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

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

The Push Model in Web-Based Network Management

Jean-Philippe Martin-Flatin

Version 1: July 1998
Version 2: October 1998
Version 3: November 1998

Technical Report SSC/1998/023

The Push Model in Web-Based Network Management

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

Abstract

The management of IP networks is currently based on the SNMP protocol, and the use of expensive network management platforms designed according to the manager/agent paradigm of the SNMP framework. It uses two different schemes to transfer management data: a request/response protocol for data collection and network monitoring (data polling), and unsolicited push to deliver SNMP notifications. This design is exposed to a number of problems, with regard to the time-to-market of vendor-specific management software, versioning, protocol efficiency, security, etc. In this paper, we propose a novel approach to network management based on the push model. This model is well-known in software engineering, and encountered a large success on the Web recently with the push technologies. It relies on the publish/subscribe/distribute paradigm, and uses a single scheme to transfer all management data. We describe why it is more efficient, in terms of network and systems resources, than the traditional pull model. We also explain in detail how to implement this model with Web technologies to deliver SNMP notifications, to handle events, and to distribute MIB data for network monitoring and data collection.

Keywords: Web-Based Management, IP Network Management, Push, Java, SNMP, HTTP, RMI.

1. Introduction

Most IP networks are currently managed with dedicated, expensive network management platforms, such as HP OpenView, Cabletron Spectrum, IBM Netview or Sun Solstice. In recent work [8], we analyzed the problems related to this way of managing networks, and put in light the advantages of going from SNMP-based to Web-based management[1]. We advocated the use of two design paradigms, the pull model and the push model, and showed that they could quickly be implemented and deployed by the industry.

A simple illustration of these models is given by the newspaper metaphor. If you want to read your favorite newspaper everyday, you can either go and buy it every morning (pull model), or subscribe to it once and then receive it automatically at home (push model).

The pull model is based on the request/response paradigm. It is a generalization of the data polling encountered in traditional SNMP-based network management. At every polling cycle, the manager (i.e., the client, or the management station[2]) sends several requests for MIB data to all agents (each agent runs one server); then, each agent answers separately to each request. The management data

1. The industry clearly favors Web-based management currently, rather than active networks, mobile agents or intelligent agents. HTTP servers have become a common feature in network equipment. This is why we assume in this paper that all agents have an HTTP server embedded (for legacy systems, we assume we go through a proxy). Conversely, very few devices have a *full* JVM embedded (that is, a JVM supporting RMI); we must take this fact into account in our engineering proposals. Whenever possible, we will only rely on a light-weight JVM (e.g., the JVM of the EmbeddedJava platform) that does not support RMI, but allows the execution of simple Java servlets.
2. For the terminology of IP network management, see [7,10,12].

transfer looks as if the client was “pulling” the data off the server. In this model, the data transfer is always initiated by the client.

The push model, conversely, is based on the publish/subscribe/distribute paradigm. It is inspired by the way SNMP notifications are delivered to the manager in traditional management. Management applications designed according to this model go through three successive phases: first, all agents advertise what MIBs they support, and what SNMP notifications they can send; second, for each agent, the administrator subscribes the management station to the MIB data or notifications he/she is interested in; for MIB data, the frequency at which the agent should send this data is also specified. Later on, each agent individually takes the initiative to “push” the data to the manager, either on a regular basis via a scheduler (for network monitoring and data collection), or asynchronously (to send SNMP notifications). In this model, the data transfer is always initiated by the server.

In [8], we showed how to implement network monitoring and data collection with either pull or push technologies. We also demonstrated why push-based network management generates less network traffic, and requires less CPU time on the management station by delegating[1] some of the processing to the agents.

In this paper, we study the push model in more detail, and show how the same communication technologies (HTTP, sockets and RMI) can be used by the agents to send unsolicited SNMP notifications or scheduled MIB data to the manager. We also explain to which extent notification handling, network monitoring and report generation are decoupled, and what are the constraints put on the location of management software when it is distributed over multiple machines (that is, management is performed by a *collapsed network management platform*).

The constraints we will try to satisfy are (i) to propose simple solutions that could be engineered and widely deployed in less than a year; (ii) to address the problems encountered by traditional SNMP-based network management [8]; (iii) to comply with the applet security model of JDK 1.1; (iv) to make it easy to go across firewalls (e.g, when the manager is inside an intranet, but some of the agents are outside, behind insecure WAN links); (v) to propose some solutions that do not require a full JVM to be embedded in all agents.

The remainder of this paper is organized as follows. In section 2, we describe the publish and subscribe phases of the push model. In section 3, we present the distribute phase, and study three communication technologies between the agent and the manager: HTTP, sockets and RMI. In section 4, we show a global picture of push-based network management, integrating all tasks. Finally, we conclude with some perspectives for future work.

2. Publish and Subscribe Phases

In the first phase, each network device (agent) must publish what MIBs it supports (generic MIBs, such as MIB-II, the ATM MIB, the RMON MIB or the FDDI MIB, or a vendor-specific MIB), and what SNMP notifications it can send to the manager (e.g., interface down, temperature of the mother board too high...). To implement this, we propose that all agents support two well-known HTML pages. The URL of these pages should be standard, in order to simplify the task of network administrators, and to allow partial automation of the subscription phase. The first URL lists all management applets stored on the agent:

1. For the rationale behind delegation, see Goldszmidt’s Management by Delegation scheme [4], or Wellens and Auerbach’s myth of the dumb agent [13]

`<URL:http://agent.domain/mgmt/mibs.html>`

where `agent.domain` is the fully qualified domain name of the agent. This requires that the directory called `mgmt` be reserved for the sole purpose of network management. The user then selects one of the entries, say MIB-II, and downloads an applet (one of the MIB data subscription GUIs that we will see next) to select data from that MIB. The format of URLs is free at this point.

