In M. Sloman, S. Mazumdar and E. Lupu (Eds.), *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, Boston, MA, USA, May 1999, pp. 3–18. IEEE Press, 1999.

Push vs. Pull in Web-Based Network Management

J.P. Martin-Flatin
Swiss Federal Institute of Technology, Lausanne (EPFL)
Institute for computer Communications and Applications (ICA)
1015 Lausanne, Switzerland
martin-flatin@epfl.ch

Abstract

In this paper, we build on the concept of *embedded management application*, proposed by Wellens and Auerbach, and present two models of network management application designs that rely on Web technologies. First, the *pull model*, based on the request/response paradigm, is a generalization of data polling in the SNMP management framework. We explain how to use it for ad hoc management (e.g, troubleshooting) and regular management (e.g., network monitoring). Second, the *push model* is a novel approach that relies on the publish/subscribe/distribute paradigm. It is better suited to regular management than the pull model, and allows administrators to conserve network bandwidth as well as CPU time on the management station.

Keywords

Web-Based Management, Network Management, IP Networks, Push Model, Pull Model, Java, Embedded Management Application.

1. Introduction

The idea of using Web technologies in IP network management is not new. Experiments with the early Web technologies (Web browsers, HTTP, HTML and CGI scripts) started in 1993-94. Initially, they were only confined to secondary tasks [4]. The first important step toward Web-based network management was taken when vendors began embedding HTTP servers in their network equipment. Bruins [1] reports some early experiments made by Cisco in 1995, whereby the entire command line interface was mapped to URLs. This opened new doors for configuration management and symptom-driven HTML forms, as there was no more need to `telnet` into network devices. Mullaney [7] also describes some work conducted by FTP Software, whereby agents send static or dynamic HTML pages back to the Network Management Station (NMS) in response to an HTTP `get` or `post` request. Many network equipment vendors, including Cisco, Nortel Networks and 3Com, now routinely embed HTTP servers in their new equipment.

The second important step was taken when Java applets appeared in Netscape's famous Web browser, in 1995. The seminal article by Wellens and Auerbach [11] introduces the concept of *embedded management application*, and shows the advantages of using HTTP rather than SNMP to vehicle data between managers and agents. Although the authors do not explicitly refer to applets in their article, the solution they propose is to transform an add-on (i.e., a vendor-specific management GUI that has to be ported to many different management platforms and operating systems) into a single applet that can run anywhere. This applet is stored in the managed device, and loaded by the administrator into a Web browser. Communication between the applet and its origin agent later relies on HTTP instead of SNMP.

Since the time of this proposal, new technologies have appeared on the Web. Today, in addition to applets and Java applications, we can use Java servlets and Remote Method Invocation (RMI), we can open persistent sockets between applets and servlets, etc. All these new possibilities enable new designs of network management applications. Leveraging on these new technologies, we propose to push Wellens and Auerbach's idea two steps further. In section 2, we show that the design paradigm they propose is just one instance of a more general paradigm, the *pull model*, which can not only be applied to ad hoc management, as they do, but also to regular management. In section 3, we introduce a novel approach based on the *push model*. Unlike the pull model, it is not based on the request/response paradigm, but on the publish/subscribe/distribute paradigm. With this scheme, management data transfers are always initiated by the agent, as SNMP notifications delivery in pre-Web network management. The push model reduces network overhead, and moves part of the CPU burden from managers to agents.

2. The Pull Model

In sections 2.1 and 2.2, we describe the pull and push models, and explain the difference between ad hoc and regular management. In section 2.3, we present the engineering details of pull-based ad hoc management, and show how Web technologies can complement the management functionalities offered by the NMS. In this case, network troubleshooting can be done from any machine running a Web browser, whereas the NMS remains in charge of regular management. Finally, in section 2.4, we show that Web technologies can also deal with regular management.

2.1. Pull vs. push: the newspaper metaphor

In software engineering, the *pull model* and the *push model* designate two well-known approaches for exchanging data between two distant entities. The newspaper metaphor is a simple illustration of these models: if you want to read your favorite newspaper everyday, you can either go and buy it every morning, or subscribe to it once and then receive it automatically at home. The former is an example of pull, the latter of push. The pull model is based on the request/response paradigm (called *data polling,* or simply *polling,* in the SNMP management framework); the client sends a request to the server, then the server answers, either synchronously or asynchronously. This is functionally equivalent to the client “pulling” the data off the server. In this approach, the data transfer is always initiated by the client, i.e. the manager. The push model, conversely, is based on the publish/subscribe/distribute paradigm. In this model, agents first advertise what MIBs they support, and what SNMP notifications they can generate; the administrator then subscribes the manager (the NMS) to the data he/she is interested in, specifies how often the manager should receive this data, and disconnects. Later on, each agent individually takes the initiative to “push” data to the manager, either on a regular basis via a scheduler (e.g., for network monitoring) or asynchronously (e.g., to send SNMP notifications).

