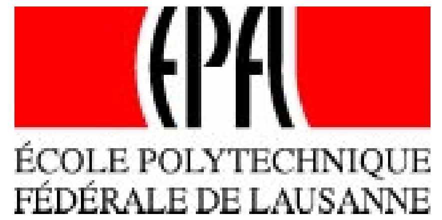
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE LAUSANNE
POLITECNICO FEDERALE DI LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY LAUSANNE

COMMUNICATION SYSTEMS DIVISION (SSC)
CH-1015 LAUSANNE, SWITZERLAND
http://sscwww.epfl.ch



JAMAP: a Web-Based Management Platform for IP Networks

Jean-Philippe Martin-Flatin and Laurent Bovet

April 1999

Technical Report SSC/1999/010

JAMAP: a Web-Based Management Platform for IP Networks

Jean-Philippe Martin-Flatin and Laurent Bovet
EPFL-ICA, 1015 Lausanne, Switzerland
Email: martin-flatin@epfl.ch Fax: +41-21-693-6610 Web: http://icawww.epfl.ch

Abstract

In this paper, we describe JAMAP, a prototype of a Web-based management platform for IP networks. It is entirely written in Java. It implements the push model to perform regular management (permanent network monitoring and data collection) and *ad hoc* management (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.

Keywords: Network Management, Systems Management, Web-Based Management, Java-Based Management, Java, Push, Pull.

1. Introduction

Web technologies have proved attractive to network and systems management for several years. Wellens and Auerbach suggested to embed applets in network equipment [25]. Bruins [6] and Deri [8] described possible mappings between URLs and command line interfaces. Harrison *et al.* proposed the HTTP Manageable MIB [10]. Maston [18] described the basics of network element management with HTML. Mullaney [19] reported work on a Web-based management agent. Thompson [23] described the Web-Based Enterprise Management (WBEM) approach and its main contribution so far, the Common Information Model (CIM). Several prototypes were built to demonstrate all these concepts, including Marvel by Anerousis [2], CyberAgent by Burns and Quinn [5], Webbin by Barillaud *et al.* [3], WbASM by Kasteleijn [13], and NetFinity by Reed *et al.* [21]. Some of these approaches are summarized in a survey by Hong *et al.* [11].

Last year, we proposed an architecture integrating push and pull communication models to manage IP networks [15–17]. 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.

In this paper, we report progress on a prototype that implements this architecture: JAMAP (JAva MAnagement Platform). It is written entirely in Java, and implements both the push and pull models. Only the push model will be presented here, as the pull model is well known and implemented by many other public-domain and commercial platforms (for references, see the excellent Web site maintained by Lindsay [14]). SNMP MIBs are the only legacy of the SNMP management framework used by JAMAP.

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 show how we managed to make software reuse a reality in this project. Finally, we conclude in Section 8 with some perspectives for future work.

2. Architecture of JAMAP

The main characteristic of JAMAP is that it implements a push model to transfer management data (that is, data extracted from SNMP MIBs at the agent) from the agent to the manager. Fig. 1 depicts push-based monitoring and data collection; the handling of notifications is represented on Fig. 2. The rationale for using Web technologies to manage IP networks and the pros and cons of the pull and the push models are detailed in a separate paper [17].

