WEAVER: REALIZING A SCALABLE MANAGEMENT PARADIGM ON COMMODITY ROUTERS

Koon-Seng Lim & Rolf Stadler
KTH Royal Institute of Technology
Stockholm

Abstract: While there is agreement on the drawbacks of centralized management, many approaches that address those do not scale well to large networks. We believe that effective management of future large-scale networks requires decentralized but coordinated control. In our recent work, we introduced the paradigm of pattern-based management, an approach that formalizes the use of graph traversal algorithms for controlling and coordinating lightweight agents that perform computations and data aggregation inside the network. We have shown analytically and through simulations that such a management system potentially scales to tens of millions of nodes, without significant performance problems regarding execution time and traffic overhead. In this paper, we report on a first implementation designed to realize the paradigm. Our system, *Weaver*, consists of active nodes constructed from small, low-cost Linux computers that are deployed onto a network of commodity routers. Management programs are written in C++ and can be validated and tested for performance on a simulator before being deployed. From the design of Weaver, we derive a simple performance model that allows us to predict the execution times of management operations on this platform. We evaluate the model through measurements on a laboratory testbed and demonstrate the efficiency of the platform. Finally, we use the model to predict the performance of a management operation running on a Weaver system for a large-scale network and thus show that our system is likely to meet the scaling potential of the paradigm.

Key words: Network management, scalability, management platform, active and programmable networks

1. INTRODUCTION

Over the last decade, the drawbacks of centralized management schemes have been recognized [2][3][22], and several approaches to distributing management tasks have been developed [9][32]. Interestingly, most of this research aimed at distributing the computations associated with a management task while keeping the overall control of a task centralized. In our recent work, we reached the conclusion that effective management of future large-scale networks requires decentralized management operations.

A significant step towards decentralized control has been made with the introduction of mobile agents for management tasks. Mobile agents can be characterized as self-contained programs that move in the network and act on behalf of a user (i.e., a human operator) or another entity. Mobile agents are generally complex, since they often need a degree of intelligent behavior for autonomous decision-making. Our approach can be understood as a variation of the mobile agent paradigm, where a management operation is realized through the coordinated actions of a swarm of lightweight mobile agents. However, in contrast to most mobile agent schemes to date, the agents in our approach are very simple; they exploit the parallel processing capability of the network, and, although they always carry state, they carry program code only when necessary

We call our approach to distributed management, which we have developed over the last two years, *pattern-based management* [16][18][19]. It centers around the concept of the navigation pattern, used for controlling and coordinating the actions of light-weight agents. Navigation patterns realize graph traversal algorithms that determine the dissemination of local management operations and the aggregation of the results of these local operations.

As our previous work shows, the approach of pattern-based management systems has interesting implications. First, management programs can be formally analyzed with respect to performance and scalability. The analysis of a management program is based on the analysis of the graph traversal algorithm of its underlying pattern. Second, navigation patterns allow the separation of the semantics of a management operation from the distributed programming aspects of the operation. From a software engineering perspective, this separation allows us to design generic patterns that can be combined with specific semantics to implement a particular management operation. A pattern, once designed, can be reused in the implementation of many management tasks. Conversely, a specific management task can potentially be built from a choice of patterns, which enables us to build management operations with different performance profiles. Ultimately, this approach frees an application programmer from developing distributed algorithms, allowing him/her to focus on the management task at hand, by selecting a navigation pattern from a catalogue that captures the requirements for that task.

Third, as our work on robust patterns indicates, the reaction to network faults, which can be complex to understand and handle, can be programmed into a pattern, thereby eliminating the need for the management application programmer to deal with faults. Finally, the degree of code mobility can be controlled in a fine-grained manner, since the execution of a management operation in a network involves distributing only those parts of the program that are not already resident in the network nodes. In other words, for a management program that is frequently executed, only the states of the distributed computation need to be exchanged between network nodes, not the code, which is locally available.

From the perspective of scalability, a pattern-based management system can eliminate bottlenecks associated with centralized processing of management data by distributing the load to network nodes via an appropriate navigation pattern. For example, the *Echo pattern* is particularly efficient at distributing and aggregating data over large networks [18]. By using an Echo pattern as a means for distributing computation to network nodes, highly scalable management programs can be implemented in compact form.

In order to support the development and study of pattern-based management programs, we have developed a PC-based discrete-event simulator, called SIMPSON [20]. SIMPSON is a C++ application that runs under Microsoft Windows (Win98, NT, 2000, XP) and is capable of simulating a large pattern-based management system of up to 60,000 nodes. Management programs on SIMPSON are written in C++ and compiled into dynamic libraries that are loaded on the fly for simulation. SIMPSON's interactive features allow the dynamics of a pattern to be visualized and recorded when its associated management program is executed. Performance data, such as completion time and volume of management traffic, can also be collected and analyzed.

In this paper, we describe the design and implementation of a pattern-based management system, called *Weaver*, on a network of commodity routers. Our design requires *neither modifications to the routers nor any special features on them*. The paper articulates and supports our belief that it is possible to build flexible and scalable network management systems at moderate cost, by using our paradigm.

