To appear in IEEE NETWORK Magazine Special Issue on Active and Programmable Networks, July 1998

Introducing New Internet Services: Why and How

David Wetherall, Ulana Legedza, and John Guttag*

Software Devices and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

Active networks permit applications to inject programs into the nodes of local and, more importantly, wide area networks. This supports faster service innovation by making it easier to deploy new network services. In this paper, we discuss both the potential impact of active network services on applications and how such services can be built and deployed. We explore the impact by suggesting sample uses and arguing how such uses would improve application performance. We explore the design of active networks by presenting a novel architecture, ANTS, *that adds extensibility at the network layer and allows for incremental deployment of active nodes within the Internet. In doing so,* ANTS *tackles the challenges of ensuring that the flexibility offered by active networks does not adversely impact performance or security. Finally, we demonstrate how a new network service may be expressed in* ANTS.

Keywords: active networks, networking protocols, distributed applications, performance.

1 Why Active Networks?

The pace of innovation in networked applications is unrelenting. New applications continue to emerge rapidly and often benefit from new network services that better accommodate their modes of use. In this paper, we address the questions of why and how to deploy such new services. We begin by observing that while it is possible to deploy new network services at end-systems (e.g., as overlays), implementing them at nodes interior to the network or at the network layer often offers better functionality and performance. This observation is supported by a number of *ad hoc* efforts to exploit such functionality:

- Multimedia applications (such as videoconferencing and Internet telephony) benefit from real-time and multicast services. For example, RSVP [3] reserves bandwidth to ensure that time-sensitive data is delivered in a timely fashion, and IP Multicast [4] reduces the bandwidth needed to communicate from one sender to multiple receivers. In the case of RSVP, bandwidth reservation functionality cannot be provided effectively above the network layer. In the case of multicast, excess bandwidth and latency costs are incurred when an overlay is used and its topology does not match the underlying topology, as was problematic in the early MBONE.
- Laptops benefit from host mobility and transport services optimized for wireless transmission. For example, Mobile IP [18] allows a laptop to be reached at different sites without the need to reconfigure address information, and Snoop-TCP [1] compensates for the fact that the congestion control mechanisms of TCP were not designed for lossy media. Mobile IP is by definition a network-level routing service; Snoop-TCP intercepts TCP packets within the network at wireless base stations.
- Web servers benefit from caching and load distribution services, e.g., Cisco's CacheDirector product reduces the amount of wide-area traffic by intercepting repeated requests, and Cisco's LocalDirector product reduces the concentration of Web traffic by distributing requests across multiple servers. By intercepting packets at routers, these products are transparent to end systems and minimize latency and bandwidth usage compared to a proxy agent.

Unfortunately, the current process of changing network protocols and processing is lengthy and difficult because it requires standardization and manual, backwardly-compatible deployment. The first two groups of examples (which are still not fully deployed despite many years and evident need) show the effects of this process. The Cisco products, on the other hand, appeared relatively quickly because they are implemented as vendor-specific products that are transparent to end-systems. This strategy may be effective for deploying a new network service at a single router, but is too restrictive to be suitable as means of widely deploying new network services.

*{djw, ulana, guttag}@lcs.mit.edu. http://www.sds.lcs.mit.edu/.

Active networks seek to address the problem of slow network service evolution by building programmability into the network infrastructure itself, thus allowing many new network services to be introduced much more rapidly. Compared with other architectures that incorporate flexibility, such as Cicso's Tag Switching [10], this approach offers a larger degree of flexibility, but at the potential expense of performance and security.

We believe that the ability to tailor network service to the application will ultimately justify the overhead necessary to protect the network as a whole. Accordingly, our research in active networks is focused on two questions:

- In what ways can new network services be used to improve application performance?
- How is it possible to make the network infrastructure programmable without compromising local forwarding performance and network security properties?

In this paper we outline our thinking to date to provide a broad introduction for the casual reader. Other papers [20, 11, 12] contain further technical detail and quantitative results about individual topics.

We begin by presenting, in section 2, some sample applications that illustrate the type of new services that we wish to be able to introduce. In section 3, we then make the case that additional work in the network may sometimes be more effective than additional work by the application. In section 4, we argue that programmability need not come at an unreasonably large performance and security cost by describing our ANTS architecture and emphasizing the tradeoffs that it embodies. In section 5, we show, by example, how new services may be deployed in our prototype implementation of the architecture. We conclude by placing our research in the context of related work.

2 Sample Applications

Here, we outline two new and different network services aimed at improving existing applications. Our intent is to suggest the diversity of services facilitated by active networking. We present these services in just enough detail to show how they can be implemented on an active network. Both are to some degree application-specific rather than general-purpose. Neither is implemented in the network layer today.

