**DIMACS Technical Report 99-25**

# An Active Network Approach to Efficient Network Management

by

*Danny Raz*[1] *Yuval Shavitt*[2]
raz@research.bell-labs.com, shavitt@ieee.org
Bell Laboratories, Lucent Technologies
101 Crawfords Corner Road
Holmdel, NJ 07733-3030

[1]DIMACS Member
[2]DIMACS Member

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

## ABSTRACT

Active networks is a framework where network elements, primarily routers and switches, are programmable. Programs that are injected into the network are executed by the network elements to achieve higher flexibility and to present new capabilities.

This work describes a novel active network architecture which primarily addresses the management challenges of modern complex networks. Its primary component is an active engine that is attached to any IP router to form an active node. The active engine we designed and implemented executes programs that arrive from the network and monitors and controls the router actions. The design is based on standards (Java, SNMP, ANEP over UDP), and can be easily deployed in todays IP networks.

The contribution of this paper is the introduction of novel architectural features such as: isolation of the active mechanism, the session concept, the ability of active session to control non-active packets, and blind addressing. Implementing these ideas, we built a system that enables the safe execution and rapid deployment of new distributed management applications in the network layer. This system can be gradually integrated in todays IP network, and allows smooth migration from IP to active networking.

# 1 Introduction

The emerging next generation of routers exhibit both high performance and rich functionality, such as support for virtual private networks and QoS [KS98, KLS98]. To achieve this, per flow queueing and fast IP filtering are incorporated into the router's hardware [KLS98, LS98]. The management of a network comprised of such devices and efficient use of the new functionality introduces new challenges.

Active networks is a framework where network elements, primarily routers and switches, are programmable [TSS+97]. Programs that are injected into the network are executed by the network elements to achieve higher flexibility for networking functions, such as routing, and to present new capabilities for higher layer functions by allowing data fusion in the network layer.

This work suggests a novel active network architecture which primarily addresses the network management challenges. At its center is an active engine that execute programs that are received through the network. The active engine is attached to an IP router, and together they form an active node. We introduce the notion of a *session*, which generalizes the soft state mechanism that appears in many active network architectures [KMH+98, Nyg98]. This enables long lasting applications that are typical to network management, to reside in the active node. For a scalable distributed operation we introduce a new addressing mode, *blind* addressing, in addition to the *explicit* mode available in IP. We explicitly allow sessions to access the router management information base (MIB) using SNMP. Overall, we introduce a modular solution that enables an easy deployment in the current IP networks.

This paper describes a working network prototype. The active engine is written mostly in C, and is demonstrated on a network comprised of software routers running on FreeBSD PCs and a Cisco 2514 router with a PC as an adjunct active engine. Currently we support Java as the programming language for the active code, but the architecture is built with handles to allow the use of other languages.

We put an emphasis on standard APIs and tools. The Java code and the data are encapsulated using the standard ANEP headers [ABG+97] over UDP. The engine communicates with the router using SNMP which enables it to monitor and control the router's operation.

The session is an important concept we introduce. It serves both as a mechanism that preserves soft state, and as a rendezvous point for data fusion. Programs and data packets are identified by a unique session id. Once a session starts execution at a node, the code and data are kept until they age. In view of the continuous duration of many management applications, we allow a session to replace its code without losing its data.

Our approach can handle the entire range of active networking, from capsules to programmable switches. Capsule applications carry their code and terminate after execution. Programmable switches are implemented by "well-known" session ids that may receive data and act on it. We also allow authorized sessions to intercept non-active packets and manipulate their data, change their routing, drop them, etc. Such authorized sessions can also change the MIB variables in the router using SNMP. Another feature of the architecture is the ability of non-active packets to request a special service, e.g., routing, that is implemented by a transparent resident session that is mapped to this service at the router.

Network management applications are traditionally centralized around some manager. The manager queries the managed objects, builds a view of the network, and sends alerts if a problem is detected. The manager can also try and take corrective actions by sending configuration commands to network entities. The recent trend in network management development environment architectures is to rely on multiple levels of abstraction. As a result, the cost of management is obscured from the application programmer, and thus neglected. If this trend will continue, management may consume increasing portions of network resources (bandwidth, buffer space).

Our approach to network management calls for the distribution of the management tasks in the network. It enables shorter control loops, deletes long haul dissemination of redundant and unimportant information ("I'm OK" messages), and facilitates new exciting applications. The framework forces the programmer to be aware of efficiency issues and thus will result in more efficient code not only due to its intrinsic capabilities to do so, but also due to the human change of focus. Other agent based approaches [KLO97, KZ97] that enable distributed computing rely heavily on bandwidth blind approaches such as Java RMI and thus do not result in efficient usage of network resources.

The rest of the paper is organized as follows. Section 2 gives a short overview of the system we built. A detailed comparison between our design and existing systems appears in Section 3. Section 4 describe the system architecture and flow of information. Section 5 demonstrates the architecture capabilities with two detailed implementation examples. We discuss future work and give our concluding remarks in section 6.

# 2 System Overview

In this section, we provide a general overview of the prototype system we built in Bell Labs.

Logically, an active node in our system is comprised of two entities: an IP router, and an adjunct active engine (AE). The IP router component performs the IP forwarding, basic routing, and filtering that are part of the functions performed by today's commercially off-the-shelf (COTS) IP routers. The active engine is an environment, where user written programs can be executed with close interaction to the router data and control variables[1].

Physically, the IP router and the active engine may either reside on different machines or coreside inside the same box. This structure enables us to upgrade any COTS IP router to an active router simply by adding an adjunct active engine.

The separation protects non-active traffic from the effects of an erroneous operation of the active part of the network, and inflict minimal additional delay on non-active traffic. It also makes gradual deployment of active nodes in current networks an easy task.

