ELSEVIER

Computer Networks 36 (2001) 21–34

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

The active process interaction with its environment

Jessica A. Kornblum [a,1], Danny Raz [b], Yuval Shavitt [b,*]

[a] Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA
[b] Bell Labs, Lucent Technologies, Holmdel, NJ 07733-3030, USA

Abstract

Adding programmability to the interior of the network provides an infrastructure for distributed applications. Specifically, network management (NM) and control applications require access to and control of network device state. For example, a routing load balancing application may require access to the routing table, and a congestion avoidance application may require interface congestion information. There are fundamental problems associated with this interaction that are apparent in current technologies. In this paper, the basic tradeoffs associated with the interaction between an active process and its environment and presenting ABLE++ as an example architecture is studied. Most notably, two design tradeoffs, *efficiency vs. abstraction* and *application flexibility vs. security* are explored. The advantages of the architecture by implementing a congestion avoidance algorithm are demonstrated. © 2001 Published by Elsevier Science B.V.

Keywords: Active networks; Network management

1. Introduction

In active networks [14], network elements, such as routers, are programmable. Code can be sent inbound and executed at the router rather than just at the edge nodes. The rationale behind active networks is that moving computation from the network edges to its core facilitates more efficient use of the network. Many of the suggested applications [20], such as congestion control and adaptive routing, require the active code to be aware of local state information in the router. When active networks are used for network management (NM) [8,16,18], interfacing with the managed router is even more important, because management applications need efficient monitoring and control functions. For example, a routing load balancing application may require access to the routing table, and a congestion avoidance application may need interface congestion information. Efficient and secure access to managed device state is especially needed when the managed router is logically, and to a greater extent, physically, separated from the management active environment [16].

The design of a control/monitoring interface to a router must balance between abstraction and efficiency. An interface with a low level of abstraction, such as simple network management protocol (SNMP) [17], can be efficient at the device level but provides a cumbersome programming infrastructure at the application level. Likewise, an interface with a high level of abstraction provides

* Corresponding author.
E-mail addresses: jkornblu@dsl.cis.upenn.edu (J.A. Kornblum), raz@lucent.com (D. Raz), shavitt@ieee.org (Y. Shavitt).
[1] This work was done while at Bell-Labs, Lucent Technologies.

simple programming constructs, such as Corba [21], but is often inefficient.

Another design tradeoff we address with our system is application flexibility versus security vulnerabilities. Management applications require a high level of flexibility to complete tasks needed to manage the network. But on the other hand, allowing an active routing application to change the forwarding tables at a router may result in global routing instability.

Our goal is to present a simple and abstract interface to the application programmer while maintaining the efficiency of the system which requires visibility of operation costs. We address this tradeoff by using a cache and an efficient abstract design. We balance the latter tradeoff by addressing security at multiple levels of our system without limiting the expressiveness of the application.

To make the discussion clearer and more concrete, we demonstrate our design by describing novel extensions to the ABLE architecture [15], ABLE++. ABLE is an active network architecture that allows the easy deployment of distributed NM applications. In ABLE++, we optimize the interaction between the active process (i.e., NM application) and the router by adding a local cache at the device level to store local router data. Caching information helps in more than one way to alleviate the load from the core router software where computing power is scarce. Obviously, it allows a single data retrieval to be used by several applications. In addition, some of the popular information we cache is 'computed' from consolidating many different 'atomic' data units, e.g., a list of all neighbors. Consolidating cached items serves several aims: it reduces the processing load from the router as a result of continuously fetching router state, eliminates the fetching, computing and storing overhead at the application layer, thus, shortening the retrieval time for applications and simplifying writing of management applications.

Our interface eliminates the management instrumentation details (e.g., SNMP, CLI or CMIP) from the application layer without sacrificing efficiency. In a similar way, we abstract the control interface that allows privileged applications to modify the router state. The control capability introduces security hazards, which we address by placing security mechanisms such as authentication and application classification at various layers in the architecture. Each application is authorized to use specific services (read and write) and resources (CPU time, bandwidth, memory). Our interfaces check for conformance to these restrictions.

To demonstrate the usefulness of our design, we present an implementation of a congestion avoidance application similar to the one suggested by Wang [22]. The application constantly monitors the load of the router interfaces. When congestion is detected, the algorithm attempts to reroute some flows around the congested interface. The algorithm works locally: a router exchanges messages with its 2-neighborhood routers (the group of routers that are no more than two hops away from it) to find a deflection route (similar ideas were also suggested in [4]) for some of the flows that pass through the congested interface. The algorithm does not change the routing tables, but instead adds temporary entries to the forwarding tables unlike Wang's original algorithm that uses tunneling on a per packet level.

1.1. Organization

In the following section, we present the new ABLE++ architecture, Section 3 discusses the system performance, and Section 4 discusses security issues. In Section 5, we demonstrate the advantages of our interface design with an application example. We discuss related projects in Section 6, future work in Section 7 and conclude in Section 8.

2. Architecture

NM applications require an infrastructure to query and set state on managed devices. Current management models support both operations but in a very inefficient and platform dependent manner. We have addressed both inefficiencies by expanding the ABLE active engine architecture to include three processing *Brokers* (see Fig. 1): a *Session Broker*, an *Info Broker* and a *Control Broker*.

