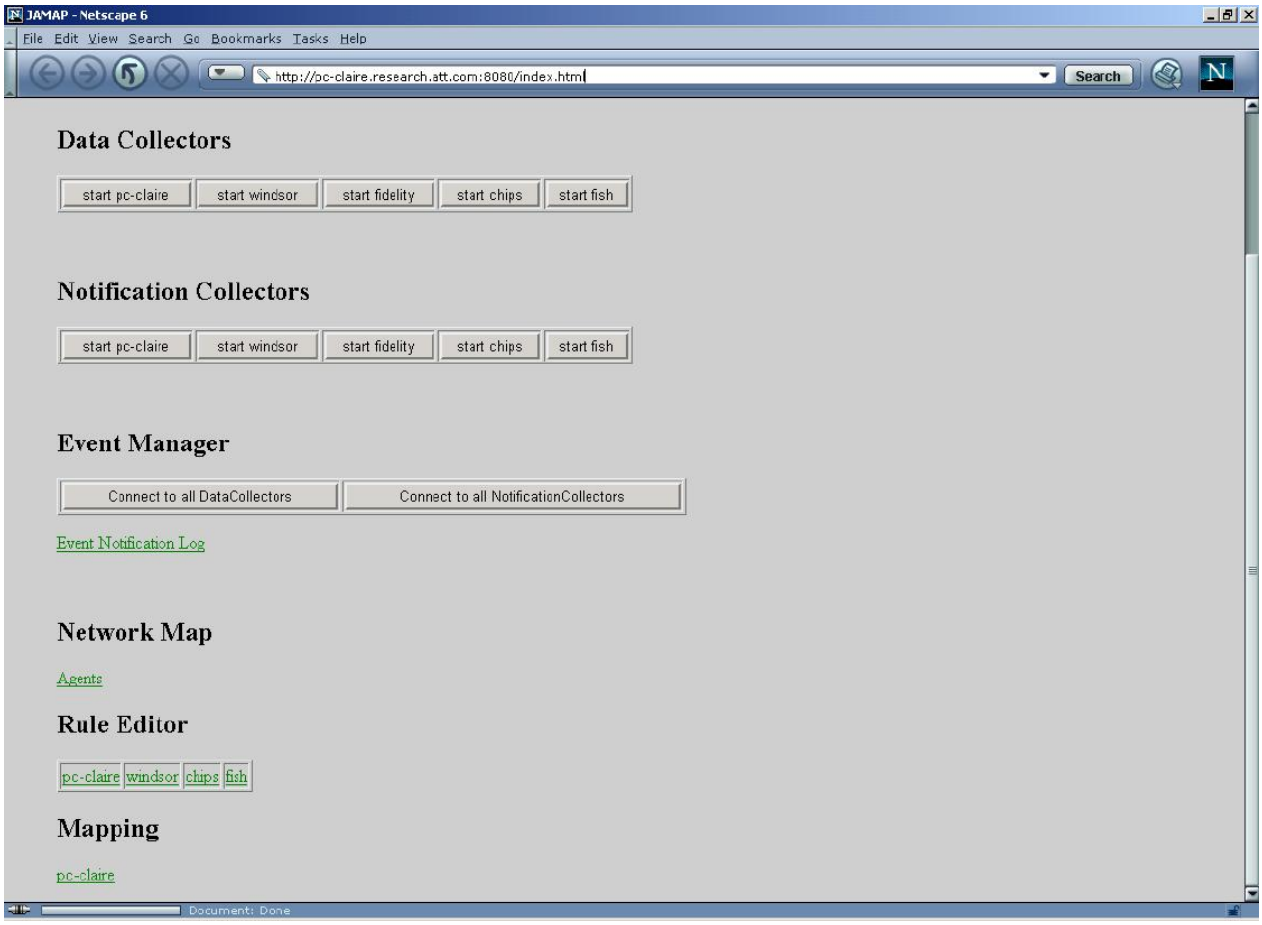

Figure 4.9: JAMAP home page

Figure 3.10 shows how flexible JAMAP is. The network system it was built for can have one or more agents, one or nore Data Collectors, a priori one Event Manager. For example, more than one agents can connect to one data collectors and one or more data collectors can receive data from the same agent as well as one or more data collector can send events to one event manager or one data collector can send events to more than one event manager, although this is not usually the case.