The IP router performs the IP forwarding and basic routing. It also performs IP filtering that enables the diversion of active packets (or other packets specified by an authorized active session) to the active engine.

The active engine is an environment in which code contained in active packets can be executed. This code can specify how code and data related to a specific task should be handled. A logical distributed task is identified by a globally unique number called a session id. When code associated with a non-existing session arrives, it is executed and creates a process that handles all the packets of that session. Such a process can either handle only a single data packet and terminate (capsule), or it can exist in the AE for a long period of time handling many data packets as required by many network management applications.

To perform network layer tasks, sessions must have access to the router's network layer data, such as, topological data (neighbor ids), routing data, performance data (packets dropped, packets forwarded, CPU usage etc.) and more. We use SNMP as the interface between the router and the AE. Standard SNMP agents exist in all routers and enable a read/write interface to a standard management information base (MIB).

In order to perform distributed tasks, an active node must have means to communicate with other active nodes. Relying on the fact that the full topology information is available for any specific node does not scale. To tackle this problem we support a topology-blind addressing mode that enables a node to send a packet to the nearest active node in a certain direction. This mode is useful for topology learning, robust operation, support of heterogeneous (active and non-active) environments, etc.. We also support the explicit addressing mode in which a packet is sent to a specific active node.

Overall, we built a system that enables the safe execution and rapid deployment of new distributed management applications in the network layer. This system can be gradually integrated in todays IP network, and allows smooth migration from IP to active networking. To facilitate this, we introduce novel architectural features such as: isolation of the active mechanism, the session concept, the ability of an active session to control non-active packets, and blind addressing.

[1] The active engine can be perceived as a an execution environment in the context of [Gro98].

# 3 Related Work

Recently, research in active networking is gaining popularity. Some of the projects that currently contribute to research in this area are: ANTS in MIT [WGT98]; [KMH+98] in the University of Kansas; SwitchWare in U. Penn. [AAH+98]; [BCZ97] in Georgia Tech.; MeGa service in Berkeley [AMK98]; NetScript in Columbia [YdS96]; DAN in Washington University [DP98], and more. In this section we compare our architecture with the menagerie of existing active network architectures and with other agent based approaches.

In our design, we separate the active engine where active code is executed from the router itself. This approach makes deployment easier and poses less threat to the non-active traffic in case the active engine breaks down. A similar approach was very recently reported by Amir et al. [AMK98], however, they are limiting the scope of their active server to the application level and thus limiting its capabilities. Bhattacharjee et al. [BCZ97] also suggested a similar approach but for a very restricted active server that can support only a given set of functions. In most other works, the active part and the non-active part are not well separated.

The use of Java in active networks seems natural, and not surprisingly it is the selection of many active network projects [WGT98, KMH+98, HPB+97], and of many more mobile agent projects [KZ97]. However, other projects selected more specialized languages such as Caml [AAH+98], or developed ad-hoc languages [YdS96, HKM+98]. ANEP [ABG+97] is incorporated into several of the existing projects. It is already used in [KMH+98] and will be used in [WGT98]. We can only speculate whether the current research projects will converge eventually to a unified environment using ANEP as the wire encapsulation. Interestingly, SNMP escaped the notice of all the active network projects but one [YdS96]. We believe the use of SNMP is the most attractive option to integrate active network technology with existing routers.

The session concept we use is a generalization of previous mechanisms that can be found in all the active network projects: state preservation and rendezvous point. Since we use a unique session id, a session can serve as a rendezvous point for data from different programs, or different versions of a program (e.g., unlike ANTS that uses the MD5 of the program itself as an identifier) and can support code upgrade on the fly. Still, we support the 'one-shot' capsule model (see example in section 5) by allowing a program to signal its termination, and trigger a clean-up.

Most agent based systems [KLO97, KZ97] reside in the application layer and thus do not have access to network layer information. A recent first step in addressing the need of agents to interface with network layer information is presented by Zapf et al. [ZHGW99]. They allow their application layer agents to access router information through an intermediate resident application in the router using SNMP interface. Another interesting work was presented by Hjálmtýsson et al. [HB97, HJ97]. They built a system where installed agents can manipulate data streams in a router. Though similar in flavor to this work, Hjálmtýsson's work suggests a new router design, while we emphasize the use of legacy routers.

Many of the modern solutions to the IP network management challenge, use high level distributed file or object environment such as: CORBA, Java ORB, Java RMI, Styx, DCOM, and Directory Enabled Networks (DENs). Often, these solutions hide the cost of communication. As is well put in the last never-to-be-reached chapter of [VD98]:

> "When CORBA is used the wrong way, the implemented applications, although they are functionally complete, can have performance and scalability problems."

We believe that our approach presents a better alternative for the current practice.

# 4 Architecture

## 4.1 Design principles

Targeting specifically the network management domain, the following principles guided our design:

**Generality and simplicity** Building applications should be easy to a large base of programmers. Thus, the system should not be limited to one language, and should support languages that are in general use. The node should also be general enough to support many levels of active networking, from capsules to programmable switches.

**Modularity** We separate the active node to modules with clearly defined API between them. In particular, we chose to separate the forwarding mechanism of a regular router from the operating environment where the packets are executed. We also use, as much as possible, well accepted standards, such as Java, SNMP, and ANEP [ABG+97], as the API in which the modules exchange information.

**Inter-operability and heterogeneity** Most likely, active nodes will co-exist with non-active routers. Furthermore, incremental deployment of active nodes with co-existing routers seems as a natural evolvement path. In such a scenario it is very unlikely to assume that an application running on an active node could explicitly know the addresses of its active neighbors. To this end we support "blind" addressing, in which the active node needs not know the address or the location of other active nodes.