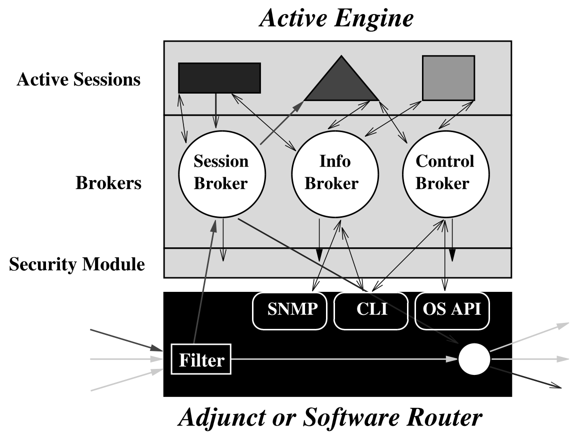

Fig. 1. ABLE++.

We define a "Broker" as a processing entity that completes a task on behalf of a requester. We have specialized the concept of a "Broker" to handle three different types of interaction between the active session and processing environment. The Session Broker creates, manages, and cleans up the processing environment as well as manages alive active session. The Info Broker provides an efficient monitoring channel by exporting local state to active sessions and mediates all queries for local state. The Control Broker provides a secure channel for control operations by exporting a set of control methods and associated policies. We will use the terms "active session" and "application" interchangeably.

Process management, information queries, and control requests are each handled by different Brokers because each requires a different set of security restrictions and/or performance metrics. The Session Broker must have complete monitoring and control access over all active sessions to prevent excess resource usage and facilitate communication between active sessions on different nodes. The Info Broker's main function is to provide efficient access to local data. Since accessing data through this channel is read-only, the Info Broker does not introduce the same security concerns as the write-enabled control channel. Hence, the design focus in the Info Broker is on efficient access to data. On the other hand, the Control Broker does introduce many security concerns because active sessions can change the state of the device. The design focus here is on preventing active sessions from leaving the network in an inconsistent state. All three Brokers communicate with active sessions using TCP or UDP sockets. Therefore, ABLE++ can support active applications written in any language that supports the socket interface. The following sections will discuss in more detail the design goals and architecture of each Broker.

2.1. The Session Broker

The Session Broker manages the initiation, behavior, communication and termination of all active sessions. Conceptually it is a meta-Broker as it can be viewed as giving session control services to the system. The communication aspect of the Broker is the only non-meta service it performs for the session, and thus, might call for a separate entity. However, the separation introduces inefficiencies in handling out-going messages and thus, was left for further research.

Most of the functionality of the Session Broker was inherited from the original design of the ABLE active engine [16]. Therefore, we will only go into a brief discussion. Both ABLE and ABLE++ architectures are designed to support long lived, mobile active sessions. NM applications must be long lived to implement any ongoing monitoring services. Mobility is essential to propagate a distributed application throughout the network. ABLE++ allows active sessions to control code distribution. Therefore, the Session Broker's responsibilities can be divided into two parts: *managing active sessions* (the meta-Broker) and *exporting mechanisms for communication and mobility*.

2.1.1. Managing active sessions

The Session Broker initializes, monitors and terminates active sessions. During the initialization phase, local resources (CPU, memory or bandwidth) are allocated to each session. "Watch dogs" or aging timers monitor all activities to prevent excess resource consumption or prevent session termination without freeing its assigned resources. All active sessions are registered with our *Security Module*. The Session Broker can terminate an

Table 1
Communication and mobility interface

Name	Description
`int Distribute ()`	Sends program and data to neighbors except original sender. Returns the number of successful messages
`int DistributeAll ()`	Sends program and data to all neighbors. Returns number of successful messages
`int Distribute (Addr)`	Sends program and data to Addr. Returns number of successful messages
`byte [] Receive ()`	Receives a packet
`void Send ( byte [], DestAddr)`	Send packet to DestAddr.
`void SendReport (String DestAddr, Port)`	Send String to DestAddr at Port

active session if it uses too many resources or attempts to violate security restrictions. We will discuss security in ABLE++ in Section 4.

2.1.2. Communication and mobility

The Session Broker controls active session communication and mobility through an exported API (see Table 1). `Distribute()` and `DistributeAll()` are used to propagate an application to every active node in the network. Session Broker will prevent multiple copies of the same application on a single node. `Distribute(Addr)` is used to send an application to a specific spot in the network. This function is especially useful for monitoring only specific regions on the network for bottleneck or congestion detection. `send(byte [])` and `byte [] receive()` are standard communication functions used to pass messages between active sessions both on the same node or different nodes. We have also included a specialized report sending function, `sendReport(String, DestAddr, DestPort)`. Event or alarm reporting is a crux of NM. Often, reports are the only ongoing documentation of the network's behavior. Therefore, we have added a specialized function for reporting.

2.2. The Info Broker

The *Info Broker* is used to export local device information to active sessions. Specifically, the Info Broker specializes in retrieving information or state from local devices for active sessions. Without the Info Broker, active session programmers must worry about how to get local information and what kind of information is available. The three components of the Info Broker, *Engine*, *Interface* and *Cache* (see Fig. 2), abstract these details away from the application programmer all while providing an infrastructure for efficient information retrieval. In the following text, we will discuss each component in more detail.

2.2.1. Engine

The engine is the "intelligence" of the Info Broker. Active sessions send requests for local state to the engine. The engine then, determines the "best" avenue to handle the request, retrieves the information, and finally translates it back to the active session.

The most interesting part of the engine is how it selects the "best" avenue. Data is classified by its volatility. In the simplest case, one can identify two types of data items: volatile, such as the number of active TCP connections or non-volatile, such as the router name. However, volatility is a continuous spectrum where each datum can be associated with the length of time it can be cached before its value is expected to change significantly. To simplify the implementation, the engine identifies a requested datum as part of three classes: volatile, quasi-volatile or static. Static and quasi-volatile data are cached in the local *Cache*. For the quasi-volatile data, we attach a time-to-live counter, that invalidates after a predetermined time period. Quasi-volatile data that expires can be either removed from the cache or be refreshed.