This paper relates to previous publications on pattern-based management as follows. In [16], we introduced the idea of a management pattern and outlined an architecture supporting a pattern-based management system. We further reported on an explorative prototype, built to validate the very concept of management patterns. That prototype was based on Voyager, a commercial mobile agent platform, and was written in Java. The system presented in this paper is completely new and has a very different design focus, which is to demonstrate the feasibility of an efficient and scalable platform. In [18], we presented the Echo pattern and analyzed its performance with focus on large networks. In [19], we discussed a possible software design for management patterns and introduced SIMPSON, the simulator we had developed to test the functionality and estimate the performance of pattern programs.

In section 2 of this paper, we present the architecture of Weaver and discuss its operation as well as pertinent design issues. In section 3 we benchmark the performance of Weaver and develop a performance model for analyzing its behavior. In section 4 we use the performance model and the results from the previous section to evaluate its scalability through simulation. Finally, we conclude with a discussion of lessons learnt and future work. An appendix summarizes the aspects on management patterns needed to understand this paper, relieving the reader from accessing [16][18] and [19].

2. THE WEAVER PATTERN-BASED MANAGEMENT SYSTEM

Figure 1 shows the architecture of a pattern-based management system, first described in [16]. The gray nodes represent physical routers, while the white nodes,

attached to them, represent the execution environments in which management programs run. The combination of a physical router and its associated logical execution environment constitute a logical network node. The execution of a pattern-based management program begins, when it is launched by a network management station onto the execution environment of the start node. When the program has completed executing on the node, its pattern determines the subsequent node (or nodes), on which the program must execute next. If that node already contains a copy of the program, only the program's state is transferred to it. Otherwise, both code and state are transferred. Alternatively, a node can download the program code directly from a secure repository called a code server.

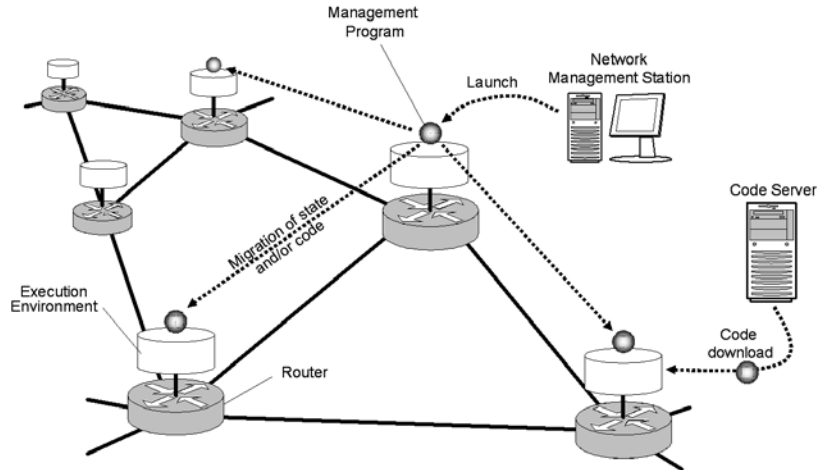


Figure 1. Architecture of a pattern-based management system

A pattern-based management system can be realized in a number of ways, for example, using a general purpose framework, such as Java, or an active networking toolkit, such as ANTS [31]. It can run internally on the processor of a router or externally on a device attached to the router, etc. However, realizing such a system at low cost on commodity routers restricts the design space and thus poses a significant challenge.

2.1 Weaver design aspects

Our most important design goals for Weaver were, first, to realize an *efficient* implementation of a pattern-based management system. By efficient, we mean that management programs complete execution quickly, even in large networks. Second, we wanted to realize a system that works with virtually all *commodity routers*.

In our system design, each router is managed by a dedicated active node that hosts the execution environment needed for running pattern programs. Such an active node, called a Weaver Active Node (WAN), is an internet-enabled, single-board computer, equipped with an Intel StrongARM 1110 microprocessor, 32MB of SDRAM, and a 10Mbps Ethernet interface. The hardware of the WAN is commercially available at a cost of an average PDA, in the form of an aluminum

cube of 3 inches per side. To provide sufficient local storage, we attached a 1GB IBM Microdrive to the onboard compact flash slot. Each WAN runs a modified distribution of the Linux kernel (version 2.4.9), as well as an Apache web server, which is used to implement the WAN's management interface.

Code mobility was a further design issue in the development of our platform. Specifically, we had (1) to choose between transferring program code in either binary or source form, and (2) to decide, if code is to be transported by patterns or downloaded from code servers. The decisions made in addressing these questions have implications on the system's performance and its vulnerability to attacks.

For instance, if a program is to be transferred in source form, then a time-consuming compilation process must be invoked, before it can be executed. In addition, every node must be equipped with sufficient disk space to store the compiler, the linker, and the header files. The advantage of transferring source code is that source programs are significantly smaller than compiled programs.

