JAMAP: A Web-Based Management Platform for IP Networks

Jean-Philippe Martin-Flatin, Laurent Bovet and Jean-Pierre Hubaux

Institute for computer Communications and Applications (ICA)
Swiss Federal Institute of Technology, Lausanne (EPFL)
1015 Lausanne, Switzerland
martin-flatin@epfl.ch
http://icawww.epfl.ch

Abstract. In this paper, we describe JAMAP, a prototype of a Web-based management platform for IP networks. It is written entirely in Java. It implements the push model to perform regular management (i.e. permanent network monitoring and data collection) and *ad hoc* management (i.e. temporary network monitoring and troubleshooting). The communication between agents and managers relies on HTTP transfers between Java applets and servlets over persistent TCP connections. The SNMP MIB data is encapsulated in serialized Java objects that are transmitted as MIME parts via HTTP. The manager consists of two parts: the management server, a static machine that runs the servlets, and the management station, which can be any desktop running a Web browser. The MIB data is transparently compressed with `gzip`, which saves network bandwidth without increasing latency too significantly.

1 Introduction

Web technologies have proved very attractive to Network and Systems Management (N&SM) for several years. In July 1996, a special issue of *The Simple Times* summarized the different ways of integrating HTTP, HTML and applets with standard IP network management platforms. Whereas most of the industry, as far as a customer could see, and most of the press then limited Web-based management to the sole use of Web browsers to display Graphical User Interfaces (GUIs), this collection of articles had the merit of widely advertising the wide range of possibilities opened up by Web technologies in N&SM. The most radical approach came from Wellens and Auerbach, who suggested to embed not only HTTP servers but also applets in network equipment, and who based the communication between managers and agents on HTTP instead of SNMP [16]. The common belief, then, as exposed by Bruins [3], was that HTTP was useful to initiate the interactive dialog between the administrator and the agent, but that further interactions should be based on SNMP.

Since then, the industry has trumpeted its adoption of Web technologies very loud, but the real achievements that we can see on the market today are more modest. HTTP servers are now routinely embedded by many router vendors, but management applets are not. Configuration management has probably been the main beneficiary so far of the recent adoption of Web technologies: several management platforms now enable administrators to run Java applications on the manager (the cheap alternative to

R. Stadler and B. Stiller (Eds.): DSOM'99, LNCS 1700, pp. 164-178, 1999.

downloading an applet from an agent) in order to configure agents. But to the best of our knowledge, all the main players in the IP systems and network management market (HP Openview, IBM/Tivoli Netview, Cabletron Spectrum...) still use SNMP for communication between managers and agents once the configuration phase is accomplished. We have attended demonstrations by several vendors of Java applications that included an SNMP stack in order to perform standard SNMP polling behind the Web interface, which proves that the full potential of Web technologies in N&SM has still not spread across the entire industry.

Despite this slow pace of the industry, the research community has been very active in the meantime, leading to a growing understanding of the issues and challenges at stake. If we ignore the revolutionary approaches that depart entirely from traditional SNMP-based management (e.g., Java-based mobile agents and multi-agent systems), we still have a lot of literature witnessing that we have gained experience in the integration of Web technologies with traditional N&SM. At the end of 1996, Deri [5] described possible mappings between URLs and command line interfaces, and Harrison *et al.* proposed the HTTP Manageable MIB [7]. In 1997, Maston [11] described the basics of network element management with HTML, while Kasteleijn [8], Barillaud *et al.* [2] and Reed *et al.* [13] reported their experiences with building management prototypes respectively for ATM backbone networks, local-area networks and PC systems. More references will be presented in Section 7.

In 1998, we proposed an architecture that goes beyond Wellens and Auerbach's [9, 10]. In addition to embedding HTTP servers and management applets in all managed devices, and to using solely HTTP to communicate between managers and agents, we suggested to push management data from agents to managers and to rewrite the managers entirely in Java, thereby leveraging on the simplicity of servlet programming. In this paper, we present the low-level design of this architecture, we indicate the design decisions which were made among all the candidates listed in [10], and we report progress on the building of a prototype called JAMAP (JAva MAnagement Platform). In particular, we describe how data is structured and encoded inside HTTP messages (Wellens and Auerbach only gave a high-level description of HTTP-based management data transfers).

In the IP world, the advantages of our architecture over the classic SNMP management frameworks (v1, v2c and v3) are fourfold. First, by going from a pull to a push model, we decrease significantly the network overhead of management data, because the manager no longer has to keep requesting the same OIDs (Object IDentifiers) to the same agents at every polling cycle. This almost halves the network overhead, because the description of the OIDs takes a lot more space than their values. Second, by grouping all the MIB data of a push cycle together and by compressing them with `gzip`, we reduce even more the network overhead without increasing latency too significantly, which has a positive effect on the scalability of the N&SM system. Third, by adopting Java, we free vendors from the burden of porting add-ons like CiscoWorks from one management platform to another (HP OpenView, Cabletron Spectrum, etc.) and from one operating system to another (Windows x.y, Solaris u.v, Linux a.b, etc.). Fourth, by using only well-known and pervasive Web technologies instead of SNMP technologies, we prove that N&SM need not rely on domain-specific

skills (SNMPv1, SNMPv2c, SNMPv3, SMIv1, SMIv2, BER...). N&SM applications are just another case of distributed applications, and can rely on standard distributed technology. Therefore N&SM platforms can be maintained by less expensive and easier-to-find Java programmers, and can reuse components developed for other application domains. By combining these advantages with the "write once, run anywhere" claim of Java, the price of N&SM platforms can be driven down significantly, and site-specific developments can be rendered much easier for administrators.