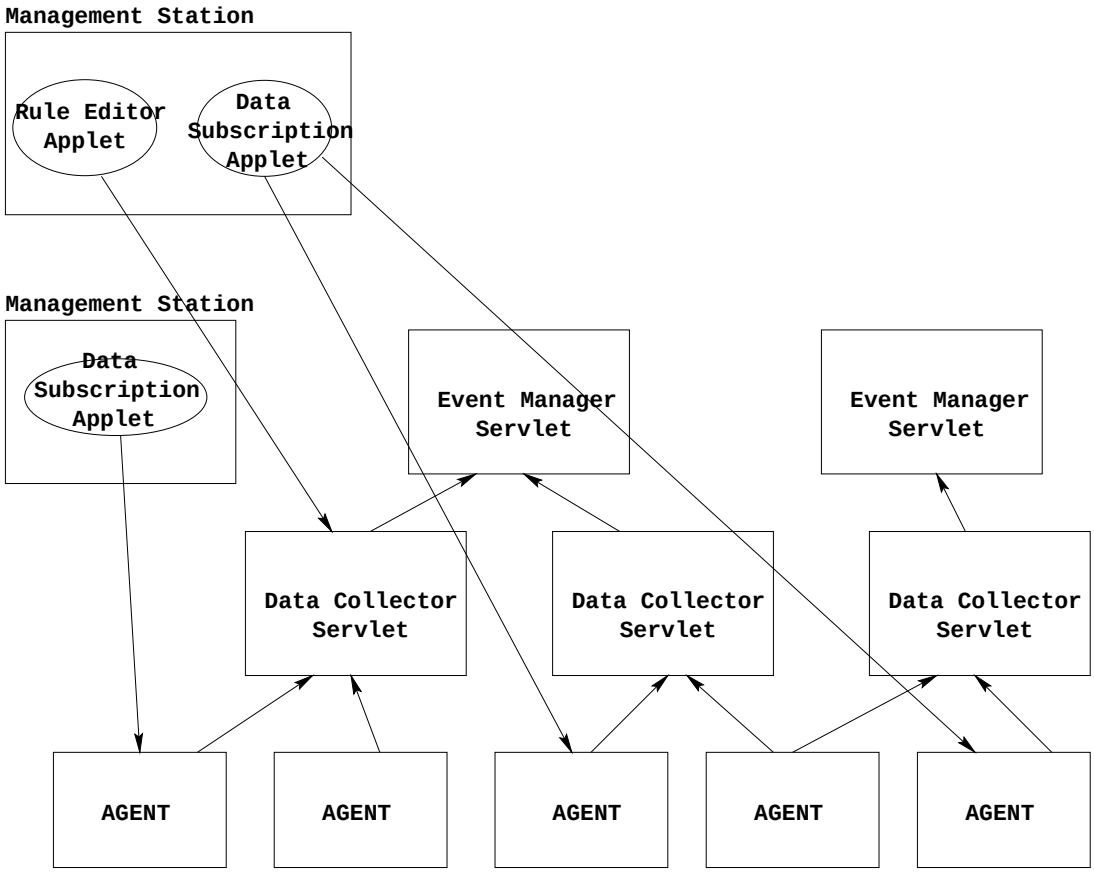

Figure 4.10: An example of Distributed Network Management Platform

Chapter 5

JAMAP Detailed Design

5.1 Communication components

As we saw before, JAMAP follows the publish/subscribe paradigm. Let us see how precisely we go through these phases using applets and servlets.

1. The publication phase corresponds to browsing on the agents' web pages and choosing an Information Base and download the corresponding SubscriptionApplet.

2. The Subscription phase coresponds to sending the subscription from the applet to the agent. In JAMAP, this is done by sending a POST request to the agent's PushDispatcherServlet. The body of the message contains a serialized object of the class Subscription, which identifies the data to push and to which data collector it should be sent.

3. The distribution phase corresponds the agent's sending data to the Data Collector through a fixed push communication path, implemented with HTTP1.0. The PushedData Collector actually sends a GET request to the PushDispatcherServlet and established an infinite connection to it. The response will be infinite, made of MIME parts containing Units. These units are either serialized or have an XML Schema representation.

The administrator can also unsubscribe to stop the data transfer from the agent. This is also done with an HTTP POST request sent to the agent's PushDispatcherServlet. JAMAP uses mostly dispatchers to create and send units and collectors on the other side to receive these units and later distribute them to the consumers concerned.

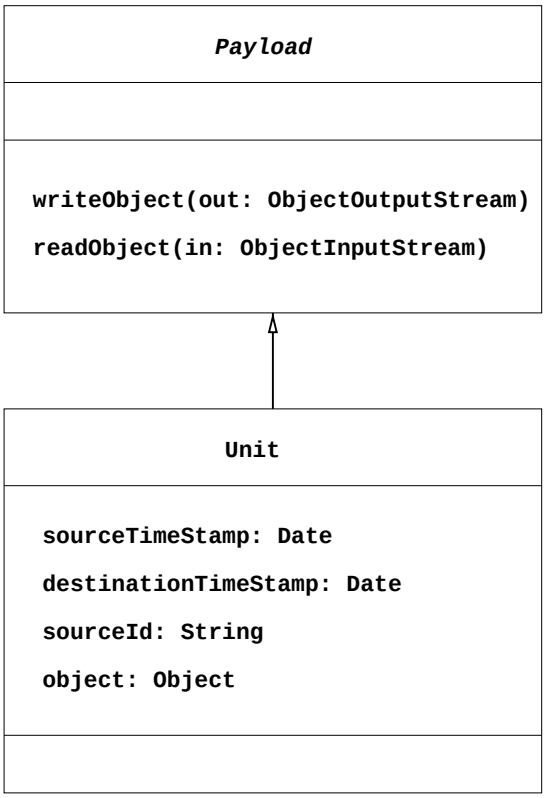

Figure 5.1: The Unit class

5.1.1 Units

Units are the base classes of communication. As shown in Figure 4.1, Unit is a specialization of the abstract class Payload which is the superclass of any class used in communication and is serializable. Units are characterized by two dates, the time they were created and the time they were received on the other side. This is useful in case the network suffers from congestion. Comparing the *sourceTimeStamp* and the *destinationTimeStamp* will tell if the network is congested or not and if yes, this will give a first idea of the hops concerned by the congestion. To identify one also needs to know where it comes from, which is given by the *sourceId* attribute of the class Unit. What this unit carries is the opaque *object* of this unit. It can either be an instance of the class PushedData or an instance of the class Event, or a simple String.

PushedData

A PushedData is sent by the PushScheduler and corresponds to a subscription. It is therefore characterized by the subscription and the value of the variable the manager

subscribed to. The value is of type Object, in case the variable represents a table.

5.1.2 Subscription

The abstract Subscription class contains all the information about the data pushed.

- **objectType**

  This describes the kind of data the manager subscribed to. For instance,*MibData* would refer to an SNMP variable or *Notification* would refer to a SNMP Trap.

- **mgmtDataId**

  This is a unique identifier of the object that the manager subscribed to, it corresponds to the OID of the variable. For instance, if the objectType was *MibData*, this would correspond to an SNMP OID.