On the second question, code servers can become bottlenecks, if the network becomes very large. Increasing the number of such servers introduces other problems, because keeping the code on all repositories up-to-date and consistent can be expensive.

In the current design of Weaver, programs are transferred in binary form by the pattern itself. We made this choice for performance reasons, as our tests indicate that simple management programs (for instance, programs based on type 1 to type 4 patterns, see appendix) take 10,000 times longer to compile than to run. Furthermore, compiled management programs in Weaver are generally less than 10 times larger than their source.

Figure 2 shows the software architecture of a WAN with its primary components. The first, the Active Node Manager (ANM), comprises an Apache SSL-enabled web server and a set of server-side PHP scripts [1]. The main functionality of the ANM is to offer a web interface to the management station for configuring and operating the node. The second component, the Active Node Daemon (AND), is a C++ application running as a background process. It implements the execution environment, which runs the pattern-based management programs on the node. In addition, there are several repositories (drawn as cylinders) with state information. For example, the node state repository holds the operational state of the node (such as the numbers and parameters of executing management programs), while the binaries repository serves as a cache of ready-to-run patterns and aggregators. When a pattern migrates to another node, it can leave local state variables in the local program state repository. Finally, the result repository provides for persistent storage of the results returned from a management operation.

2.2 Executing a pattern program on Weaver

In order to start a management program on Weaver, the management station downloads the source code of the pattern and aggregator, as well as the run-time parameters, via http onto a WAN, which will be the start node of this operation. The WAN's ANM then saves the code into its source repository and relays the source file names and the parameters to the preprocessor module through a local socket (Figure 2).

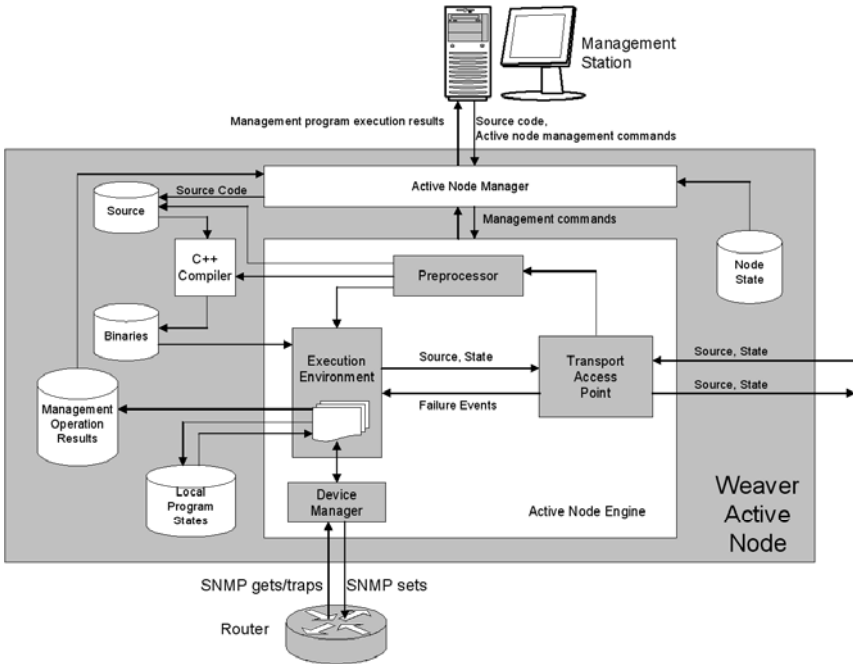


Figure 2. Software architecture of a Weaver Active Node (WAN)

The preprocessor module invokes the compiler to process the program source, computes the MD5 checksum of the resulting binary, and invokes the execution environment to run the program. If the compiler encounters an error, the execution is aborted, and an error code is returned to the management station via the ANM. If the program binaries are already in the WAN's repository, the preprocessor invokes the execution environment directly, passing to it the filenames and paths of the compiled binaries.

The execution environment dynamically loads the program and instantiates the pattern and the aggregator objects. It also generates a system-wide unique cookie, which associates the distributed state of the program with its current execution. Finally, it relinquishes control to the program, passing to it the arguments as specified by the management station.

A program accesses the management interface of the attached router through the WAN's device manager. In addition to the specific access protocol, the device manager also implements low-level monitoring procedures, such as heartbeats, to detect failures in the attached device. In principle, a WAN may include multiple device managers, one for each access protocol supported by a router. Our current prototype, though, has only a single device manager for SNMP.

When the management program has completed its execution on a node, it returns control back to the execution environment, along with a list of node addresses, to which to migrate next. After that, the execution environment stores the local program variables in the local state repository, serializes the mobile state variables, and passes them to the transport access point, along with the list of node addresses and the cookie.

The main function of the transport access point is to securely transfer program code and states between adjacent nodes. Whenever a WAN is initialized, it connects to its neighboring WANs by establishing secure channels.

When a transport access point receives a request to send program code and (mobile) state to a neighboring WAN, it first checks, whether the destination already has a copy of the program. If so, only the state of the program and the cookie are sent. Otherwise, the program code is sent, as well.