The remainder of this paper is organized as follows. In Section 2, we present an overview of the architecture of JAMAP. In Section 3, we introduce three advanced technologies used in JAMAP. In Sections 4, 5 and 6, we describe the different applets and servlets run by the agent and the two constituents of the manager: the management station and the management server. In Section 7, we present some related work. Finally, we conclude in Section 8 with some perspectives for future work.

2 Architecture of JAMAP

Our architecture integrates push and pull communication models to manage IP networks [9, 10]. For regular management, i.e. when tasks are repetitive and performed identically at each time step, we use the push model and the publish-subscribe paradigm. Initially, managers subscribe to some MIB data published by the agents. Later, the agents push this data at regular time intervals, without the manager requesting anything else. For *ad hoc* management, i.e. when tasks are performed over a short time period (e.g. troubleshooting), we use the pull model for one-shot retrievals, and the push model otherwise (e.g. short-term monitoring).

For the information model, we keep SNMP MIBs unchanged in the agents because we believe that they constitute the main achievement of SNMP. It took years for the industry to define and deploy these MIBs in all sorts of network devices and systems, and it would not make sense to change them. Their main limitation is that, due to the SNMP management frameworks, most (all?) of them are confined today to the instrumentation level, that is, offer low-level semantics. But there is no reason why we should not see higher level MIBs appear in the future. From standard SNMP-based management, we also borrowed the organizational model, with managers and agents, but we changed the SNMP 2-tier architecture into a 3-tier architecture.

The main novelty demonstrated by JAMAP is that it uses a push model to transfer management data (i.e., data extracted from SNMP MIBs at the agent) from the agent to the manager. Fig. 1 depicts push-based monitoring and data collection, while the handling of notifications is represented on Fig. 2. The push model, also known as the publish-subscribe paradigm, involves three phases:

- *publication*: each agent announces the MIBs that it manages and the notifications that it may send to a manager;
- *subscription*: agent by agent, the administrator (the person) subscribes the manager (the program) to different MIB variables and notifications via subscription applets; the push frequency is specified for each MIB variable;

– *distribution*: at each push cycle, the push scheduler of each agent triggers the transfer of MIB data from the agent to the manager; unlike what happens with traditional polling, the manager does not have to request this data at each push cycle; the transfer of notifications is triggered by the health monitor; notifications and MIB data use independently the same communication path.

