Predictable, Lightweight Management Agents*

Jonathan T. Moore[1], Jessica Kornblum Moore[1], and Scott Nettles[2]

[1] Computer and Information Science Department
University of Pennsylvania
{jonm,jkornblu}@dsl.cis.upenn.edu
[2] Electrical and Computer Engineering Department
The University of Texas at Austin
nettles@ece.utexas.edu

Abstract. In this paper we present an *active*, or programmable, packet system, SNAP (Safe and Nimble Active Packets) that can be used to provide flexible, lightweight network management with predictable resource usage. SNAP is efficient enough to support both centralized polling and mobile agent approaches; furthermore, its resource predictability prevents malicious or erroneous agents from running amok in the network.

1 Introduction

Mobile agents have often been proposed as a solution for network management [5]. By moving computation away from a central network operations center (NOC) via mobile code, these schemes use distribution to achieve scalability [4, 5, 6, 17, 20]. Furthermore, having a programmable management platform in theory makes it easier to customize management applications to particular networks or to quickly take new management actions (for example, a response to a new virus or denial-of-service attack).

If mobile agents are such a panacea for distributed management, why do they not appear more frequently in actual management systems? We believe there are two reasons: existing systems are too heavyweight, and network managers fear losing control over runaway agents.

Several researchers have explicitly considered how the performance of agents compares to SNMP and similar approaches. Rubinstein and Duarte [20], for example, simulate the performance of an agent designed to gather values from several nodes and compare it to SNMP on a simple topology. Their simulations show that for several metrics when a large number of nodes are visited (over 200), their agent outperforms SNMP. However, for a small number of nodes, SNMP is superior; because their agent requires about 5,000 bytes to be transmitted, the mobile agent approach can overcome its constants of proportionality only for the largest of networks. Baldi and Picco [3] develop a set of analytical models concerning the performance of several approaches, including both centralized polling and mobile agents and are thus able to characterize when one

* This work was supported by DARPA under Contract #N66001-96-C-852 and by the NSF under Contracts ANI 00-82386 and ANI 98-13875.

technique will perform better than another. They then examine a specific case study involving SNMP and Java Aglets [13]. Here the Java code is again about 5,000 bytes in size, with a similar impact on the trade-offs compared to SNMP.

A further problem concerns bounding resource usage, which is always an issue where mobile agents are concerned. Indeed, many agent platforms suffer from the problem of runaway agents that are difficult to track down and stop once unleashed [27]. Network operators are extremely hesitant about surrendering control to agents whose behavior is not entirely predictable.

In this paper, we describe the use of an *active* (or, programmable) packet system, SNAP (Safe and Nimble Active Packets) [15], for use in network management. SNAP packets contain code that is executed at SNAP-aware routers, thus adding significant flexibility over IPv4 or simple client-server protocols. The SNAP language has been designed to provide lightweight execution as well as predictable resource usage. As a result, SNAP is an ideal candidate for realizing the mobile agent approach to network management.

We begin in Section 2 with a brief overview of SNAP, stressing its security features and providing a micro-benchmark demonstrating its lightweight implementation. Then in Section 3, we illustrate the detection of a Distributed Denial-of-Service (DDoS) attack using special SNAP monitoring agents. We discuss related work in Section 4 before concluding and suggesting future work in Section 5.

2 SNAP: Predictable, Lightweight Mobile Agents

SNAP is an active packet language designed to address the open problem of providing a flexible programming language with high performance, yet safe (and in particular, safe with respect to system resources), execution. In this subsection, we present merely a high level overview; the reader is referred to [15] for more details.

SNAP is a simple stack-based bytecode language designed to require no unmarshalling (and in most cases very little or no marshalling) and to permit in-place execution. The packet contains code, heap, and stack segments; the stack is the last segment in the packet, allowing us to execute SNAP *in-place* in a network buffer, as the stack can grow and shrink into the available space at the end of the buffer (the heap can also grow down from the end of the buffer if needed). By design, SNAP does not require a garbage collector.