- **consumerId**

  The consumerId identifies the final consumer the data must be sent to. For example, for a MIB variable, this will identify the rule that will be applied to the value of this variable. The consumerId for a MIB variable will have the form: *agent.research.att.com__oid_frequency*. This way, the final consumer, ie the rule corresponding to this data is uniquely identified by its Id.

- **representation**

  The agent needs to know which representation(up to now, XML Schema or Serialized Java)it is supposed to use when sending the data.

The DataSubscription class inherits from the Subscription class and stands for regular management push and needs therefore a fourth field to determine the frequency at which the variable with the mgmtDataId given needs to be sent to the Data Collector. The NotificationSubscription class, up to now, simply inherits from the Subscription class.

5.1.3 Event

An Event is sent by the DataCollector to the Event Manager after applying a rule to a PushedData. An event is characterized by:

- its EventType: what kind of event was generated. For now, only two types were implemented, *OVERFLOW* and *Notification*.
- the name of the Rule that generated this event.

- its Severity whether this event is just Informative, Warning, Critical or Fatal.

5.2 The Management Station

The Management Station loads all the applets which will allow the manager to configure JAMAP and actually manage the whole network from a simple PC.

5.2.1 DataSubscription applet

Operations

This is the first step in order to configure the network management platform. This applet uses AdventNet classes for a sophisticated GUI. First it 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 using HTTP GET requests and sending them to the agent in a standard pull model. Secondly, it provides the subscription and unsubscription system for any variable. Subscribing to a variable or unsubscribing is performed using HTTP POST requests sent to the agent. The following figure shows which classes are involved in the Subscription phase. For all applets that we will describe later, the model is the same. Mainly, the applet has one separate GUI, and is a realization of a *Listener* interface, here *DataSubscriptionListener*, that will listen to events.

Classes

The classes used in the DataSubscriptionApplet are depicted in Figure 4.2.

The Get System

From the DataSubscriptionApplet, the administrator can also choose to simply retrieve the value of the MIB variable he wants.

Figure 4.3 shows the Interaction Diagram used when simply getting the value of a variable, without subscribing. The SnmpBase class then uses AdventNet classes to get the value of the given variable. When the administrator actually wants to subscribe, Figure 4.4 shows the Interaction Diagram used when subscribing to a MIB variable. More classes are involved in the Subscribe system than in the simple Get system.