2.2. Ad hoc management vs. regular management

The simplest and most intuitive application of the applet technology is ad hoc management, which requires a user (administrator or operator) to interact with the management software via some GUIs. Ad hoc management is typical of transient tasks: you connect to a network device, retrieve some data to check something, and disconnect shortly after. Regular management, conversely, is concerned with ongoing data collection, network monitoring and event handling. It is automated to a large extent, and generally runs continuously.

Ad hoc management takes place in virtually all companies. In large organizations who can afford staff dedicated to monitoring the network (operators), or who rely on entirely automated regular management, ad hoc management is complementary to regular management. Conversely, in Small and Medium-sized Enterprises (SMEs), ad hoc management generally replaces regular management. There is no operator and no dedicated NMS: the management software is only used occasionally, on an ad hoc basis. Ad hoc management typically consists in troubleshooting (i.e., a network problem just showed up, and the administrator tries to identify and fix the problem manually), or configuration management (e.g., the administrator sets up a new router, or checks if a router is configured properly).

2.3. Pull-based ad hoc management

Applets address several issues in SNMP-based network management [4]. They decrease vendors’ development costs for management GUIs; they address the issue of having different versions of a vendor-specific MIB in the same network; they cut the time-to-market of management GUIs down to zero; and they are independent of the machine where the Web browser runs. Applets can also replace all the management GUIs that we find in pre-Web network management platforms [6].

Vendor-specific management GUIs coded as applets

In the approach proposed by Wellens and Auerbach, the vendor-specific management GUI is coded as an applet. They call it the *embedded management* *application*. The uploading of the applet by the Web browser is depicted in Figure 1. The HTTP server running on the agent retrieves its vendor-specific management applet from local storage, e.g. from EPROM, and sends it back to the Web browser. Once the applet is uploaded by the Web browser, there are two ways to proceed: either use SNMP or HTTP.

If we use SNMP, the interactions between the manager (Web browser running on any machine) and the agent (network device) are depicted in Figure 1. Steps 1 and 2 describe the applet transfer; they occur only once. Steps 3 and 4 describe the management data transfers; they are an iterative process. The dotted arrow for step 2 is a visual aid that shows that the applet is transferred from the agent to the manager. In reality, this transfer takes place between the HTTP client of the Web browser and the HTTP server of the agent.

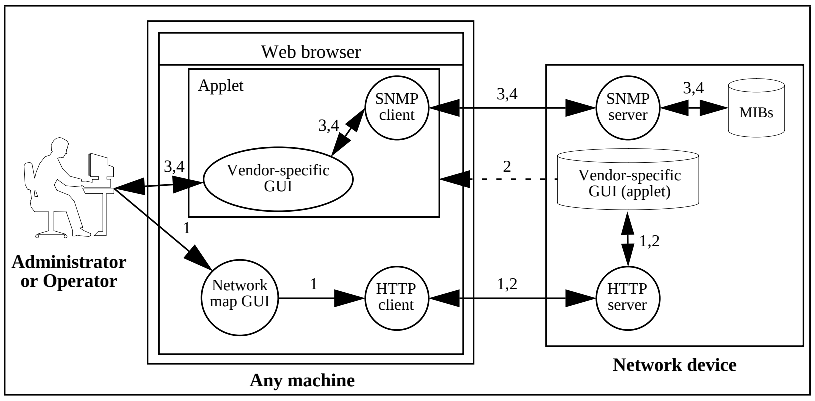

Figure 1: Pull model: HTTP together with SNMP

Once the applet is uploaded, the user has the equivalent of an add-on in a pre-Web network management platform to interact with. Graphical interactions are translated into SNMP commands by the applet (e.g., a mouse click on the drawing of a reset button can be mapped to an SNMP `set`). In other words, we have a Java API making SNMP calls underneath.

How can we load this SNMP stack into the browser? The simplest approach is to include an SNMP stack in the applet, as described by Bruins [1], because the applet security model prevents it from retrieving this stack from the local file system. So each time a management applet is uploaded from a network device, the entire SNMP stack needs to be moved along. This is clearly inefficient, especially if this transfer takes place across a WAN link. An improvement on this is to retrieve the SNMP stack separately, via a socket. Because of the applet security model, sockets may only be opened between an applet and its *origin server*, in HTTP parlance; so we need a proxy to act as the origin server [4]. The management applet is first requested by the Web browser to the proxy; second, the proxy contacts the network device and retrieves the applet without the SNMP stack; third, the applet is passed along to the Web browser which executes it; fourth, the applet opens a socket to the proxy, which runs the server side of the socket; fifth, the SNMP stack is transferred via the socket; and sixth the socket is closed. All subsequent SNMP traffic between the manager and the agent is simply relayed "as is" by the proxy.