The second well-known URL lists all notifications supported by the agent:

`<URL:http://agent.domain/mgmt/notifications.html>`

This HTML page is vendor specific, and even device specific. Typically, it would be an applet (the notification subscription applet that we will see next), with a nice GUI describing the notifications supported by the network device, and allowing the administrator to select or unselect each notification. Alternatively, it could simply be an HTML form with radio buttons or check boxes for each notification.

As depicted in Fig. 1, the *modus operandi* for the user (administrator or operator), in this first phase, is to upload the network map applet in a Web browser running on its local machine, then select an agent on the map, and load from that agent one of the well-known HTML pages stored in EPROM by the agent.

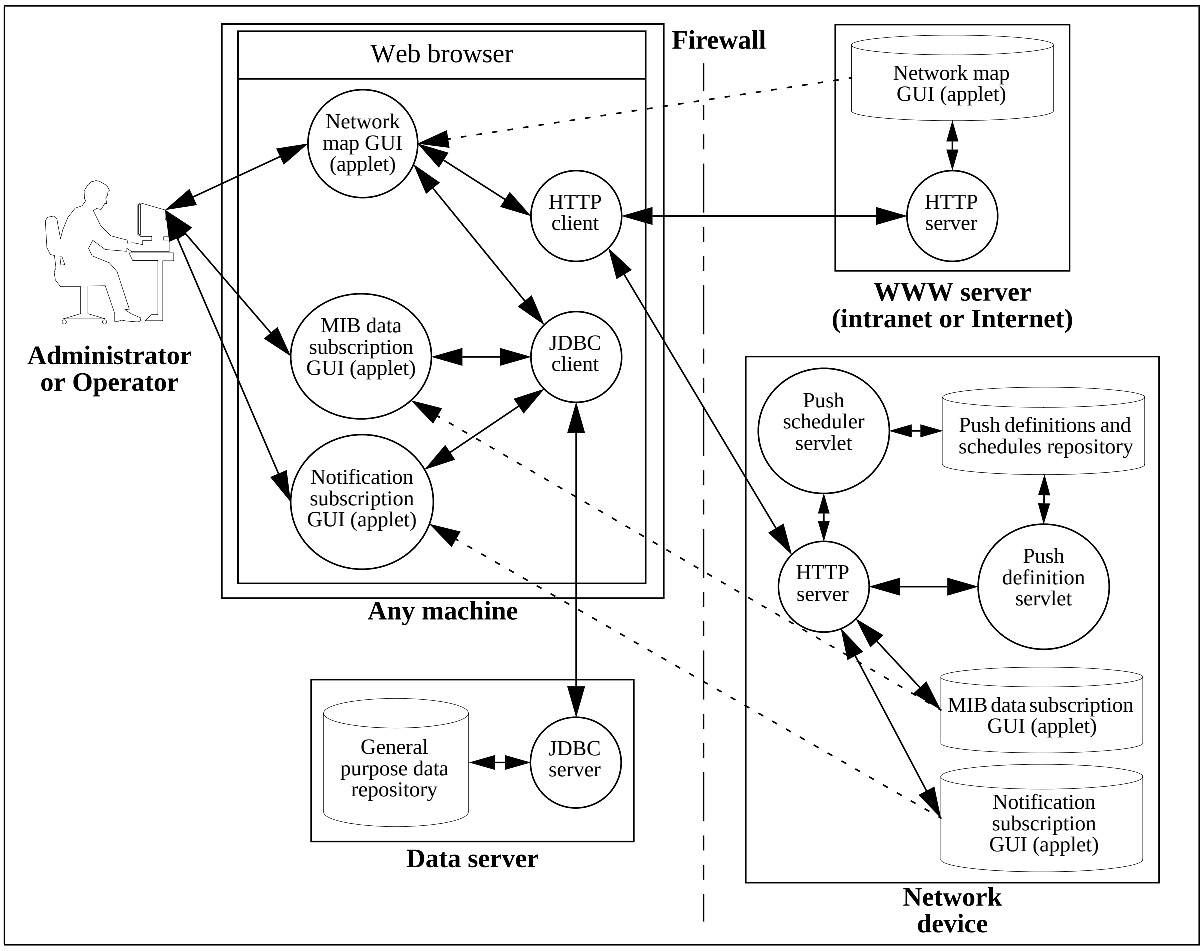
Web browser
Firewall
Network map GUI (applet)
HTTP client
MIB data subscription GUI (applet)
JDBC client
Notification subscription GUI (applet)
Any machine
Administrator or Operator
Network map GUI (applet)
HTTP server
WWW server (intranet or Internet)
Push scheduler servlet
Push definitions and schedules repository
HTTP server
Push definition servlet
MIB data subscription GUI (applet)
Notification subscription GUI (applet)
Network device
General purpose data repository
JDBC server
Data server

Fig. 1. Publish and subscribe phases

In the second phase, the administrator subscribes the manager to MIB variables and SNMP notifications. The so-called *MIB data subscription applets* allow him/her to select MIB variables as well as push frequencies. The push frequency can be different for each individual MIB variable, or it can be the same for a whole MIB, or a whole device. Unlike its counterpart, the *notification subscription applet* does not have to specify a push frequency, as notifications are inherently asynchronous. In fact, this notification subscription applet can be considered as a simple filter. Notifications for which the administrator showed no interest are discarded by the notification generator (see Fig. 2).