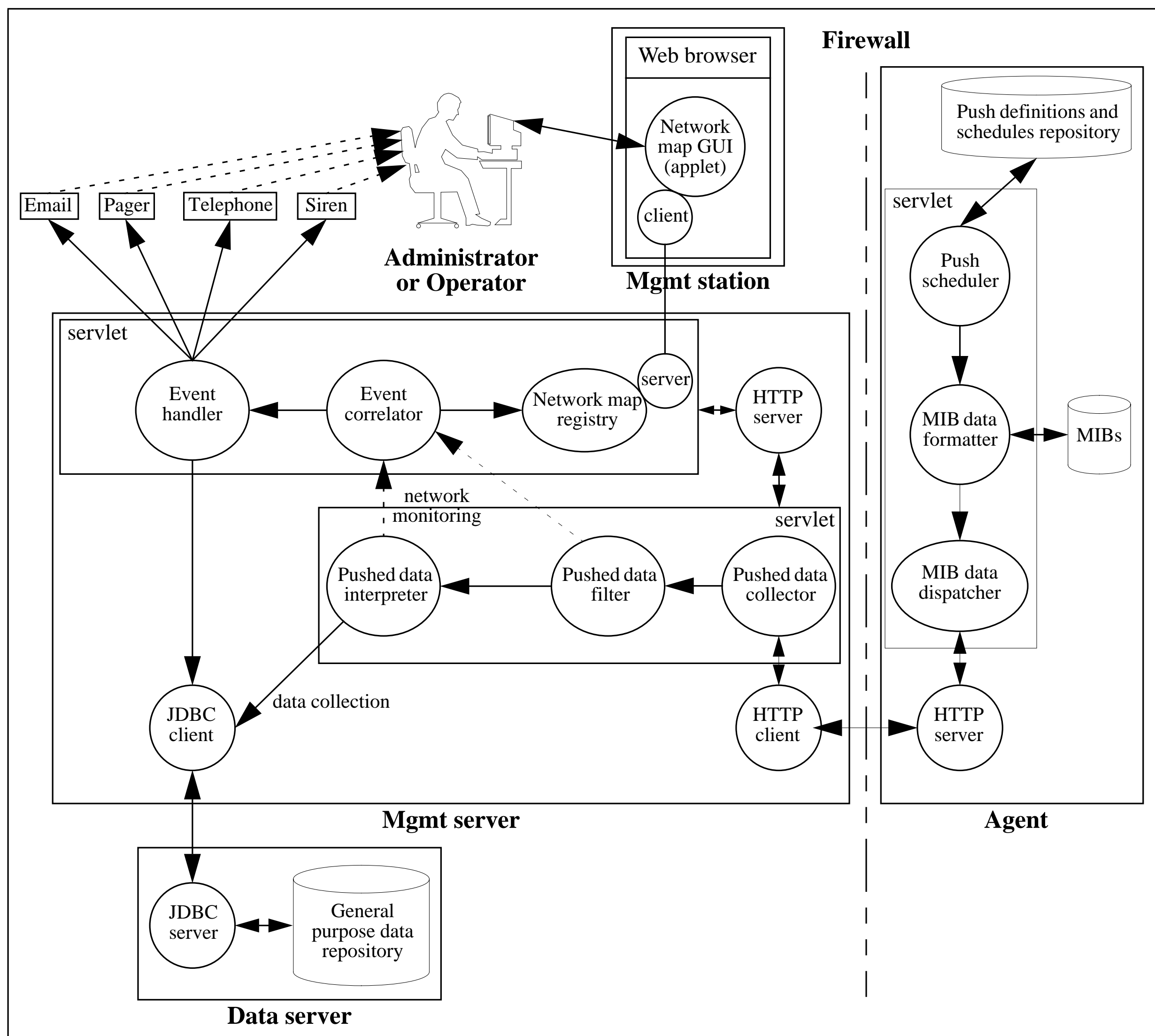


Fig. 1. Push-based monitoring and data collection

The push model is also known as the publish-subscribe paradigm. It 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.