Every transport access point keeps a record of the nodes, to which it has sent programs, by saving their MD5 checksums. If no record exists for a particular node and a program checksum, then the program code is sent to the neighboring node, despite the chance that the neighbor might actually have a copy of the program. While this scheme incurs a (usually small) overhead, it is simple and requires no handshake.

When program code is sent, the receiving transport access point saves it into the local binary store, if necessary, and invokes the execution environment. Using the cookie, the execution environment determines, if the node has participated in the current execution of the program. If so, it checks for the program's local state variables, before passing control to the pattern and aggregator objects. Otherwise, new instances of the pattern and aggregator objects are created prior to program execution.

3. BENCHMARKING WEAVER

In this section we give a performance model for pattern-based management programs that are executed on the WAN architecture described in the previous section. The metric of interest in our model is the *execution time* of a management operation. It is measured as the time period from when a program is launched on a start node to when the results are returned to the management station.

We have conducted two series of experiments to obtain a delay profile for Weaver management programs. Table 1 and 2 show results from these experiments. The first series focuses on measuring the delay incurred by a management program based on the type 1 pattern, which models a simple polling operation, where control passes from a node to one of its neighbors before returning (see appendix). Other types of patterns (e.g. the type 2 and type 4 pattern, see appendix) can be expressed as serial compositions of type 1 patterns. In a similar manner, the type 3 pattern (see appendix) constitutes the basic building block for patterns, in which control is passed to neighboring nodes in parallel. The Echo pattern (see appendix), for instance, can be built from a type 3 pattern.

For accurate measurements, the experiments have been carried out on an isolated testbed of four Cisco 2621 routers, which are interconnected via a Cisco Catalyst 2900 fast Ethernet switch. Each router is equipped with two fast Ethernet ports, one of which is connected directly to a WAN. A 1.13 GHz DELL Inspiron 8100 notebook serves as the management station. Static routes have been set up from each router to the fast Ethernet switch, so that all nodes are able to communicate with each other.

In order to understand the delay profile of a management program that is based on a type 1 pattern, we decompose the execution of the program into a series of phases, listed in Table 1.

As we are only interested in measuring the delay from the point of view of the management program, the first phase begins when control is passed to the management program (T1). We call this the execution phase. When the program completes its execution, the serialization phase (T2) is invoked, in which the program's mobile state is serialized and then, during the dispatch phase (T3), sent to the remote WAN. The receiving phase (T4) begins, when the mobile state has been received on the remote node. Depending on whether the management program has been executed on this node before, the next phase can be either the loading phase (T5) or the instantiation phase (T6).

The loading phase occurs the first time a management program is executed on the node. Typical tasks performed include invoking the dynamic linker to load the program code as a shared library and instantiating the pattern and aggregator objects. Also, the program code, if received during T4, is saved in this phase. If the management program has already been loaded because of a previous execution, only the instantiation of the objects are performed. This is referred to as the instantiation phase.

If the program is still active on a node (i.e., the pattern will traverse the node again during its current execution), the pattern and aggregator objects are not deleted when the program migrates to another node. In this case, only a lookup will be needed to return their object references. We refer to this phase as the resolving phase (T8). Finally, the de-serialization phase (T7) creates (or recreates) the mobile state variables in the program's address space.

Table 1. Overhead incurred by each phase of execution of the type 1 pattern

	Duration in ms	Performed by Module
Execution (T1)	1.57 (σ = 0.48)	Execution Environment
Serialization (T2)	3.46 (σ = 0.71)	Execution Environment
Dispatch (T3)	1.67 (σ = 0.49)	Transport Access Point
Receiving (T4)	0.62 (σ = 0.30)	Transport Access Point
Loading (T5)	23.42 (σ = 0.70)	Execution Environment
Instantiation (T6)	0.77 (σ = 0.015)	Execution Environment
De-serialization (T7)	2.04 (σ = 0.49)	Execution Environment
Resolving (T8)	0.15 (σ = 0.001)	Execution Environment
Communications Delay (T_C)	4.04 (σ = 0.10)	---

Table 1 gives the mean and standard deviation of the delay for each of the phases (T1 through T8 and T_C), as measured over 40 runs. The communication delay on the last row of the table includes transmission delay, propagation delay and operating system overhead. The size of the mobile state is 207 bytes. The pattern program contains the minimal code necessary to implement the type 1 pattern and does not perform any other computations. The aggregator program contains only empty functions.

Based on the above discussion, we can derive the (average) completion time of a type 1 pattern as:

$$T_{type1} = 3T1 + 2(T2 + T3 + T4 + T7) + T6 + T8 + T_C$$

For a more detailed explanation of the above formula, see [21]. When the pattern is executed for the first time, an additional delay of T5-T6 incurs, because the execution environment needs to invoke the dynamic linker. Also, the estimate given

by the above equation does not take into account the situation when the node daemon is swapped out by the operating system. Such instances appear rarely during our measurements, because of the small testbed and the light system load.

Following the above approach, we can derive similar expressions for the average completion times of management programs based on the type 2, 3, and 4 patterns. (See [21] for more details).

