# The Active Process Interaction with its Environment

Jessica Kornblum[1]*, Danny Raz[2], and Yuval Shavitt[2]

[1] Computer and Information Science
Univ. of Pennsylvenia
jkornblu@dsl.cis.upenn.edu
[2] Bell Labs, Lucent Technologies
Holmdel, NJ
{raz, shavitt}@research.bell-labs.com

**Abstract.** Adding programmability to the interior of the network provides an infrastructure for distributed applications. Specifically, network management 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 fundemental problems associated with this interaction.
In this paper, we study the basic tradeoffs associated with the interaction between an active process and its environment and presenting ABLE++ as an example architecture. Most notably, we explore two design tradeoffs, *efficiency vs. abstraction* and *application flexibility vs. security*. We demonstrate the advantages of the architecture by implementing a congestion avoidance algorithm.

## 1 Introduction

In active networks [12], network elements, such as routers, are programmable. Code can be sent in-bound 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 [17], 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 [16, 14, 6], 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 [14].

* This work was done while in Bell-Labs, Lucent Technologies.

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) [15], 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 simple programming constructs, such as Corba [18], but is often inefficient. Another design tradeoff we address with our system is application flexibility versus security vulnerabilities. For example, 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.

To make the discussion clearer and more concrete, we demonstrate our design by describing novel extensions to the ABLE architecture [13], ABLE++. ABLE is an active network architecture that allows the easy deployment of distributed network management (NM) applications. In ABLE++, we optimize the interaction between the active process 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 load from the router, shortens the retrieval time for applications, and simplifies the 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 [19]. 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 [3]) 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.

*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 [8, 20] SNMP is missing from most past system design. Recently, there where several suggestion for agent systems that incorporate an SNMP interface [21, 11].

*Organization* In the next 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.

## 2 Architecture

Network management 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 Figure 1): a *Session Broker*, an *Info Broker* and a *Control Broker*.

"Brokers" are used to complete a task on behalf of a requester. We have specialized the concept of a "Broker" to handle three different active processing and network management tasks. The Session Broker creates, manages, and cleans up the processing environment as well as manages active session. The Info Broker provides an efficient monitoring channel by exporting local state to active sessions and mediating 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.