2.1 Stock quotes

Many people use the Internet to obtain stock quotes. Fast access to up-to-date quotes is crucial, especially during periods of heavy server load (because it usually signifies an important market fluctuation). Web caches (e.g. Squid[19]), which are used to reduce latency and increase the throughput of accesses to servers, are not helpful in this context. First, most Web caches do not cache quotes because they are dynamic data. Second, even if the difficulty of caching rapidly changing data were overcome, the granularity of objects stored in Web caches (i.e., entire Web pages) is inappropriate for this application. Each client generally requests a page with a short customized list of quotes. With hundreds of stocks, the number of possible unique Web pages is enormous, so the likelihood of a cache hit is small.

In an active network, the caching strategy may be customized to suit the application needs. First, an active protocol specialized for stock quotes can cache quotes at network nodes using a per-stock name granularity. This allows all requests for popular (i.e., cached) stock quotes to hit in the cache, no matter what subset or order is requested. Second, requests can specify a client-controlled degree of currency (i.e., up-to-date-ness) of the desired quotes. This allows each client to trade response-time against the currency of the quotes. For example, some clients may prefer near-instantaneous access to quotes that are up to 15 minutes out of date, while other clients may require the latest quotes, even if it means waiting longer to get them.

The basic idea is that quotes are cached (along with timestamps) at network nodes as they travel from the server to a client. Subsequent client requests are intercepted at the nodes where the local cache is checked to determine whether the desired quotes (with desired degree of currency) are available. If so, the quotes are sent to the client, where they are assembled into a viewable web page. If not, the request is forwarded to the server.

2.2 Online Auctions

Web servers hosting online auctions are currently among the most popular sites in the Internet [6]. A server running a live online auction collects and processes client bids for the available item(s). This server also responds to requests for the current price of an item. Because of the network delay experienced by a packet responding to such a query, its information may be out of date by the time it reaches a client, possibly causing the client to submit a bid that is too low to beat the current going price. Thus, unlike auctioneers in traditional auctions, the auction server may receive bids that are too low and must be rejected, especially during periods of high load when there are many concurrent bids.

Current implementations of such servers [6] [17] perform all bid processing at the server. In an active network, low bids can be filtered out in the network, before they reach the server. This capability can help the server achieve high throughput during periods of heavy load. When the server senses that it is heavily loaded, it can activate filters in nearby network nodes and periodically update them with the current price of the popular item. The filtering active nodes drop bids lower than this price and send bid rejection notices to the appropriate clients. This frees up server resources for processing competitive bids and reduces network utilization near the server. The filtering active nodes could also keep track of the number of rejected bids at each price, and ship those to the auction server at the end of the auction.

The auction server is similar to the stock quote application in that it also performs caching (of current price information) in network nodes. However, its protocol is necessarily different because it also delegates application-specific tasks, e.g., bid rejection, to the active network nodes. More details about how to implement this service are provided in section 5.

3 Rethinking Performance

Paradoxically, despite increasing the amount of processing performed within the network, the sample applications above can lead to improved overall system performance.

Traditional network performance measures, such as throughput (bits or packets per second) and packet latency, are aimed at evaluating the performance of the network rather than the performance of the applications using it. However, it is not necessarily the case that network performance is positively correlated with application performance. An active network can perform operations that can cause fewer packets to be sent or delivered and packets to experience longer per-hop latencies. While these effects would appear to degrade performance, they may actually result in improved overall application performance because of reduced demand for bandwidth at endpoints, reduced network congestion, etc. Therefore, performance should be evaluated in terms of application-specific metrics.

We first consider application-specific notions of throughput. In the stock quote example, the active approach clearly reduces the server's throughput in terms of client requests processed per second at the server, but increases the number of client requests serviced per second. In the active online auction, the relevant measure could be either the number of bids processed per second or, perhaps, the total number of *winning* bids processed per second. Preliminary experiments with this application indicate that the active implementation will increase both these measures. Both of these improvements in throughput are brought about by parallelism resulting from delegating some of the application's functionality to internal network nodes.

While active processing slows down packets slightly, this time can be recovered by improved latency of application-level operations. Caching in the network, as in the stock quote example, can reduce the latency of data accesses when the server is busy. When network nodes in the auction application reject low bids, they inform the "losing" end nodes more quickly than could the overloaded (and farther away) server.

The cost of these performance improvements is the increased consumption of computational and storage resources in the network, which may slow down other network traffic traveling through the busy active nodes. However, this competing traffic could also benefit from active processing. Because the active processing can reduce the bandwidth utilization in some regions of the network, other traffic will benefit from the resulting reduction in congestion-related loss and delays. This claim is supported in [11], which reports on simulations of a stock quote server, and in [12], which reports on simulations of an active reliable multicast protocol.