Table 2 compares the estimated completion time (based on the above formula and table 1) with the actual measurements on the testbed for all 4 basic pattern types. As can be seen, the estimations lie below the measured delays in all cases, with a margin of error between 8.3% and 10%.

Table 2. Comparison of estimated vs. actual measurements for the four basic patterns

	Average completion time (estimated)	Average completion time (measured on testbed)
Type 1	25.2 ms	27.6 ms
Type 2	72.6 ms	78.4 ms
Type 3	44.3 ms	47.6 ms
Type 4	49.5 ms	55.0 ms

4. EVALUATING THE SCALABILITY OF WEAVER

In this section, we investigate the scalability of the Weaver architecture when the network to be managed becomes large. Specifically, we estimate the completion times of pattern-based management programs on large networks using SIMPSON and the delay profiles shown in the previous sections. For comparison purposes, we also estimate the time of the same operation executed on a centralized SNMP-based management system, polling nodes serially or in parallel.

For simplicity, we assume the network topology to be a full b-ary tree with height h. Each node in the network is a router with b+1 ports, one of which is connected to a WAN. This way, each WAN manages exactly one router. We assume that the latency between two adjacent routers is identical to that experienced in our testbed (i.e. $T_{C1}=T_{C2}=0.5T_C=2.022$ms). We also assume that routes taken by packets are symmetrical; that is, PDUs of an SNMP request take the same path, from the manager to the managed device and vice versa. We assume that the management station is attached to the root of the tree and that all management program executions use the root as the start node. Finally, we choose the management task to be an operation that computes the average value of a specific MIB variable across all nodes in the network.

Given the above topology, the total number of nodes in the network, N, is therefore given by

$$N = \frac{n^{h+1} - 1}{n - 1}$$

The most scalable manner to implement the task in a centralized management scheme is to compute the network-wide average value of the desired variable incrementally from the values obtained from each node that is polled (via SNMP GET). If the polling is performed serially on *N* nodes (i.e., each GET operation must complete before the next GET is initiated), and if we neglect the time needed for the simple averaging computation, the total completion time is given by:

$$T_{centralized\_S} = (T_C + T_S) + (2T_C + T_S)n + ... + ((h+1)T_C + T_S)b^h$$

which evaluates to:

$$T_{centralized\_S} = \frac{(T_C(b+1) + T_S)b^{h+2} - (T_C(h+2) + T_S)b^{h+1} + T_C + T_S(1-b)}{(b-1)^2}$$

where T_S is the time required by the SNMP agent on a node to process a request. Our measurements on the Cisco 2621 routers puts this to be approximately 1.9 ms. On the other hand, if the polling is performed in parallel (i.e., the system does not wait for the completion of a GET before polling the next node) and nodes farther away are polled first before nearer nodes, the total completion time is given by:

$$T_{centralized\_P} = 2hT_C + (b^h - 1)T_P$$

where T_p is the polling interval between nodes. Our measurements on the WANs indicate that this is approximately 1.5ms.

In the case of a pattern-based management solution, we employ the Echo pattern described in section 2 to accomplish the same task.

We compare the performance of Weaver against centralized management using the serial polling scheme as a common yardstick. Specifically, we define the scalability measure S to be the ratio between the average completion time of a management task using the serial polling scheme and the scheme underlying the specific management task (such as parallel polling or Weaver/Echo). Figure 3 plots S against h, the number of levels in the tree network, (a) for parallel polling (dashed lines) and (b) Weaver/Echo (solid lines), for the two cases where b, the number of children of each node, is 2 and 6, respectively.

From the plot it is evident that parallel polling always outperforms serial polling, since its scalability measure S never falls below 1. Furthermore, for networks of small to moderate size (i.e., b=2, h<7 and also b=6, h<3), it also outperforms Weaver/Echo because of its lower overhead. However, for large networks, i.e., (b=6, h>4) Weaver/Echo yields completion times that are several orders of magnitude lower than the other schemes.

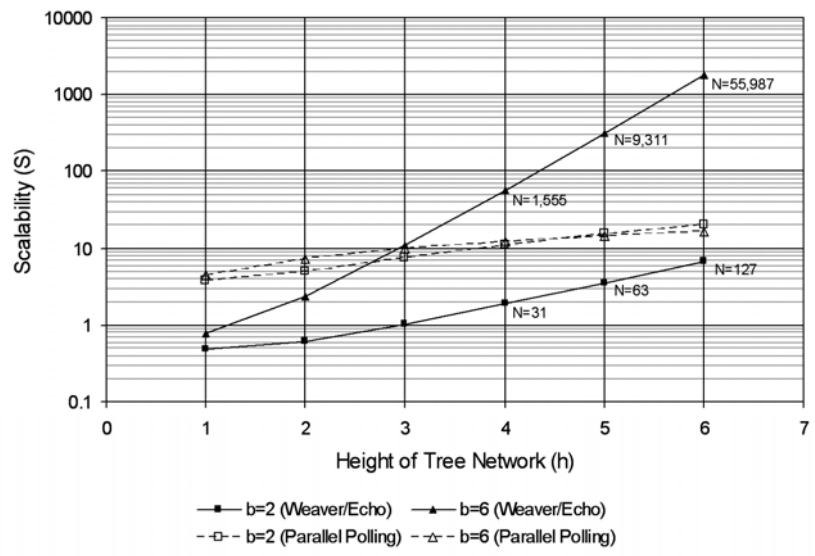


Figure 3. Scalability (*S*) versus Height (*h*) for parallel polling and Weaver/Echo. *N* indicates the number of network nodes.