SNAP is indeed a low-level, assembly-style language (see Figures 1 and 2 for examples), making tightly hand-tuned packet programs possible. However, for programming convenience, a compiler [10] exists to translate the higher-level, strongly-typed active packet language PLAN into SNAP.

In its current implementation, SNAP is carried in an IPv4 packet with the Router Alert [11] option. Thus, legacy routers will simply forward a SNAP packet toward its destination. On the other hand, SNAP-aware routers can detect the router alert, check the IP protocol field to see that the packet contains SNAP, and then call the SNAP interpreter. Kernel and user-space implementations exist for Linux systems, and an interpreter has also been built for IBM's PowerNP 4GS3 network processor [12].

A key aspect of SNAP's design is its model of predictable resource usage. In particular, a single packet execution on a node is guaranteed to use time and space proportional to the packet's length. This guarantee is derived from two facts: all SNAP instructions execute in (possibly amortized) constant time and space, and all branches must move forward. Thus, unlike other approaches based upon watchdog timers or memory limits, SNAP routers know that incoming SNAP programs will behave reasonably with respect to resources *without even having to examine them.* Furthermore, this means that SNAP programmers can know that their programs will not be unexpectedly terminated; if they can express their programs in SNAP, those programs are resource safe.

In addition, each packet has an associated *resource bound* (RB) field that is decremented for every network hop, including sending to the current node via a loopback interface. This loopback behavior can be used to emulate backward branches at the cost of 1 RB. Furthermore, a packet must donate some of its RB to any child packets it spawns (conservation of resource bound). As a result, we can examine the initial length of a SNAP program and its initial RB count, and place a strict upper bound on the amount of network resources it or its descendants can ultimately use.

Because SNAP cannot express infinite loops, nor can SNAP packets roam around the network indefinitely, it is not possible to write a stationary daemon-style agent that "settles in" at a given node to monitor it, nor can we write permanently circulating "sentry" agents. Stationary agents are more appropriately designed using existing approaches like AgentX [7] and RMON [25]. Long-running agents that traverse a given circuit repeatedly can be composed from multiple SNAP agents, each of which traverses the circuit a fixed number of times before reporting back into the NOC, which then sends out a new agent. Our example DDoS detection application in Section 3 would be naturally constructed this way. The benefit, of course, is that to control incorrect agents, we can simply stop injecting new agents at the NOC; all outstanding agents will then fairly quickly run out of resources and die.

For added security, access to management functions may be protected by strong authentication, in the style of PLAN [9]. Here, there is some minimal set of "safe" services available to all packets without authentication. A packet can carry cryptographic credentials that can be presented to the node in exchange for an expanded service namespace—access to additional services. This style of *namespace security* allows packets to only pay the costs of cryptographic authentication on an as-needed basis. Although we have not yet implemented this service for SNAP, it would be straightforward: the credentials could be carried in the packet's heap as a byte array and then handed to an "`auth`" service that would side-effect the service namespace for the packet.

```
        forw                         ; move towards destination
        bne done                     ; if returning to source, branch
        push "interfaces.ifNumber.0" ; MIB variable
        calls "snmpget"              ; invoke service
        push 1                       ; push 1 to indicate return
        getsrc                       ; retrieve source address
        forwto                       ; forward back toward source
done:
        demuxi <portnum>             ; deliver payload
```

Fig. 1. SNAP program for the polling micro-benchmark.

2.1 Micro-benchmark

In this section, we support our claim that SNAP can provide a lightweight network monitoring system. Our experiments were performed on a two PCs, called *hera* and *athena*; each machine has a Pentium III (Coppermine) 1 Ghz CPU, 256 MB of RAM, and a SuperMicro Super 370 DE6 motherboard with on-board Intel Speedo3 100 Mbps Ethernet card. The cluster runs RedHat Linux 7.3, with kernel version 2.4.18-5. Both machines are on the same LAN, switched by an Asanté Fast100 Ethernet hub. The measurements we present are the median of 21 trials.