Sometimes, doing work within the network also reduces the total amount of work that needs to be done by an application. Consider the following example. Some number of sensors (e.g. microphones, antennas, devices measuring emissions of pollutants) are continuously collecting large amounts of information that must be combined (mixed) for one or more receivers. An active network implementation offers the opportunity to reduce the work done at end nodes by more than it increases the work within the network, i.e., when there are multiple receivers there is a reduction in total work as well as in endpoint work. Consider a situation in which N sources send signals to M destinations. If each end node does all of its own mixing, the work, summed over all end nodes, is proportional to $N*M$. In the best case, by mixing pairs of signals within the network the end nodes can be completely freed of the need to mix signals. Furthermore, the total amount of mixing done can be reduced – in the best case to N.

The degree to which intra-network processing improves performance depends on where in the network it is deployed. In the stock quote example, it is important to place the caches where they will serve a large number of client requests; otherwise, they are not very effective. In the sensor fusion example, the greatest decrease in bandwidth utilization occurs when the splitting of multicast streams is performed as late as possible and mixing is performed as early as possible. In the online auction, filters should be far enough away from the server to turn back low bids as early as possible, but close enough to the server to get reasonably up-to-date price information. In all our examples, placing processing and storage near a bottleneck link is likely to decrease delay and loss due to congestion. Note that placement strategy not only differs from application to application, but is also likely to change with change in network load. Fortunately, the flexibility offered by an active network enables the implementation of dynamic and application-specific placement policies.

4 How to Introduce New Services

We now turn to the question of how a network may support the deployment of the kind of new services discussed in section 2. This is a difficult design problem that must balance flexibility against service expectations: the architecture must be expressive enough to accommodate new services, but restrictive enough to allow performance goals and security properties to be met.

Our work on the ANTS architecture is focussed on exploring this tension. It presents a point in the design space that we believe is a workable starting point for a real active network. Further, it allows for incremental deployment in the Internet. In this section, we outline our architecture; greater detail is provided in [20]. We begin with an overview of its goals and service programming model, and then describe the main components of the system. In doing so, we emphasize the tradeoffs that we have made.

4.1 ANTS Overview

Like most networks, an ANTS-based network consists of an interconnected group of nodes that execute a common runtime; the nodes may be connected across the local or wide area and by point-to-point or shared medium channels. The system builds on the link layer services of the channels to provide network layer services to distributed applications.

Unlike IP, the network service provided by ANTS is not fixed – it is flexible. In addition to providing IP-style routing and forwarding as the default network-level service, ANTS allows applications to introduce new protocols into the network. Applications accomplish this by specifying the routines to be executed instead of IP forwarding at the active network nodes that forward their messages. In effect, applications may push a portion of their processing into the network – either processing such as caching that is traditionally performed at end-systems or novel kinds of processing such as bid rejection that only make sense in the context of active networks.

In designing ANTS, we set three goals for network protocol innovation. All describe more flexible forms of innovation than are currently achieved in the Internet. The network should support:

- Simultaneous use of a variety of different network layer protocols.
- The construction and use of new protocols by mutual agreement among interested parties, rather than requiring new protocols to be registered in a centralized manner.
- The dynamic deployment of new protocols, since it is unreasonable to take portions of the network "off-line" in order to reconfigure them, especially as the scale of the network increases.

Our architecture meets these goals through the use of three key components.

- The packets found in traditional networks are replaced by *capsules* that refer to the processing to be performed on their behalf at active nodes. Capsule types that share information within the network are grouped into *protocols*; a protocol provides a service and is the unit of network customization and protection.
- Selected routers within the Internet and at participating end nodes are replaced by *active nodes* that execute the capsules of a protocol and maintain protocol state. Unlike ordinary routers, active nodes provide an API for capsule processing routines, and execute those routines safely by using operating system and language techniques.
- A *code distribution* mechanism ensures that capsule processing routines are automatically and dynamically transfered to the active nodes where they are needed. This component does not exist in traditional networks, and is handled by the system, not the service programmer, in our network.

Each of these components is described more fully in the following section. The model that they collectively support for programming new network services is a generalized form of packet forwarding. It has the following characteristics:

- The forwarding routine of a capsule is set at the sender and may not change as it traverses the network; nor may capsules belonging to one protocol create capsules or access state belonging to a different protocol within the network. Given this, one user may not control the processing of another user's capsules in unintended ways.

- Some active nodes may elect not to execute particular forwarding routines, depending upon the node's available resources and security policies. When this happens, the node performs "default" IP-like forwarding on these capsules. Additionally, forwarding routines may self-select nodes at which it is useful to perform their specialized processing depending on the location of the node and its capabilities.

- Since forwarding routines may be defined by untrusted users, they are limited in their capabilities. In particular, like traditional forwarding routines, they are expected to run to completion locally and within a short time. Further, their global memory and bandwidth consumption is bounded by a TTL (Time-To-Live) scheme.