5. DISCUSSION AND CONCLUSIONS

In this paper, we described a possible realization of the pattern-based management paradigm, using a network of low-cost, single-board computers that are attached to commercial routers. Our goal was to engineer a system that (a) would be lightweight, fast, and scalable, and (b) would be deployable on existing networks. The first goal dissuaded us from using a general-purpose mobile agent platform, which, while probably cutting down the development effort, would have provided unneeded flexibility at the cost of performance. The second goal necessitated the design of an execution environment external to a router, since the current generation of commodity routers does not permit the execution of user-supplied code internally. We chose a low-cost solution in terms of both hardware costs and space needed for the management devices inside the network. The choice of SNMP for router access also arose from our desire for a widely deployable solution.

Since every testbed limits the performance predictions of a system for different configurations, sizes, etc., we developed a performance model for a pattern-based operation on Weaver and used the testbed to instantiate and validate the model. Based on the measurements we took so far, we believe the model given in Section 3 to be basically sound. However, the accuracy of +/-10% for the predicted execution times of the type 1 to 4 patterns on our testbed tells us that our model must be refined. We are currently investigating, why the current model seems to

systematically underestimate execution times, and why a large variance in certain sub operations occurs.

In Section 4, we gave evidence that pattern-based management operations on a large-scale Weaver system are likely to be very scalable. To prevent misunderstanding, we add two comments here. First, we measure scalability by comparing an Echo-based operation to a centralized operation. This means that we use the centralized management solution as a point of reference. Specifically, we are not implying that traditional or current solutions to managing very large networks are centralized in this manner. Second, one could probably come up with a traditional management system that is built on a hierarchy of SNMP agents and would exhibit similar scaling properties as shown in Figure 3. Note though that, while scalability is a key design goal of Weaver, its underlying paradigm of pattern-based management potentially has many advantages over traditional management systems. For instance, the communication and computation structure of a management operation is dynamically constructed by the pattern during its execution, which makes patterns independent of network topology and network size. Also, the discussion in Section 4 is based on the Echo pattern, which exhibits a specific hierarchical communication structure. Other patterns establish different communication structures during execution.

As with any system that includes mobile code, there is a danger of unsafe or malicious code being introduced. In its current implementation, Weaver addresses this issue as follows. First, all communication between the management station and a WAN occurs through an SSL-enabled web interface. This reduces the risk of unauthorized access and protects against masquerade attacks. Second, the compiled code of a management program is only executed within the context of a separate process, with restricted rights and resource quotas. This prevents management programs from interfering with one another or from crashing the daemon, should a fatal error occur. Finally, the communication channel between Transport Access Points of peer nodes is implemented using a simple TLS-like protocol [12], thereby preventing a third party from altering the program code or state while the data is in transit. In addition, all Linux services that are not needed to run Weaver have been disabled in our platform. While the above measures introduce security elements into the design of Weaver, a thorough study of this system's vulnerabilities and how to reduce them effectively still needs to be carried out.

There have been many efforts into building efficient management platforms by recent research into active and programmable networks. First, high-performance active network platforms have been used to implement management applications where management traffic is processed (close to) wire speed. Systems implemented with this philosophy typically have low-level, assembly-style instruction sets [14][23][29], optimized for space and speed. Unfortunately, developing management programs on such systems is difficult, due to their low-level nature, and thus limited to applications, such as programmable traffic probes. Furthermore, these platforms are built on customized or special network nodes that require features not available in commodity routers [10][11][17]. In order to apply some of these techniques to the management of traditional IP networks, (PC-based) software routers have to be used [5][14][17][23][25][29]. In these environments, the (operating system) kernel intercepts packets for processing in a management execution environment, usually through an IP option called Router Alert [15].

The second approach suggests that scalability can be achieved through mobile agent platforms that support intelligent preprocessing and data aggregation inside

the network [3][6][22][24]. Many of the systems developed along this line emphasize flexibility over performance. They generally require the support of a heavyweight infrastructure, which is often Java-based [4][6][24].

Our current and planned work in pattern-based management follows several tracks. On the one hand, we are further exploring the potential of pattern-based management, by designing patterns for dynamic construction of network hierarchies and routing schemes, as well as investigating how basic patterns can be combined into complex ones with desirable properties. At the same time, we are accelerating our work on Weaver. We are currently extending our testbed from 4 to 16 nodes, which will help us in better evaluating and refining our performance model. Since Weaver is a decentralized system, initializing it is a non-trivial task, even in a medium-size network. We began working on a (pattern-based) scheme that would automatically configure Weaver on any network topology and dynamically integrate new WANs into the system. Finally, we have begun studying the use of patterns in policy-based management systems for disseminating policies in large networks and for dynamically re-computing policies, when triggered by state changes or network faults.