The advantage of placing the policies in the engine is that it abstracts the instrumentation details away from the programmer, thus, reducing

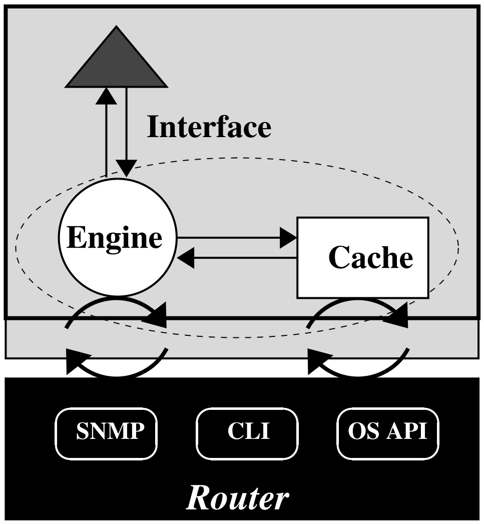

Fig. 2. Info Broker.

the complexity of applications. For example, if an application needs local state from three different managed devices and each device requires a different communication protocol, the programmer must implement each protocol to manage those devices, thus resulting in a large, complicated program. Not only does the engine abstract those details from the application layer, it also reduces response latency by choosing an appropriate avenue for retrieving information. Currently, we have two modules, SNMP and control line interface (CLI), implemented. We also understand there are some instances in which the application would like control over how the information is retrieved. Thus, we have added small instrumentation hooks like SNMP's `get(oid)` and `getnext(oid)` to force the engine to use a particular instrumentation. This brings us to an important question. How do active sessions communicate queries to the engine?

2.2.2. Interface

The engine exports local state to active sessions via a predefined interface (see Table 2 for a list of methods).

As mentioned previously, the main design goal of the Info Broker is efficient access to local device state. Current information retrieval models allow applications to ask for small "atomic" pieces of local state. However, applications monitor and make decisions at a much higher level. For example, a load balancing application needs to calculate load on managed interfaces. In the current practice, interface "load" is not exported at the device level. It must be derived from atomic pieces of local state such as MIB-II's `ifOutOctets` and `ifSpeed` over a given period of time. The application must query for several variables and compute the load before continuing with the algorithm. This is an inefficient use of system resources and time. The application must wait until all variables have been received before proceeding. Waiting through several possible round-trip transmit times prevents the application from

Table 2
Info Broker interface methods

Name	Description
`int getNumIf()`	Number of interfaces
`String getName ()`	Local machine name
`String [] getIpAddrs (Interface)`	List of IP addresses for Interface
`int getIfNumber (IPAddr)`	Interface number for IPAddr
`String getNextHopAddr (DestAddr)`	Next Hop IP address towards DestAddr
`String getNextHopIf (DestAddr)`	Interface number of Next Hop towards DestAddr
`float getLoad (Interface)`	Load of interface
`int getStatus (Interface)`	Operation status of Interface
`String [] getNeighbors ()`	List of IP addresses for all "alive" neighbors except loopback
`String [] getNeighbors (Interface)`	List of neighbor IP addresses for Interface
`Boolean isLocalLoopback (IPAddr)`	True if IPAddr is loopback
`Boolean isLocalLoopback (Interface)`	True if Interface is loopback
`String [] getDestAddrs (Interface)`	Destination IP addresses for Interface

working at fine grained time scale. We have addressed this problem by exporting high level data. Applications can now ask "Who are my neighbors?" with one query, "How many interfaces are alive?" with two queries, and finally "What is the load on this interface?" with one query, just to name a few. We have pushed the data correlation into the device level, thus reducing the number of queries generated at the application level as well as the total round-trip latency.

2.2.3. Cache

We have added a small local cache to decrease query response time at the application level and reduce the query load from the managed device. Our cache reflects the design of the Interface and engine policies. As mentioned before, we only cache popular static and quasi-volatile local state.

The access time to the router local information is inherently much larger than accessing the local cache (see Section 3). This is because routers are not optimized to answer queries and treat them with low priority. On the other hand, accessing the local cache does not require crossing of the user/kernel boundary and is much faster. When multiple sessions require the same local data, using the cache reduces the number of queries the router receives.

Caching state introduces a trade-off between retrieval efficiency and information freshness. A high update frequency benefits from storing fresh state, but at an increased retrieval cost. The cache policies must be tuned to achieve an appropriate trade-off between resource usage and stale state. Different caching policies can be set for each type of information. For instance, volatile data should be refreshed frequently, whereas static data will not change as often.

Another related design issue is what triggers data updates. One option is to update the data periodically at some rate (that may depend on the data change rate [7]). Another option is to update stale data only if a new request arrives. Periodic updates may result in unnecessary overhead but maintain fresh information at all times, while the former approach may have a longer reaction time if an application is accessing stale data.

2.3. Control Broker

The *Control Broker* is a bidirectional channel that enables active sessions to control the behavior of the router. This allows authorized sessions to perform network engineering tasks such as changing an IP route, or modifying the QoS parameters in the router. Giving such an extended power to an active session comes with an heavy cost; unauthorized, or malicious sessions can "take over" the router and destroy the entire network. Thus, one should be very careful with the use of this feature. Only authorized sessions should be allowed to use it, and the possible side effects should be minimized. Even when sessions do use this privilege correctly, there is a problem of coordinating the overall local (and also global) effect of the combined control action. Consider two NM applications: one is designed to monitor the status of all interfaces, and once a down interface is detected it tries to reset it; the other application is a security application that turns off an interface when an attack is detected. The results of these two applications working together in the same node might be that the interface is turned on and off alternately.