To develop a new service with this model, the service developer defines the different capsule types and their processing routines as a protocol structure. To use a new service, an application need only supply the protocol definition to the local node and start sending and receiving capsules of the appropriate types.

Service programming requires consideration of how processing will interact with packet loss, changing routes, protocol state loss, and concurrency. Based on our experience with this model, we are developing guidelines for constructing new services. While active networks clearly do not make the task of protocol design any easier, this difficulty should not be seen as a barrier to the widespread use of active protocols: we do not expect all users to construct new protocols directly, but rather to choose between protocols offered by third party software vendors. On the other hand, it is not necessary for protocol designers to consider how to transfer the processing routines themselves around the network, nor worry about interactions with other protocols (except indirect resource consumption).

This approach embodies several tradeoffs in order to be flexible enough to support novel protocols, while being restrictive enough to guarantee some level of protection, allocation and performance of shared resources in an untrusted environment. Protection is based on the inability to specify processing for another user's packets and the encapsulation of protocol state by the associated capsule processing routines. Allocation is based on the limited resources that will be granted to each packet by nodes. Performance is based on the simple event-driven processing model and the ability to tailor processing to the diversity of heterogeneous networks, including those in which only some nodes may be active.

protocol/ capsule	common header	rest of header ... payload

Figure 1: Capsule Format

4.2 Components of the Architecture

The three architectural components that we have mentioned support the ANTS programming model. We now describe each of them in turn.

4.2.1 Protocols and Capsules

The basis for customizing network processing is the protocol, which is composed of a set of related capsule types. The format of capsules as they are carried across a generic link-layer channel is sketched in Figure 1. When deployed in an IP network with a mix of active and IP-only routers, the format must be compatible with the IP packet format. IP-only routers then appear to be active routers that have elected not to run additional services. Murphy [14] describes how this can be accomplished by carrying ANTS-only fields as IP options.

The most important architectural function of the capsule format is to contain an identifier for a protocol and forwarding routine within that protocol. This identifier is based on a fingerprint of the protocol code. For example, the MD5 message digest algorithm converts an arbitrarily long sequence of bytes to a short and (with extremely high probability) distinct identifier. This identifier is used for demultiplexing to a forwarding routine in the same sense as the Ethernet type and IP version and protocol fields.

That the capsule identifier is derived from the code description of the protocol of which it is a part is crucial for two reasons:

- It greatly reduces the danger of protocol spoofing. This is because a fingerprint based on a secure hash is effectively a one-way function from code to identifier. Each active node can verify for itself (and without trusting external parties) that a particular set of programs maps to a given identifier.

- It allows protocols and capsule types to be allocated quickly and in a decentralized fashion, since their identifier depends only on a fingerprint of the protocol code.

Because of these properties, our architecture uses protocols as the unit of protection, preventing one protocol from interfering with the state of another within active nodes. Per protocol protection provides a much more efficient model than per user protection (since user authentication for each data manipulation is not required), yet still provides semantics that we have found to be useful. Within a protocol, however, it is up to the protocol programmer to separate information about concurrent sessions; to do otherwise would require that the architecture understand the definition of a session, which is application dependent.

Some forwarding routines are "well-known" in that they are guaranteed to be available at every active node. These primarily include routines for common case processing, i.e., unreliable data transfer with standard routing, and for bootstrapping network services, such as the code distribution scheme to be described shortly. Other routines are "application-specific." Typically, they will not reside at every node, but must be transfered to a node by the code distribution scheme before the first capsules of that type can be processed.

The remainder of the capsule format is comprised of a common header that contains fields present in all capsules, a type-dependent header that may be updated as the capsule traverses the network, and a payload. The important components of the common header are source and destination addresses and information about resource limits to be enforced by nodes. We use IPv4 style addresses for convenience.

4.2.2 Active Nodes

A key difficulty in designing a programmable network is to allow nodes to execute user-defined programs while preventing unwanted interactions. Not only must the network protect itself from runaway protocols, but it must offer co-existing protocols a consistent view of the network and allocate resources between them.

Our approach has been to execute protocols within a restricted environment that limits their access to shared resources. Active nodes play this role in our architecture. They export an API for use by application-defined processing routines, which combine these primitives using the constructs of a general-purpose programming language rather than a more restricted model, such as layering. They also supply the resources shared between protocols and enforce constraints on how these resources may be used as protocols are executed. We describe our node design along these two lines.

Capsule Forwarding API

We chose an initial API based on our experience with a predecessor system [21]. This work suggests that a relatively small set of operations is sufficient to express a number of different and useful forwarding routines. We currently support the categories of node primitives listed below. There are also some obvious additions (namely authentication, fingerprinting, compression, etc.) that we plan to experiment with in the future.