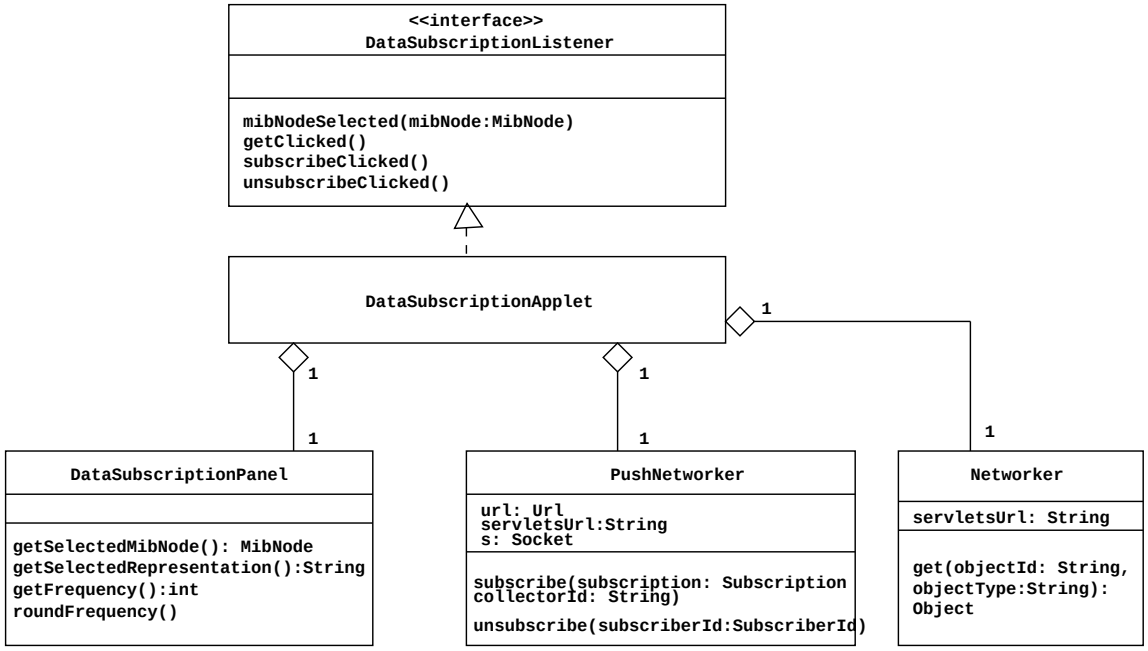

Figure 5.2: The classes used by the DataSubscription applet

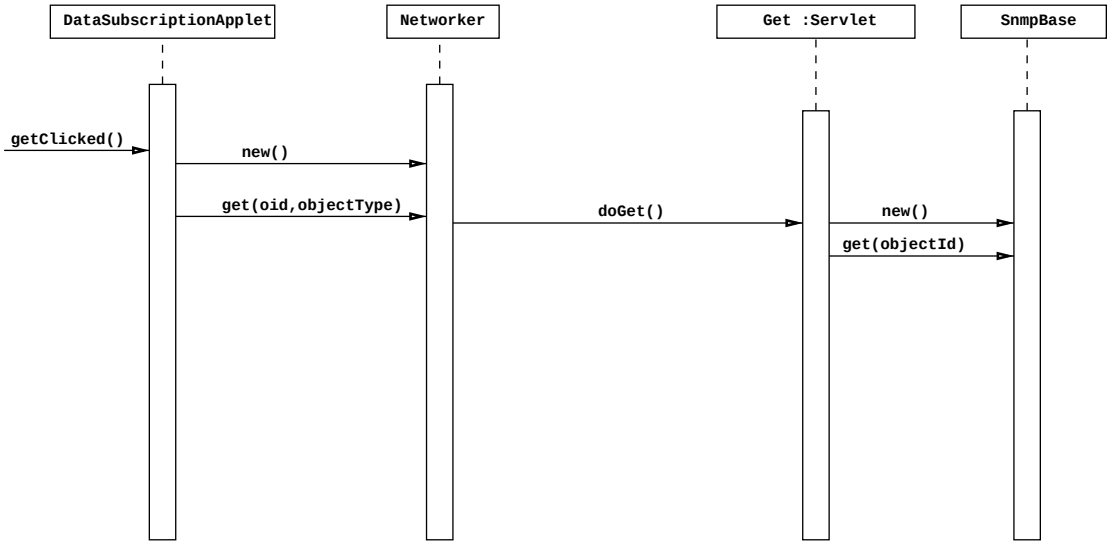

Figure 5.3: Interaction Diagram for the Get system

5.2.2 RuleEditorApplet

After subscribing to some MIB variables, the administrator has to tell the system whats he/she wants to do with the values of these variables. This is considered the

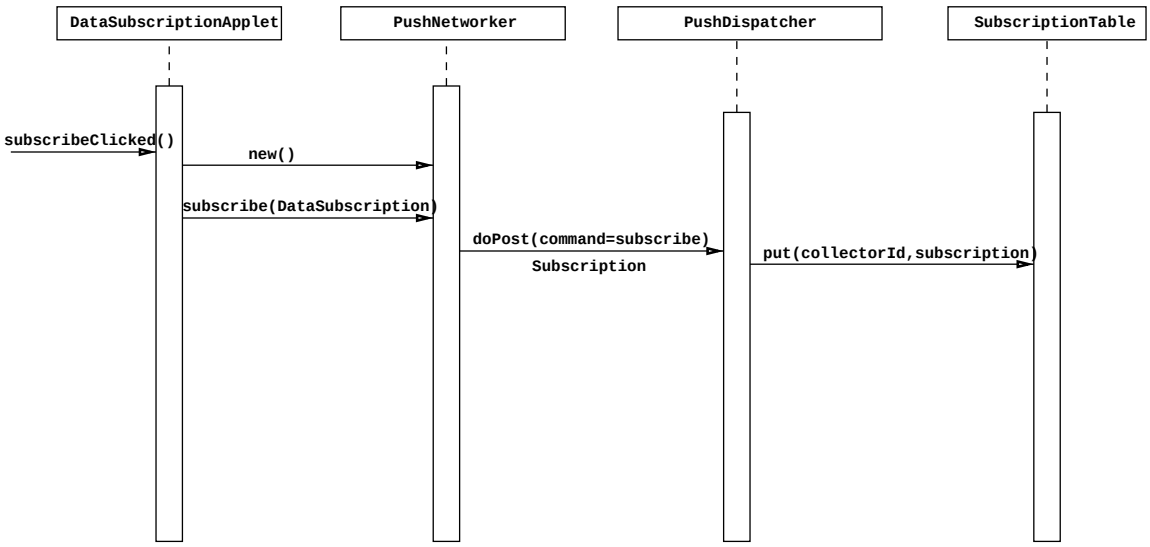

Figure 5.4: Interaction Diagram for the Subscribe system

second step in configuring the network management platform. The manager write rules and saves them on the Data Collector. Here this task is performed using HTTP POST from the Management Station to the Data Collector. Figure 4.5 shows the Interaction diagram used when editing and saving a rule.

5.2.3 Operations

The manager can choose to write a new rule or modify an existing one. If he chooses to modify an existing rule, then the list of all the rules saved on the Data Collector appears on the screen and he can choose which one he wants to edit.

Saving a rule

1. The GUI notifies the applet that implements RuleEditorListener of a click on the SaveRule button.
2. The main controller retrieves the source string of the rule fron the *Source* area.
3. It tells the networker to post the source.
4. It tells the GUI to display the response from the Data Collector in the *Dialog* area.

Classes

The design of this simple applet is depicted in Figure 4.5.