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

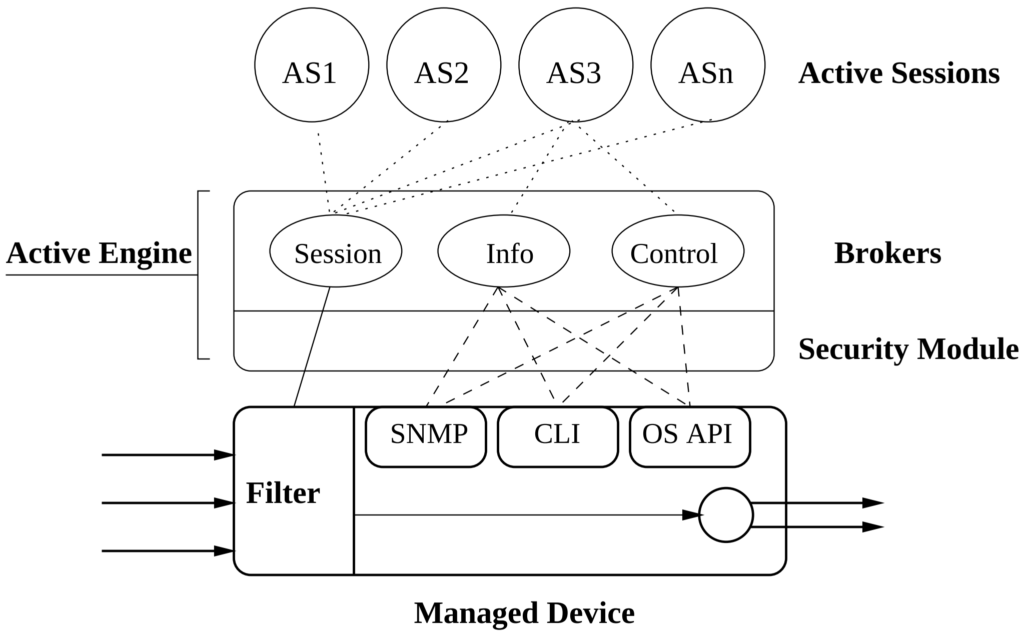


**Fig. 1.** ABLE++

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 [14]. Therefore, we will only go into a brief discussion. Both ABLE and ABLE++ architectures are designed to support long lived, mobile active sessions. Network management 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 twoparts: *managing active sessions* (the meta-Broker) and *exporting mechanisms for communication and mobility.*

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

*Communication and mobility* The Session Broker controls active session communication and mobility through an exported API (See Table 1). *`Distribute()`* and

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

**Table 1.** Communication and Mobility Interface

*`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 network management. 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 Figure 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.

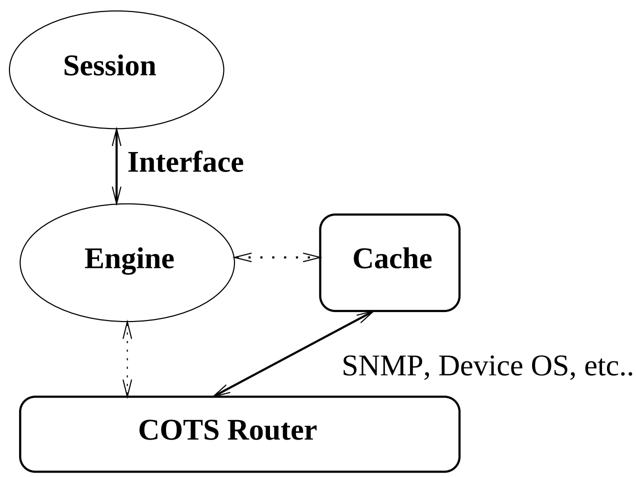


**Fig. 2.** Info Broker

*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 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?

*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 possibly several round-trip transmit times prevents the application from 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.

*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

| 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 address 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 |

**Table 2.** Info Broker Interface methods

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 [5]). Another option is to update staled 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 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 allow 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 preform 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* milliseconds. The function returns immediately and does not wait until the new route has been cancelled. 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

In this section, we demonstrate the efficiency of the Info Broker by analyzing the response time between an active session and a managed router's SNMP agent. We are using the following three functions, each differing 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 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 and for each of the following 3 monitoring channels (total of 9 trials):

**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.

**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 interprocess communication (IPC) between the session and the Info Broker.)

**Broker SNMP (Cache)** opens a connection with the Info Broker and polls the local Cache for computed result.

Figure 3 displays the median response times for the different monitoring channels and functions. Our implementation of the Info Broker as well as active

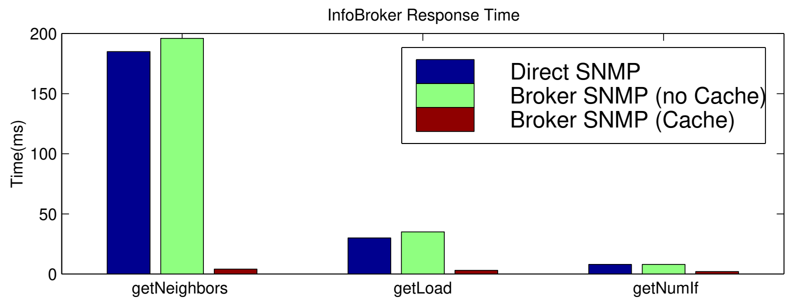


**Fig. 3.** Response Time Between Active Session and Router

sessions run on JVM 1.1.8, 200 MHz Pentium PC's 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. We suspect the increase is due to Java garbage collection and application thread scheduling algorithms in the JVM. Therefore, we have plotted the median of the 200 values to better represent our data. 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.

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 Figure 3 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 [2]. 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 pre-defined 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 [9] and techniques like Proof Carrying Code [10] can be used to address some of these problems.