- *environment access*, to query the node location, state of links, routing tables, local time and so forth;
- *capsule manipulation*, with access to both header fields and payload;
- *control operations*, to allow capsules to create other capsule and forward, suspend or discard themselves;
- *storage*, to manipulate a soft-store of application-defined objects that are held for a short interval.

The node API is important because it determines the kinds of processing routines that can be composed by applications. For example, without the ability to store and access node state, individual capsule programs would be unable to communicate with each other. Further, the compactness and execution efficiency of capsule programs is affected by these primitives. Both are enhanced if the primitives are a good match for the processing, and degraded otherwise. For example, the neighbors at a given node may be found either by walking the entire routing table looking for adjacent nodes, or by asking the question directly of the node, depending on which topological queries are supported. The direct query can be represented compactly and executed efficiently as a built-in node primitive, while the other program cannot.

Capsule Execution

The node must provide an environment that executes the processing routines that use this API while meeting network security and resource management goals. Since capsule processing resembles a distributed programming system in which there are many legitimate users with small tasks, authentication and other traditional security schemes are likely to be too heavyweight to be used for common-case forwarding programs. Instead, we rely on the safety mechanisms of mobile code technologies

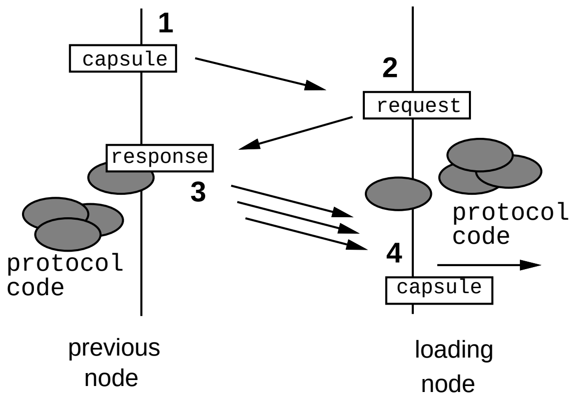

Figure 2: Demand Loading of Protocol Code

(e.g., sandboxing and Java bytecode verification) to execute untrusted routines efficiently in a contained manner. Conversely, the occasional use of primitives that manipulate shared logical resources, e.g., updates to the system-wide routing tables, must be authenticated.

In addition, the node limits the resources that capsule programs may consume (both at individual nodes and across nodes) and handles any errors that arise. Recall that the capsule format includes a resource limit that functions as a generalized TTL (Time-To-Live) field. This limit is carried with the capsule and decremented by nodes as resources are consumed; only nodes may alter this field, and nodes discard capsules when their limit reaches zero. In order to reason about total resource bounds, care must be taken to transfer resources when one capsule creates another inside the network. It is straightforward to charge for resources as they are consumed. Processing time and link bandwidth are allocated by time and capsule quanta, respectively; node memory is allocated by cached objects, since caching converts memory into a renewable resource. We hope, however, that it will prove feasible to enforce static limits at nodes with a scheme similar to [5] or by using proof-carrying code techniques [15].

4.2.3 Code Distribution

The third component of our architecture is a code distribution system. Given a programmable infrastructure, a mechanism is needed for propagating program definitions to where they are needed. A good scheme must be scalable and efficient, adapt to changes in node connectivity, and limit its activity so that the network remains robust.

Many different mechanisms are possible. At one extreme, programs may be carried within every capsule. This scheme is only suited to transferring extremely short programs when bandwidth is not at a premium. At the other extreme, programs may be pre-loaded into all nodes that may require them by using an out-of-band or management channel prior to using a new protocol. This scheme is not suited to our goals of rapid and decentralized deployment.

Instead, our approach has been to couple the transfer of code with the transfer of data as an in-band function. We believe this has several advantages. It limits the distribution of code to where it is needed, while adapting to node and connectivity failures. It improves startup performance and facilitates short-lived protocols by overlapping code distribution with its execution.

We have designed a scheme that is suited to flows, i.e., sequences of capsules that follow the same path and require the same processing. At end-systems, applications may begin to use a new protocol at any time by registering the code definition at their local node. Capsules of the new type may then be injected into the network and received from it. At intermediate nodes, caching of protocol code is used to achieve high performance by amortizing the cost of code loading across all of the capsules of a flow. To fill the caches initially, a lightweight protocol is used to transfer protocol code incrementally from one node to the next in response to capsules of that protocol traveling through nodes of the network. This code distribution protocol is described below and shown in Figure 2.

1. When a capsule arrives at a node, a cache of protocol code is checked. If the required code is not all present, a load request for the missing portion based on the capsule type and protocol is sent to the "previous" node, i.e., the last active node in the capsule's path. The capsule execution is suspended, awaiting the code, for a finite time.