**Long lasting sessions** In many network management applications there is a natural need for an application to reside in a node for a long period of time (for example to do monitoring and billing). This cannot be efficiently implemented with capsules. As we do target the network management domain we specificly designed the system to support such applications. It is also very important for an application to have an easy and standard access to the local information at a node, since in many applications the action taken by the packet depends on this information.

**Cost visibility** Although we wish to abstract most of the technical details in order to simplify the development of applications, we think that the application must be aware of the costs, both in terms of node resources (CPU, memory, etc.), and in terms of global network resources (bandwidth and delay). Therefore, we do not use advanced distributed tools such as CORBA and Java RMI, which in general hide much of the actual cost from the user.

**Safety and security** Non-active traffic should not be affected by the new active ability. Further more, an active application should not be able to affect any other application. The system should support security and robustness at all levels.

The above principles directed us to make the following architectural decisions:

- The active node is composed of a regular router with a *diverter*, which detects and diverts active packets to the main separate component - the Active Engine.
- The Active Engine is a separate entity (which may reside on a different card, or a separate machine), which performs most of the active node's task.

This simple modular structure supports inter-operability, and does not require that the specific address of the next active hop be known. The diverter part is fairly simple, and can be carried out using IP filtering which is supported in the API level by most of the router vendors. This structure also allows an easy incremental deployment in heterogeneous networks. Another advantage of this design is robustness; non-active traffic could not be effected by active traffic. Even if for some reason the active engine stops working the router will still route non-active packets correctly.

The second significant entity in our design is the session. Logically, a session is a distributed task preformed in the network. A session has a unique network id, thus, different programs, on various nodes can belong to the same session. These programs may exchange information using active data packets, and they can distribute (and/or update) their code by sending active programs. This notion of a session is general enough to support both long lasting processes, and short term capsules. The fine details of the design are described in the following subsections.

## 4.2 Detailed Design

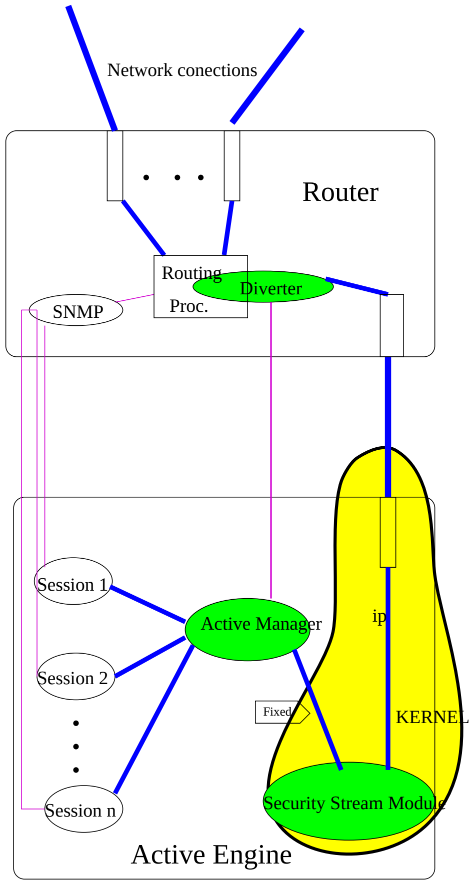



Figure 1: The general architecture. Thick lines between components represent a possible flow of data, thin lines represent a logical connection.

The main components of the system are (see figure 1):

**Diverter**— A part of the router that enables it to divert packets to the Active Engine if their protocol type and port number match a mask. The new generation of high-performance IP routers [KLS98] has this option implemented as part of the router hardware. Edge routers that handle low bandwidth links may perform this function in software, as we demonstrate in our prototype.

**Active manager** — The core of the Active Engine is the Active Manager. This part generates the sessions, coordinates the data transfer to and from the sessions, and cleans up after a session when it terminates. While a session is alive, the Active Manager monitors the session resource usage, and can decide to terminate its operation if it consumes too much resources (CPU time or bandwidth) or if it tries to violate its action permissions.

**Security stream module** — This module resides in kernel space below the IP output routine. Every connection that the session wishes to open, must be registered with this module to allow monitoring of network usage by sessions. The registration will be done by our supplied objects transparently to the application developer.

**Router interface** — This module allows sessions to access the router Managed Information Base (MIB). It is implemented as a Java object that communicates with the router using SNMP. In the future we plan to enhance performance by caching popular MIB objects.

The design allows multiple languages to be implemented simultaneously, but since the current implementation handles only Java packets we will restrict the description to the details of the Java implementation. Implementation of other languages may require some adaptations according to the language specifics.

In the following we describe the flow of packets through the system. Note that a non active packet does not pass through the Active Engine since the diverter recognizes it as such and thus the packet takes the fast-track to its output port.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              00H              |              04H              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |     lang      |     flags     |      seg      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            ses_seq                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            ses_id                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: The structure of the default option in our ANEP header. `version` is the software version (currently 1); `lang` is the language id, 0 identifies data, 1 is for Java; only the most significant bit in the `flags` field is currently assigned to identify last segment; `seg` is the segment number. `ses_seq` and `ses_id` comprises the session id.

All active packets include a default option that contains the unique session id of the packet, a content description (data, language), and more (see figure 2). All the diverted packets are sent to the active manager. If a packet does not belong to an existing session and it contains code it triggers a creation of a session. If it is a data packet it is discarded. A session creation involves, among others, authentication (not implemented), creation of a control block for the session, creation of a protected directory to store session files, opening of a private communication channel through which the session receives and sends active packets, and execution of the code.

Methods that are associated with the session object allow the Java program to easily send itself to another node, and to send and receive data. New arriving programs are passed to the session to allow it to perform code updating without losing its state.