The overall structure of the Control Broker is very similar to that of the Info Broker: it has an *Interface* that is, a collection of classes used by the active session to request the service, and an *engine* which processes requests. The engine receives a request, checks whether it is legal by both authenticating the sender session and verifying no conflicts with previous, active requests exist, and then executes it using the best available underlying method or indicates an error has been found. The *Security Module* checks the authorization of a session to take certain control actions as well as detect conflicts. The Control Broker can communicate with the router to perform control tasks via SNMP, the router's CLI, or a special-purpose control protocol. Note that the requirements of many control functions include the retrieval of information, so the control action functions, in many cases, return values to the active session. However, the focus and thus, the performance considerations in the design of the Control Broker and the Info Broker are substantially different.

A basic, but extreme example of a control function we have implemented is `cliControl (String command)`. This function takes as an argument a CLI command, executes it on the router, and returns the outcome string. This is a very basic, low level device-dependent function and the application (the active session) has to know (by checking possibly through the Info Broker) what type of a router is located at the node, and then uses a router specific CLI to perform the control task. It is also very dangerous, as it allows the session to perform harmful operations such as the unix *shutdown* command.

A higher level example is the function `tmpSetRoute(destination, gateway, routeTTL)`. This function creates a manual entry in the local router's forwarding table for `destination` with `gateway` as the next hop, for a period of `routeTTL` ms. The function returns immediately and does not wait until the new route has been canceled. This requires that the Control Broker engine maintain state for each route deflection request that is in progress. It also has to resolve and raise alerts for possible conflicts. This design ensures the atomicity of the Control Broker service which has an important safety aspect. Since the Broker is expected to be more stable than any session, atomicity ensures that any temporary change a session makes in the router state will indeed be cleared in due time. However, if the Control Broker fails, the router may be left in an unstable state. We discuss this further in Section 4.

3. Performance

As stated before, a main design goal of ABLE++ is to improve the efficiency of the interaction between the application layer (active sessions) and the managed device. In this section, we demonstrate how the ABLE++ architectures achieves this goal.

3.1. Reduced network traffic

Current management technologies, such as SNMP, export atomic pieces of data which are available to the application via an interface or protocol. Management applications must query many of these atomic pieces of data before deriving the computation result needed to reason about the state of the device or network. Such a low level interface generates an unnecessary amount of traffic in the network. ABLE++ alleviates this traffic by including a device level cache. Table 3 lists the number of messages needed to compute three functions, `getNeighbors()`, `getLoad()` and `getNumIf()` (described below), using both SNMP library directly and the Info Broker interface from the application layer.

Table 3
Reduced message count

	Direct SNMP	Info Broker
`getNeighbors()`	4[a]	1
`getLoad()`	8[b]	1
`getNumIf()`	1	1

[a] Per routing entry
[b] Per polling interval

3.2. Info Broker performance

Now that we have determined that using the Info Broker can reduce the number of messages needed to query for device state, there are two performance questions left to answer:

- How fast can the application retrieve device state?
- Does the Info Broker incur additional overhead for the application?

In this section, we answer both of these questions by analyzing the response time between an active session and a managed router's SNMP agent. We measure the response time for the following three functions, each decreasing in complexity (see Table 2 for the entire list):

- `getNeighbors()` walks the SNMP routing table (ipRouteTable) looking for directly linked IP addresses (ipRouteType). The function returns a list of neighbors for all local interfaces.
- `getLoad(interface)` polls the router for number of received octets (ifOutOctets), system time (sysUpTime), and link speed (ifSpeed) over a period of time. The load is computed by calculating the rate of received octets divided by the

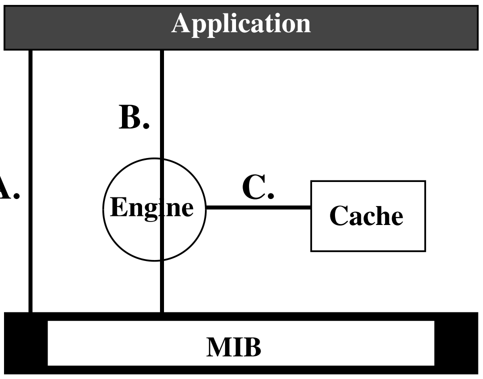

Fig. 3. Experimental setup.

link speed and returned. We ignore the waiting interval between measurements.

- `getNumIf()` simply polls the router for the number of interfaces (ifNumber) and then returns this number.

Our test program performed 200 consecutive function calls for each above three functions using three different communication channels between the application and the router (see Fig. 3):

A. *Direct SNMP*. Opens a connection directly with the SNMP agent, polls for needed state using only SNMP `get` and `getnext` functions, then computes the result within the active session.

B. *Broker SNMP* (*no cache*). Opens a connection with the SNMP agent through the Info Broker, polls for needed state using only the Info Broker `get` and `getnext` functions, then computes the result within the active session. (The only difference between this channel and Direct SNMP is an additional inter-process communication (IPC) between the session and the Info Broker.)