REFERENCES

1. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. Wilson, R. Lerdorf, A. Zmievski and J. Ahto, PHP Manual, at http://www.php.net/manual/en.
2. M. Baldi, S. Gai and G. Picco, "Exploiting Code Mobility in Decentralized and Flexible Network Management," First International Workshop on Mobile Agents (MA'97), Berlin, Germany, April 1997, pp. 13-26.
3. M. Baldi, G. Picco, "Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications," First International Working Conference on Active Networks (IWAN'99), June/July 1999, Berlin, Germany.
4. C. Baurner and T. Magedanz, "The Grasshopper Mobile Agent Platform: Enabling Short-term Active Broadband Intelligent Network Implementation," First International Working Conference on Active Networks (IWAN'99), June/July 1999, Berlin, Germany.
5. S. Berson, B. Braden and L. Riciulli, "Introduction to ABone", June 15, 2000, available at http://www.isi.edu/abone/DOCUMENTS/ABarch/
6. A. Bieszczad, T. White and B. Pagurek, "Mobile Agents for Network Management," IEEE Communications Surveys, Vol. 1, No. 1, September 1998, pp. 2-9.
7. E. J. H. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs," IEEE Transactions on Software Engineering, Vol. 8, No. 4, pp. 391-401, July 1982.
8. J. Case, M. Fedor, M. Schoffstall and J. Davin, "A Simple Network Management Protocol (SNMP)," RFC 1157, IETF, May 1990.
9. R. Preshun (chair), Activities and Results of the IETF Working Group on Distributed Management (disman), http://www.ietf.org/html.charters/disman-charter.html.
10. D. Decasper and B. Plattner, "Dan: Distributed Code Caching for Active Networks," IEEE INFOCOM'98, San Francisco, California, March/April 1998, pp. 609-616.
11. D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf and B. Plattner, "A Scalable High Performance Active Network Node," IEEE Network, Vol. 13, No. 1, January 1999, pp. 8-19.
12. T. Dierks and C. Allen, "The TLS protocol version 1.0", RFC2246, January 1999.

13. J. Gosling, B. Joy and G. Steele, The Java Language Specification, Addison-Wesley, 1996.
14. M. Hicks, J. Moore, D. Alexander, C. Gunter and S. Nettles, "PLANet: An Active Internetwork," INFOCOM'99, New York, New York, March 1999, pp.1124-1133.
15. D. Katz, "IP Router Alert Option," RFC 2113, IETF, February 1997.
16. R. Kawamura and R. Stadler: "A Middleware Architecture for Active Distributed Management of IP networks," IEEE/IFIP NOMS 2000, Honolulu, Hawaii, April 2000, pp. 291-304.
17. D. Larrabeiti, M. Calderon, A. Azcorra and M. Uruena, "A Practical Approach to Network-Based Processing," 4th International Workshop on Active Middleware Services (AMS'02), Edinburgh, U.K., July 2002.
18. K.-S. Lim and R. Stadler, "A Navigation Pattern for Scalable Internet Management," 7th IFIP/IEEE IM'01, Seattle, USA, May 2001, pp. 405-420.
19. K.-S. Lim and R. Stadler: "Developing pattern-based management programs," 4th IFIP/IEEE International Conference on Management of Multimedia and Network Services (MMNS'01), Chicago, Illinois, October/November 2001, pp. 345-358.
20. K.S. Lim, "SIMPSON—A simple pattern simulator for large networks," source code and documentation, http://www.comet.columbia.edu/adm/software.htm.
21. K.S. Lim, R. Stadler, "Weaver: Realizing a Scalable Management Paradigm on Commodity Routers," KTH/IMIT/LCN Technical Report Nr. 02-5021, August 2002.
22. A. Liotta, G. Knight, G. Pavlou, "On the Performance and Scalability of Decentralized Monitoring Using Mobile Agents," DSOM '99, Zurich, Switzerland, October 1999.
23. J. Moore, M. Hicks and S. Nettles, "Practical Programmable Packets," IEEE INFOCOM'01, Anchorage, Alaska, April 2001.
24. A. Puliafito and O. Tomarchio, "Using Mobile Agents to Implement Flexible Network Management Strategies," Computer Communications Journal, Vol. 23, No. 8, April 2000.
25. D. Raz and Y. Shavitt, "An Active Network Approach for Efficient Network Management," First International Working Conference on Active Networks (IWAN'99), June/July 1999, Berlin, Germany, pp. 220-231.
26. M. Rose, The Simple Book. New Jersey: Prentice Hall, 1994.
27. E. Rosen, A. Viswanathan and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, IETF, March 1998.
28. A. Segall, "Distributed Network Protocols", IEEE Transactions on Information Theory, IT-29, pp. 23-35, 1983.
29. B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell and C. Partridge, "Smart Packets: Applying Active Networks to Network Management," ACM Transactions on Computer Systems, Vol. 18, No. 1, February 2000, pp. 67-88.
30. G. Tel, Introduction to Distributed Algorithms, Cambridge University Press, 2nd Edition, 2000.
31. D. Wetherall, J. Guttag and D. Tennehouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," First Workshop on Open Architectures and Network Programming (OPENARCH'98), San Franciso, California, March 1998.
32. Y. Yemini, G. Goldszmidt and S. Yemini, "Network Management by Delegation," IM'91, Washington, DC, April 1991, pp. 95-107.
33. E. Gamma, R. Helm, Ralph Johnson, and John Vlissides: Design Patterns—Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