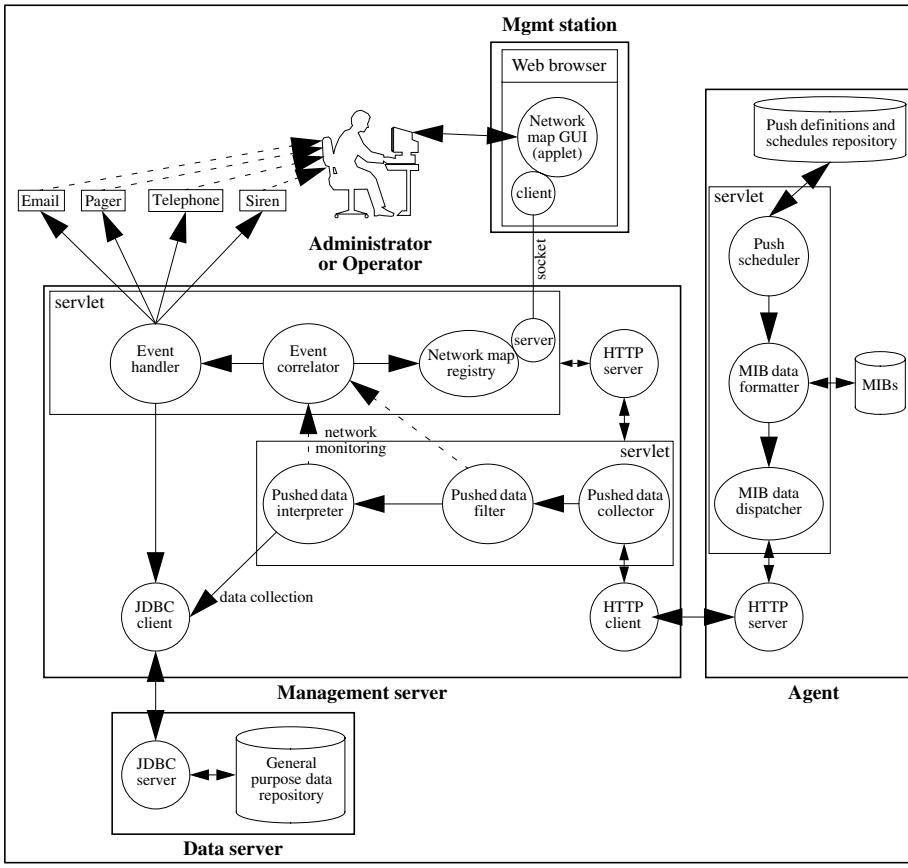

Fig. 1. Push-based monitoring and data collection

Our main motivation for developing of JAMAP was to prove the feasibility and relative simplicity of our design innovations. The core of JAMAP, that is, the push engine and the communication between the agent and the manager, was implemented in only two weeks, thereby demonstrating that our design was simple to implement. The different servlets and applets depicted in Fig. 1 and Fig. 2 took a lot longer to write and debug, as expected, and we had to make a number of simplifications to finish the first version of JAMAP in time to demonstrate it internally in March 1999. First, the persistence of data (MIB data, log of events, agents' configuration files, network topology, etc.) currently relies on flat files. Eventually, it will be ensured by a public-domain RDBMS (Relational DataBase Management System, e.g. `msql`) accessed via JDBC (Java DataBase Connectivity), as represented on the figures. Second, we have

not yet written a network map GUI applet. Instead, as illustrated by Fig. 3, we use an event notification applet that simply displays incoming events, line by line, in a window. For the future, we plan to generate automatically a network map from a file describing the network topology, and to change the color of the icons according to the events received. Third, our event correlator is still very simple, with many rules hard-coded on an *ad hoc* basis. We are currently investigating whether we could integrate a full-blown event correlator written in Java by another research team.

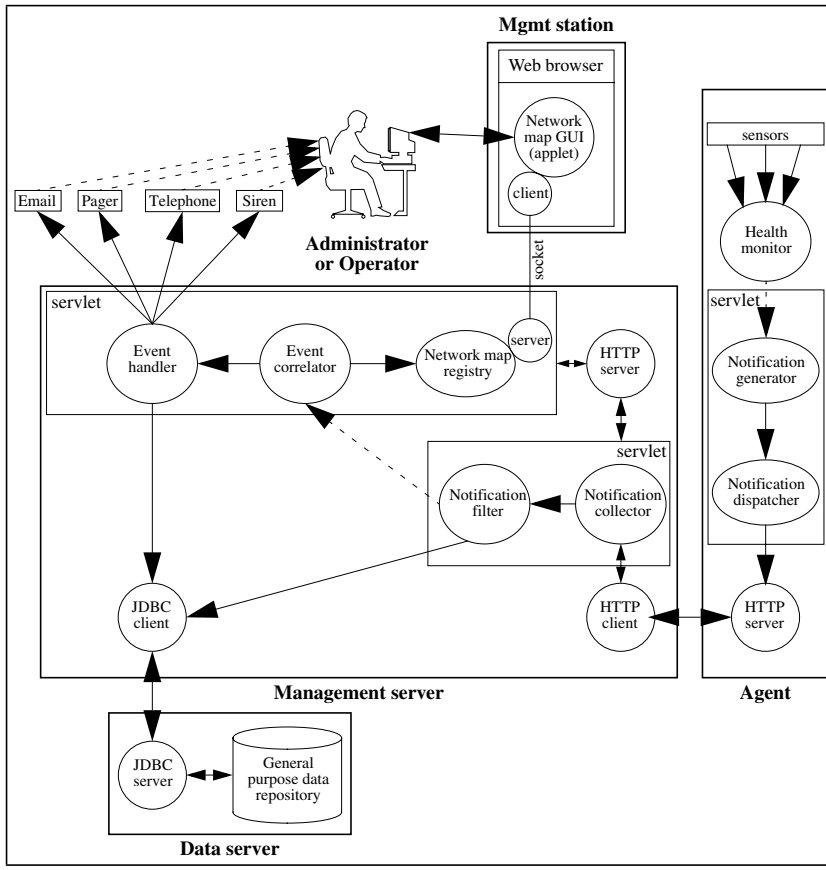

Fig. 2. Push-based notification handling

Fig. 3 and Fig. 4 are synthetic views of the communication between the different Java applets and servlets running on the agent and the manager. They both illustrate our 3-tier architecture with the management station, the management server and the agent. The push arrow between the MIB data dispatcher servlet and the MIB data subscription applet represents the path followed by MIB data retrieved for *ad hoc* management. The other push arrows depict regular management. The dotted arrows represent the applet-to-servlet dialogs that take place at the subscription phase. These figures will be explained in detail in Sections 4, 5 and 6.