## 5 An Application Example - Congestion Avoidance

In 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) [4] 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 `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 `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 `ARF`$(n,\ d)$ indicating that an **A**alternative **R**oute was **F**ound. In 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 figures 4 and 5.

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.

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

**Fig. 4.** 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 ∈ neighbors()
5.          if j ≠ c then
6.            send CONGESTED(c, n, d, 1) to j
7.    else    // offstream
8.      send ARF(n, d) to s

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

**Fig. 5.** The algorithm pseudo-code for node $n$.

### 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 Figures 4 and 5. The application can learn its local machine name for the sender id (line 5 in Figure 4, and lines 6 and 8 in Figure 5) 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 Figure 4 is $load(i)$ which checks the load on interface $i$. In our implementation, this function has an optional parameter that define 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 [7] 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 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 suggest 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.

**Acknowledgement**

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. D. S. Alexander, B. Braden, C. A. Gunter, A. W. Jackson, A. D. Keromytis, G. J. Minden, and D. Wetherall. The active network encapsulation protocol (ANEP). URL `http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt`, 1997.
3. I. Cidon, R. Rom, and Y. Shavitt. Bandwidth reservation for bursty traffic in the presence of resource availability uncertainty. *Computer Communications*, 22(10):919–929, June $25^{th}$ 1999.
4. S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, Oct. 1994.

5. J. Jiao, S. Naqvi, D. Raz, and B. Sugla. Toward efficient monitoring. *IEEE Journal on Selected Areas in Communications*, 18(5), May 2000.
6. R. Kawamura and R. Stadler. Active distributed management for ip networks. *IEEE Communications Magazine*, 38(4):114–120, Apr. 2000.
7. K. McCloghrie and M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II, Mar. 1991. Internet RFC 1158.
8. K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, and Y. Yemini. Decentralizing control and intelligence in network management. In *The Fourth International Symposium on Integrated Network Management*, May 1995.
9. J. T. Moore. Mobile code security techniques. Technical Report MS-CIS-98-28, University of Pennsylvania, May 1998.
10. G. C. Necula. Proof carrying code. In *POPL97*, Jan. 1997.
11. B. Pagurek, Y. Wang, and T. White. Integration of mobile agents with SNMP: Why and how. In *2000 IEEE/IFIP Network Operations and Management Symposium*, pages 609 – 622, Honolulu, Hawaii, USA, Apr. 2000.
12. K. Psounis. Active networks: Applications, security, safety, and architectures. *IEEE Communications Surveys*, 2(1), First Quarter 1999. http://www.comsoc.org/pubs/surveys/1q99issue/psounis.html.
13. D. Raz and Y. Shavitt. An active network approach for efficient network management. In *IWAN'99, LNCS 1653*, pages 220 –231, Berlin, Germany, 1999.
14. D. Raz and Y. Shavitt. Active networks for efficient distributed network management. *IEEE Communications Magazine*, 38(3), Mar. 2000.
15. M. T. Rose. *The Simple Book: An Introduction to Networking Management.* Simon & Schuster Trade, 2nd edition, 1995.
16. B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart packets for active networks. In *OPENARCH'99*, pages 90–97, New York, NY, USA, Mar. 1999.
17. J. M. Smith, K. L. Calvert, S. L. Murphy, H. K. Orman, and L. L. Peterson. Activating networks: A progress report. *IEEE Computer*, 32(4):32–41, Apr. 1999.
18. A. Vogel and K. Duddy. *JAVA Programming with CORBA*. Wiley, 2nd edition, 1998.
19. Z. Wang. *Routing and Congestion Control in Datagram Networks.* PhD thesis, University College London, Jan. 1992.
20. Y. Yemini and S. da Silva. Towards programmable networks. In *Workshop on Distributed Systems Operations and Management*, Oct. 1996.
21. M. Zapf, K. Herrmann, K. Geihs, and J. Wolfgang. Decentralised snmp management with mobile agents. In *sixth IFIP/IEEE International Symposium on Integrated Network Management - IM'99*, May 1999. Boston, MA, USA.