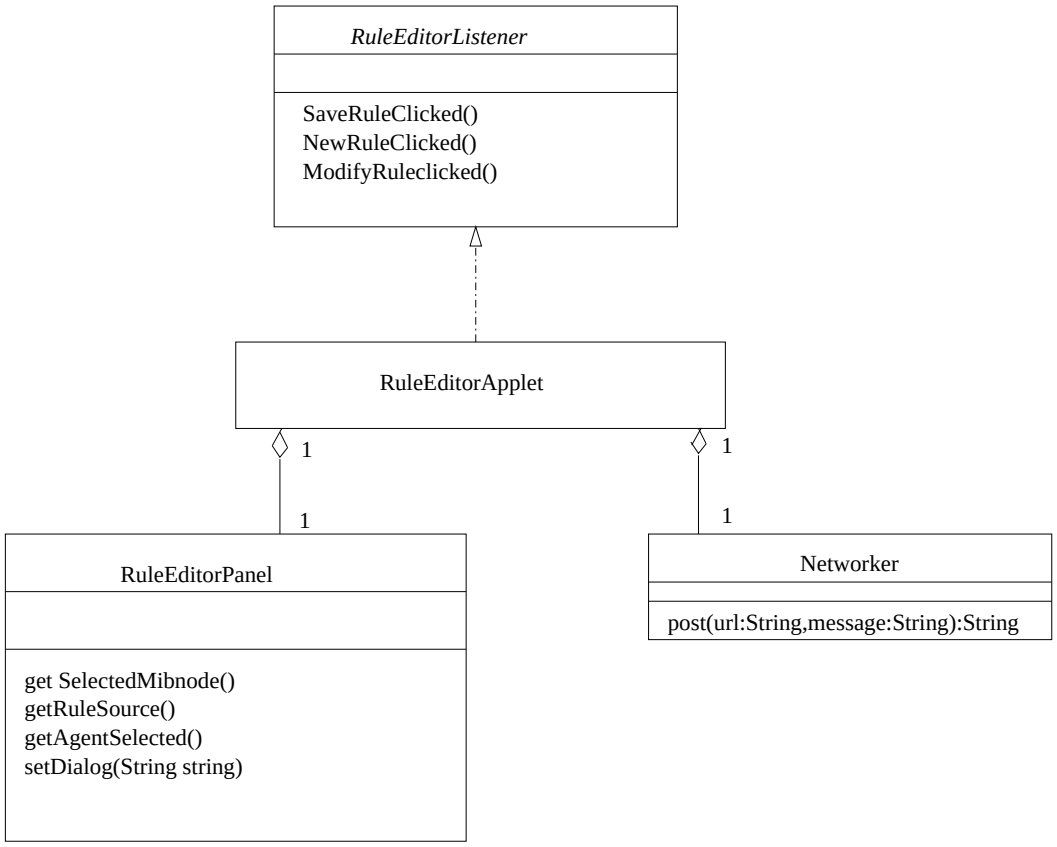

Figure 5.5: Class Diagram for the RuleEditor system

- RuleEditorApplet: Main controller class.
- RuleEditorPanel: Graphical User Interface
- RuleEditorListener: Interface for receiving events from a RuleEditorPanel
- Networker: used for transfer information to the Data Collector.

With the RuleEditorApplet, the manager saves the rule source, which is compiled and the rule is given a name, and saved on the DataCollector. The rule is also added to the RuleTable, a hashtable where the flag activated is set to false.

5.2.4 MappingApplet

This is the last step in rule configuration. The manager associates a rule with an agent, an oid and a frequency. Here this task is performed using HTTP POST from the

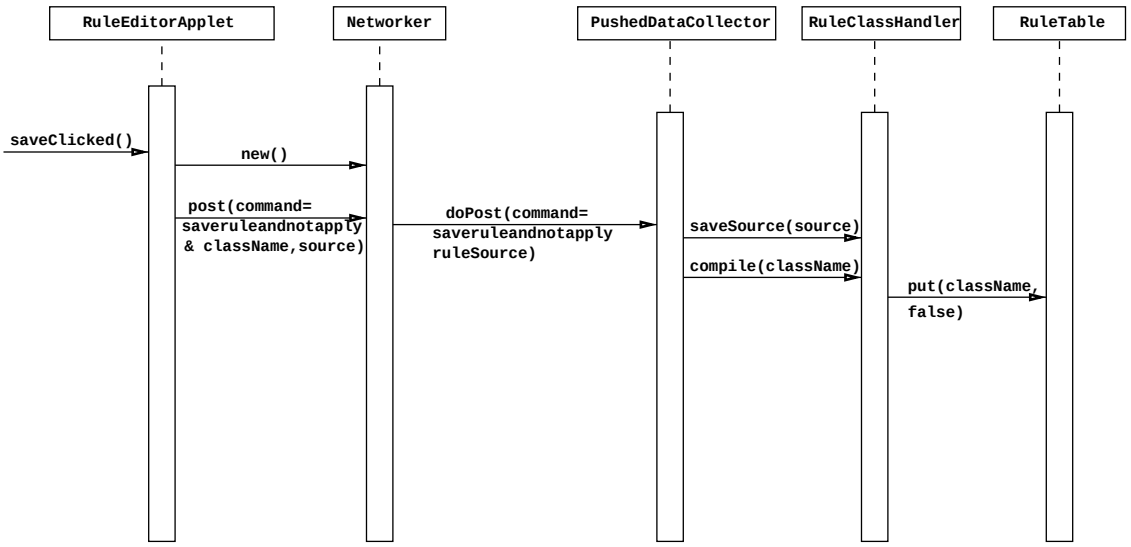

Figure 5.6: Interaction Diagram for the RuleEditor system

Management Station to the Data Collector after loading the frequencies corresponding to the given oid from the agent.

5.2.5 Operations

The manager must go through a few operations before activating a rule.

1. First the manager must choose the agent. Then the list of the rules created for this agent appears.

2. He must choose the oid of the variable he wants to apply the rule to. This is done by browsing the *MibTree* provided by AdventNet.

3. Then he loads the frequencies of the different subscriptions registered for this agent and for this oid. Briefly, this task is performed by sending a Get to the SubscriptionSheet Servlet that logs all the subscriptions realized.

4. He chooses a rule among those on the list

5. He clicks on the *Activate* button.

After that, the sequence of operations performed is similar to those performed after clicking on the *SaveRule* button on the Rule Editor Applet.

Classes

The design of the Mapping Applet is similar to the one for Rule Edition.

- MappingApplet: Main Controller class.
- MappingPanel: Graphical User Interface
- MappingListener: Interface for receiving events from a Mapping Panel
- PushNetworker: used for transfer information to the Data Collector.

5.2.6 EventNotificationApplet

Operations

First, the manager clicks on the *Connect* button and then waits for events.

1. The UnitCollector gets the next available unit from the PushNetworker.
2. It passes it to the main controller, *feeding* the applet considered a Consumer
3. the applet tells the GUI to display the event, by adding a line to the list of previously received events.
4. The manager can eventually delete certain lines from the list of events by selecting a line of the event list and clicking on the Remove button.

A blinking light and a sound system should notify the user of occurence of events.

Classes

The design of this applet is depicted in Figure 4.8.