This approach works fine; it is of limited interest though, because we do not benefit from the advantages of using HTTP instead of SNMP [4]: reduced network overhead, improved security, etc. So let us now study the case when the recurrent steps 3 and 4 are based on HTTP rather than SNMP (see Figure 2). This time, there is no need for an SNMP client in the Web browser, thus no need for a proxy.

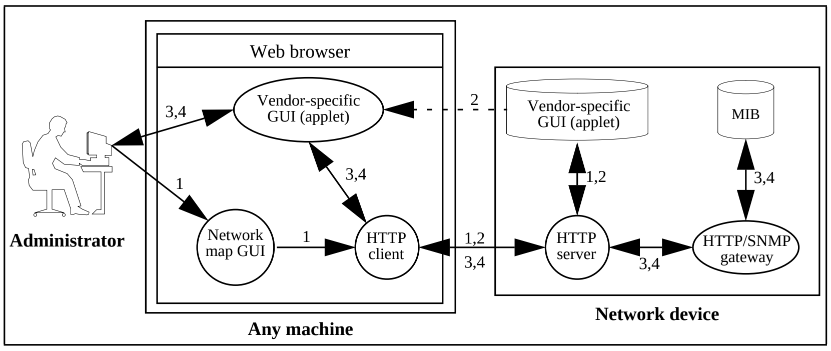

Figure 2: Pull model: HTTP instead of SNMP

When a MIB variable is requested by the applet, the request is made to the HTTP server run by the agent. The HTTP server then launches an HTTP-to-SNMP gateway to access the local MIB. The gateway can be a CGI script, a Java servlet, etc. Depending on the degree of optimization of the code run by the agent, the gateway can either directly access the MIB data structures in memory, or do an explicit SNMP `get` or `set`. This gives a useful migration path to network equipment vendors.

Generic management GUIs coded as applets

So far, we followed the original idea of Wellens and Auerbach: only vendor-specific management GUIs are coded as applets. The next step is to code generic GUIs as applets, too. (A generic GUI supports a generic vendor-independent MIB such as MIB-II, the RMON MIB or the ATM MIB [6].) At this point, ad hoc management relies entirely on Web technologies (see Figure 3).

The Web server can be any machine on the Internet or the intranet; for robustness, it should clearly be attached to the intranet. The "MIBs" icon in the network device represents all the generic MIBs supported by the agent, plus its vendor-specific MIB.

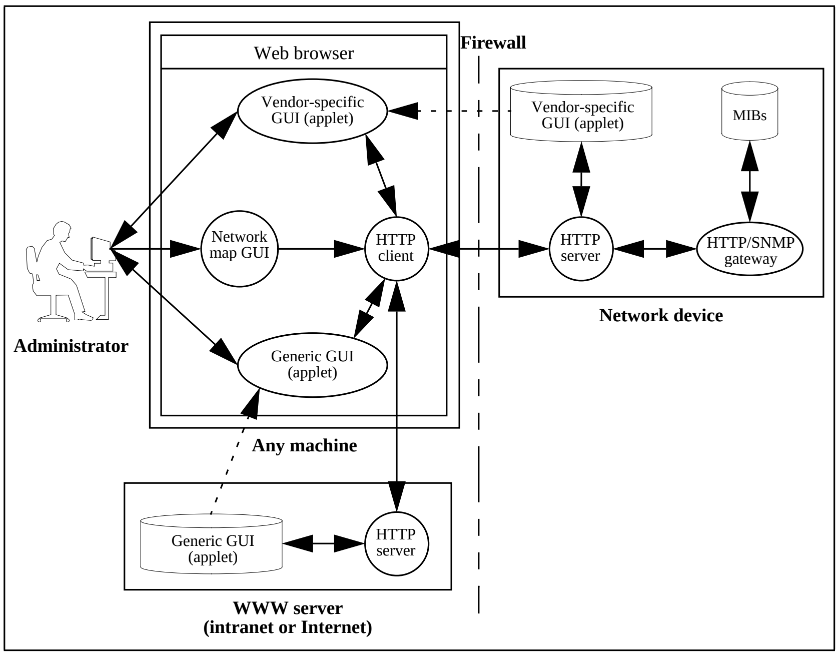

Figure 3: Pull model: ad hoc management based on HTTP

In this section, we showed that SMEs that only rely on ad hoc management for their network equipment can save the cost of a network management platform by using Web technologies instead. Thus, depending on the pricing policy of network equipment vendors, the move from SNMP-based to Web-based network management can completely reshape the SMEs segment of this market.

2.4. Regular management

In addition to ad hoc management, many organizations need to perform regular management. For them, network management must be automated to a large extent, including data polling and event handling which are not dealt with by the solutions we presented so far. Let us concentrate on data polling in this section. Let us suppose we have two separate management platforms, side by side: one for regular management, based on SNMP, and one for ad hoc management, based on Web technologies. Let us now describe, step by step, how to integrate them.