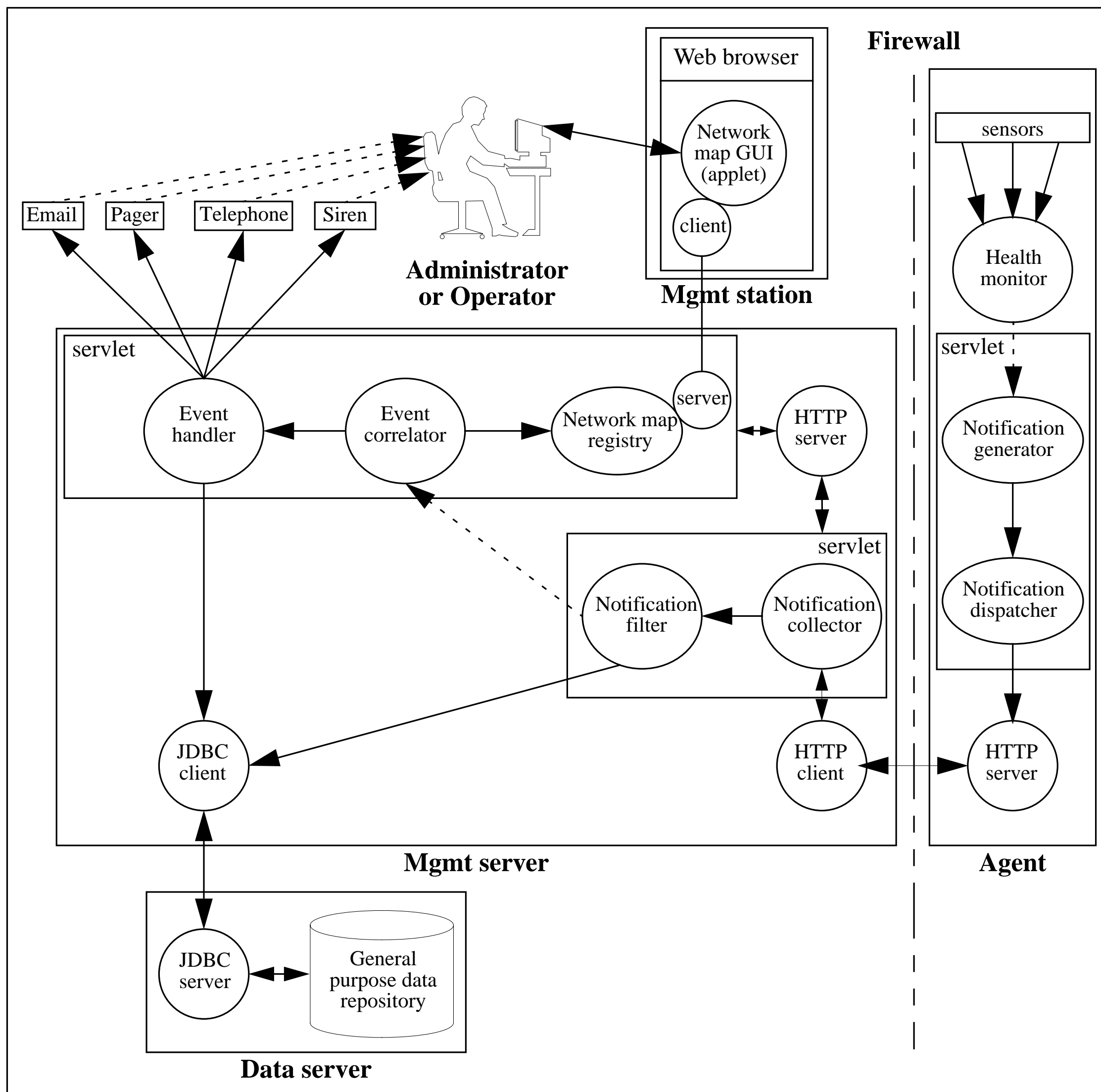


Fig. 2. Push-based notification handling

Our main motivation for developing 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, 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 [4]. 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 (e.g. `msql`) accessed via JDBC, as represented on the previous figures. Second, we have not yet written a network map GUI applet. Instead, as shown by Fig. 3, we use an event notification applet that simply displays incoming events, line by line, in a window. In the future, we will generate automatically a network map from a file describing the network topology, and will 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.