In our first test, we ran the `snmpd` from `ucd-snmp` version 4.2.5 on *hera*, and ran a client program on *athena* to retrieve the `interfaces.ifNumber.0` MIB variable from *hera*. The median latency was 575 μs (the ping latency between the machines accounts for 148 μs of this).

In our second test, we ran our user-space `snapd` on both *hera* and *athena*, and injected the SNAP program[1] shown in Figure 1 from *athena*. The program proceeds to *hera*, queries the MIB variable via the "snmpget" service, and then returns back to *athena*. The median latency for this program was 660 μs, an extra overhead of only 85 μs (15%) over plain `snmpd` above. However, this overhead is exaggerated by the fact that the implementation of the "snmpget" service simply contacts the local `snmpd` via a network socket; this overhead could be avoided in an integrated SNAP+SNMP daemon, such as we discuss in future work (Section 5).

3 Application Example: DDoS Detection

We now present a concrete example using SNAP as a lightweight mobile agent platform for network management. Distributed denial-of-service (DDoS) attacks are an increasing problem, targeting well-known e-commerce and government sites; easy to use tools for carrying out these attacks [1], are becoming widely available. For e-commerce sites, response time for such attacks is critical, as

[1] For an explanation of SNAP semantics, the reader is referred to [15].

serious revenue losses can accrue during down-time, not to mention the impact on customer satisfaction. The first step of a response is detecting the attack in the first place. In this section, we describe a DDoS detection mechanism using SNAP.

To detect a DDoS attack, we can measure the amount of incoming traffic T into our administrative domain. Generally, as a network manager, we will have some traffic threshold T_{alarm}; if incoming traffic exceeds T_{alarm}, we want to sound an alarm and take action. We need to query incoming octet counts on multiple interfaces of multiple nodes. The usual centralized polling approach will quickly overwhelm the NOC with management data as the number of managed nodes grows, so this solution does not scale. The key to scalability lies in being able to distribute the computation of T. Fortunately, this particular computation can be performed incrementally, making it especially well suited for the use of lightweight active packets.

3.1 SNAP Surveyors

Figure 2 shows the SNAP "surveyor" programs that we use to address the scaling problem. This packet program carries a list of nodes to query and visits each in sequence. At each node i it queries the MIB variable `interfaces.ifTable.ifEntry.ifInOctets` for the external interface and keeps a running sum, T_{sum}. Once all nodes have been visited, the surveyor returns to the NOC and reports the current value of T. The algorithmic intuition is the following: with n nodes to manage, a centralized polling approach requires $O(2n)$ network hops (out to each node and back), whereas the surveyor approach requires $O(n)$ hops. Perhaps more importantly, in the centralized approach, all $O(2n)$ hops involve the NOC, whereas in the surveyor approach, only 2 hops involve the NOC. Thus, not only is the network traffic reduced, but it is also distributed.

The surveyor program consists of just 21 instructions, for a total of 84 bytes of code in a SNAP packet. This leaves significant room in the packet for carrying accumulated data and/or addresses of nodes to visit. Even with a maximum transmission unit (MTU) as small as 256 bytes, there are still 128 bytes of room left over in the packet after headers and code (enough to visit 32 nodes, assuming 4 bytes of accumulated data as in the above example). With more realistic autonomous domain MTUs of 1500 bytes, one packet could easily visit over 300 nodes.

3.2 Discussion

The specific example presented here is just one of a class of *distributed threshold detection* problems; Raz and Dilman [18] point out several such problems, including monitoring general network traffic, Web mirror loads, software licenses, bandwidth brokerage, and denial-of-service attacks. In each case, we want to know whether some global network threshold has been exceeded.