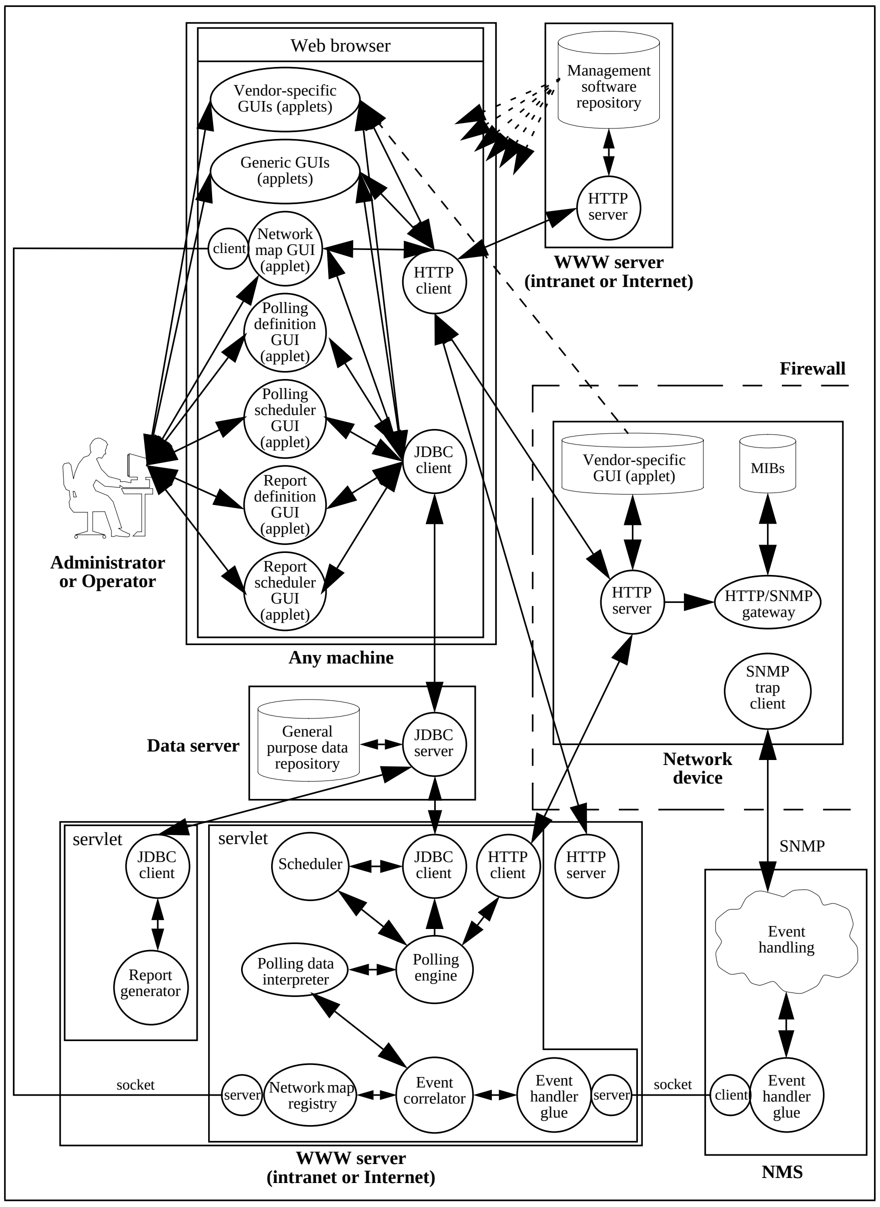

Figure 4: Pull model: data polling based on HTTP

The first step is to integrate a Web browser in the network management platform, in order to have a unified interface for ad hoc and regular management. This feature is now offered by some commercial network management platforms. The network map GUI is an applet that is uploaded by the Web browser. It can be updated dynamically by the event correlator, with icons turning red, green, etc. An easy way to update this map would be to open a socket directly between the applet and the event correlator. But we do not necessarily want to have a network map GUI: regular management can rely entirely on event handlers, when it runs in unattended mode. In such a case, the administrator is automatically paged, mailed or telephoned when a serious problem is detected. To cope with this case, we add an intermediary, the network map registry, between the network map GUI(s) and the event correlator. Thus, we can have zero, one or several network map GUIs registered independently.

The applet security model mandates that the applet be downloaded from the machine with the server side of the socket. So the WWW server at the bottom of Figure 4 and the WWW server at the top right are actually the same machine. They are represented separately to avoid too many overlapping arrows.

The second step is to replace all the GUIs of the network management platform [6] with applets. The vendor-specific applet is still loaded from the network device. All other GUIs are retrieved from the management software repository; the WWW server must then act as a proxy because of the applet security model.

The third step is to make the data repository independent of the network management platform. We assume here that data is stored in a third-party RDBMS; note that we may as well use a plain text file system, or an object-oriented database. The data repository resides on a machine that we call the data server, which can be any machine of the intranet. To store or retrieve data, we use JDBC. In order to make Figure 4 easier to read, we assume that we have a single data repository for polling and report definitions and schedules; but this is not mandatory.