Four UDP port numbers (3322-5) are assigned to active network research. The first, the *blind* addressing port, is used to send active packets to an unspecified node in a certain direction, i.e., towards some distant destination. The diverter in the first active node that is on the route to that destination intercepts the packet and sends it to the active engine. Therefor, the sender is not required to know the address of the next active node. The second UDP port number (the *explicit* active port) is used to send an active packet to a specific active node. This packet is forwarded through the fast-track of all the intermediate active nodes, and is not diverted until reaching its destination.

A session that requires non-active communication with the outside world, such as using SNMP, HTTP, etc., must use our interfaces to the relevant Java objects. These interfaces register the communication channel in the security stream module, transparently. An attempt to bypass these registrations, e.g., by using the original Java methods, will result in the security module dropping the packets of the unregistered connection.

The active manager keeps track of the resource consumption of the session in the node (CPU time, bandwidth, disk space). The monitoring is done to avoid a situation in which a session takes a large portion of the system resources, due to an error or malicious intent. For this reason, the accounting is done on a per-node basis and not globally. A session that consumes excessive resources is aborted.

The manager aborts a session in two more cases. The first is when the session does not show activity within an aging period. Different sessions may have varying aging periods, but none will be able to reside dormant in the CPU forever. This way we can assure garbage collection of unused code. A session might also ask to be aborted gracefully. The manager allows a graceful clean up of all the objects associated with this session, especially packets that are waiting for transmission in the node.

Since we expect most network programming to be stable, we do not try to optimize the capsule model. Thus, we are less concerned about the program size as programs are not going to be transmitted frequently. A mechanism to reassemble a program from a chain of up to 256 UDP packets is currently implemented in the active engine.

## 4.3 Inter process communication

For simplicity all the inter-process communication in the system is done using UDP datagram sockets. The following sockets are used in the system. They are all transparent to the session which uses them through our supplied Java methods.

**Administration socket** — A socket used by the manager to receive administrative information from the sessions. The sessions are using this socket to report their creation and termination, and to request access to services such as opening external connections.

**Manager internal data socket** — A socket used by the manager to receive data or programs from the sessions to be sent outside.

**Session Socket** — A socket used by the session to receive active packets.

**Manager external data socket** — A socket used by the manager to receive active packets (data or programs) from the router through the diverter.

**Security socket** — A socket used by the security stream module to receive information about permitted outside connections.

Initially, a session can only communicate with the manger. All delivery and receipt of active packets is done through the manager for security reasons. Sessions that wish to open a TCP connection to the outside world (e.g., for HTTP) have to register (in a transparent way using our wrappers). In addition, no exchange of information between sessions is allowed.

## 4.4 Security

Security and Safety are of major concern in the deployment of active networks. A system is safe if no application can destroy or damage the appropriate execution of other applications. In particular, the active engine as a whole should not effect the routing of non-active packets. A system is secure if all operations including access to data are authenticated, i.e., only authorized sessions can perform actions and/or access private data.

Our architecture supports both security and safety, although currently it is not fully implemented. In any design, one faces the dilemma of choosing between the freedom to allow more sophisticated session behavior (e.g., setting MIB variables, diverting non-active packets) and the fear of a possible safety/security hole. Our approach allows multiple levels of security via authentication and session classification. Each session is authorized to use specific services (MIB access for read or write, divert non-active packets) and resources

(CPU time, bandwidth, memory). As it is important to ensure both safety and security in order to promote the use of active network, one can select initially to be more restrictive in authorizing services, and gradually allow more sophisticated services.

Our first concern is to make sure that non-active packets are not affected by the active packets. This is easily achieved by the logical separation of the active engine from the router. In the case where the active engine is a separated, i.e., it resides on a seperate machine, which we foresee as the natural evolution path of IP networks, even a crash of the active engine will not effect non-active data.

Next step in safety, is to ensure that a session will not corrupt or even poke at other session data. We achieve this through the use of Java SecurityManager. It allow us to control the session running environment, in particular we prevent sessions from using native methods and restrict the use of the file system.

Malicious or erroneous over use of system resources is of great concern. To this end, we intend to monitor the use of CPU time by sessions. We implemented a tight control over the usage of the communication channel to the outside world. TCP connections can be only opened by a permitted session using our supplied methods that monitor the bandwidth consumption. An attempt to use Java methods is blocked by controlling the IP layer in the active engine. An unauthorized connection will be dropped. UDP packets can be sent only through the manager, which again can monitor the bandwidth usage.

## 4.5 Test bed description

We built a small heterogeneous network as described in figure 3. The network is comprised of both FreeBSD based active routers and COTS routers (currently we use CISCO 2500 routers and Lucent Technologies RABU PortMaster3) with an adjunct active component. The FreeBSD routers are PCs running FreeBSD, using `routed` for routing. In these PCs, the active engine and the router co-reside in the same machine.

The packet diverting to the active engine is done with filters according to the active port id. In the FreeBSD routers, active packets are diverted to the active engine by FreeBSD's packet filter software, `ipfw`. In the Cisco router, the diverting is done using Cisco IOS features. In the Lucent PortMaster, the diverting is done with tunneling.