We could imagine to use a single applet to subscribe to MIB variables of all MIBs, instead of using one applet per MIB. But just like people dislike using MIB browsers in traditional SNMP-based management platforms because they are too basic, and prefer to use management GUIs customized for each MIB, people would not be happy if they had to subscribe to MIB data without visual aids customized for each MIB.

The subscription phase we described so far is entirely manual. Since it would be very tedious for the administrator to enter all over again all this subscription data, if an agent were to lose its configuration, it is important to store this data in a persistent data repository. As shown by Fig. 1, we use the data server for that. The details of the different data repositories (see [7]) are not shown on this figure, to keep it readable. All repositories are merged into a single *general-purpose data repository*. If an agent loses its push configuration data, the manager can then resend all the definitions and schedules for that agent in an unattended mode. The general purpose data repository of the data server includes (i) the definitions and schedules of the MIB data subscribed to by the manager, (ii) the definitions of the notifications subscribed to by the manager, and (iii) the network topology definition used by the network map applet to construct its GUI. In practice, these three logical data repositories may be stored into one or several databases.

3. Distribute Phase

In the distribute phase, the handling of data collection and network monitoring is only marginally different from that of notification delivery and events. The communication issues between the agent and the manager are the same; only the Java applications running on the manager side are different.

Let us consider notification delivery and event handling first. These tasks are depicted in Fig. 2. The network device runs a process monitoring its own health (the *health monitor*): its Ethernet interfaces still sense a carrier, the ventilation of the power supply is still working, etc. When an abnormal condition is detected, the health monitor contacts the notification generator, which translates a vendor-specific data structure in memory into a standard SNMP notification. This notification is then sent by the network dispatcher to the manager, where it is handled by the *notification collector*. To be precise, it is not the manager, like in traditional network management, but the host of part of the management application; this application is coded in Java, and runs on any machine supporting a JVM in the intranet. The notification collector passes this notification on to the notification filter, whose role is to detect misbehaving, misconfigured or malicious agents. If the manager is bombarded with notifications by an agent, this filter silently drops them, and possibly warns the administrator that something is going wrong with this agent.

Once the notification filter has checked an incoming notification, it sends it to the event correlator. Like in traditional network management platforms, this is the central point of network monitoring: events generated by the pushed data interpreter (as we will see further), on the manager, and events directly received from agents, are analyzed, and the source of the network problem is identified (if a router is down, the hosts behind that router will appear to be down, but no corrective action should be taken for the hosts: only the router should be repaired).

If the administrator or an operator has registered a network map GUI with the *network map registry*, the event correlator forces an update of that GUI (icons will turn red, green, yellow...). If no network map is registered, that is, no one is monitoring the network right now, and network management is entirely automated, we entirely rely on *event handlers*. These may simply log the problem in a file, or they may take more drastic actions such as paging the administrator, starting off a siren, etc. Event handlers are configured by the administrator via a GUI not displayed here (see [7]).