C. *Broker SNMP* (*cache*). Opens a connection with the Info Broker and polls the local cache for computed result.

Our implementation of the Info Broker as well as active sessions run on JVM 1.1.8, 200 MHz Pentium PCs with 64 MB RAM running on FreeBSD 3.2 operating system. Response times were measured using Java's system clock method (*System.currentTimeMillis()*). Over 200 consecutive trials, we notice periodic increases in response times (see for example Fig. 4). We suspect the increase is due to Java garbage collection and application thread scheduling algorithms in the JVM. Therefore, we have plotted in Fig. 5 the median of the 200 response times for the different monitoring channels and functions. For `getNeighbors()` using Direct SNMP and Broker SNMP (no cache), the median value is within 7.14% of 25% quartile value and 12.4% of 75% quartile value. In the rest of the experiments, the median, 25% quartile and 75% quartile values differed by only 1 ms. The reason for the higher difference in the first two experiments may be attributed to the longevity of the algorithm. The full cumulative distribution is plotted in Fig. 6. It is clear that for all the three monitoring channels most of the 'probability mass' is concentrated around the median value, and that the median is very close to the minimum. This spikes observed in Fig. 4 are responsible for the tails.

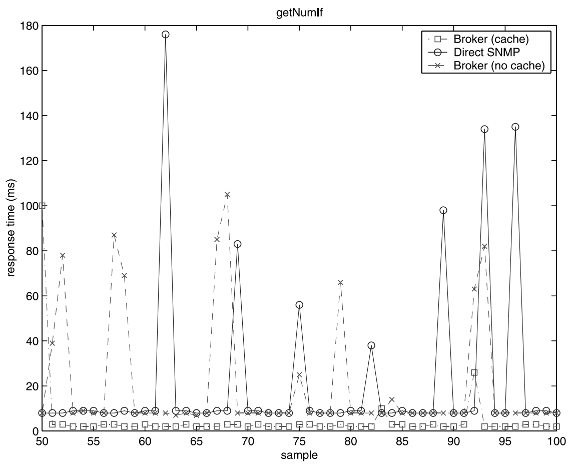

Fig. 4. Sample of the response time between active session and router for getNumIf.

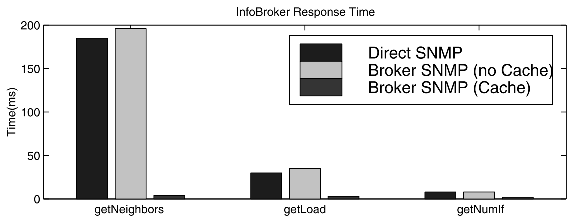

Fig. 5. Median response time between active session and router.

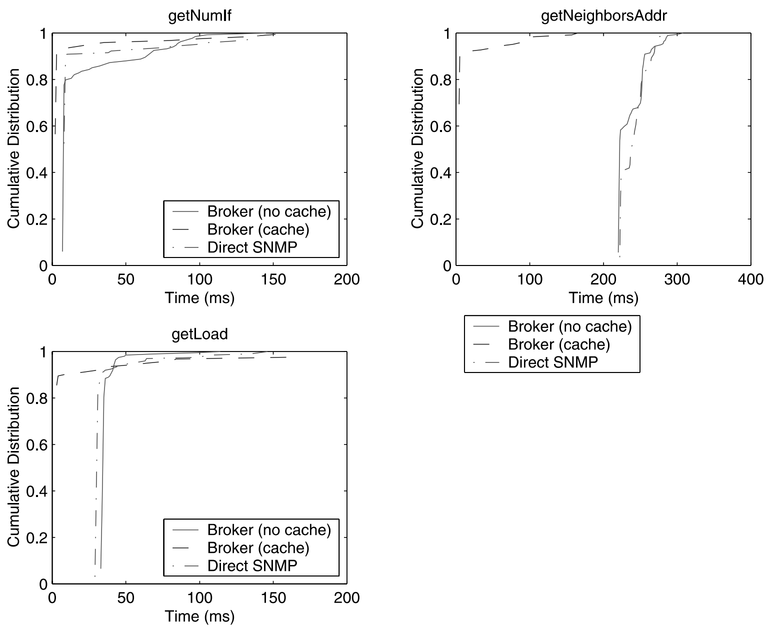

Fig. 6. Response time between active session and router – cumulative distribution.

In all three functions, using the cache decreased response latency. In the most complex function, `getNeighbors()`, the cache improved performance by approximately 98% over both Direct SNMP and Broker SNMP with no cache. This demonstrates that caching the right data can be extremely effective in reducing response latency, especially for complex functions.

Another important point to note in Fig. 5 is the difference between Direct SNMP and Broker SNMP (no cache) response times. Both monitoring channels are the same except for the IPC cost between the active session and the Info Broker. Thus, the difference of response times is the overhead of using the Info Broker. The Info Broker overhead is insignificant as shown by getNumIf() method which simply retrieves one atomic piece of state. (Our measurement tools could not measure time at a finer granularity than 1 ms.)

4. Security

As mentioned before, security and safety are crucial components of ABLE++. The separation between the active engine (AE) and the router plays a significant role in asserting safety. However, when an active session has the power to manipulate the router's state, and to divert non-active streams, separation is not sufficient. Therefore, we have built a multi-level mechanism to ensure the secure and safe operation of ABLE++.

The main idea is that sessions (at the thread or process level) will be isolated, much as the *sandbox* concept of JAVA applets. However, in our case, sessions may need to have access to resources such as the file system, communication to the outside world or access to the router's state. Thus, we have to tailor the right envelope around each session, in a way that will allow it to perform restricted actions in a controlled way, and deny the use of unauthorized services. We call this flexible envelope the *rubber box*.

As explained in Section 2.1, all the communication to or from the session is done through the Session Broker. This serves several purposes simultaneously: first, at each given time there is only a *single* copy of each session, additional code belonging to the same session will be sent to the session and will not create a new copy. It also prevents a session from interfering with other sessions' communication, or manipulating other sessions' code.