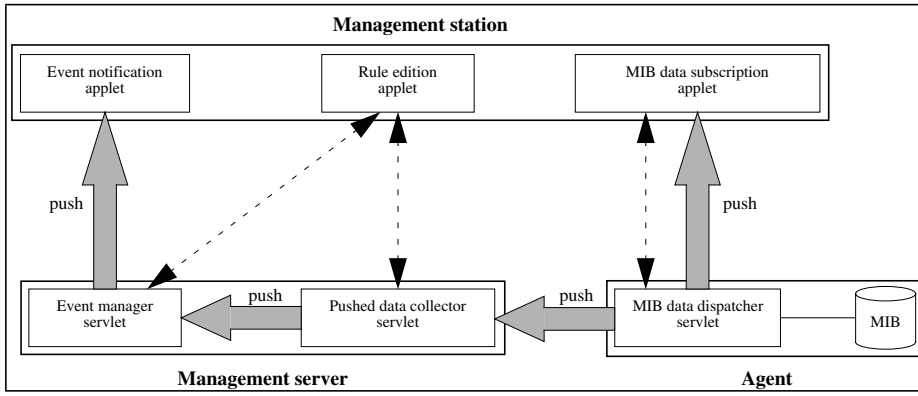

Fig. 3. Communication between Java applets and servlets: monitoring and data collection

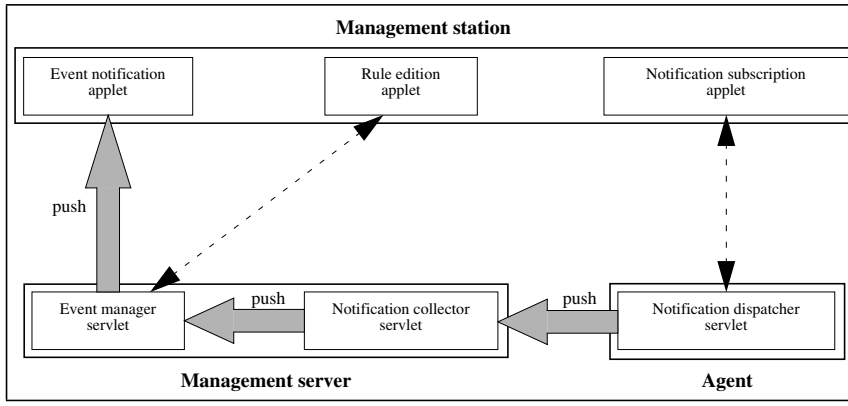

Fig. 4. Communication between Java applets and servlets: notifications

3 Advanced Technologies Used in JAMAP

In this section, we describe three advanced technologies used in JAMAP: MIME-based push, Java servlets and Java serialization.

3.1 MIME multipart and MIME-based push

Unlike other distributed application technologies such as sockets and Java Remote Method Invocation (RMI), HTTP offers no native support for bidirectional persistent connections [9]. With HTTP, a connection is always oriented: it is not possible to create a persistent connection in one direction (from the client to the server) and to send data afterward in the opposite direction (from the server to the client). Before an HTTP server can send data to a client, it must have received a request from this client. In other words, an HTTP server cannot send unsolicited messages to an HTTP client.

This is an important difference between SNMP and HTTP. SNMP implements a generalized client-server communication model, whereby the request from the client can either be explicit (e.g., pull-based `get` and `set` operations) or implicit (e.g., push-based `snmpv2-trap` operation). Conversely, HTTP implements a strict client-server model: all its methods adhere to the request-response paradigm, and the request cannot be implicit. As a result, the implementation of the push model is not natural in HTTP.

A simple and elegant way to circumvent this limitation was proposed by Netscape [12] in a different context: How can a GUI displayed by a Web browser be automatically updated by an HTTP server? Netscape's idea was to initiate the data transfer from the HTTP client, and send an infinitely long response from the HTTP server, with separators embedded in the payload of the response (see Fig. 5). These separators enable the HTTP client to work out what, in the incoming stream of data, is the data for a given push cycle. To achieve this, Netscape recommended to use the `multipart` type of MIME (Multipurpose Internet Mail Extensions [6]).

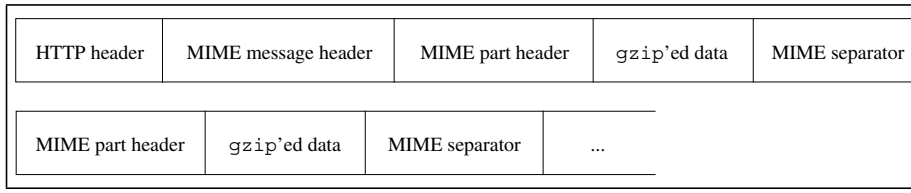

Fig. 5. TCP payload of the infinite HTTP response

We proposed to do the same in Web-based network management [9], and implemented it very simply in JAMAP. At every push cycle, the agent sends a new MIME part including a number of {OID, value} pairs, as specified by the push scheduler. A MIME separator delimits two consecutive push cycles; the manager interprets it as metadata meaning "end of push cycle". In the case of notifications, we encode only one notification per MIME part. In this case, the MIME separator is considered by the manager as metadata meaning "end of notification".