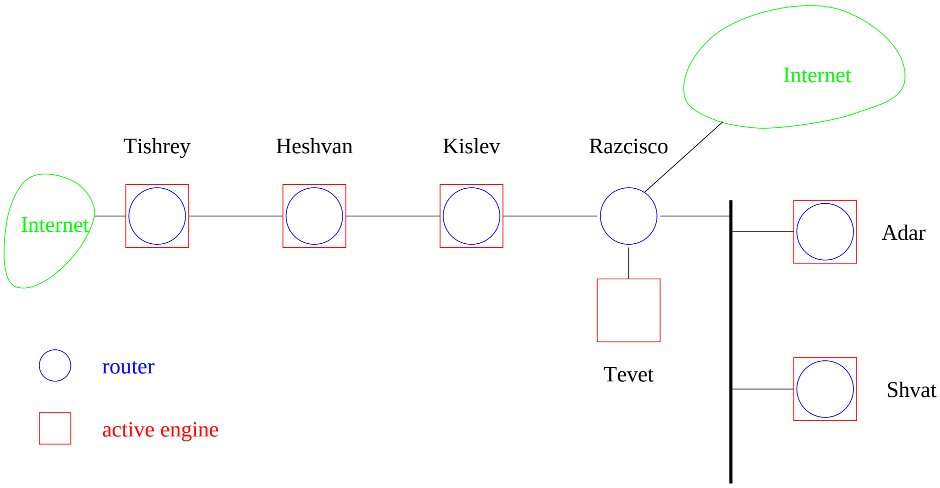



Figure 3: The prototype network architecture.

### 4.5.1 Performance

In building the prototype we did not aim at performance. Our first target was to build a concept system that will enable us to test design ideas and applications. Thus, many parts of the system where not optimized for performance. Nevertheless, we tested the capabilities of our prototype. Thus the delay of an active packet through the system should be treated as an upper bound on what can be accomplished and the load measures as lower bound.

Our first experiment was to measure the delay of an active packet through one active node. To this end we used a session on `heshvan` that forwards every packet it receives to the destination on the packet. Java applications on `tishrey` and `kislev` (see figure 3) exchange UDP packets using either an active port that is diverted to `heshvan`'s active engine, or a non-active port that is forwarded by `heshvan`'s router. The packets were about 500 bytes long, and the network was kept without additional traffic to prevent queueing delay that could effect the results.

The average round trip delay (RTD) for a packet without diversion was 1.37ms. The 90% confidence interval is $1.37 \pm 0.0047$ms, about 0.68% wide. The average RTD for a packet with diversion was 11.20ms. The 90% confidence interval is $11.20 \pm 0.022$ms, about 0.39% wide. Thus the average delay through `heshvan`'s active engine was $(11.2 - 1.37)/2 = 4.915$ms. Note that `heshvan` is a Pentium machine with 64Mbytes memory, that runs at 200MHz.

It is obvious that the active engine performance is bounded by the memory access speed and not by the computation power of the processor. We did not conduct a full scale stress test, but we repeated the above experiment with ten sessions active in `heshvan`, and the delay through the active engine did not change.

# 5 Application Examples

To get a better feel of the architecture and its usage, we demonstrate in this section application examples in full detail. We consider two problems both related to network management but represent basic problems that can be used in various applications. The first one is bottleneck detection, which is a special case of collecting information or calculating a function along a route between two nodes. The second application is a message dissemination to a large group of receivers. It is useful for automatic configuration of network elements or any other application that requires dissemination of messages to a large population.

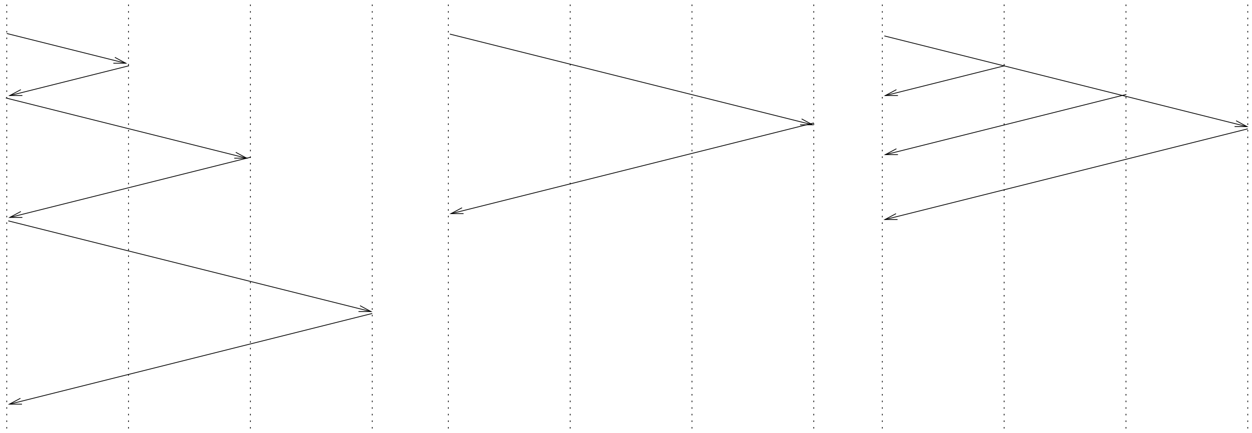
## 5.1 Bottleneck detection

Bottleneck detection is an important problem faced in network management. It is a building block for higher level applications, e.g., video conferencing, that require QoS routing. It is also an example for any problem related to gathering information along a given path between two network nodes. We will describe this application in greater detail to give the reader a better understanding on how the system works.

In today's IP networks there is only one ad-hoc technique to examine one specific QoS parameter, namely the delay along a path. It is the well-known `traceroute` program that enables a user at a host to get a list of all the routers on the route to another host with the elapsing time to reach them (see example is figure 4). The use of the `traceroute` program for network management has several drawbacks: it can only retrieve the hostname and the delay along a path; it is extremely inefficient in its use of network resources; and it is slow.

```
tishrey 268 act> traceroute shvat
traceroute to shvat (192.180.140.203), 30 hops max, 40 byte packets
 1  heshvan (135.180.142.2)  0.348 ms  0.236 ms  0.219 ms
 2  kislev (135.180.142.10)  0.708 ms  0.624 ms  0.599 ms
 3  razcisco1 (135.180.142.18)  2.620 ms  382.160 ms  7.236 ms
 4  shvat (192.180.140.203)  160.888 ms  4.052 ms  4.196 ms
```

Figure 4: An example of a `traceroute` execution from host `tishrey`.