When a session is created, a list of authorized services is built in the session control block, and the required authentication is checked, using the ANEP header options [3]. The granularity of the authorized service list can be very fine, e.g., describing each of the Control Broker functions, or coarse with a few predefined security permission levels.

When the session's JAVA code is first run, we execute it inside a Class Loader, modified to work in the appropriate level. For example, if the session is unauthorized to access the file system, it will be run inside a loader that blocks all classes that access the file system. This adds a second level of safety around the session.

When services (such as Control Broker or Info Broker) are requested, the appropriate broker checks the authorized services list and acts accordingly.

In addition, we use an IP level packet filter for the outgoing packets. A packet that originated from one of the active sessions, and was *not* sent through the Session Broker, is blocked. This prevents a session from trying to communicate outside the execution environment without proper authorization.

The problem of deadlock prevention, and safe execution of the code of each session, is of course a very hard problem. We do not intend to address it here. Safe languages [11] and techniques like Proof Carrying Code [12] can be used to address some of these problems.

5. An application example – congestion avoidance

In the current Internet, congestion is treated by end-to-end flow control. The network signals to the end-points the existence of congestion. As a result, the end-points reduce the rate at which they inject new packets into the network. Usually congestion is signaled via packet drop, which is a wasteful process because the network already consumed resources in getting the packet to the point in the network where they were dropped. Recent suggestions to use explicit congestion notification (ECN) [6] allow signaling about congestion before packets must be discarded, enabling a more efficient use of the network bandwidth. In both cases, the network reaction time depends heavily on the algorithm at the end-points, namely TCP, and on the round trip delay of the various connections.

These end-to-end flow control mechanisms have proved to be stable and efficient; however, due to the long control loop they do not prevent transient congestion in which packets are dropped. We suggest a local congestion avoidance algorithm to augment current mechanisms. The main idea behind the algorithm is to find a temporary deflection route for a portion of the traffic passing through a congested link. The algorithm thus, eases the load at this link and reduces the amount of packet loss until the end-to-end flow control decreases the flow rates. It is important to note that the algorithm does not interfere with the routing algorithm and can work in conjunction with any existing routing algorithm. The deflection is done by adding entries to the *forwarding* tables, temporarily overriding the route chosen by the routing algorithm.

5.1. General description

After initiation, each node locally monitors the load on each of its outgoing interfaces. When an interface is identified as congested, e.g., by comparing the interface current transmission rate to the maximum transmission rate, the node tries to find deflection routes. The first step is to identify a destination (d) of some flow that passes through the congested interface. Then the node (denoted by c) sends to all its neighbors (not connected through the congested interface) the message $\texttt{CONGESTED}(c, c, d, 0)$. The first field identifies the congested node id, the second field is the message sender id, the third field is the chosen destination, and the last field is a counter denoting the hop distance of the sender from the congested node. A node, n, that receives a $\texttt{CONGESTED}(c, c, d, 0)$ can be either upstream from the sender, i.e., it forwards packets to destination d through node c, or it can be off-stream, meaning it forwards its packet to d via some other route. In the latter, node n sends node c the message $\texttt{ARF}(n, d)$ indicating that an Alternative Route was Found. In

```
1.  foreach interface $i$
2.    if $load(i) > threshhold$ then
3.      $d \leftarrow finddest(i)$
4.      foreach interface $j \neq i$
5.        send CONGESTED($c$, $c$, $d$, 0) to all $neighbor(j)$
```

Fig. 7. The pseudo-code for the load detection algorithm at node *c*.

```
1.  For CONGESTED($c$, $s$, $d$, $cnt$)
2.    if $nexthop(d) = c$ OR $nexthop(d) = s$ then // upstream
3.      if $cnt = 0$ then
4.        foreach $j \in neighbors()$
5.          if $j \neq c$ then
6.            send CONGESTED($c$, $n$, $d$, 1) to $j$
7.    else   // off-stream
8.      send ARF($n$, $d$) to $s$

9.  For ARF($m$, $d$)
10.   $settemproute(d, m)$
```

Fig. 8. The algorithm pseudo-code for node *n*.

the first case, node *n* propagates the search by sending message CONGESTED$(c, n, d, 1)$ to all its neighbors except *c*.

A node, n', that receives the message CONGESTED$(c, n, d, 1)$ sends ARF(n', d) to node *n* if the next hop to destination *d* is neither node *c* nor *n*. Otherwise, it ignores the message.

A node that receives the message ARF(n, d) adds a temporary static entry to its *forwarding* table indicating that the next hop to destination *d* is node *n*. This static route is automatically removed after a pre-configured time interval. The algorithm's pseudo-code appears in Figs. 7 and 8.

The algorithm design details and its global performance are beyond the scope of this paper. We concentrate here, on the services required from the active node environment to efficiently facilitate the algorithm's execution.

5.2. Implementation discussion

In this section, we discuss services the application receives from ABLE++. We believe these services are typical for many applications running above any NodeOS.

The simplest group of services contains functions that require atomic data such as the local host name. This is omitted for brevity from the code of Figs. 7 and 8. The application can learn its local machine name for the sender id (line 5 in Fig. 7, and lines 6 and 8 in Fig. 8) by calling Info Broker `getName()` function. Other functions of the group are defined to get the router OS or IP statistics. The main advantage of these simple services is the abstraction of the interface type, the simplification of obtaining the data (one single call), and, in case of static data such as hostname, the ability to use caching.