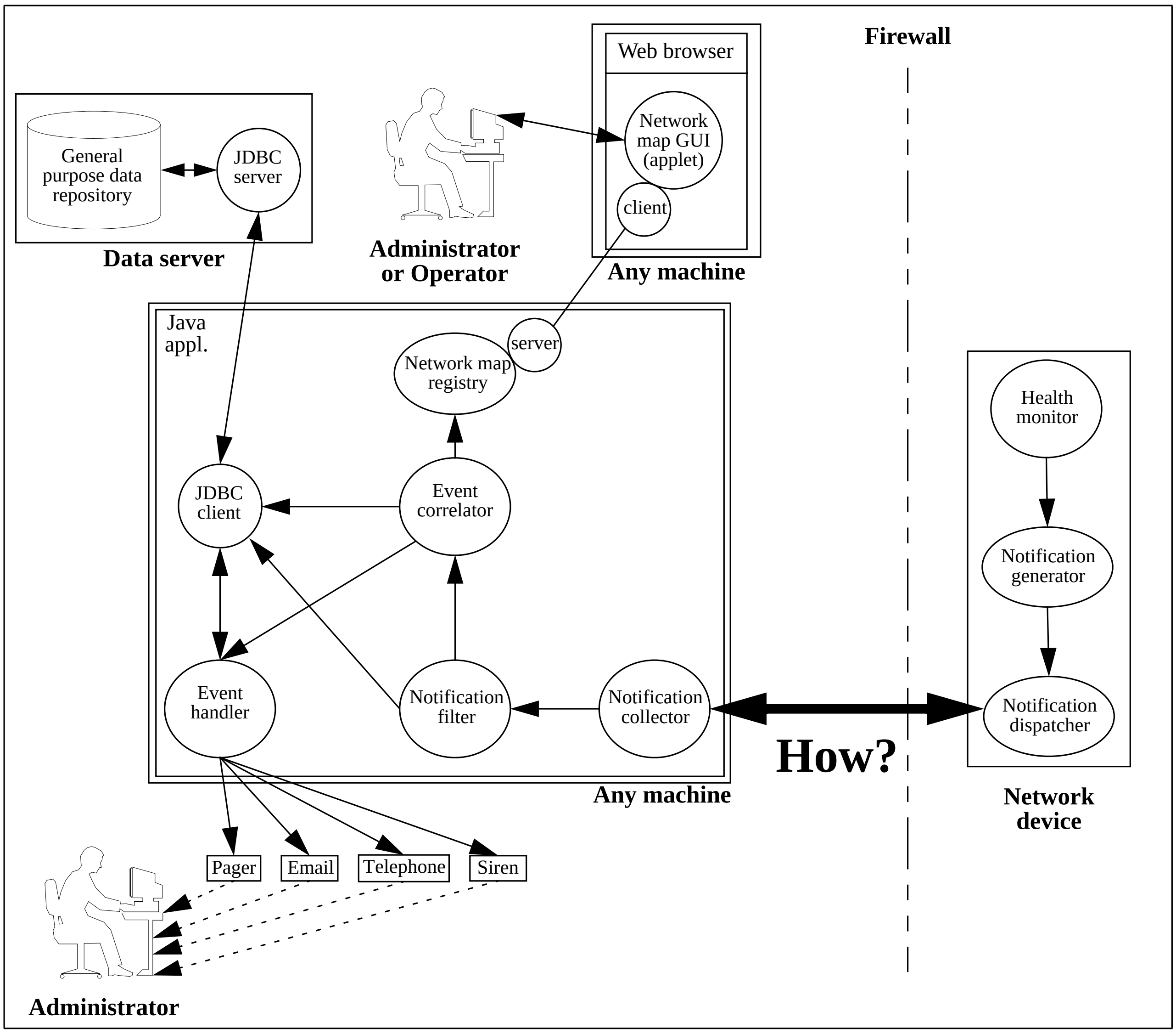

Fig. 2. Distribute phase for notification delivery

Some events may also be stored in the data repository, although administrators should be careful to keep only relevant data. Event statistics may be more useful than the actual events. The general purpose data repository depicted in Fig. 2 includes seven different repositories: (i) the definitions and schedules of the notifications subscribed to by the manager; (ii) the definitions of the MIB data subscribed to by the manager; (iii) the network topology definition used by the network map applet to construct its GUI; (iv) the event handler definitions repository; (v) the event handlers invocation log; (vi) the pushed data repository; and (vii) the pushed notifications repository. Like previously, in real life, all these logically different data repositories may actually reside in one or more databases.

If we consider network monitoring and data collection instead of notification delivery and event handling, the main difference is in the Java application running on the manager, as we see clearly by comparing Fig. 2 and Fig. 3. This time, instead of a notification collector, we have a *pushed data collector*, that collects data related to network monitoring or data collection. Instead of the notification filter, we have a *pushed data filter*, which plays a similar role and increases the robustness of the system. For data collection, that is, data whose sole purpose is to build statistics and daily, weekly or monthly reports, the pushed data filter sends the data directly to the data server, via JDBC[1]. For network monitoring, we go via an extra level of indirection, the *pushed data interpreter*, which can generate events when, for instance, a network device no longer sends a heart beat — typically its `sysObjectID` (MIB-II). Events generated by the pushed data interpreter are sent to the event correlator, where they are mixed with notifications and processed as described earlier.