2. When a node receives a load request that it can answer, it does so immediately. It sends load responses that contain the portion of protocol code that is implicated.

3. When a node receives a load response, it incorporates the code into its cache. If the required code is now all present, it wakes sleeping capsules. If the required responses are not forthcoming within some time bound, sleeping capsules are discarded without further action.

This scheme embodies an important tradeoff compared with simply using TCP to transfer code when it is needed. Under normal load, either our protocol or a reliable transport should work well. Our scheme suffers from the apparent disadvantage that load requests or responses that are lost or too slow will result in capsule loss and require the intervention of end-to-end reliability mechanisms (as does congestion loss today). However, our scheme has the property that (given a bounded protocol size) the amount of processing and bandwidth that the network will expend loading code is bounded to within a constant factor of that used for forwarding. Further, this work is localized because the loading is done incrementally. The intent is to ensure that the network will remain robust under high load.

We believe that our code distribution scheme has qualities that will prove it efficient, adaptive, and robust, though this must be borne out by experimentation. In order for it to best accommodate the largest number of scenarios, we also include a number of special cases. First, for very small protocols, the code may be carried along with every capsule if desired. Second, capsules may be constructed to "prime" a path with protocol code to reduce the startup period. Finally, popular protocols may simply be preloaded to avoid dynamic code distribution.

5 Developing New Services

The architecture just described has evolved in parallel with an implementation of a toolkit. In this section, we briefly describe our ANTS toolkit and how it may be used to develop a new service.

5.1 The ANTS Toolkit

The ANTS toolkit provides both an active node runtime and support for combining nodes into a network complete with distributed applications. It is written entirely in Java and runs as a user-level process on commodity hardware [1]. While we do not expect to run user-level Java on real routers (a faster and equivalent binary version would be needed), we have found our prototype to be useful for the purposes of research and experimentation.

We chose Java bytecodes and classfiles as a transfer format for processing routines because of Java's support for safety and mobility and the likely emergence of higher performance runtimes for evaluating it. Java's flexibility as a high-level language and support of dynamic linking/loading, multi-threading, and standard libraries has allowed us to evolve our design rapidly while maintaining a small code base (≈10000 lines).

[1] ANTS is publicly available – see http://www.sds.lcs.mit.edu/activeware.

The toolkit provides a class-based model for constructing new services. The abstract classes `Capsule` and `Protocol` provide required and useful functionality, and are subclassed once for each type of capsule and protocol. The programmer manipulates each capsule as an instance of the appropriate subclass to express the processing that should occur at nodes. The processing routine takes a parameter of class `Node` (representing the local node) to access the node API. Capsule instance variables may be carried along with the capsule and accessed within the network by providing methods to encode them for transmission and decode them on reception.

Performance measurements indicate that the base performance of our system is reasonable for a high-level prototype and fast enough for experimenting with distributed applications. The throughput of a single node was measured to be 1680 capsules/second for capsules with minimal IP-style forwarding. This measurement was taken on a Sun Ultrasparc 1 (167 MHz) running Sun's JDK 1.1 with a just-in-time compiler. Nygren [16] provides evidence that the ANTS model is lightweight enough such that the overhead of implementing it is low. He reports on a Linux-based (PC) implementation of the ANTS architecture in which capsule code is transported as Intel binary code instead of as Java bytecodes. Comparison of the performance of Nygren's implementation with the performance of Linux IP routing shows little additional overhead for forwarding capsules over IP packets: less than an 8% decrease in throughput, and a small increase in latency that corresponds to 20% for 512 byte packets.

5.2 The Auction Service

As an example of how our architecture is intended to be used, we explain how the auction service described in section 2 is implemented in the ANTS toolkit. We focus on this one application to provide reasonable detail, and in doing so restrict ourselves to the implementation of in-network processing since bid processing at the clients and server is straightforward.

The essential feature of the auction service is that low bids may be rejected at nodes within the network when server load is high. The basic form of this functionality can be realized in ANTS with a protocol comprised of four capsules:

- a FILTER capsule for the server to set a filtering price
- a BID capsule for clients to submit bids
- a SUCCEED capsule for the server to notify a client that a bid succeeded