(A) (B) (C)

Figure 5: Three traceroute executions on a three-hop path: (A) the current program; (B) collect-en-route; and (C) report-en-route

| Algorithm Used | No. of messages used | time of data arrival from node $i$ |
|---|---|---|
| `traceroute` | $n(n+1)$ | $i(i+1)$ |
| collect-en-route | $2n$ | $2n$ |
| report-en-route | $n(n+3)/2$ | $2i$ |

Table 1: Performance comparison (time is measured in hop count).

In an active network, and specifically in our architecture, there are several options to gather information along a given path between two network nodes, each optimize a different objective function. One option (collect-en-route) is (see figure 5(B)) to send a single packet that will traverse the route and collect the desired information from each active node. When the packet arrives at the destination node, it sends the data back to the source (or to any management station). This design minimizes the communication cost since a single packet is traveling along each link in each direction.

Another option (report-en-route) is (see figure 5(C)) to send a single packet along the path. When the packet arrives at a node, it sends the required information back to the source and forwards itself to the next hop. This design minimizes the time of arrival of each part of the route information, while it compromises communication cost.

Note that the `traceroute` program has (see figure 5(A)) time and communication complexities that are quadratic in the path length. Table 1 compares the three options.

The use of general programs in the capsule enables the application programmer to query any available variable (e.g., a MIB variable) from the router. Our solutions can collect any desired datum rather than just the router IP address. For example, for bottleneck detection we can collect statistics about TCP packet loss along a route to a certain host in order to identify the bottleneck link.

In addition, the program can be generalized to allow a node to perform the data collection on the path between any other two active nodes in the network. This is facilitated by our two addressing modes. As mentioned before, the reports can be sent to any host (not necessarily active).

```
hop 1: Object ID : .1.3.6.1.2.1.1.1.0  (Syntax : Octet)
Value : FreeBSD tishrey.dnrc.bell-labs.com 2.2.2-RELEASE FreeBSD 2.2.2-RELEASE #
0: Mon Feb 23 04:01:12 EST 1998     root@tishrey.dnrc.b

hop 2: Object ID : .1.3.6.1.2.1.1.1.0  (Syntax : Octet)
Value : FreeBSD heshvan.dnrc.bell-labs.com 2.2.2-RELEASE FreeBSD 2.2.2-RELEASE #
0: Mon Jun  8 13:15:35 EDT 1998     root@heshvan.dnrc.b

hop 3: Object ID : .1.3.6.1.2.1.1.1.0  (Syntax : Octet)
Value : FreeBSD kislev.dnrc.bell-labs.com 2.2.2-RELEASE FreeBSD 2.2.2-RELEASE #0
: Thu Dec  3 05:04:35 EST 1998     root@kislev.dnrc.bel

hop 4: Object ID : .1.3.6.1.2.1.1.1.0  (Syntax : Octet)
Value : Cisco Internetwork Operating System Software
IOS (tm) 2500 Software (C2500-I-L), Version 11.3(3), RELEASE SOFTWARE (fc1)
Copyright (c) 1986-1998 by cisco Systems, Inc.
Compiled Mon 20-Apr-98 18:23 by phanguye

hop 5: Object ID : .1.3.6.1.2.1.1.1.0  (Syntax : Octet)
Value : FreeBSD shvat.dnrc.bell-labs.com 2.2.2-RELEASE FreeBSD 2.2.2-RELEASE #0:
 Wed Dec 23 04:27:46 EST 1998     root@shvat.dnrc.bell-
```

Figure 6: An example of a router id report generated by our program. [2]

Figure 6 shows the router id report[3]generated by the implementation of option report-en-route, executed on a six node active network depicted in figure 3. The active packet that generates this report can be sent from any host as long as its path goes through an active node. The first active node (`tishrey` in our case) diverts it to its active engine, as the packet uses the well known active port number (3322). The packet contains the class file of the Java code we show in Fig 7 as well as 9 bytes of data, which contain the report destination IP address, the IP address of the destination end-point of the path, and a hop count.

As the session number of this packet does not match any existing session in this node, a new session will be created using the Java code in the active packet. The packet itself is then delivered to this session as the first packet. The session reads the data from the capsule, generates a copy of the active packet to be sent towards a destination, sends a report home, and terminates. The generated copy is then intercepted by the next node on the route to the destination in which exactly the same scenario repeats. The reports are sent to the destination specified in the code (it is easy to have the report destination as part of the data carried in the capsule), which may as well be different from the host that originated the application.

The Java code of Fig 7 is straight forward; `session` is a new instance of the class Act, the constructor takes $-9$ as an argument that indicates the number of data bytes in the capsule. The program and the data are then retrieved using our Act class methods. A new active packet with the appropriate hop count is then prepared, and sent to the destination address. We then generate a report; local information from the router is gathered using the SNMP interface. Currently, we require here a full MIB specification of the requested values. In the future, part of this interface may be overridden by a different Java interface to retrieve some of the most important information.