MIME parts transferring MIB data are compressed with `gzip` (MIME content transfer encoding). This saves a lot of network bandwidth when the manager subscribes to many MIB variables, and does not increase latency too significantly. MIME parts carrying notifications are not compressed because the compression ratio would be poor for so little data, and the increased latency would not be worth the meager savings in network overhead.

3.2 Java servlets

JAMAP relies heavily on HTTP-based communication between Java applets and servlets. Servlets [4] only recently appeared on the Web; they are an improvement over the well-known CGI (Common Gateway Interface) scripts. Unlike the CGI scripts that are typically written in a scripting language like Perl or Tcl/Tk, servlets are Java classes loaded in a Java Virtual Machine (JVM) via an HTTP server. The HTTP server must be configured to use servlets and associate a 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 HTTP server invokes a method of the servlet depending on the HTTP method used by the request. All servlets implement one method per HTTP method. For instance, the `doGet()` method is invoked when an HTTP `GET` request comes in for the corresponding URL.

Modern operating systems generally support multithreading. As a result, most HTTP servers now support concurrent accesses. Several HTTP clients may therefore invoke concurrently the same method of the same servlet. This allows the sharing of the same servlet by multiple persistent connections. We used this feature extensively in JAMAP when we tested it with several agents. Like any URLs, Java servlets can also leverage on the general-purpose features of HTTP servers (e.g. access control).

As we write this paper, servlet environments are still in constant evolution. During our work, Sun's specification of the servlets changed from version 2.0 to version 2.1, but public-domain implementations remained at 2.0. For JAMAP, we first used the Apache HTTP server version 1.3.4 and the Apache servlet engine Jserv 0.8. But we had problems because Jserv 0.8 did not support concurrent accesses to servlets and the response stream was buffered (both problems were later corrected in Jserv 1.0). In the meantime, we switched to another HTTP server, Jigsaw 2.0.1, which offered good support for servlets.

3.3 Java serialization

Serialization is a feature of Java that allows the translation of an arbitrarily complex object into a byte stream. In JAMAP, we used it for ensuring the persistence of the state of an object and for transferring objects over the network. Objects containing references to other objects are processed recursively until all necessary objects are serialized. The keyword `transient` can be added to the declaration of an attribute (e.g. an object reference) to prevent its serialization.

For network transfers, instead of defining a protocol, one can use serializable classes dedicated to communication. Such classes offer a `writeObject()` method on one side, and a `readObject()` method on the other. For persistence, serialization proved very useful to store rules and agents configurations.

In the next three sections, we will describe the different applets and servlets running on the different machines of the management system (see Fig. 3 and Fig. 4): the management station, the management server and the agent.

4 Management Station

The management station is the desktop of the administrator or operator. It can be any machine (a Linux PC, a Windows PC, a Mac, a Unix workstation, etc.) as long as it runs a Web browser and supports Java. Unlike the management server, it is not static: the administrator can work on different machines at different times of the day. In the subscription phase of the push model, he/she configures the agent via the MIB data subscription applet and the notification subscription applet. The rules used by the pushed data collector and the event manager servlets can be modified at any time via the rule edition applet. Events are displayed by the event notification applet.

4.1 MIB data subscription applet

The MIB data subscription applet communicates directly with the agent. It provides the subscription system for regular management. It is also used to perform push-based and pull-based *ad hoc* management. Its main tasks are the following:

- browse MIBs graphically;
- select MIB variables or SNMP tables and retrieve their values once (pull model);
- select MIB variables and monitor them for a while (text fields, time graphs or tables);
- monitor some computed values (e.g. interface utilization); and
- subscribe to MIB variables or SNMP tables and specify a push frequency (per MIB variable).

Computed values are typically the results of equations parameterized by multiple MIB variables. We implemented a sort of multiplexer to support them. This kind of simple preprocessing could be delegated to the agent in the future.

4.2 Notification subscription applet

Similarly, the notification subscription applet also communicates directly with the agent. It enables the administrator to set up a filter for notifications at the agent level. Notifications that have not been subscribed to by the manager are silently discarded by the agent.

4.3 Rule edition applet

The rule edition applet controls the behavior of two objects:

- the pushed data interpreter object living in the pushed data collector servlet; and
- the event correlator object living in the event manager servlet.

The administrator can write rules in Java via the applet, or can edit them separately and apply them via the applet. (Java is used here as a universal scripting language.) For instance, an event can be generated by the pushed data interpreter if the value of a MIB variable exceeds a given threshold. A typical rule for the event correlator would be that if a system is believed to be down, then all applications running on it should also be down, so events reporting that NFS (Network File System) is not working or that an RDBMS is not working should be discarded.

More complex rules can easily be written. For instance, the pushed data interpreter can check if the average value of a given MIB variable increased by 10% or more over the last two hundred push cycles. In fact, these rules can be arbitrarily complex, as there is no clear-cut distinction between what is in the realm of offline data mining and what should be performed immediately, in pseudo real-time. The trade-off is that the pushed data interpreter should not be slowed down too much by an excessive amount of rules, otherwise it might not be able to apply all the relevant rules to incoming data between two consecutive push cycles.