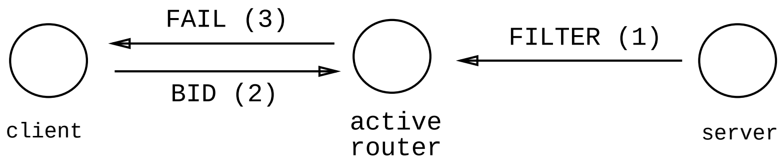

Figure 3: Rejection Processing in the Auction Service

```
// FILTER capsule, carries price = filter price, thing = auctioned object, hops = travel limit

Object info = n.getCache().get(thing);                              // cached price info?
if (info != null) {
    int old = ((Integer)info).intValue();                           // update not needed?
    if (price <= old) return true;
}
n.getCache().put(thing, new Integer(price))                         // otherwise, update
if (--hops > 0) return n.sendToNeighbors(this);                     // and spread the word
else return true;


// BID capsule, carries bid = an offered price, and thing = auctioned object

Object info = n.getCache().get(thing);                              // cached price info?
if (info != null) {
    int price = ((Integer)info).intValue();                         // will the bid fail?
    if (price > bid) {
        AuctionFailCapsule ack = new AuctionFailCapsule(this, price);// if so, reject it now
        return n.forward(ack, ack.dpt);
    }
}
return n.forward(this, dpt);                                        // otherwise, continue
```

Figure 4: Auction Service Capsule Processing

- a FAIL capsule to notify a client that a bid failed or would have failed

During normal operation, BID capsules are sent from clients to the server, and SUCCEED and FAIL capsules returned from the server to clients. Recall that, unlike traditional auctions, bids may fail to be accepted because they are out-of-date by the time they are processed at the server. During periods of high load, many bids may fail, and the server may delegate some rejection processing to active network nodes. It does this by sending FILTER capsules to nearby active nodes. These capsules store the current price in the node, and subsequent BID capsules passing through the node compare the price of their bid with a known bid. If it is lower, then a FAIL capsule may be returned from within the network indicating failure, and the BID capsule need not be forwarded to the server. This sequence is shown in Figure 3. Note that the SUCCEED capsule is generated only by the server, never by interior network nodes; it need not form part of the network service, but was included for the purpose of exposition.

The processing routines of two of these capsules are shown in Figure 4. The FILTER capsule uses a flooding algorithm to update the current price of the item at all network nodes within a certain radius of the server ; the size of the radius in hops is selected by the server depending on load. At each node it reaches, it updates the item's price in the cache, decrements its own hop limit, and then forwards copies of itself on all outgoing links. Forwarding stops when the hop limit is exhausted, or if it reaches a node that has filter that supersedes the one being forwarded. The BID capsule forwards itself towards the server, comparing its bid with any known prices it discovers along the way. Strictly lower bids are rejected by creating a FAIL capsule and returning it to the sender in place of forwarding the failed BID. The processing routines for the FAIL and SUCCEED capsules are not shown, since these capsules are simply forwarded at nodes until they reach their destinations.

Early simulation experiments confirm this service works as intended and suggest that it can improve performance, at least for simple topologies. At times of high load, the server sees a higher ratio of in-the-money bids. The roundtrip latency for failed bids is also reduced, though this improvement is limited by the placement of filters near the server.

There are also several noteworthy aspects of the functional organization of the protocol. First, to be compatible with end-to-end reliability, successful bids must always be accepted by the server; only rejection processing is handled within the network. With this organization, the protocol is correct despite packet loss, duplication or reordering. Second, the only step requiring authentication is the updating of the known price by the FILTER capsule (this is not shown in the code in Figure 4). It is not necessary for BID and FAIL processing to authenticate prices or senders since our protection model ensures that no other mechanism can update (or even observe) the price. Thus, most auction related capsules are forwarded with a minimum of overhead.

In closing, we note that there are many possible enhancements: reporting of failed bid statistics to the server, aging of known price information to provide a better indication of the current price, integration of known price updates with current price queries and replies to successful bids, timestamping of bids, and so forth. Since our description is intended to convey how an overall service may be implemented in terms of forwarding routines, we have omitted these enhancements, though none is incompatible with the basic scheme.

6 Where We Are Now

The long-term goals of our work are: to understand how new network services can improve performance and functionality; and to construct a framework within which these services can be expressed – easily, safely, understandably, and with minimum impact on other network users. In this paper, we have,

- described the kind of new services that we expect an active network to be able to introduce;
- argued qualitatively that these services are useful for improving overall application performance;
- presented an architecture for deploying new services that balances security and performance concerns;
- demonstrated how a particular new service can be developed for this architecture with our toolkit.

Our work is complementary to several other active network efforts. The use of general-purpose Java bytecodes and virtual machine has allowed us to evolve our architecture quickly, but at the cost of less control over resource usage and lower absolute performance. Research at the University of Arizona on Liquid Software [7] and Scout [13] enable a finer granularity of local resource management as well as competitive performance through the construction of a specialized node operating system. Research at the University of Pennsylvania on PLAN [8] and BBN on Sprocket [9] enables stronger resource management and security guarantees across the nodes of a network through the use of language design techniques. Research at Georgia Tech [23] is examining the composability of services within the network. Finally, research on active signaling at USC ISI [2] and NetScript at Columbia University [22] explore alternative models of active networks in which new services are introduced for control rather than data transfer purposes, or by network management agents rather than all users.