The fourth step is to implement data polling with HTTP. This is achieved by a servlet. Upon start-up, the polling engine retrieves all polling definitions and schedules from the RDBMS. The scheduler makes it poll all agents on a regular basis. The data retrieved for network monitoring is checked by the polling data interpreter, which may generate an event in case a problem is inferred. This event is then dispatched to the event correlator, which can decide to update the network map(s). The communication between the manager and the agent is based on HTTP.

The fifth step is to migrate reports generation to Web technologies. This is done by another servlet, which accesses the data repository via JDBC. This data is stored by the polling servlet, and is accessed independently by the report generator. For the sake of simplicity, we pictured the two servlets, report generation and data polling, as running on the same machine; but this does not have to be the case.

At this stage, data collection and network monitoring rely entirely on Web technologies. They no longer require an expensive NMS, dedicated to network management. SNMP notifications delivery and event handlers are the only tasks still performed in the traditional way.

3. The Push Model

The push model generalizes to network monitoring and data collection the way SNMP notifications are delivered today. Despite its large success in software engineering, it has always been confined to SNMP notification delivery in IP network management. To the best of our knowledge, no network management platform uses it today for network monitoring or data collection. Yet, we claim that its very design makes it better suited to regular management than the pull model.

The chief advantages of using the push model are to conserve network bandwidth and move part of the CPU burden from managers to agents. Much of the network overhead caused by the pull model is due to the fact that there is a lot of redundancy in what the manager keeps asking all agents at every polling cycle for data collection and network monitoring. With the push model, the manager contacts each agent once, subscribes to an OID once (push data definition), and specifies at what frequency (push frequency) the agent should send the value of this OID (push data schedule). Afterward, there is no more traffic going from the manager to the agent (except in the rare cases when the manager wishes to change its subscription). The agent "remembers" what the manager subscribed to by keeping the push definitions and schedules on local storage; if this is EPROM, the agent can retrieve these definitions and schedules by itself after a reboot; if this is RAM, it needs to retrieve them from the manager, which stores them in the data server.

The point of moving part of the CPU burden from managers to agents is to decrease the requirements put on NMSs in terms of CPU and memory. Network management platforms for large networks are often big Unix or Windows NT servers, which cost a fortune to buy and maintain. Agents, on the other hand, are more powerful than they used to be, and many of them can reasonably do a bit of processing locally (see the rationale behind Goldszmidt's Management by Delegation scheme [3], or Wellens and Auerbach's myth of the dumb agent [11]).

Compared to the pull model, the push model introduces a new issue: synchronization. If the manager and the agent have internal clocks that do not synchronize regularly, they will probably drift apart. This is not a problem for network monitoring, but it can be for data collection [4]. For push technologies to work, it is therefore recommended to synchronize the clocks of network equipment on a regular basis, e.g. with NTP. Let us now delve into the engineering details of the push model.

3.1. Publication and subscription phases

In the first phase, the network device (agent) publishes what MIBs it supports, and what SNMP notifications it can send to the manager. A simple way to implement this is to use applets, as depicted in Figure 5. First, the user selects an agent on the network map applet, and loads from that agent a well-known HTML page, e.g. <http://agent.domain/mgmt/mibs.html>, which lists the applets stored on the agent.

Every applet publishes one MIB (vendor-specific or generic) supported by the agent, except one, which publishes the SNMP notifications supported by this agent.

In the second phase, the administrator subscribes the manager to MIB variables and SNMP notifications. MIB data subscription applets allow him/her to select MIB variables as well as push frequencies. The push frequency can be specified at the MIB variable level: it need not be the same for all variables of a given MIB. The push frequency is equal to the polling frequency considered in section 2. Obviously, the notification subscription applet does not have to specify a push frequency, as notifications are inherently asynchronous. In fact, the notification subscription applet is simply a filter: it specifies what notifications the manager is interested in. Other notifications are discarded by the agent.