4.4 Event notification applet

The event notification applet is connected to the management server to receive events. We use it as a debugger, as we do not manage a production network with our platform. This applet displays a simple list of events and manages a blinking light and sound system to grab the operator's attention in case of incoming events. It is intended to remain permanently in a corner of the administrator's and operator's desktop screens. Eventually, it will be complemented by the network map GUI applet.

5 Management Server

The management server runs three servlets: the pushed data collector, the notification collector and the event manager. In principle, this management server could easily be distributed over multiple machines if need be (e.g. for scalability reasons), as the communication between servlets relies on HTTP, and the data server is already a separate machine. For instance, we could run the three servlets on three different machines, and data mining on a fourth. But so far, we have only tested our software with a single management server.

5.1 Pushed data collector servlet

The pushed data collector servlet consists of three core objects (see Fig. 1), plus a number of instrumentation objects not represented on that figure. The pushed data collector object connects to the agent upon startup, and enters an infinite loop where it listens to the socket for incoming data and passes on this data "as is" to the pushed data filter object. If the connection to the agent is lost, e.g. due to a reboot of the agent, the pushed data collector immediately reconnects to it so as to ensure a persistent connection [9].

The pushed data filter object controls the flow of incoming data. If it detects that too much traffic is coming in from a given agent (that is, from a given socket), it tells the pushed data collector object to close permanently the connection to that agent (that is, the collector should not attempt to reconnect to the agent until the administrator explicitly tells it to do so). The rationale here is that a misbehaving agent is either misconfigured, bogus, or under the control of an intruder pursuing a denial of service attack, and that the good health of the management system should be protected against this misbehaving agent. When this happens, the administrator is informed via email.

If the pushed data filter object is happy with the incoming data, it passes it on "as is" to the pushed data interpreter object. The latter unmarshalls the data and checks, MIB variable by MIB variable, whether it was subscribed to for monitoring, data collection or both.

In the case of data collection, the MIB variable is not processed immediately. Instead, it is stored in a persistent repository (an NFS-mounted file system currently, an RDBMS in the future) via a logger object. We assume that an external process will process it afterward to perform some kind of data mining (e.g., it could look for a trend in the variations of the CPU load of an IP router to be able to anticipate when it should be upgraded).

In the case of monitoring, the MIB variable is processed immediately. The pushed data interpreter object applies the rules relevant to that agent and that MIB variable. If it notices something important (e.g., a heartbeat is received from an IP router which was considered down), the pushed data interpreter object generates an urgent event and sends it via HTTP to the event correlator object living in the event manager servlet. We took special care for the case where the same MIB variable is used for both monitoring and data collection. The data is then duplicated by the pushed data interpreter.

A nice feature of our rule system is that rules may be dynamically compiled and loaded in by the servlet. 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 in the form of a string. The class loader of the JVM searches the class file in the file system, and loads it into the JVM's memory. This enables the servlet to load a class at runtime without knowing its name in advance. Once a class is loaded, it behaves just as any other class. We are limited only by the fact that a class cannot be modified at runtime. This means that if a rule is already registered under a certain class name and that rule is modified by the administrator, another class name must be used for that new version of the rule.

To solve this problem, we implemented a simple technique that consists in postfixing the class name with a release number and incrementing this release number automatically. As a result, the administrator can create, modify and debug rules dynamically. The drawback is that the memory used by loaded classes (especially those corresponding to the "old" rules) is freed only when the JVM is restarted. The administrator should therefore be careful not to fill up the memory in the rule debugging phase. Clearly, this feature should be used with special care on a production system; but it proved to be particularly useful for debugging rules.

5.2 Notification collector servlet

As depicted in Fig. 2, the notification collector servlet consists in principle of two core objects, the notification collector and the notification filter. Contrary to pushed data, we do not need an interpreter for notifications because we know already what happened: we do not have to work it out.

The notification collector object works exactly as the pushed data collector object. The notification filter object also works as the pushed data filter object. In fact, in the current version of JAMAP, the notification collector servlet and the pushed data collector servlet are just one single servlet. This enables us to use a single persistent connection between the agent and the manager for transferring MIB data and notifications. (Note that this would not be the case if we were to distribute the servlets over several machines.) Notifications received by the pushed data interpreter object are currently passed on "as is" to the event correlator object living in the event manager servlet, without any further processing.

5.3 Event manager servlet

The event manager servlet connects to one or more pushed data collector servlets (one, in the case depicted in Fig. 3 and Fig. 4) and waits for incoming events. Events are

processed by the event correlator object. This object performs a simple correlation with regard to the network topology, in order to discard masked events. For instance, if a router is down, all machines accessed across it will appear to be down to the pushed data interpreter. Based on its knowledge of the network topology (which is hardcoded in the current version of JAMAP), the event correlator is able to keep only those events that cannot be ascribed to the failure of other equipment.