We believe that networks today are poised to become increasingly malleable, as virtual overlays proliferate and rapid adaptation to changing requirements becomes more and more important. Today, we are at the beginning of our exploration of active networking techniques as a means of providing such flexibility. Much work remains to reach the conclusion of our current line of research: a demonstration of improved performance of several useful distributed applications running over an active network.

Acknowledgments

We thank our fellow members of the Software Devices and Systems Group. The research presented here is ongoing within our group, and has benefited from the contributions of many people. In particular, we wish to acknowledge Steve Garland, Frans Kaashoek, Li Lehman, Erik Nygren, Jon Santos, Liuba Shrira and David Tennenhouse. We also wish to thank members of the wider active network community at the University of Arizona, Georgia Tech, USC ISI, Columbia University and University of Pennsylvania for their support and assistance with deploying ANTS nodes.

This work was supported by DARPA under contract No. N66001-96-C-8522, and by seed funding from Sun Microsystems Inc.

References

[1] H. Balakrishnan et al. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *SIGCOMM '96*, 1996.

[2] B. Braden. Active Signalling Protocols. http://www.isi.edu/active-signal/, June 1997.

[3] R. Braden et al. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Internet Draft, Nov 1996.

[4] S. E. Deering. Host Extensions for IP multicasting. Request For Comments 1112, Aug 1989.

[5] P. Deutsch and C. A. Grant. A Flexible Measurement Tool for Software Systems. In *Information Processing*, 1971.

[6] eBay Inc. AuctionWeb server. http://www.ebay.com/.

[7] J. Hartman et al. Liquid Software: A New Paradigm for Networked Systems. Technical Report TR96-11, Dept. of Computer Science, Univ. of Arizona, 1996.

[8] M. Hicks et al. PLAN: A Programming Language for Active Networks. http://www.cis.upenn.edu/~switchware/papers/plan.ps, July 1997.

[9] A. Jackson and C. Partridge. Smart Packets. http://www.net-tech.bbn.com/smtpkts/.

[10] D. Katz et al. Tag Switching Architecture – Overview. Internet Draft, Aug 1997.

[11] U. Legedza, D. Wetherall, and J. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *INFOCOM '98*, 1998.

[12] L.-W. Lehman et al. Active Reliable Multicast. In *INFOCOM'98*, 1998.

[13] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *2nd Symp. on Operating System Design and Implementation*, 1996.

[14] D. Murphy. Building an Active Node on the Internet. M.Eng Thesis, Massachusetts Institute of Technology, June 1997.

[15] G. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *2nd Symp. on Operating System Design and Implementation*, 1996.

[16] E. Nygren. The Design and Implementation of a High-Performance Active Network Node. M.Eng Thesis, Massachusetts Institute of Technology, February 1998.

[17] ONSALE Inc. ONSALE web server. http://www.onsale.com/.

[18] C. Perkins, Ed. IP Mobility Support. Request For Comments 2002, Oct 1996.

[19] D. Wessels. The Squid Internet Object Cache. http://squid.nlanr.net/Squid/, 1997.

[20] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *OPENARCH'98*, 1998.

[21] D. J. Wetherall and D. L. Tennenhouse. The ACTIVE IP Option. In *7th SIGOPS European Workshop*, 1996.

[22] Y. Yemini and S. da Silva. Towards Programmable Networks. In *FIP/IEEE Intl. Workshop on Distributed Systems Operations and Management*, 1996.

[23] E. Zegura and K. Calvert. CANES: composable active network elements. http://www.cc.gatech.edu/projects/canes.

Biographies

David Wetherall is a Ph.D. candidate at the MIT Laboratory for Computer Science. His research interests span the area of computer systems with a focus on networking. His thesis research is helping to pioneer active networks, an approach in which customized network services may be deployed rapidly within a programmable network infrastructure. David came to MIT after working at QPSX Communications, a high speed networking company that led the development of the IEEE802.6 (DQDB) switching technology. He received his B.E. in electrical engineering from the University of Western Australia in 1989, and his M.S. and E.E in computer science from MIT in 1994 and 1995, respectively.

Ulana Legedza is a Ph.D. candidate at MIT Laboratory for Computer Science. Her research interests are in computer systems, networking, and parallel computing. Her current focus is on the design of network-level support for application-specific routing functions. She received the B.S.E. degree in Computer Science from Princeton University in 1992, and the M.S. degree in Computer Science from MIT in 1995.

John Guttag is Associate Department Head, Computer Science, of MIT's Electrical Engineering and Computer Science Department, and head of the Laboratory for Computer Science's Software Devices and Systems Group. The group does research in networking, distributed computing, computer and communications security, and wireless communications. Professor Guttag has also done research, published, and lectured in the areas of software engineering, mechanical theorem proving, hardware verification, and compilation.