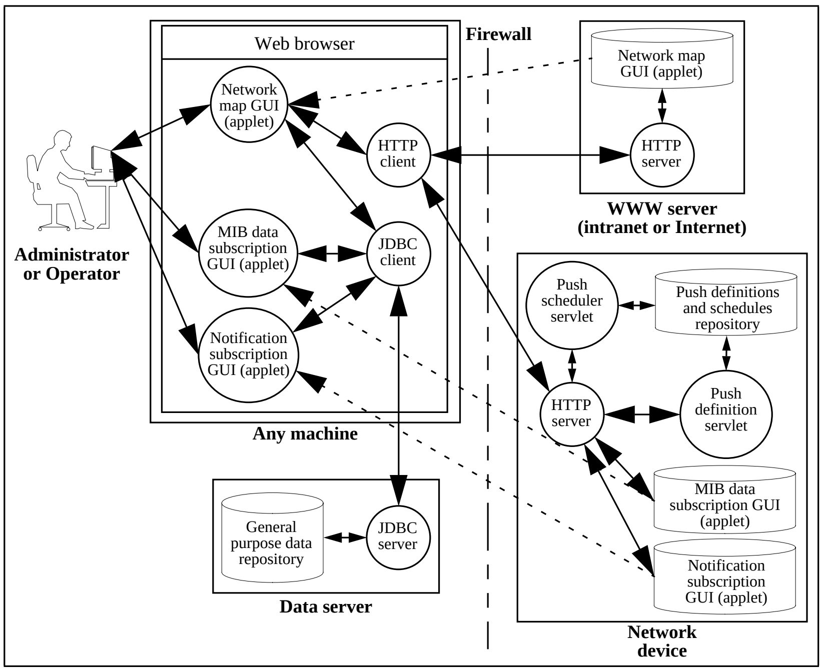
Web browser
Firewall
Network map GUI (applet)
HTTP client
MIB data subscription GUI (applet)
JDBC client
Notification subscription GUI (applet)
Any machine
Administrator or Operator
Network map GUI (applet)
HTTP server
WWW server (intranet or Internet)
Push scheduler servlet
Push definitions and schedules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

Figure 5: Push model: publication and subscription phases

The publication and subscription phases are depicted in Figure 5. The details of the MIB and notification subscriptions are stored in the data server. In case an agent loses all its push configuration data, this allows the manager to resend all the definitions and schedules for that agent in unattended mode: the administrator does not have to enter it all over again manually, via a GUI. 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 real life, these three logical data repositories may actually be stored into different databases, or a single database.

3.2. Distribution phase

In the distribution phase, the case of data collection and network monitoring is only marginally different from the case of notification delivery and event handling. In order to facilitate the comparison with the pull model, we will concentrate on data collection and network monitoring in this paper, that is, how to replace data polling with push technologies. Notification delivery and event handling are presented in detail in [5]. Let us stress that the solutions we will describe below apply equally well to both cases: the communication issues between the agent and the manager are the same, only the servlets running on the manager side are different.

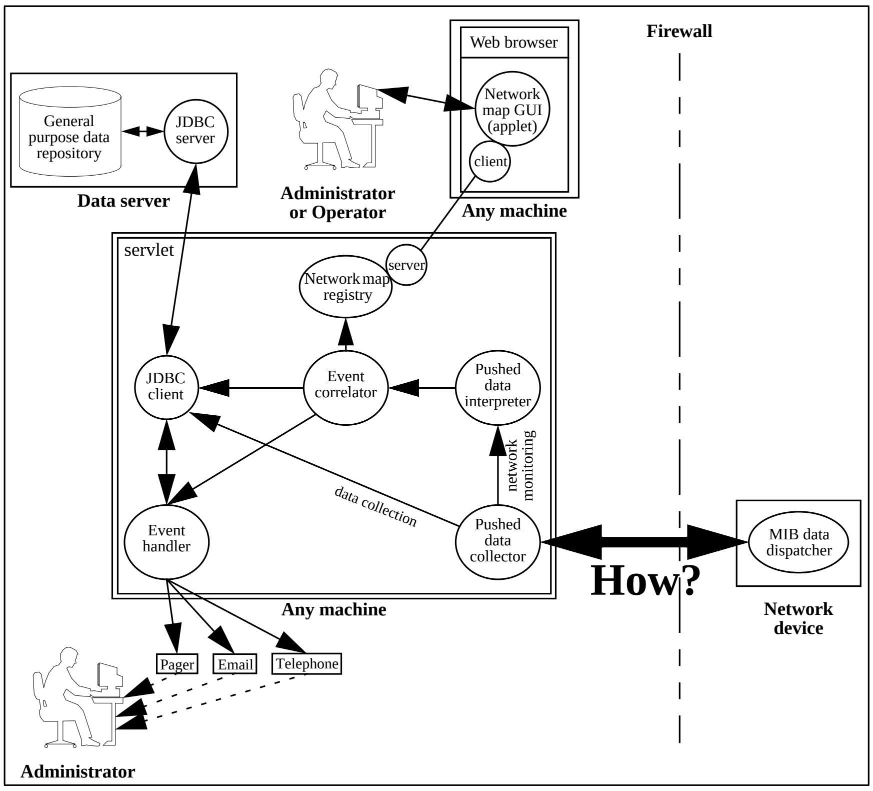

Figure 6: Push model: distribution phase

The general purpose data repository depicted in Figure 6 includes seven different repositories: the three listed in section 3.1, plus (iv) the event handler definitions repository, (v) the event handlers invocation log, (vi) the pushed data repository, and

(vii) the pushed notifications repository. Again, in real life, all these logically different data repositories may actually reside in one or more databases.