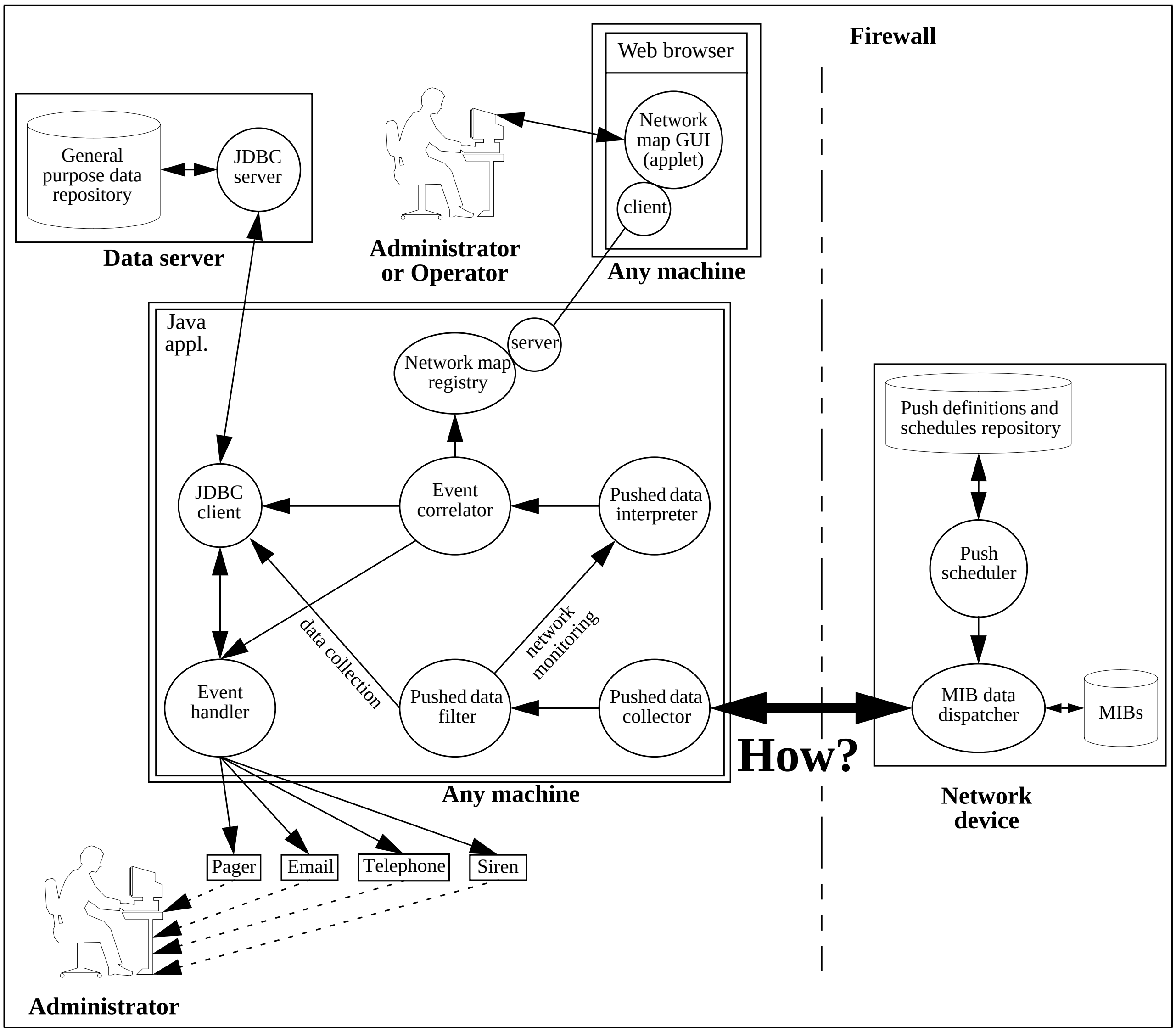

Fig. 3. Distribute phase for network monitoring and data collection

1. Since the execution speed of Java code is slow, the performance may be significantly increased by storing data in bulk. Depending on how tables of the RDBMS are organized, this can lead to small or very significant speed-ups.

One point we did not mention so far is how the agent and the manager communicate with each other. Both scenarios have a dispatcher on the agent and a collector on the manager, but how do they exchange data? We showed in [8] the advantages of using a persistent connection between the manager and the agent. For security reasons, especially if we need to go across a firewall, this persistent connection must be initiated by the manager, not by the agent. But when we go from the pull model, which underlies traditional SNMP-based management, to the push model that we advocate in this paper, the client/server roles are swapped. The transfer of management data is now initiated by the agent, instead of the manager; but the client side of the persistent connection remains on the manager, and the server side on the agent. Compared to the usual mapping between the manager/agent paradigm and a client/server architecture, the client and the server are on the wrong sides! Somehow, we want the server to initiate the communication, whereas communication must be initiated by the client in a client/server architecture.

To address this issue, distributed Web programming gives us three communication technologies [11]: HTTP, sockets and RMI. For each of them, let us now study how to ensure some kind of persistent connection between the client and the server. In particular, we will pay attention to the repercussions when we need to go across a firewall.

3.1. Sockets

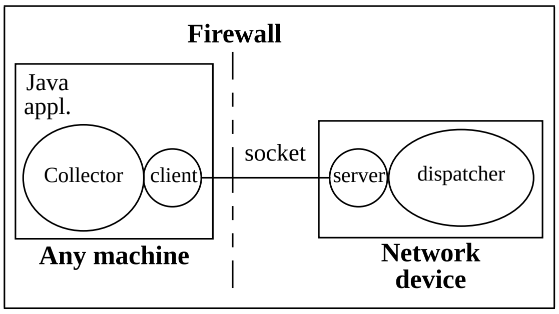

Fig. 4. Distribution via sockets

Sockets present a very interesting property: they are bidirectional. When a socket is created, the client side of the socket contacts the server side, as usual in the client/server architecture. But once this socket is established, be it a TCP or a UDP socket, the client can send data to the server, but the server can also independently send data to the client via the same socket. This property solves our problem of server-initiated communication: the manager can create a socket to each agent, that is, create a virtual pipe between the manager and all the agents managed by this manager; but later on, only the agents will use these pipes to send data across. This data can either be SNMP notifications or MIB data: everything goes across the same virtual pipe.

To ensure that this connection remains persistent, the collector, on the manager side, must set an infinite time-out value on the socket when it creates it. If the underlying TCP connection times out for whatever reason, it is the responsibility of the manager (i.e., the collector) to reconnect to the agent, by creating a new socket.

This solution presents a big advantage: simplicity. Programming with sockets is very easy, especially in Java. But it also has some drawbacks. First, if the operating system of the machine hosting the manager or the agent keeps timing out the connection, then this solution is clearly inappropriate. Typically, this would happen if the time-out value of the socket was lower than the push frequency of the agent, and if the operating system of this agent would not allow the administrator to change the time-out value. In this case, not only do the repeated socket creations and time-outs cause network and CPU overhead, but even worse, it is not reasonable to take the risk of making notifications delivery depend on such a versatile type of persistent connection; there must be a way for the agent, not the manager, to create a new connection if the previous times out.

The second problem is related to firewalls. if we need to go across a firewall between the manager and the agent, there is a potential issue with sockets. Most firewalls filter out UDP, and let only a few TCP ports go through [1]. So whether we use TCP or UDP sockets, firewalls will generally not let sockets go through by default. Thus, in order for this socket-based solution to work, the firewall system needs to be modified. This may not be a problem for large organizations, because they generally have in-house expertise to set up UDP relays or update TCP filtering rules, or can afford consultants to do the job if necessary. But it is likely to be a problem for SMEs, who generally lack such expertise, and for whom expensive external consultants are only a last resort option.

We might face a third problem with persistent TCP sockets if the number of agents to monitor from a single manager is large: the number of concurrent TCP descriptors required might exceed the maximum allowed by the operating system. On many Unix systems, this problem can be solved by modifying a single configuration parameter of the kernel and rebuilding it—something routinely done on servers running very busy WWW servers.