Our implementation is not limited to collecting only node ids. We could, for example, check some of the IP counters in a router, instead of (or in addition to) its name. The change in the code is minimal: all we

[3]Note that the reports from the FreeBSD machines are truncated due to implementation problem of the SNMP agent in FreeBSD.

```
import Act.*;
import OurSnmp.*;
public class capsule
{
    public static void main ( String args[] ) throws Exception {

        DatagramPacket  udppacket;
        Act   session = new Act(-9);
        byte[] p = session.getProg();
        byte[] v = session.getInitVars();
        byte[] destip = new byte[4];
        byte[] udpmsg;

        //  get target IP address
        for (int i=0;i<4;i++) destip[i] = v[i+4];

        //  get hop number
        int  hopnum = (int) v[8];
        if (v[8]>127) System.out.println("too big.........");
        else v[8]++;

        //  prepare a new message
        byte[] newpck = new byte[p.length+9];
        for (int i=0;i<p.length;i++) newpck[i] = p[i];
        for (int i=0;i<9;i++) newpck[i+p.length] = v[i];

        // send a new message forward
        session.send(newpck,Act.IPaddr(destip));

        // get some local status (via SNMP)
        String oid = ".1.3.6.1.2.1.1.5.0";  //host name
        String res1 = OurSnmp.Get(oid);

        // send a UDP datagram to report your status
        String udpmsgtext = "hop " + v[8] + ": " +  res1;
        session.sendUDP(udpmsgtext, "inbar.dnrc.bell-labs.com", ReportPortNum);

        // be nice, report you are done.
        session.killme();
    }
}
```

Figure 7: The Java code in the active packet that implements the data collection along a path (option report-en-route)

have to do is to request a different MIB variable. If we chose to request `.1.3.6.1.2.1.4.6.0` which is a MIB variable counting the number of IP packets forwarded by the router we get a report like the one in Fig 8. Note that reports are received at `inbar` which is not an active node.

In figure 8, one can see that the number of forwarded packets is increased between the two executions. The counter value is 0 for the last router, as currently only one of its interfaces is connected, and thus, it does not forward any packets. Note also that the reports may arrive out of order due to the difference in response time between the SNMP servers in the machines. It is the application's GUI responsibility to display the information in a convenient way to the user.

It is easy to generalize the program to start the data collection from any active node in the network. This can be done by using the explicit port number (see page ) to send the program to the desired starting node. As explained before, the reports can be sent to any host (not necessarily active)[4]. Using this, a reverse traceroute can be easily implemented.

As our generalized data collection program will become popular it can be assigned a well-known session id and be flooded to all the routers. At this stage, small data packets will be sent to collect the data between two end nodes. Such data packets can contain the IP address of the start and end nodes for the path, the report destination, the required MIB variable(s), and a hop count.

## 5.2 Message dissemination

In many network management applications there is a need to deliver a message to an ad-hoc group of machines. For example, using an autoconfiguration application, a group of routers might need to be reconfigured due to a change in the network. A monitoring application may periodically query all the hosts it did not hear from in the last period. A security application might collect information from a group of routers based on the attack pattern it suspects.

In these applications, the machine group is ad-hoc defined for the purpose of a single message dissemination and not a long lasting group as in multicast applications. Since the group is defined by the recipient list of a single message, it is not efficient to form a multicast group or to invest in any other long term infrastructure. Without active network, the two ways to implement a message dissemination to a large group of receivers is by either sending a unicast message to each receiver, or by broadcasting the message to the entire network.

We assume that a message is comprised of a header with a list of receivers, and a body which, for a large group of receivers, is much smaller than the header. We use the fact that the union of all the routes from the originator to the receivers is a directed tree rooted at the originator, termed *the dissemination tree.* To simplify the discussion, we assume this tree is a binary balanced tree with the receivers at the leaves (see Fig 9).

Our active solution is to partition the receiver list at the source according to the first hop on the path to each receiver. We continue this partitioning at every intermediate node until the message arrives at the tree leaves. This way, exactly one copy of the message traverses each link in the dissemination tree. For a balanced binary tree with $n$ leaves, our message complexity $2n$ and the bit complexity is $O(n \log n)$, while the unicast solution has a message complexity of $n \log n$ and the bit complexity is $O(n \log n)$.

In each active node, we have to partition a possibly large list of addresses. This requires more processing at each node than the one required in the bottleneck detection example. In fact, the processing time will be linear in the receiver list length, since we have to check the next hop of every receiver. To be able to do that, we must have access to the routing table at the router which is also a feature we supply through the MIB access. Such an access is typically not available to higher layer software agents.

The overall processing cycles needed in the entire network to accomplish this task for a full binary tree with $n$ leaves is $O(n \log n)$. If the message is distributed using unicast, we only need to have $O(n)$ processing

[4]To reserve this option, we relinquished the ability to display the message travel time that is calculated by the `traceroute` program. However, it can be easily done by the option report-en-route program in the cases where the originator is the node that receives the reports.

```
inbar:try> java UDP
hop 1: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 1178310

hop 2: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 491882

hop 3: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 213

hop 4: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 1066733

hop 5: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 0



hop 1: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 1178361

hop 2: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 491911

hop 3: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 215

hop 4: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 1066771

hop 5: Object ID : .1.3.6.1.2.1.4.6.0  (Syntax : Counter32)
Value : 0
```

Figure 8: An example of a report generated by our active data collection program for the IP forwarding counter.

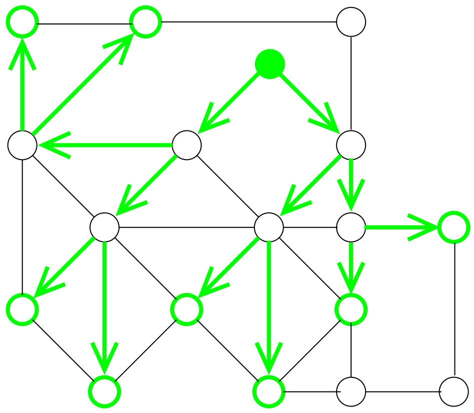


Figure 9: An example of a binary balanced dissemination tree.

cycles at the sender. The delay due to the processing is about twice in our active solution than the unicast solution. This is since at every level of the tree the processing delay is halved.