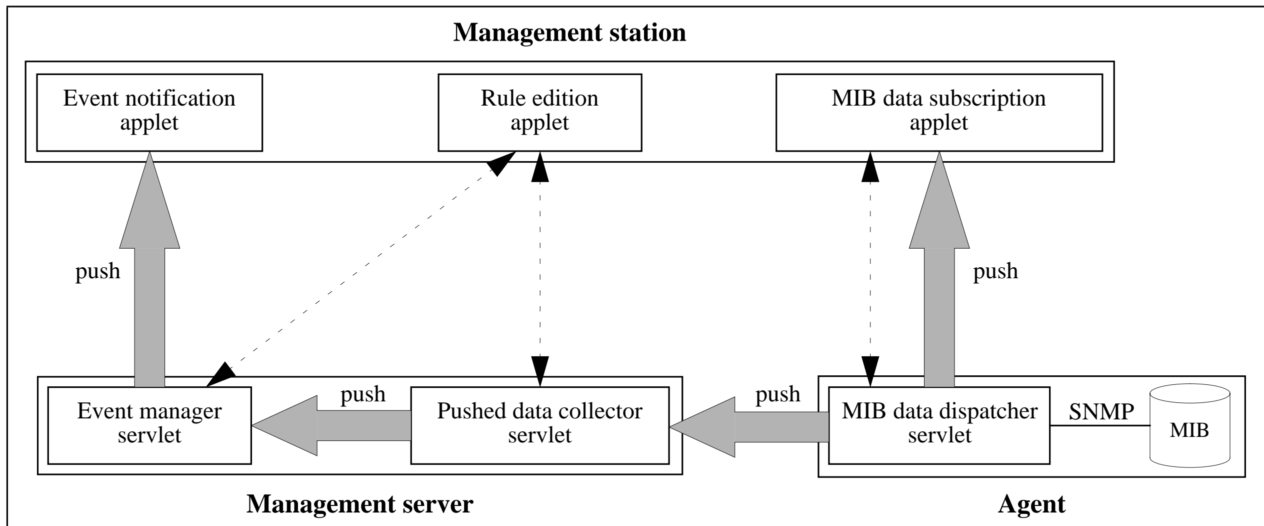


Fig. 3. Communication between Java applets and servlets: monitoring and data collection

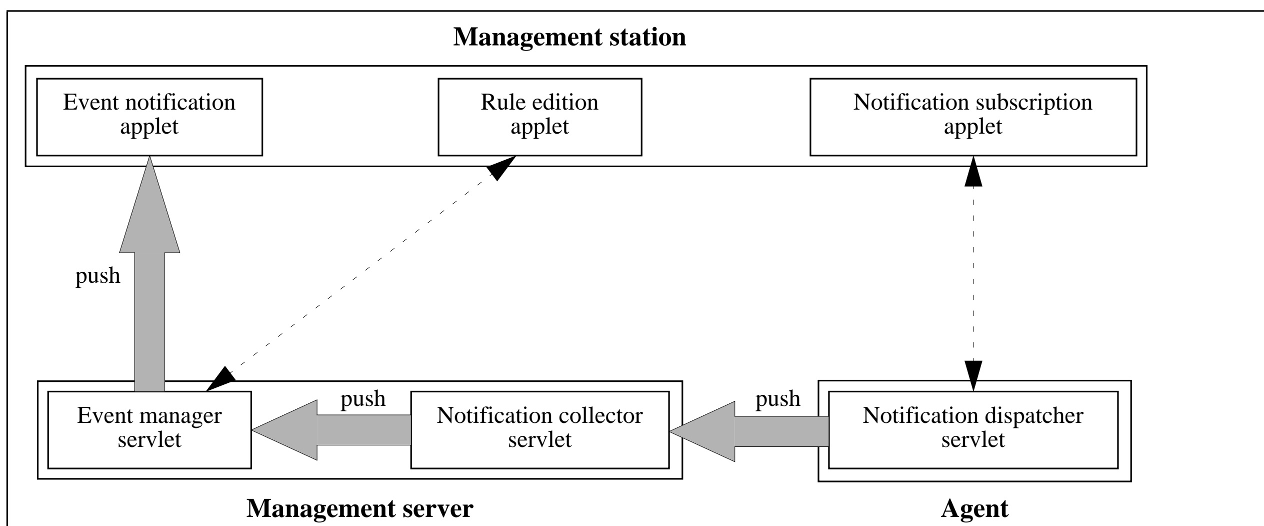


Fig. 4. Communication between Java applets and servlets: notifications

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. 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.