Raz and Dilman's approach, efficient reactive monitoring, apportions some "ration" of the global threshold to each monitored node; the node monitors its

```
main:
        forw            ; get to next hop
        bne     athome  ; if homeward flag set, just deliver data

        ;; else, we need to update load sum
        push  "interfaces.ifTable.ifEntry.ifInOctets.5"
        calls "snmpget" ; get current octet count
        add             ; running sum

        ;; any more nodes to visit?
        pull    1       ; get n (number of remaining nodes)
        bez     gohome  ; if out of addrs, go home

        ;; re-arrange stack state in preparation for transit
        pull    2       ; get next node's address
        pull    2       ; get n
        subi    1       ; n--
        store   4       ; put new n over old next hop
        pull    1       ; pull load sum
        store   3       ; put load sum over old n
        push    0       ; still more hops to go; unset homeward flag
        store   2       ; put flag over old load sum
        forwto          ; move on to next hop

gohome:
        push    1       ; set homeward-bound flag
        getsrc          ; find out where home is
        forwto          ; go there

athome:
        getspt          ; get port number for delivery
        demux           ; deliver octet total
```

Fig. 2. SNAP "surveyor" program

own state and, if the ration is used up, triggers an alarm to the NOC, which then issues a global poll. If none of the nodes exceed their ration, then the global usage cannot have exceeded the global threshold.

This approach can be adapted to use SNAP surveyors without requiring extra code to be installed at the monitored nodes (except for the one-time installation of a SNAP interpreter). Furthermore, the surveyor can make use of domain-specific knowledge to shortcut its route: the surveyor may be able to determine, based on its current incremental result, the number of remaining nodes to visit, and an upper bound on the queried value, that the overall threshold will not be exceeded[2].

[2] A centralized poll could also short-circuit its search, but as we noted earlier, this requires on the order of twice as many network hops as the surveyor approach.

4 Related Work

There is a significant body of work concerning mobile agents in network management [2, 4, 6, 16, 24, 17, 19, 21, 22, 26]. Bieszczad *et al.* provide a survey [5] that identifies a number of areas in which mobile agents are applicable to network management, including network modeling, fault management, configuration management, and performance management.

Existing mobile agent systems [4, 16, 24, 17, 19] geared toward network management tend to be based largely upon Java. Unfortunately, there is no good way to bound the resources consumed by a Java agent, as infinite loops can be expressed, yet Hawblitzel *et al.* have shown that it is unsafe to simply terminate runaway threads in the JVM [8].

The IETF ScriptMIB [14] provides an SNMP-based interface for installing and running scripts, although it does not specify a particular script programming language. Furthermore, multiple SNMP round-trips are necessary to set up and invoke a new script, whereas SNAP-based agents are more "light on their feet," being self-contained.

Perhaps the most closely related project to ours is the Smart Packets project from BBN [23]. Indeed, their system closely resembles ours in having a bytecode interpreter for active packets. The main difference from our work is their approach to resource control. Smart Packets rely on instruction counters and memory limits to prevent packets from consuming too many local resources, whereas SNAP can provide the same guarantees via language design. In the end, this impacts programmer convenience: with Smart Packets, it is possible that a packet program may accidentally and unpredictably exceed its resource allotment and be prematurely terminated, whereas SNAP programs will always run to completion (barring other sorts of errors). One other important difference is that Smart Packets cannot direct themselves; they are sent from a source to a destination, and may execute on intermediate nodes, but may not deviate from the original path. As a result, Smart Packets do not offer quite the agility needed for truly mobile agents. Finally, no performance data is available to determine whether the Smart Packets execution environment is lightweight or not.

5 Conclusions and Future Work

We have described an active packet system, SNAP, and have argued that it can be used to add flexible mobile agent capabilities wherever SNMP is already used. We have shown experimentally that SNAP execution overheads are small compared to SNMP: thus SNAP offers the flexibility of mobile agents with the efficiency of standard centralized polling. Furthermore, our language design guarantees that SNAP agents have predictable (and finite) resource usage. Finally, we have presented an example application, using SNAP to detect distributed denial-of-service attacks (DDoS).

The main thrust for future work revolves around providing a generic SNMP service interface for SNAP. Walter Eaves of University College London has already added an SNMP service interface to a user-space SNAP implementation.

In this system, services send appropriate SNMP requests to a separate SNMP daemon on the same host. Willem de Bruijn at the Leiden Institute of Advanced Computer Science is currently merging a SNAP user-space implementation with the `net-snmp` SNMP daemon, creating a single program with a MIB backend and two frontends, one for vanilla SNMP and one for SNAP. The resulting program will be a SNAP-enabled drop-in replacement for a standard `snmpd`.