Our solution for the message distribution problem demonstrates how active networks can be used to trade-off delay and network utilization. Note, that we achieve a logarithmic improvement in the utilization by paying only a constant factor in delay.

# 6 Discussion and Future Work

Since the inception of the active network idea there is a search for the "killer application", the one that will strongly require active network technology. We believe that network management is a domain where using active network technology could be proved to be very significant. Applications like adaptive control, router configuration, element detection, network mapping, security management (intruder detection, fighting deny of service attacks) are only some examples where active network technology can be successfully applied. Our architecture also supports solutions to other problems not necessarily part of network management, such as, search worms, smart mail, multicast, hop to hop flow control, etc.

The additional delay seen by packets in active networks, is an issue that was addressed in the past, and as others pointed out we believe the small slow-down is compensated by the big saving that can be achieved in traffic volume. The analysis presented in section 5 is a first step towards a more formal framework to address this point. In our prototype, non-active packets suffer only the negligible additional delay of the diverter. And although the active engine is not optimized in the current version, as the emphasis is on functionality, the delay through the AE is reasonable.

The work presented here is in progress and we intend to keep building our prototype and enhance it with more capabilities. Our initial aim was to build a prototype that will enable us experimenting with new applications. Based on our experience, we plan to add functions to the engine, to make the programming easier, and programs smaller.

Altogether, we presented a prototype implementation of a network management engine built using active network technology. We believe our approach can enable new and efficient ways to manage today's (and tomorrow's) networks.

# References

[AAH+98] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith.

The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[ABG+97] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. The active network encapsulation protocol (ANEP). URL `http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt`, 1997.

[AMK98] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. In *SIGCOMM'98*, September 1998.

[BCZ97] Samrat Bhattacharjee, Ken Calvert, and Ellen W. Zegura. An architecture for active networking. In *HPN'97*, April 1997.

[DP98] Dan Decasper and Bernhard Plattner. DAN: Distributed code caching for active network. In *INFOCOM'98*, March 1998.

[Gro98] AN Working Group. Architectural framework for active networks. URL `http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps`, August 31 1998. Version 0.9.

[HB97] Gísli Hjálmtýsson and Samrat Bhattacharjee. Control on demand - customizing control for each application. Technical report, AT&T Technical Memorandum, 1997. presented at the OpenSig 1997, Columbia University, NY, NY, October 6. 1997.

[HJ97] Gísli Hjálmtýsson and Ajay Jain. Agent-based approach to service management - towards service independent network architecture. In *fifth IFIP/IEEE International Symposium on Integrated Network Management - IM'97*, pages 715 – 729, May 1997. San Diego, CA, USA.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. In *International Conference on Functional Programming (ICFP) '98*, pages 86–93. ACM, September 1998.

[HPB+97] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Joust: A platform for communications-oriented liquid software. Technical Report TR97-16, Dept. of Computer Science, Univ. of Arizona, December 1997.

[KLO97] Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4):68–77, July/August 1997.

[KLS98] Vijay P. Kumar, T. V. Lakshman, and Dimitrios Stiliadis. Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow's Internet. *IEEE Communications Magazine*, 36(5):152–164, May 1998.

[KMH+98] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, S. Sheth, H. Pindi, F. Wahhab, A. Gopinath, and A. Nagarajan. Implementation of a prototype active network. In *OPENARCH'98*, pages 130–143, April 1998.

[KS98] S. Kesav and Rosen Sharma. Issues and Trends in Router Design. *IEEE Communications Magazine*, 36(5), May 1998.

[KZ97] Joseph Kiniry and Daniel Zimmerman. A hands-on look at java mobile agents. *IEEE Internet Computing*, 1(4):21–30, July/August 1997.

[LS98] T. V. Lakshman and D. Stiliadis. High speed policy-based packet forwarding using efficient multi-dimentional range matching. In *SIGCOMM'98*, September 1998.

[Nyg98] Erik L. Nygren. The design and implementation of a high-performance active network node. Master's thesis, Massachusetts Institute of Technology, February 1998.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[VD98] Andreas Vogel and Keith Duddy. *JAVA Programming with CORBA*. Wiley, 2nd edition, 1998.

[WGT98] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *OPENARCH'98*, pages 117–129, April 1998.

[YdS96] Yechiam Yemini and Sushil da Silva. Towards programmable networks. In *IFIP/IEEE Intl. Workshop on Distributed Systems Operations and Management*, October 1996.

[ZHGW99] Michael Zapf, Klaus Herrmann, Kurt Geihs, and Johann Wolfgang. Decentralised snmp management with mobile agents. In *sixth IFIP/IEEE International Symposium on Integrated Network Management - IM'99*, May 1999. Boston, MA, USA.

# Appendix

# A Methods of the class `Act`

In this appendix we describe shortly some of the methods supplied with the class `Act`.

**The constructor** `Act(int len)` — The constructor opens a datagram socket, and sends an opcode to the manager administration socket to announce that the session is active. The opened socket is used for all the outgoing communication to the manager. The constructor saves the program code and the initial data. `len` is optional and it indicates the amount of initial data carried with the program packet (capsule).

`public byte[] getProg()` — Returns the program code from the capsule. Used by a session to send itself.

`public byte[] getInitVars()` — Returns the initial data from the capsule.

`public byte[] rec()` — Used to receive a packet without the active header.

`public byte[] recraw()` — Used to receive a packet with the active header. This method may be useful when the header information is required, e.g., for debugging.

`public void send(byte[] m, InetAddress destAddress)` — Used to send a packet `m` to address `destAddress`. The ANEP header and our default option are generated by the manager and are not part of the buffer `m`.

`public void killme()` — Used to signal the manager that the session terminated and can be cleaned.

`public void keepme()` — Used to signal the manager that the session is still alive.