Next in complexity are services that require a modest amount of queries. An example in the code of Fig. 7 is *load*(i) which checks the load on interface *i*. In our implementation, this function has an optional parameter that defines the measurement granularity, by specifying the time difference between two queries that check how many bytes have been send (ifOutOctet). Together with the interface speed, one can calculate the load.

The most complex services are the ones involving neighbor lists. MIB-II [9] does not hold this list explicitly, and one needs to search for the neighbors in the tables that were designed for other purposes, e.g., in the IP routing tables. The task is further complicated by small variants in the implementation of these tables among manufacturers. As a result, obtaining the neighbor list (*neighbors*()) seems cumbersome and hard for an application developer. Other services we supply in this problem domain are the list of all neighbors attached to a specific interface (*neighbor*(j)), and the nexthop neighbor on the route to some destination (*nexthop*(d)).

In addition, we supply a service which is somewhat tailored to this specific application: *finddest*(i) returns some destination to which packet are routed through interface *i*. Our strong belief in local algorithms to solve global function suggests that other such applications can benefit from this service.

In order to react to problems by taking corrective actions, an application (with the right permission) must be able to control the router. In this example, the function *settemproute*(d, m) adds a temporary static route to node *d* through the neighbor *m*. This service is more than abstracting the CLI (command line interface) from the programmer. A session can specify the duration of the temporary route leaving the service responsible for deleting the static route after the specified time period has expired. In addition, the service validates that no conflicts between requests exist, and is responsible for eventually cleaning up the temporary changes even if the application was terminated incorrectly.

In general, control services are more complex than monitoring since (as we see in the example above) one needs to make sure that an application error will not have a long term negative effect on the router operation. Specifically, one needs to check for state coherency at all times. Thus, control services need to be at higher level of abstraction. To illustrate this, suppose an application wants to change the route to some destination, *d*. One way to do this is to supply the application with two services: $deleteroute(d)$ and $setroute(d)$. In this case, if due to an error the application terminates after deleting the route and before adding the new one packets to *d* will be routed through the default route which might cause routing loops. A better design of the interface is to supply a higher-level abstraction, say a service $forceroute(d)$ that performs the delete and set operation at the Broker Interface level.

6. Related work

The IEEE P1520 [1] is an ongoing effort to standardize the interface between the managed element and the control software. The current drafts define three levels of interfaces: CCM between the hardware and the control software (drivers), L (low level) between the control level software and the low level implementation software (IP routing, RSVP), and U (upper level) between the low level software and the high level services.

Since the IEEE P1520 is not standardized, yet, other standards are being used. Due to the wide spread of SNMP it is clearly a current attractive candidate for a standard agent-node interface. However, with a few exceptions [10,23] SNMP is missing from most past system design. Recently, there where several suggestion for agent systems that incorporate an SNMP interface [13,19,24].

The SmartPackets Project [18] at the BBN takes a language based approach to implementing an active network system. The active process, coded in capsule form, interacts with the device through built-in language constructs. The current implementation can query router supported MIBS using a *Mib-type*. The SmartPackets active network does not support persistant NM applications needed for long term management tasks, e.g., resource or load monitoring.

Distributed management framework (DMF) [5] by IBM Zurich is a distributed processing environment for management applications. DMF focuses on the mobility and communication of applications by augmenting the ObjectSpace Voyager [2] platform with a distributed directory service. DMF does not attempt to address the efficiency or security of the interaction between the application and the managed device.

7. Future work

Areas of future work for ABLE++ includes, among others, the following.

7.1. Dynamically extensible cache

We have shown that caching consolidated management information results in a performance boost at the application layer. Thus, the performance of the application is tied to the behavior and policies of the cache. Both the behavior and policies of our current cache implementation are static. Our cache has a predefined list of information to retrieve from the router at specified intervals. It would be interesting to look at how a "dynamically extensible cache" or "learning cache" could boost application performance over a wide variety of applications.

7.2. Distributed cache

ABLE++ currently focuses on the interaction between application (active sessions) and the local router. Since only local router state is available, applications must send messages to other active nodes to get a global picture of the network. Another approach to exporting global network state to a local application is to replace our current local cache with a distributed global cache.

7.3. Fine grain resource control

The session broker implements rudimentary resource control. When the execution environment is created for an active session, the session broker allocates a finite amount of resource (CPU, Bandwidth and memory). If the active session behaves outside the given constraints, the session broker simply terminates its execution. Unfortunately, such abrupt termination could result in a global instability of a distributed application. Currently, we leave it up to the application programmer to handle such situations.

8. Conclusions

In this paper, we have presented the NM distributed framework ABLE++. We have discussed various trade-offs including abstraction versus efficiency and application freedom versus security constraints. The design of ABLE++ allows it to be efficient while presenting a simple, abstract interface for programming ease. We use a cache to store correlated data at the device level. Caching at the local level reduces information retrieval latency, amortizes computation among applications and reduces resource consumption. Our interface abstracts instrumentation details from the application programmer without giving up performance. We have also added security mechanisms into the layered fabric of our architecture to prevent applications from harming the network. We have demonstrated the performance of ABLE++ with the following three performance metrics: reduced message count, efficient application retrieval time and low processing overhead. Finally, we have demonstrated the ease of programming using ABLE++'s interfaces by presenting a novel congestion-avoidance algorithm.

Acknowledgements

We would like to thank Jonathan T. Moore, Michael Hicks and Steve Muir for many helpful comments.