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 RMI, HTTP offers no native support for bidirectional persistent connections [16]. 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 [20] 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 [9]).

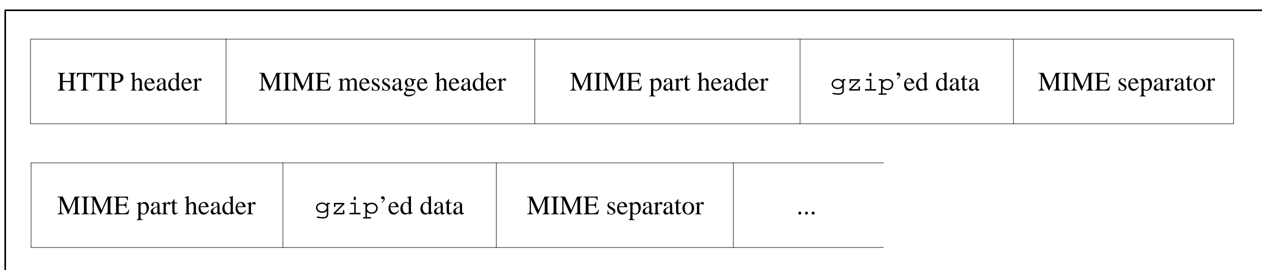


Fig. 5. TCP payload of the infinite HTTP response

We proposed to do the same in Web-based network management [16], 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 [7] only recently appeared on the Web; they are an improvement over the well-known CGI (Common Gateway Interface) scripts. Unlike CGI scripts that are typically written in a scripting language like Perl or Tcl/Tk, servlets are Java classes loaded in a 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 such as access control, secure transactions and virtual hosting.

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. We had problems with Jserv 0.8 because it did not support concurrent accesses to servlets and the response stream was buffered. Both of these problems were corrected in Jserv 1.0. In the meantime, we used another HTTP server, Jigsaw 2.0.1, developed by the World-Wide Web Consortium (W3C); this version of the server 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 agent configurations, rules and compiled MIBs.

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 the 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, the administrator 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 by the administrator 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 is not working or that a DBMS 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 [16].

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 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 (e.g. EPROM), 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 a collection of sensors. When a problem is detected, the health monitor asynchronously fires an alarm to the notification generator object in the servlet via some proprietary, tailor-made mechanism. The notification generator object checks with the filter if this alarm should be discarded or kept. 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. Software Reuse

Instead of reinventing the wheel, we strived to reuse publicly available software as much as possible in JAMAP. From the AdventNet SNMP package [1], we used:

- the `MibTree` class (GUI bean displaying the MIB tree);
- the `MibNode` class (node of a MIB tree);
- the `SnmpTarget` class (bean abstracting an SNMP-compliant device);
- the `SnmpVar` class (serializable SNMP variable); and
- the `SnmpTable` class (serializable bean representing an SNMP table).

We also used the `Util` class of the HTTPClient package written by Tschalär [24]. Compared to JDK 1.1.6, this package provides additional features such as HTTP header parsing. In the event mailer consumer (an event handler that sends email to operators and administrators), we used the SMTPConnection class of IBM's AlphaWorks SMTP package [12]. Finally, we used the `sun.tools.javac.Main` class of Sun's JDK Java compiler [22] to implement the dynamic compilation of the rules.

8. Conclusion

We have presented JAMAP, a network and systems management platform entirely written in Java. It implements the push model to perform regular management (permanent monitoring and data collection for offline analysis) and *ad hoc* management (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. In JAMAP, the only legacy of the SNMP framework is the MIBs; everything else uses Web technologies.