3.2. Java RMI

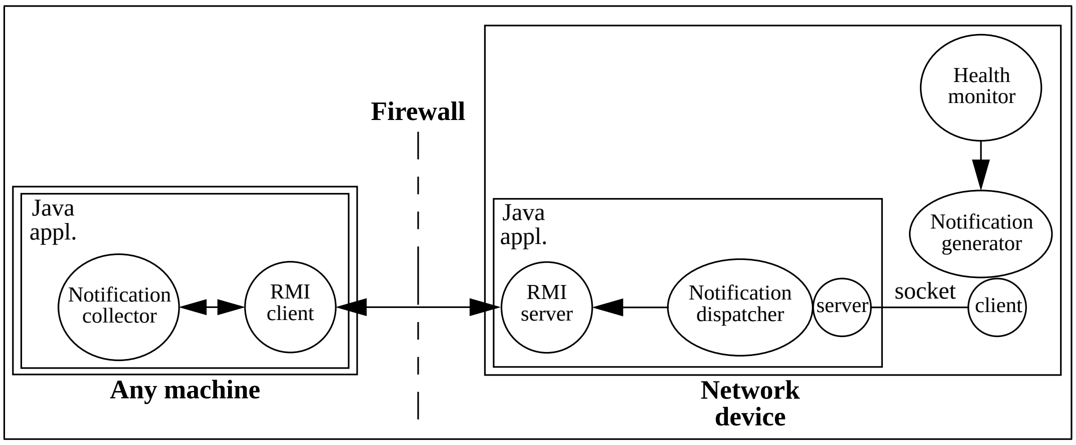

Fig. 5. Distribution via RMI for notification delivery

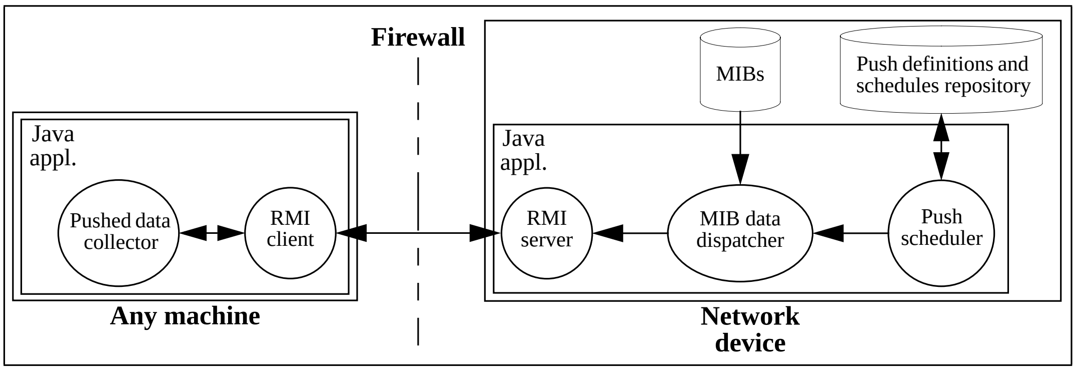

Fig. 6. Distribution via RMI for data collection and network monitoring

Like sockets, Java RMI offers a bidirectional association: once an RMI client has bound to an RMI server, both of them can send data to the other. RMI is an elegant solution in terms of design, because it gives a fully object-oriented view of network management. It offers semantics to the network management application designer that are higher than mere MIB variables, and makes it easier to design complex applications [6].

With RMI, things are slightly different for notification delivery and data collection. This time, we have a Java application running on the agent, thus we must decide what should be coded in Java and what should not. In Fig. 5, we show a solution where the health monitor and the notification generator are non-Java programs, compiled and executing fast, while the notification dispatcher is coded in Java. The notification generator and dispatcher exchange data via a socket. Conversely, for data collection and network monitoring (see Fig. 6), the solution we propose is to code everything in Java: only the data repositories (be they virtual, like MIBs, or real, like push definitions and schedules) are non Java. To improve the efficiency, the agent-side Java application can use native code to access the data repositories.

Unfortunately, RMI presents severe drawbacks. First, it requires that all agents embed a full JVM. Very few do today, and the large footprint of a full JVM makes it unlikely that bottom-of-the-range, price-sensitive devices will offer one before long. Second, current RMI implementations are slow to execute, and use much CPU and memory; as it currently stands, RMI-based network management is not scalable. Things may improve in future implementations, especially if we keep in mind that RMI is a fairly recent technology, which has not gone through many upgrade cycles yet. But the fact that other distributed object-oriented platforms, like CORBA or DCOM, suffer from the same problems, incites us to believe that fully object-oriented IP network management will remain, at best, a niche market for the years to come. The third and last problem with RMI is that the communication between RMI clients and RMI servers is based on sockets, which are transparent to applications. So once again, we have a problem with firewalls. Actually, things are even worse with RMI, because we no longer control what ports are used by sockets. RMI sockets are transparent to the application, so even if RMI servers run on a well-known port (`1099/tcp` or `1099/udp` [5]), RMI clients may bind to any port (whereas in the previous solution, the administrator was in control of the ports used by the client and the server). As a result, specific software must be added to the firewall system in order to go across it; and RMI relays are not widely supported by firewall systems today (they may be in the future, if RMI proves successful over time).

For all these reasons, we cannot reasonably base IP network management on RMI in the near future, although it is a neat solution on paper.

3.3. HTTP

HTTP does not exhibit the property that we exploited for sockets and RMI. With HTTP, connections are oriented: it is not possible to create a persistent connection in one direction, from the client to the server, and later send data in the opposite direction. All HTTP methods rely on a strict request/response protocol: for an HTTP server to send a response to an HTTP client, it must have received a request from this client beforehand. It cannot send unsolicited messages.