When an event is not discarded by the event correlator object, it is transmitted to the event handler object corresponding to its level of emergency (this level of emergency is encapsulated inside the event). Each event handler is coded to interface with a specific notification system (e.g., an email system, a pager, a telephone, a siren, etc.). In our prototype, we only implemented an email-based notification system.

6 Agent

The agent runs a lightweight JVM [9] and two servlets: the MIB data dispatcher and the notification dispatcher.

6.1 MIB data dispatcher servlet

The MIB data dispatcher servlet consists of three core objects (the push scheduler, the MIB data formatter and the MIB data dispatcher) plus a number of instrumentation objects not represented on Fig. 1. During the subscription phase, the push scheduler object stores locally the subscription sent by the MIB data subscription applet (we call it the agent's configuration). Later, during the distribution phase, the push scheduler object uses this configuration to trigger the push cycles. It tells the MIB data formatter object what MIB variables should be sent at a given time step. The MIB data formatter object accesses the in-memory data structures of the MIBs via some proprietary, tailor-made mechanism, formats the MIB data as a series of {OID, value} pairs, and sends it to the MIB data dispatcher object. The latter compresses the data with `gzip`, assembles the data in the form of a MIME part, pushes the MIME part through and sends a MIME separator afterward to indicate that the push cycle is over.

In the future, the MIB data dispatcher servlet will be able to retrieve the agent's configuration from the data server via the management server. Thus, the agent will not necessarily have to store its configuration in nonvolatile memory, a useful feature for bottom-of-the-range equipment.

6.2 Notification dispatcher servlet

The notification dispatcher servlet consists of two core objects (the notification generator and the notification dispatcher) plus a number of instrumentation objects not represented on Fig. 2. During the subscription phase, the notification generator object stores locally the subscription sent by the notification subscription applet. In other words, it sets up a filter for notifications coming in from the health monitor. During the distribution phase, the health monitor checks continuously the health of the agent based on input from sensors. When a problem is detected, the health monitor asynchronously fires an alarm to the notification generator object in the servlet via

some proprietary mechanism. The notification generator object checks with the filter if this alarm should be discarded. If it was not subscribed to by the manager, the alarm is silently dropped. If it was, the notification generator object formats it as an SNMPv2 notification and sends it to the notification dispatcher object, which, in turn, wraps it in the form of a MIME part, pushes it to the management server via HTTP, and sends a MIME separator afterward to indicate that this is the end of the notification.

As we do not manage a real-life network with JAMAP, the notifications that are generated by the health monitor are all simulated. Instead of using real sensors, we fire, from time to time, one notification taken in a pool of predefined notifications; the selection of this notification is based on a random number generator. As with the previous servlet, the notification dispatcher servlet will eventually be able to retrieve the agent's notification filter from the data server via the management server.

7 Related Work

In the recent past, two very promising contributions came from industrial research: the Web-Based Enterprise Management (WBEM) initiative and Marvel.

WBEM came to life in 1996, by making sensational marketing announcements that it would unify (at last) N&SM by defining a new information model and a new communication model. By obsoleting all existing technologies and management frameworks, it did not gain much credibility. In1997, the WBEM Consortium became more realistic: it adopted HTTP as its transfer protocol, selected the Extensible Markup Language (XML) to structure management data, and delivered the specifications for a new information model: the Common Information Model (CIM) [15]. Then, the WBEM initiative was taken over by the Distributed Management Task Force (DMTF) and integrated into its more global work plan—a guarantee of its independence toward any particular vendor. A lot of work is currently under way, split across 14 Technical Committees. Apart from CIM, whose specifications were updated several times already and are now fairly stable, most of the technical specifications are still ongoing work, e.g. the definition of CIM operations over HTTP. Several of our proposals could fit into this framework, such as the use of push rather than pull technologies and the encapsulation of XML into MIME parts.

The most interesting prototype freely available to date is probably Marvel by Anerousis [1]. The main difference with JAMAP is that Marvel relies on RMI for manager-agent communication, and builds on it to offer a distributed object-oriented N&SM platform. This enables a very elegant architecture and a clean design, but is exposed to a well-known criticism: can we reasonably expect all managed devices to support RMI in the future? For the Jini camp [14], which advocates universal plug-and-play based on Java, the answer is clearly *yes*. We have doubts about it: bottom-of-the-range devices are very price sensitive, and despite the decreasing prices of CPU and memory, the extra cost of embedding Java software still makes a difference today—enough to win or lose customers. For top-of-the-range devices, the support for Java RMI makes perfect sense. But if we want to have a unified N&SM framework for all devices, we should be careful not to have too stringent requirements—otherwise, our proposals will simply be rejected by the industry. Our requirement that a

lightweight JVM be embedded in all devices seems to be the farthest we can reasonably go for the next couple of years. It should be noted that Sun's EmbeddedJava technology might bring an answer to this question in the future.