Compared to Figure 4 (pull model), we no longer have a polling engine; instead, we have a pushed data collector, which receives the data necessary to monitor the network or build usage statistics reports. This data is stored on the data server via JDBC. Since the execution speed of Java code is slow, it may be a good idea to increase the performance by storing data in bulk. As the polling engine in the pull model, the pushed data collector sends data collected for network monitoring to the pushed data interpreter. If an abnormal condition is detected by the pushed data interpreter, e.g. a device no longer sends any data, an alarm is generated in the form of an event sent to the event correlator. The event correlator also receives events in the form of notifications (not shown here), and identifies the problem with the network. It can invoke an event handler, when an event is not masked by another, in which case the call to the event handler is logged in the data server.

The main difficulty when going from pull to push is that the data transfer is now initiated by the agent, instead of the manager, while the client remains on the manager side, and the server on the agent side. Somehow, the client and the server are on the wrong sides! We would like the server to initiate the communication, whereas communication is always initiated by the client in a client/server architecture. To address this issue, we have the choice between three communication technologies: HTTP, sockets and RMI [9].

Sockets

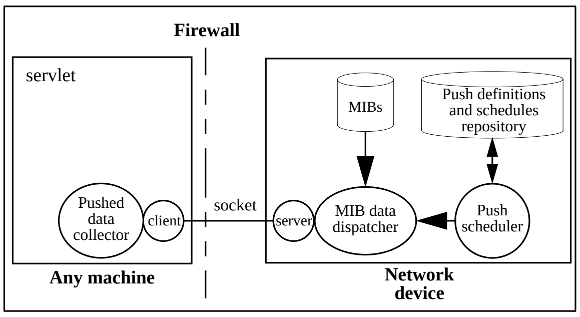

Figure 7: Push model: distribution via sockets

The fact that sockets are bidirectional solves the problem of server-initiated communication: we can open a socket as usual, from the client to the server, and, later on, only use it to send MIB data from the socket server to the socket client. To ensure that this connection remains persistent, the pushed data collector, on the manager side, sets 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 (pushed data collector) to open a new socket to the agent.

This socket-based solution presents a big advantage: simplicity. Sockets are very simple to program, especially in Java. But it also presents two potential drawbacks. First, if the underlying operating system of either the manager or the agent keeps timing out the connection (e.g., because the administrator has no control over the time-out value of the socket, and this time-out value happens to be lower than the push frequency), then this solution is clearly inappropriate. Not only do the repeated socket creations and time-outs cause network and CPU overhead, but even worse, we cannot take the risk to make 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 timed out. Second, 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 [2]. 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 either have in-house expertise in firewalls to set up UDP relays or change TCP filtering rules, or they can afford expensive external consultants to do the job. But it may well be a problem for SMEs, who generally lack this kind of expertise, and for whom expensive external consultants may not be an option.

RMI

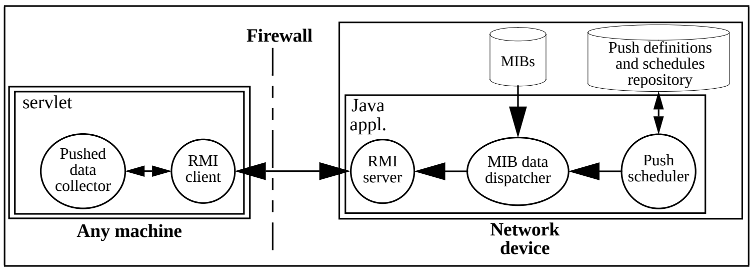

Figure 8: Push model: distribution via RMI

Just like sockets, 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. But RMI also presents several drawbacks. First, it requires a full JVM to be embedded in all agents (as opposed to a light-weight JVM such as the one included in the EmbeddedJava platform [10]). Very few network devices offer this feature today. And many will not have a full JVM for some time, because of the large footprint of this software on bottom-of-the-range devices that are very price sensitive. Second, current RMI implementations are very slow, and use many CPU and memory resources; today, RMI-based network management is not scalable. Third, RMI communication is actually based on sockets, which are transparent to applications; so once again, we face a problem with firewalls. In fact, things are even worse with RMI, because the administrator no longer controls what port is used on the client side. So, in order to use RMI across a firewall, it is necessary to add RMI-specific software to the firewall system; and RMI relays are not supported by all firewall systems today. Therefore, sockets appear to be better than RMI for IP network management.

HTTP

HTTP does not share the property we exploited for sockets and RMI. It is not possible for the HTTP server to initiate a data transfer via a pre-existing persistent connection. All HTTP 1.1 methods rely on a strict request/response protocol, so a server cannot send a response without having received a request beforehand.