References

[1] IEEE P1520 homepage, http://www.ieee-pin.org/.
[2] Objectspace voyager homepage. http://www.objectspace.com/products/voyager/.
[3] D.S. Alexander, B. Braden, C.A. Gunter, A.W. Jackson, A.D. Keromytis, G.J. Minden, D. Wetherall, The active network encapsulation protocol (ANEP), URL http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt, 1997.
[4] I. Cidon, R. Rom, Y. Shavitt, Bandwidth reservation for bursty traffic in the presence of resource availability uncertainty, Computer Communications 22 (10) (1999) 919–929.
[5] M. Feridun, W. Kasteleijn, J. Krause, Distributed management with mobile components, in: The Sixth IFIP/IEEE International Symposium on Integrated Network Management, Boston, MA, May 1999.
[6] S. Floyd, TCP and explicit congestion notification, ACM Computer Communication Review 24 (5) (1994) 10–23.
[7] J. Jiao, S. Naqvi, D. Raz, B. Sugla, Toward efficient monitoring, IEEE Journal on Selected Areas in Communications 18 (5) (2000).
[8] R. Kawamura, R. Stadler, Active distributed management for IP networks, IEEE Communications Magazine 38 (4) (2000) 114–120.
[9] K. McCloghrie, M. Rose, Management information base for network management of TCP/IP-based internets: MIB-II, Mar. 1991, Internet RFC 1158.
[10] K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, Y. Yemini, Decentralizing control and intelligence in network management, in: The Fourth International Symposium on Integrated Network Management, May 1995.
[11] J.T. Moore, Mobile code security techniques, Technical Report MS-CIS 98-28, University of Pennsylvania, May 1998.
[12] G.C. Necula, Proof carrying code, in: POPL97, Paris, France, 15–17 January 1997, ACM press, New York, 1997.
[13] B. Pagurek, Y. Wang, T. White, Integration of mobile agents with SNMP: why and how, in: 2000 IEEE/IFIP Network Operations and Management Symposium, Honolulu, Hawaii, April 2000, pp. 609–622.
[14] K. Psounis, Active networks: applications, security, safety, and architectures, IEEE Communications Surveys 2 (1) (1999), http://www.comsoc.org/pubs/surveys/1q99issue/psounis.html.

[15] D. Raz, Y. Shavitt, An active network approach for efficient network management, in: IWAN '99, LNCS 1653, Springer, Berlin, Germany, 1999, pp. 220–231.

[16] D. Raz, Y. Shavitt, Active networks for efficient distributed network management, IEEE Communications Magazine 38 (3) (2000).

[17] M.T. Rose, The Simple Book: An Introduction to Networking Management, second ed., Simon and Schuster Trade, New York, 1998.

[18] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, C. Partridge, Smart packets for active networks, in: OPENARCH '99, New York, March 1999, pp. 90–97.

[19] P. Simoes, L.M. Silva, F. Boavida-Fernandes, Integrating SNMP into a mobile agent infrastructure, in: IEEE/IFIP DSOM '99, Zurich, Switzerland, October 1999.

[20] J.M. Smith, K.L. Calvert, S.L. Murphy, H.K. Orman, L.L. Peterson, Activating networks: a progress report, IEEE Computer 32 (4) (1999) 32–41.

[21] A. Vogel, K. Duddy, JAVA Programming with CORBA, second ed., Wiley, New York, 1998.

[22] Z. Wang, Routing and congestion control in datagram networks, PhD thesis, University College London, January 1992.

[23] Y. Yemini, S. daSilva, Towards programmable networks, in: Workshop on Distributed Systems Operations and Management, October 1996.

[24] M. Zapf, K. Herrmann, K. Geihs, Decentralised SNMP management with mobile agents, in: The Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99), May 1999, Boston, MA.

Jessica Kornblum is a Ph.D candidate at the University of Pennsylvania working with Professors Jonathan Smith and David Farber. She received her Master of Science degree from University of Pennsylvania in 1998 and her Bachelor of Arts degree in computer science with a minor emphasis in mathematics from DePauw University in 1997. While at Bell Laboratories/Lucent Technologies, she worked on the ABLE (Active Bell Labs Engine) project. Jessica has also worked on performance analysis techniques for IP networks at IBM Zurich. Her main research interest is in the intersection of Network Management and Active Network technologies.

Danny Raz received his doctoral degree from the Weizmann Institute of Science, Israel, in 1995. From 1995 to 1997 he was a post-doctoral fellow at the International Computer Science Institute, (ICSI) Berkeley, CA and a visiting lecturer at the University of California, Berkeley. Since October 1997 he is with the Networking Research Laboratory at Bell-Labs, Lucent technologies. From October 2000, Danny Raz will join the faculty of the computer Science department at the Technion, Israel. His primary research interest is the theory and application of management related problems in IP networks.

Yuval Shavitt received the B.Sc. in Computer Engineering (cum laude), M.Sc. in Electrical Engineering and D.Sc. from the Technion – Israel Institute of Technology, Haifa in 1986, 1992 and 1996, respectively. From 1986 to 1991, he served in the Israel Defense Forces first as a system engineer and the last two years as a software engineering team leader. After graduation he spent a year as a Postdoctoral Fellow at the Department of Computer Science at Johns Hopkins University, Baltimore, MD. Since 1997 he is a Member of Technical Stuff at the Networking Research Laboratory at Bell-Labs, Lucent Technologies, Holmdel, NJ. From October 2000, Dr. Shavitt is also a faculty member in the department of Electrical Engineering at Tel-Aviv University. His recent research focuses on active networks and their use in network management, QoS routing and Internet mapping and characterization. He serves as TPC member for INFOCOM 2000, 2001 and 2002, and on the executive committee of INFOCOM 2000 and 2002.