In this respect, SNMP and HTTP behave differently. Both are based on the client/server model of communication. But SNMP implements a request/response protocol for some of its operations (`get`, `set`, `inform`...), and a one-way asynchronous transfer protocol for others (`snmpv2-trap`). HTTP, conversely, implements a strict request/response protocol for all of its methods (`get`, `post`, `head`...).

How can we work around this design limitation? How can we have an HTTP server send an infinitely large number of replies to a single request from an HTTP client? The trick is to make the server pretend that it is sending a single endless reply, and to embed separators in the payload of the HTTP messages. To do that, Netscape proposed to use the `multipart` type of MIME [3, 9] as early as 1995, in the context of the Web. We propose to use it in IP network management too. In our case, we send one MIME part at each new time interval, and the MIME boundary is interpreted as an *end of time interval* marker. The main issue here is to control the time-out value of the embedded HTTP server: persistent HTTP/1.1 connections are assumed to be short-lived by the Web community, typically a few seconds, whereas we typically need several minutes in IP network management. The second issue is that of the operating system timing out inactive TCP sockets, which we already presented in section 3.1. Either vendors allow their customers to change these two time-out values, or we need another solution.

Let us suppose that we need to find another solution. In order to allow HTTP-based communication between the manager and the agent, we must find a new answer to the challenge of server-initiated communication. The one we propose is to add an HTTP/1.1 client on the agent, and an HTTP/1.1 server on the manager, so as to re-establish a normal client/server communication.

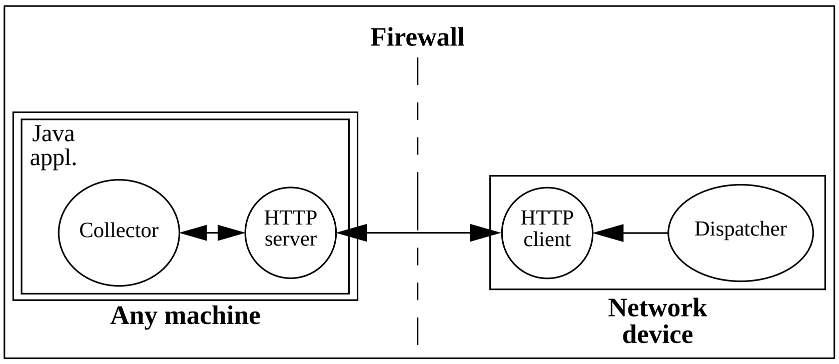

Fig. 7. Distribution via HTTP

This solution presents several advantages. First, it does not rely on non-intuitive designs, which stretch the client/server architecture to its limits: the client is on the agent side, and the server on the manager side. Second, the agent can reconnect immediately in case the persistent connection times out: it does not have to count on the manager to do that; this improves the robustness, and avoids time windows when the agent wants to send data to the manager, but the manager has not reconnected to the agent yet. Third, no change at all is required on the firewall system, if the management application runs on the external Web server of the organization; if it runs on a different machine, a minor change in the setup of the firewall system is needed.

The main drawback of this solution is that it requires an HTTP server to be included in the Java application running on the manager side. This makes a large program (the network management application) even larger, more difficult to debug, slower to execute, and induces a larger footprint on the machine where it is running.

4. Push-Based Network Management: the Global Picture

In the previous sections, we presented different snapshots of a push-based network management application. But we did not show how the different Java applications coexist on the manager side, for event handling and network monitoring. In this section, we integrate all these partial views, and show that a coherent Web-based design model can deal with all the management tasks performed by traditional network management platforms [7].