APPENDIX: PATTERN-BASED MANAGEMENT

The pattern-based management paradigm is a distributed management approach based on the use of graph traversal algorithms to control and coordinate the processing and aggregation of management information inside the network. From the perspective of a network manager, the algorithms provide the means to 'diffuse' or spread the computational process over a large set of nodes. A key feature of the approach is its ability to separate this mechanism of diffusion and aggregation from the semantics of the management operation. The paradigm achieves this through the development of two important concepts; the navigation pattern and the aggregator. The former represents the generic graph traversal algorithms that implement distributed control while the latter implements the computations required to realize the task. A pattern-based management program includes both components.

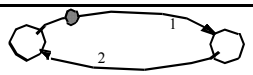
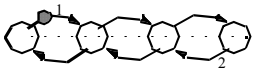
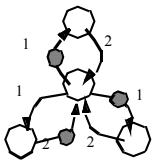
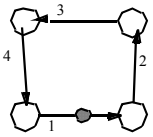
Pattern	Typical Application	Visualization
type 1: node-to-node	1 node control/monitor (e.g. get/set of variables)	
type 2: visit all nodes along a path/flow	1 flow/path control (e.g. traceroute, bottleneck detection, signalling, VPN operation)	
type 3:distribute agents to all nodes in subnet (parallel control)	subnet control, message broadcast (e.g. congestion location detection)	
type 4: visit all nodes in subnet (sequential control)	subnet control (e.g. topology detection)	

Figure 4. Examples of simple navigation patterns

Figure 4 presents the simplest examples of navigation patterns. The most basic pattern is the type 1 pattern, where control moves from one node to another and returns after triggering an operation. The manager-agent interaction is an example of this pattern. A type 2 pattern represents the scenario where control moves along a path in the network, triggers operations on the network nodes of this path, and returns to the originator node along the same path. A possible application of this pattern is resource reservation for a virtual path or an MPLS tunnel [27]. In a type 3 pattern, control migrates in parallel to neighboring nodes, triggers operations on these nodes, and returns with result variables. This pattern can be understood as a parallel version of the type 1 pattern. Finally, in a type 4 pattern, control moves along a circular path in the network. As these examples illustrate, navigation patterns can be defined independently of the management tasks performed in an operation.

For further clarification, we briefly discuss the *Echo pattern*, first introduced in [18]. It is an extension of the basic type 3 pattern and is based on a class of distributed graph traversal algorithms known as wave algorithms [7][28][30]. The behavior of the Echo pattern can be described as follows. The pattern starts out from a single start node, migrating to all its neighbors for further execution. This forward migration of the pattern from a node to each neighbor is called an explorer. An explorer, arriving on a node for the first time, marks the node as 'visited' and generates an explorer for each neighbor, except for the one from which it arrived (which is called its parent). Explorers arriving on a node that has been marked as 'visited', terminate at that node (i.e., they do create more explorers). If the node has no neighbors other than its parent, the program returns to its parent node. This return of the pattern from a node to its parent is called an echo. When a node has received an echo from each of its neighbors, it returns an echo to its parent. The Echo pattern terminates, when the start node has received an echo from each of its neighbors. Figure 5 shows the Echo pattern in pseudo code. It is a refined version of the code given in [19].

```
var visited_i : boolean                    init : false;
    G_i       : set of integers            init : neighbors();
    parent_i  : integer                    init : -1;

1   Echo(inmsg: bytes, from: integer) {

2      G_i := G_i - from;
3      If visited_i = false {
4         parent_i := from;
5         visited_i := true;
6         OnInitiate(inmsg, outmsg);
7         if G_i != empty {
8            dispatch(G_i, outmsg, i);
          }
       } else {
9         OnAggregate(inmsg);
       }
10     if G_i = empty {
11       OnComplete(outmsg);
12       if parent_i >= 0
13          dispatch(parent_i, outmsg, i);
14       else OnTerminate(inmsg);
       }
    }
```

Figure 5. Echo pattern in pseudo code

Note that the concept of a navigation pattern is very different from that of a design pattern as used in software engineering [33]. While a navigation pattern captures the flow of control of executing a distributed operation, a design pattern describes communicating objects and classes for the purpose of solving a general design problem in a particular context.