- EventNotificationApplet
- EventNotifierPanel
- EventNotifierListener
- PushNetworker

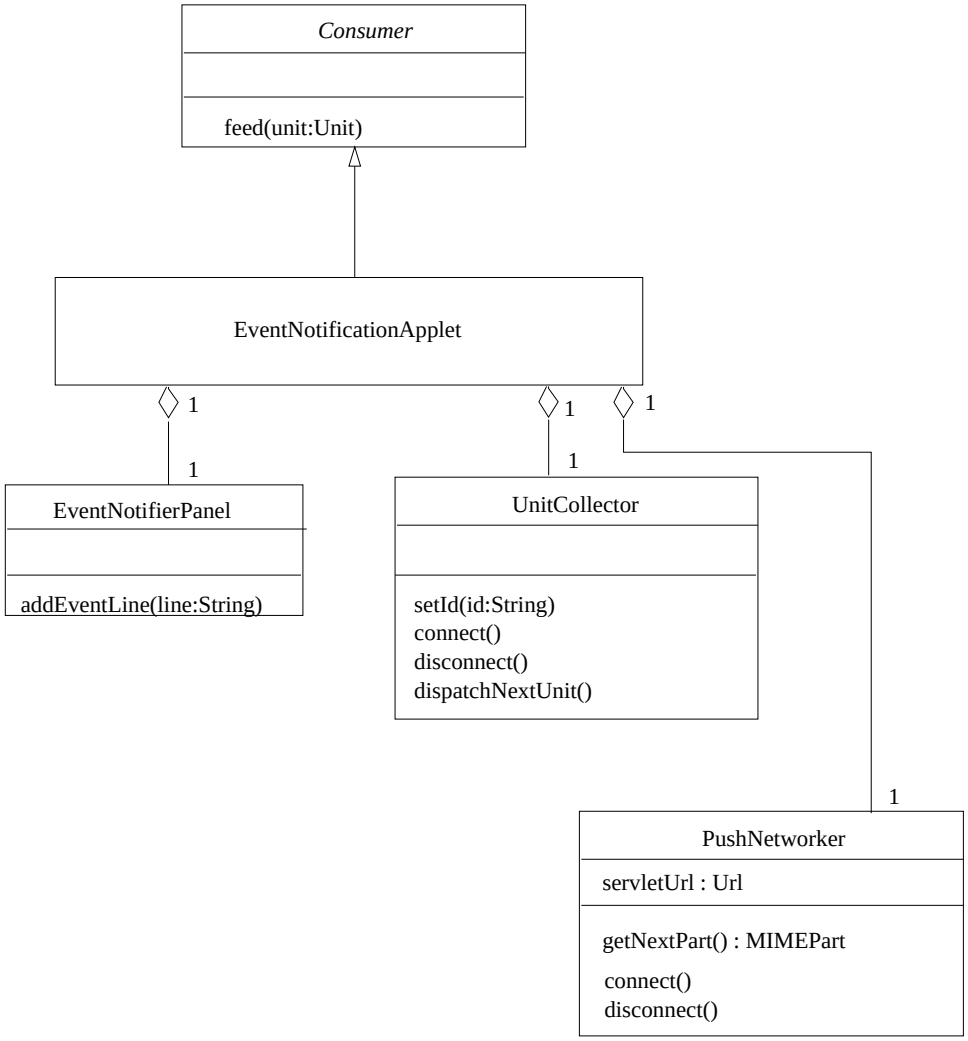

Figure 5.7: Class Diagram for the EventNotification applet

5.3 The Agent

5.3.1 Get and GetTable Servlets

This servlet is used for ad hoc management. It uses the usual pull system. The servlet retrieves data given their OID and ObjectType as the URL query parameters.

- ManagementBase This is the interface to any class handling management information base.

- SnmpBase Implementing ManagementBase, this is an object implementing the underlying SNMP protocol and using AdventNet classes to retrieve the requested data.

Operations

The servlet only has a doGet method to answer a GET request from the Management Station. To reply to the request, the servlet retrieves the value of the variable from the ManagementBase. Up to now, JAMAP supports SNMP only so the objectType can only be *MibData*. The reply consists in a Serialized Java object giving the value of the oid the manager asked for. If the Get Servlet is called, the object in the response will be a SnmpVar, representing a single MIB variable. If the GetTable Servlet is called, the object in the response will be a table meaing that the oid given referred to an SnmpTable.

5.3.2 PushDispatcherServlet

This servlet provides the push system on the agent and gives some intelligence to the usually dumb SNMP agent.

Classes

The classes of the PushDispatcher are depicted in Figure.

- SubscriptionTable This is a hashtable containing subtables as values and collectorIds as keys.
- PushScheduler This is the center of the push system, it dialogs with the ManagementBase to retrieve values and pushes units to the outputstreams accordingly with its schedules.
- PushedDataFormatter This is the class that decides which outputstream to use: Multipart or XML, after reading the representation field in the subscription given.
- MultiPartOutputStream This class handles a MIMEMultipart stream.Objects written to it are serialized and sent in one MIME Part.
- XMLOutputStream This class handles a MIMEMultipart stream too. Objects written to it are represented using XML Schema.

Operations

- Adding a subscription to the SubscriptionTable The ManagementStation, via the Management Server, sends an HTTP POST on the servlet with *subscribe* as URL

query string.The servlet then invokes the Put method on the SubscriptionTable which will add the subscription to the subtable identified by the collectorId given.

- Removing a subscription from the SubscriptionTable The ManagementStation, via the Management Server, sends an HTTP POST on the servlet with *unsubscribe* as URL query string.The servlet then invokes the remove method on the SubscriptionTable which will remove the subscription from the subtable identified by the collectorId given.
- PushScheduling All the interactions between components of the scheduling system are depicted in Figure.