Acknowledgments

The authors would like to thank Michael Hicks and Aaron Marks for comments on early drafts of this paper, Sotiris Ioannidis for useful discussions about using active network technology for network monitoring, and Walter Eaves for discussions about an SNMP service interface for SNAP and for implementing such an interface for the user-space version of SNAP. We would also like to thank the anonymous reviewers and Rolf Stadler for their useful comments.

References

[1] Denial of service tools. CERT Advisory CA-1999-17, December 1999.

[2] M. Baldi, S. Gai, and G. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Proceedings of the First International Workshop on Mobile Agents*, April 1997.

[3] Mario Baldi and Gian Pietro Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 1998.

[4] C. Baumer and T. Magedanz. The Grasshopper Mobile Agent Platform Enabling Shortterm Active Broadband Intelligent Network Implementation. In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, June/July 1999.

[5] A. Bieszczad, T. White, and B. Pagurek. Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1), September 1998.

[6] M. Breugst and T. Magedanz. Mobile Agents—Enabling Technology for Active Intelligent Network Implementation. *IEEE Network*, 12(3):53–60, May/June 1998.

[7] M. Daniele and B. Wijnen. Agent Extensibility (AgentX) Protocol, Version 1. RFC 2741, IETF, January 2000.

[8] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[9] M. Hicks and A. Keromytis. A Secure PLAN. In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, June/July 1999.

[10] M. Hicks, J. Moore, and S. Nettles. Compiling PLAN to SNAP. In *Proceedings of the IFIP-TC6 Third International Working Conference on Active Networks (IWAN'01)*, pages 134–151, September/October 2001.

[11] D. Katz. IP Router Alert Option. RFC 2113, IETF, February 1997.

[12] A. Kind, R. Pletka, and B. Stiller. The potential of just-in-time compilation in active networks based on network processors. In *Proceedings of the 5th Workshop on Open Architectures and Network Programming (OPENARCH'02)*, pages 79–90, June 2002.

[13] Danny B. Lange. Java aglet application programming interface (J-AAPI) white paper—draft 2, 1997.

[14] D. Levi and J. Schoenwaelder. Definitions of Managed Objects for the Delegation of Management Scripts. RFC 3165, IETF, August 2001.

[15] J. Moore. *Practical Active Packets.* PhD thesis, University of Pennsylvania, 2002.

[16] Objectspace, Inc. Voyager application server 4.0 datasheet, October 2000.

[17] A. Puliafito and O. Tomarchio. Using Mobile Agents to Implement Flexible Network Management Strategies. *Computer Communications Journal*, 23(8), April 2000.

[18] D. Raz and M. Dilman. Efficient Reactive Monitoring. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, April 2001.

[19] D. Raz and Y. Shavitt. An Active Network Approach for Efficient Network Management. In *Proceedings of the First International Working Conference on Active Networks (IWAN'99)*, June/July 1999.

[20] M. Rubinstein and O. Duarte. Evaluating Tradeoffs of Mobile Agents in Network Management. *Networking and Information Systems Journal*, 2(2), 1999.

[21] A. Sahai, C. Morin, and S. Billiart. Intelligent Agents for a Mobile Network Manager. In *Proceedings of the IFIP/IEEE International Conference on Intelligent Networks and Intelligence in Networks (2IN'97)*, September 1997.

[22] C. Schramm, A. Bieszczad, and B. Pagurek. Application-Oriented Network Modeling with Mobile Agents. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98)*, February 1998.

[23] 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*, 18(1), February 2000.

[24] P. Simões, L. Moura Silva, and F. Boavida-Fernandes. Integrating SNMP into a Mobile Agent Infrastructure. In *Proceedings of the Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, October 1999.

[25] William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2.* Addison-Wesley, third edition, 1999.

[26] G. Susilo, A. Bieszczad, and B. Pagurek. Infrastructure for Advanced Network Management based on Mobile Code. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS'98)*, February 1998.

[27] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3), September 1997.