For future research, we plan to investigate different schemes to represent 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.

Acknowledgments

This research was partially funded by the Swiss National Science Foundation (FNRS) under grant SPP-ICS 5003-45311. The authors wish to thank H. Cogliati for proofreading this paper.

References

[1] AdventNet. *The AdventNet SNMP Package*. Available at <http://www.adventnet.com/products.html>. March 1999.

[2] 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.

[3] 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.

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

[5] R. Burns and M. Quinn. “The Cyber-Agent Framework”. *The Simple Times*, 4(3):12–15, 1996.

[6] B. Bruins. “Some Experiences with Emerging Management Technologies”. *The Simple Times*, 4(3):6–8, 1996.

[7] J.D. Davidson and S. Ahmed. *Java Servlet API Specification. Version 2.1a*. Sun Microsystems, November 1998.

[8] L. Deri. *HTTP-based SNMP and CMIP Network Management*. Internet draft <draft-deri-http-mgmt-00.txt> (work in progress). IETF, November 1996.

[9] N. Freed and N. Borenstein (Eds.). *RFC 2046. Multipurpose Internet Mail Extensions (MIME). Part Two: Media Types*. IETF, November 1996.

[10] B. Harrison, P.E. Mellquist and A. Pell. *Web Based System and Network Management*. Internet draft <draft-mellquist-web-sys-01.txt> (work in progress). IETF, November 1996.

[11] J.W. Hong, J.Y. Kong, T.H. Yun, J.S. Kim, J.T. Park and J.W. Baek. “Web-Based Intranet Services and Network Management”. *IEEE Communications Magazine*, 35(10):100–110, 1997.

[12] IBM. *AlphaWorks*. Available at <http://www.alphaworks.ibm.com/>. March 1999.

[13] W. Kasteleijn. *Web-Based Management*. M.Sc. thesis, University of Twente, Enschede, The Netherlands, April 1997.

[14] J. Lindsay. *The Web Based Management Page*. Available at <http://www.mindspring.com/~jlindsay/webbased.html>. March 1999.

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

[16] 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.

[17] J.P. Martin-Flatin. “Push vs. Pull in Web-Based Network Management”. To appear in *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM’99), Boston, MA, USA, May 1999*.

[18] 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.

[19] P. Mullaney. “Overview of a Web-based agent”. *The Simple Times*, 4(3):8–12, 1996.

[20] Netscape. *An Exploration of Dynamic Documents*. 1995. Available at <http://home.mcom.com/assist/net_sites/pushpull.html>.

[21] 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.

[22] Sun Microsystems. *Java Developer Kit 1.1*. Available at <http://www.javasoft.com/products/jdk/1.1/>. March 1999.

[23] J.P. Thompson. “Web-Based Enterprise Management Architecture”. *IEEE Communications Magazine*, 36(3):80–86, 1998.

[24] R. Tschalär. *HTTPClient version 0.3*. Available at <http://www.innovation.ch/java/HTTPClient/>. March 1999.

[25] C. Wellens and K. Auerbach. “Towards Useful Management”. *The Simple Times*, 4(3):1–6, 1996.

Biographies

J.P. Martin-Flatin is currently preparing for a Ph.D. thesis at EPFL. From 1990 to 1996, he was with the European Centre for Medium-Range Weather Forecasts in Reading, England, where he worked in network and systems management, security, Web management and software engineering. From 1988 to 1990, he worked on the Geographic Information System of a large city in France. In 1986, he received an M.S. in a mix of EE and ME from ECAM, Lyon, France. His main research interest is in distributed network management. He is a member of the IEEE and the ACM.

L. Bovet received an M.S. in computer science from EPFL in 1999. He developed the first version of JAMAP in the course of his M.S. thesis. In 1998, he did a summer internship in network management at Siemens in Renens, Switzerland. He will soon work with ELCA Informatik AG in Zurich, Switzerland.