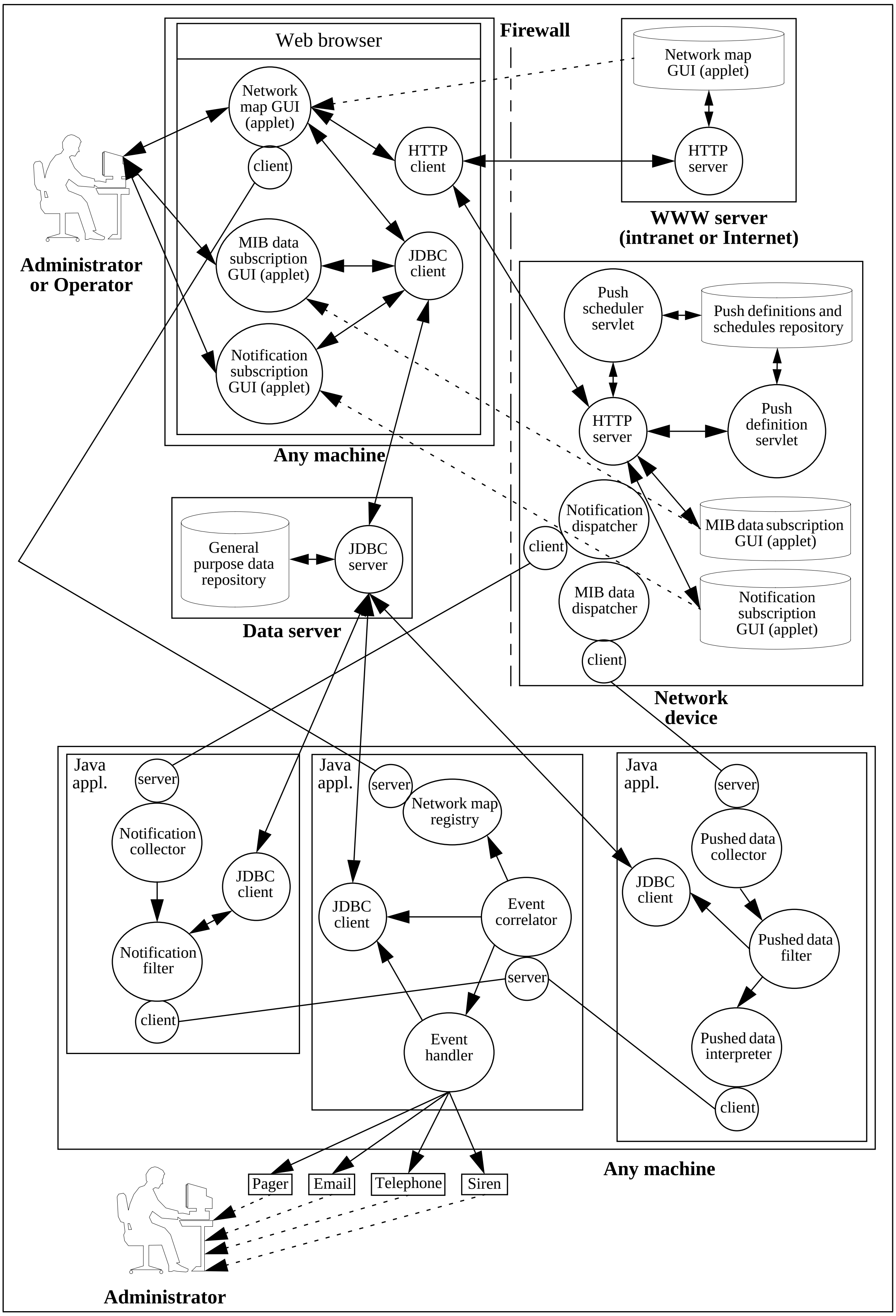

Fig. 8. The collapsed network management platform

This leads us to the concept of *collapsed network management platform* [8], whereby the network management platform no longer exists per se. Instead, the management application is spread over several machines, each fulfilling a particular task. Several network map GUIs can be viewed concurrently by Web browsers running on different machines; logical data repositories can be physically stored in one or several databases in the intranet; the core of the management application is divided into three different modules, which can be integrated into one large Java application running on a single machine, or than can run separately as three different applications on three different machines (in Fig. 8, they communicate via sockets).

Among the three communication technologies presented in the previous section, we selected the first, based on sockets, to make Fig. 8 more readable. This is not to say that this technology is the one we advocate: depending on site-specific needs, administrators may select HTTP instead, or even RMI for top-of-the-range devices.

5. Conclusion

In this paper, we presented the engineering details of a new design paradigm for IP network management applications: the push model. This model addresses a number of deficiencies in traditional SNMP-based network management, and does not require expensive network management platforms. We explained how to go across firewalls, how to use existing RDBMSs, and how to distribute the management application across several machines. We described a coherent framework integrating SNMP notification delivery, event handling, data collection and networking monitoring. In a companion paper [8], we describe how to integrate the pull model, better suited for ad hoc management (manual mode), and the push model, better adapted for regular management (automatic mode).

For future work, it would be worth investigating active databases. In our current proposal, agents send data to the collector object of the Java application running on the manager side, and then rely on this application to take some actions if needed, and to store data in a repository. Instead, agents could directly send data to an active database, and then rely on trigger-based actions taken by this database.

Acknowledgments

This research was partially funded by the Swiss National Science Foundation (FNRS) under grant SPP-ICS 5003-45311. The author wishes to thank G. Madhusudan for discussions on communication in distributed Java applications, and L. Bovet on MIME multipart.

Acronyms

ATM	Asynchronous Transfer Mode	MIB	Management Information Base
CORBA	Common Object Request Broker Architecture	MIME	Multipurpose Internet Mail Extensions
CPU	Central Processing Unit	RDBMS	Relational DataBase Management System
DCOM	Distributed Common Object Model	RFC	Request For Comment
EPROM	Electrically erasable Programmable Read-Only Memory	RMI	Remote Method Invocation
FDDI	Fiber Distributed Data Interface	RMON	Remote MONitoring
GUI	Graphical User Interface	SME	Small or Medium-sized Enterprise

HTML	HyperText Markup Language	SNMP	Simple Network Management Protocol
HTTP	HyperText Transfer Protocol	TCP	Transmission Control Protocol
IETF	Internet Engineering Task Force	UDP	User Datagram Protocol
IP	Internet Protocol	URL	Uniform Resource Locator
JDBC	Java DataBase Connectivity	WAN	Wide-Area Network
JDK	Java Development Kit	WWW	World-Wide Web
JVM	Java Virtual Machine		

References

[1] D.B. Chapman and E.D. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Sebastopol, CA, USA, 1995.

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

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

[4] G. Goldszmidt. *Distributed Management by Delegation*. Ph.D. thesis, Columbia University, New York, NY, USA, December 1995.

[5] IANA. *Protocol Numbers and Assignment Services*. Available at <URL:http://www.iana.org/numbers.html>. This Web site updates RFC 1700 which is now obsolete.

[6] J.P. Martin-Flatin, S. Znaty and J.P. Hubaux. "A Survey of Distributed Enterprise Network and Systems Management". To appear in *Journal of Network and Systems Management*, vol. 7, no. 1, March 1999.

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

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

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

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

[11] P. Sridharan. *Advanced Java networking*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[12] W. Stallings. *SNMP, SNMPv2 and CMIP: the Practical Guide to Network Management Standards*. Addison-Wesley, Reading, MA, USA, 1993.

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

Biography

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.Sc. 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.