5.4 The DataCollector

5.4.1 The PushedDataCollectorServlet

Classes

The classes of the PushedDataCollector are depicted in Figure.We can distinguish two different phases in this servlet. One is used for configuration sake and the other one is used in any push cycle. This is why in the future, it would be better to separate these two different parts in two different servlets. For configuration, we use:

- CollectorTable A hashtable of all the unit collectors in the servlet,indexed o, their collectorId.
- RuleTable This is a hashtable where all the rules are registered, whether they are active or not.
- RuleClassHandler This is the static class reponsible for saving and compiling the rules.
- RuleThis is the generic rule. During a push cycle we use:
- UnitCollector The UnitCollector reads form the stream and determines if the data are encoded in XML or if they are in Serialized Java. Then, the unit collector forwards the interesting MIME Part of the MIME Multipart Stream to an XML Handler or a Serialized Java Handler.
- XMLHandler This class handles a MIME Part where objects are read using a XML Parser.
- SerializedJava Handler This class handles a MIME Part where objects are deserialized using readObject methods.

- PushedDataFilter It controls the number of units received per minute. If the number of units received exceeds a certain threshold, the Unit Collector closes the connection to the agent.

- PushedDataAnalyzer It has a set of rules identified by the key agentId/type/oid/frequency and decides which rule to apply to one unit received depending on the agentId, the type, oid and frequency corresponding to this unit..

- PushForwardConsumer This consumer waits for units and forwards them to a SerializedJava OutputStream. Up to now events are written in Serialized Java only. We need to add an applet for event subscription where the administrator chooses the format which will be used to forward events.

5.4.2 Operations

1. RuleConfiguration Creating a rule from a received rule source. Activating the rule and putting the rule into production through associating it with a PushedDataAnalyzer. The Mapping applet allows the administrator to deactivate or activate the rules he chooses.

2. Push data and generate units encapsulating events.

For one push cycle, the interaction diagram is given in Figure 4.6

5.5 The Notification Collector

The big picture is the same as with the Data Collector, the only difference being that we have not implemented any rules yet. We can distinguish two different phases in this servlet. One is used for configuration sake and the other one is used in any push cycle. This is why in the future, it would be better to separate these two different parts in two different servlets. For configuration, we use:

- CollectorTable A hashtable of all the unit collectors in the servlet,indexed on their collectorId. During a push cycle we use:

- SerializedJava Handler This class handles a MIME Multipart stream where objects are deserialized using readObject methods.

- NotificationFilter It controls the number of units received per minute. If the number of units received exceeds a certain threshold, the Unit Collector closes the connection to the agent.

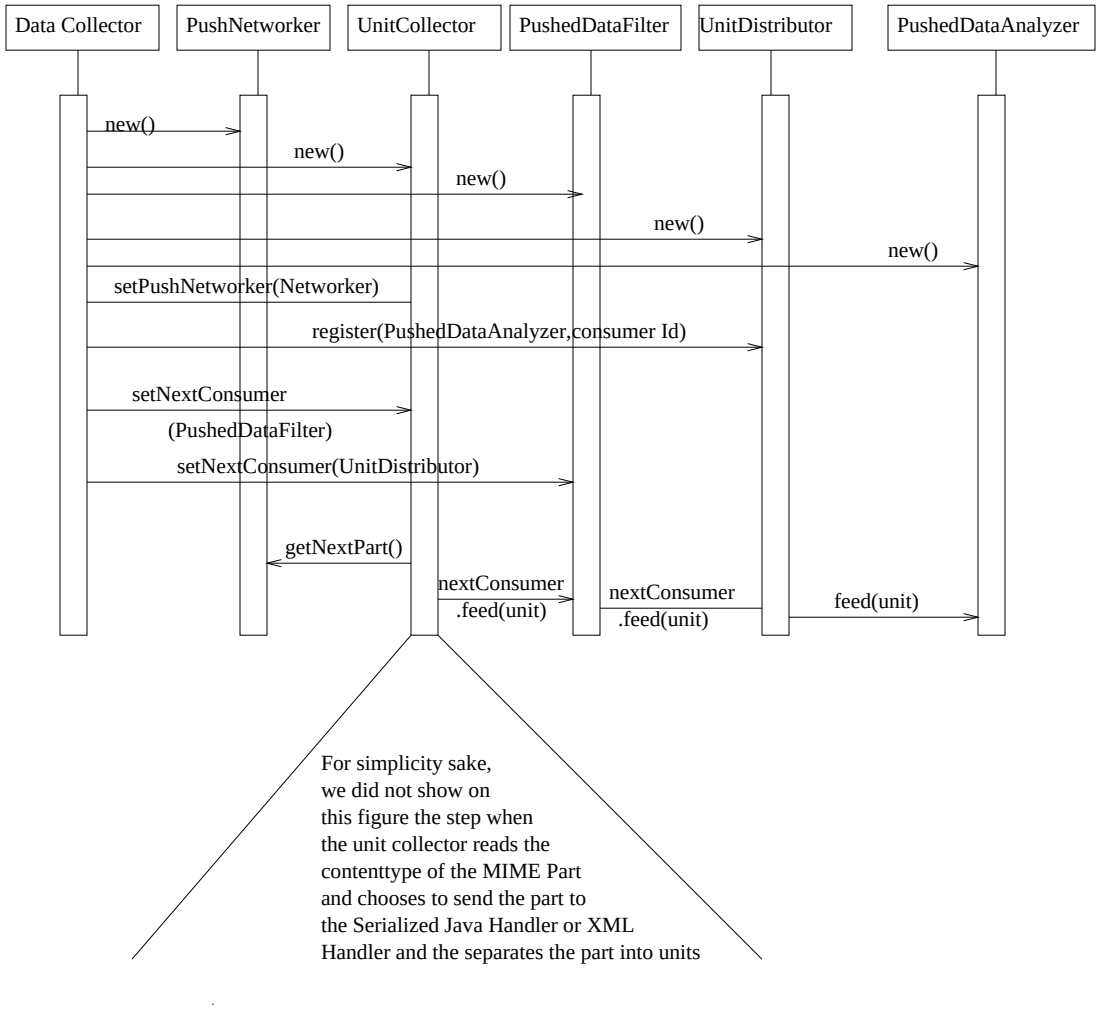

Figure 5.8: Interaction Diagram for one Push cycle

- PushForwardConsumer This consumer waits for units and forwards them to a SerializedJava OutputStream . Actually, up to now, events are described in Serialized Java only.
- SerializedJavaOutputStream describes the events in Serialized Java and sends them to the next consumer.