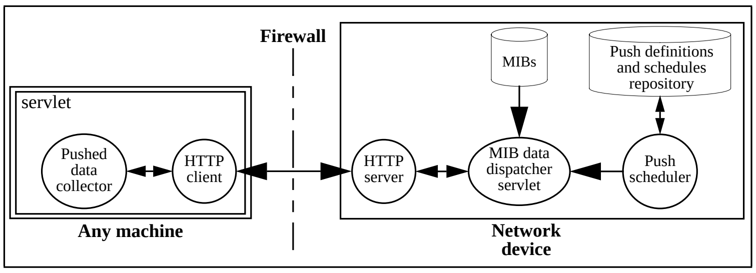

Figure 9: Push model: distribution via HTTP (MIME)

We can work around this by having the HTTP server send an infinitely large number of responses to a single request from an HTTP client. More precisely, the HTTP server sends a single endless reply, with separators embedded in the payload of the HTTP messages. Netscape proposed to use the `multipart` type of MIME in the context of the Web [8]. We propose to use it in IP network management. In the case of push, we send one MIME part at each new time interval; the MIME boundary is then interpreted as an *end of time interval* marker.

The main problem here is the control of the time-out value of the TCP connection underlying the persistent HTTP/1.1 connection. First, persistent HTTP/1.1 connections are assumed to be short-lived by the Web community, typically a few seconds, because they were created primarily for busy Web servers. In IP network management, we typically need several minutes, so we must be able to change the time-out value of the HTTP server. Second, the operating system can time out idle TCP sockets (as we saw with sockets). Will vendors allow their customers to change these two time-out values? For the HTTP server, the answer is probably *yes*. Apache, reportedly the most successful HTTP server to date, already allows it. As to the operating system, the answer is likely to be *no*, because it would require a modification of the kernel on most machines – a very sensitive task that many vendors would not want their customers to do. In fact, they would rather configure their equipment with a rather high default time-out value, e.g. 20 minutes, to cope with the most common push frequencies.

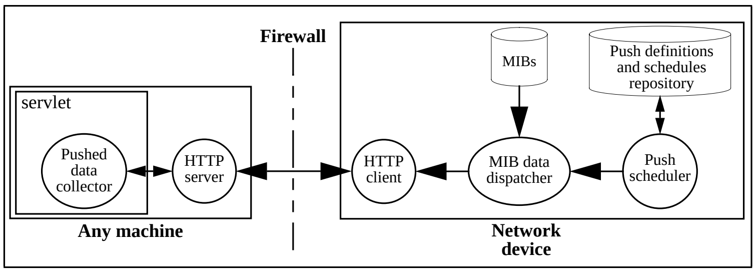

Figure 10: Push model: distribution via HTTP (client and server swapped)

Alternatively, we can swap the positions of the HTTP client and server, that is, put 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 (see Figure 10). This solution presents several advantages. First, it does not rely on counterintuitive designs as we saw previously: the client and the server are on the right 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 this, which improves the robustness. Third, no change 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, only a minor change is needed. The main drawback of this solution is that the firewall must be configured to let external agents create new persistent HTTP connections to the internal manager; this is less secure than letting the manager create new connections.

4. Conclusion

We presented two complementary design approaches based on Web technologies: the pull model, well suited to ad hoc management, and the push model, well adapted to regular management. Engineering solutions were presented for data collection and network monitoring for both models. A companion paper [5] describes how to perform notification delivery and event handling with the push model. More technical details can be found in [4].

To implement the push model, the changes required in managed devices are limited when data distribution relies on HTTP or sockets. For the pull model, they are even more limited: we only need an HTTP server and an HTTP-to-SNMP gateway. Top-of-the-range network equipment can optionally use RMI, but none of our models mandates that a full JVM be embedded in all agents. We therefore believe that the solutions we described can be deployed in the industry in a relatively short time frame. Early contacts with network equipment vendors confirmed that.

Acknowledgments

This research was partially funded by the Swiss National Science Foundation (FNRS) under grant SPP-ICS 5003-45311. The author would like to thank G. Madhusudan for discussions on communication in distributed Java applications, and H. Cogliati for proofreading this paper.

References

[1]. B. Bruins. “Some experiences with emerging management technologies”. In *The Simple Times*, 4(3):6–8, 1996.
[2]. D.B. Chapman and E.D. Zwicky. *Building Internet firewalls*. O’Reilly & Associates, Sebastopol, CA, USA, 1995.
[3]. G. Goldszmidt. *Distributed management by delegation*. Ph.D. thesis, Columbia University, New York, NY, USA, December 1995.
[4]. 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.
[5]. 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.
[6]. J.P. Martin-Flatin. *IP network management platforms before the Web*. Technical Report SSC/1998/021, version 2, SSC, EPFL, Lausanne, Switzerland, December 1998.
[7]. P. Mullaney. “Overview of a Web-based agent”. In *The Simple Times*, 4(3):8–12, 1996.
[8] Netscape. *An Exploration of Dynamic Documents*. 1995. Available at <http://home.mcom.com/assist/net_sites/pushpull.html>.
[9]. P. Sridharan. *Advanced Java networking*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[10]. Sun Microsystems. EmbeddedJava. Available at <http://www.javasoft.com/products/embeddedjava/>.
[11]. C. Wellens and K. Auerbach. “Towards useful management”. In *The Simple Times*, 4(3):1–6, 1996.