8 Conclusion

We have presented JAMAP, a prototype of N&SM platform written entirely in Java. It implements the push model to perform regular management (i.e. permanent monitoring and data collection for offline analysis) and *ad hoc* management (i.e. temporary monitoring and troubleshooting). The communication between agents and managers relies on HTTP transfers between Java applets and servlets over persistent TCP connections. The SNMP MIB data is encapsulated in serialized Java objects that are transmitted as MIME parts via HTTP. The manager consists of two parts: the management server, a static machine that runs the servlets, and the management station, which can be any desktop running a Web browser. The MIB data is transparently compressed with `gzip`, which saves network bandwidth without increasing latency too significantly.

Our approach offers many advantages over traditional SNMP-based management: it reduces the network overhead of management data transfers; it delegates part of the processing overhead from the manager to the agents; it reduces the development costs of management software for both equipment and N&SM platform vendors (and consequently the cost of the N&SM platform for the customers); and it makes it easier to find engineers with the expertise necessary for customizing a management platform to specific sites. Other advantages not developed in this paper include the better potential for distributing management with mobile code, the simplicity to implement low-level security, the potential for high-level semantics, and the usual advantages of 3-tier over 2-tier architectures. The main disadvantage is the slow speed of execution of Java code, especially JDBC, which may cause scalability problems. More work is necessary to assess if we can work around these difficulties, or if we have to resort to alternatives to manage large production systems and networks.

For future research, we plan to investigate different schemes to structure and encode management data, instead of serializing Java objects that encapsulate SNMP MIB data. Our objective is to get a higher level of semantics while keeping both network overhead and end-to-end latency reasonably low. In particular, we want to study the pros and cons of going from a string-based to an XML-based representation of MIB data, and to measure the effects of `gzip` compression in both cases. It would also be useful to perform a detailed performance analysis of these different structuring and encoding schemes, and to compare them with SNMP.

Acknowledgments

This research was partially funded by the Swiss National Science Foundation (FNRS) under grant SPP-ICS 5003-45311. We wish to thank H. Cogliati for proofreading this paper. We are also grateful to AdventNet, IBM and R. Tschalär for making useful Java classes freely available to academic researchers.

References

1. N. Anerousis. "Scalable Management Services Using Java and the World Wide Web". In A.S. Sethi (Ed.), *Proc. 9th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'98), Newark, DE, USA, October 1998*, pp. 79–90.
2. F. Barillaud, L. Deri and M. Feridun. "Network Management using Internet Technologies". In A. Lazar, R. Saracco and R. Stadler (Eds.), *Integrated Network Management V, Proc. 5th IFIP/IEEE Int. Symp. on Integrated Network Management (IM'97), San Diego, CA, USA, May 1997*, pp. 61–70. Chapman & Hall, London, UK, 1997.
3. B. Bruins. "Some Experiences with Emerging Management Technologies". *The Simple Times*, 4(3):6–8, 1996.
4. J.D. Davidson and S. Ahmed. *Java Servlet API Specification. Version 2.1a*. Sun Microsystems, November 1998.
5. L. Deri. *HTTP-based SNMP and CMIP Network Management*. Internet draft <draft-deri-http-mgmt-00.txt> (now expired). IETF, November 1996.
6. N. Freed and N. Borenstein (Eds.). *RFC 2046. Multipurpose Internet Mail Extensions (MIME). Part Two: Media Types*. IETF, November 1996.
7. B. Harrison, P.E. Mellquist and A. Pell. *Web Based System and Network Management*. Internet draft <draft-mellquist-web-sys-01.txt> (now expired). IETF, November 1996.
8. W. Kasteleijn. *Web-Based Management*. M.Sc. thesis, University of Twente, Enschede, The Netherlands, April 1997.
9. 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.
10. J.P. Martin-Flatin. "Push vs. Pull in Web-Based Network Management". In *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston, MA, USA, May 1999*, pp. 3–18. IEEE Press, 1999.
11. M.C. Maston. "Using the World Wide Web and Java for Network Service Management". In A. Lazar, R. Saracco and R. Stadler (Eds.), *Integrated Network Management V, Proc. 5th IFIP/IEEE Int. Symp. on Integrated Network Management (IM'97), San Diego, CA, USA, May 1997*, pp. 71–84. Chapman & Hall, London, UK, 1997.
12. Netscape. *An Exploration of Dynamic Documents*. 1995. Available at <http://home.mcom.com/assist/net_sites/pushpull.html>.
13. B. Reed, M. Peercy and E. Robinson. "Distributed Systems Management on the Web". In A. Lazar, R. Saracco and R. Stadler (Eds.), *Integrated Network Management V, Proc. 5th IFIP/IEEE Int. Symp. on Integrated Network Management (IM'97), San Diego, CA, USA, May 1997*, pp. 85–95. Chapman & Hall, London, UK, 1997.
14. Sun Microsystems. *Jini*. Available at <http://www.sun.com/jini/>.
15. J.P. Thompson. "Web-Based Enterprise Management Architecture". *IEEE Communications Magazine*, 36(3):80–86, 1998.
16. C. Wellens and K. Auerbach. "Towards Useful Management". *The Simple Times*, 4(3):1–6, 1996.