5.5.1 Operations

1. NotificationConfiguration The Notification Collector is used during the configuration phase when the administrator chooses what type of notifications he accepts to receive and which format should be used.

2. Push data and generate units encapsulating notifications.

5.6 The EventManager

5.6.1 Classes

- EventSink This is where all events arrive. It serves the same purpose as the PushedDataFilter for events this time. If too many events are received, something is wrong in the network and it closes its connection to the PushedData Collector and Notification Collector.
- EventCorrelator This is the core of the EventManager. By correlating events it allows to drop some events and therefore it allows to reduce redundancy. Its task is to relate events one to another. By now, no rule was implemented on the event correlator since correlating events is a hard task and we had rather take some free code and adapt it to JAMAP than write en event Correlator from scratch.
- EventQueue In our configuration, we supposed that there were more than one Data Collectors and more than one notification collectors. So we have to manage the case when two Data Collectors or Notification Collectors send events at the same time. When events are received from collectors, they are queued in one of 4 different EventQueues depending on their severity, which can be either fatal, warning, critical or simply informative.
- SeverityThreads When one event is received the thread with highest priority wakes up, looks into its EventQueue and processes the events in its queue. If its queue is empty, then the thread with the second highest priority wakes up and does the same. This goes on until the thread with the lowest priority has processed its events.
- EventLogger Processing events means sending them to the next consumer that was registered during the configuration phase. Whatever the severity of an event is, this event is logged giving the agentId, the collectorId, the rule that generated the event and the severity of the event. This way the administrator knows from looking at the logs what generated the event.
- EventMailer If the severity of the event received is either fatal or critical, the event is then forwarded to an EventMailer that will send a mail the the administrator in charge of the system or the network. In this mail, we also find the agentId, the collectorId, the rule that generated the event, the type of the event and its severity.
- PushForwardConsumer In case the administrator loaded the EventNotificationApplet, this applet is then considered a consumer of events and all events are formatted from the EventLogger and sent to the Management Station through a PushForwardConsumer.

5.6.2 Operations

The Event Manager is used to process events and forwards them to the right event consumers depending on the severity. It can also forward events to the Management Station applet if the administrator is running the EventNotification applet.

Chapter 6

Conclusion

In this last chapter, we summarize what has been implemented and see what future work remains to be done.

6.1 Summary and Contributions

Finally, JAMAP is a research prototype and not a full-fledged management platform. We could have developed nicer GUIs but we decided to concentrate on the Communication Model and the distribution aspects. JAMAP is now available on the web at : http://www.research.att.com/ jpmf and fully implements the WIMA architecture described by J.P. Martin-Flatin in his Ph.D. thesis.

JAMAP is platform independent and has been tested under Windows 2000, Solaris 5.8, Irix 6.5 and Red Hat Linux 6.2. We used JDK1.2 and JDK1.3 as well as HTTP/1.0 and HTTP/1.1. My main contributions to this project were :

- Distribution of the platform to support multiple agents, data collectors and notification collectors
- Implementation of a new design for the event manager and notification handling
- JAMAP now supports XML as a means for representing management data
- Encapsulate numerous data in one MIME Part
- Implementation of a new design for the rule edition and the mapping between rules and incoming data/events
- Reduction of the network overhead and the CPU/ memory overhead of the PushedDataAnalywers running on the data collectors.

6.2 Future Work

Many enhancements could be made to JAMAP. One of them could be to implement a real-life event correlator with rules and do some performance testing. We chose not to implement a complete event correlator since event correlation is a complicated task we did not have time to tackle during this internship. We also need to implement the new design of the PushScheduler in order to send data more efficiently. The HTTP Server we used buffered the data before sending them and we have contacted the W3C team about this problem but have got not response yet.

Bibliography

[1] R. Anderson, M. Birbeck *et al. Professional XML*, Wrox Press, 2000.

[2] G.Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[3] J.P. Martin-Flatin. *Web-Based Management of IP Networks and Systems*, PhD thesis, EPFL, Switzerland, October 2000.

[4] J.P. Martin-Flatin. *Push vs.Pull in Web-Based Network Management.* In M.Sloman, S.Mazumdar and E.Lupu (Eds), *Proc 6th IIFIP/IEEE International Symposium on Integrated Network Management (IM' 99), Boston, MA, USA, May 1999.*

[5] J.P. Martin-Flatin. *La gestion des réseaux IP basée sur les technologies Web et le modèle push.* In O.Cherkaoui (Ed), *Proc. 3e Colloque Francophone sur la Gestion de Réseaux et de Services (GRES' 99), Montreal, QC, Canada, June 1999.*

[6] J.P Martin-Flatin, L. Bovet and J.P. Hubaux. *JAMAP: a Web-Based Management Platform for IP networks.* In R.Stadler and B.Stiller (Eds), *Active Technologies for Network and Service Management, Proc 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99), Zurich, Switzerland.*

[7] L. Bovet. *The Push Model in a Java-Based Network Management Application*, M. S. thesis, Computer Science Dept, EPFL, March 1999.