17th ACM Symposium on Operating Systems Principles (SOSP '99)
Published as *Operating Systems Review* **34**(5):64–79, Dec. 1999

Active network vision and reality: lessons from a capsule-based system

David Wetherall
Department of Computer Science and Engineering
University of Washington
(*djw@cs.washington.edu)*

Abstract

Although active networks have generated much debate in the research community, on the whole there has been little hard evidence to inform this debate. This paper aims to redress the situation by reporting what we have learned by designing, implementing and using the ANTS active network toolkit over the past two years. At this early stage, active networks remain an open research area. However, we believe that we have made substantial progress towards providing a more flexible network layer while at the same time addressing the performance and security concerns raised by the presence of mobile code in the network. In this paper, we argue our progress towards the original vision and the difficulties that we have not yet resolved in three areas that characterize a "pure" active network: the capsule model of programmability; the accessibility of that model to all users; and the applications that can be constructed in practice.

1 Introduction

Active networks are a novel approach to network architecture in which customized programs are executed within the network. They were first described in [41], where the authors postulated that this approach would provide two key benefits: it would enable a range of new applications that leveraged computation within the network; and it would accelerate the pace of innovation by decoupling services from the underlying infrastructure. Active networks have generated much interest because of their appeal as a means of creating new Internet services. They have also resulted in at least as much controversy because of serious performance and security concerns raised by the presence of untrusted code within the network.

Can active networks deliver their claimed benefits in terms of new and novel services, while at the same time keeping the network efficient and secure? At least eight active network prototypes have been developed in addition to our own to study this question in one form or another [1, 11, 43, 48, 2, 16, 30, 39]. Hence the concept of active networking (if not its actual deployment) is now "old hat." However, very little evidence has been presented to support or refute the original vision for active networks to date.

In this paper, we reconsider the vision for active networks in light of our experience designing, implementing and using the ANTS toolkit [46] over the past two years; ANTS is well-suited to this purpose because it is based on an aggressive "capsule" design that adds extensibility at the IP packet level. We unabashedly present a mix of fact and opinion, experimental results and qualitative analysis, that reports on the progress we have made reconciling network flexibility with performance and security. Our findings are based on having implemented two large active network applications and numerous smaller examples on top of ANTS, and on a comparison of the properties of ANTS with the Internet. Additionally, ANTS is widely used within the research community (having been publicly available since late '97) and has resulted in considerable design feedback.

To highlight what we have learned, we contrast our findings with the vision originally stated in [41] in three areas that characterize a "pure" active network: the capsule model of programmability; the accessibility of that model to all users; and the applications that can be constructed in practice. Specifically, we present the following findings in this paper:

Capsules. Profile measurements suggest that capsules can be a competitive forwarding mechanism wherever software-based routers are viable; this is despite the fact that our prototype is limited to an unimpressive 10 Mbps, in large part due to being implemented in Java. To implement capsules efficiently, we have replaced a naive code carrying scheme with one in which code is carried by reference and so depends, to a larger extent than suggested in [41], on demand loading and traffic patterns for which caching is effective. We have also found it necessary to revise our architecture to accommodate heterogeneous types of nodes, and in particular to be compatible with nodes that are not active. As a result, routers that would not otherwise perform forwarding

SOSP-17 12/1999 Kiawah Island, SC

in software are not slowed by capsule processing.

Accessibility. We have partly succeeded in allowing each user the freedom to control the handling of their packets within the network. We are able to isolate the code and state of different services to guarantee their correctness to the same degree as do today's static and trusted protocols. This is possible without restrictions on who can program the network, despite the fact that active network code is mobile and untrusted. However, it remains an open problem to prevent misbehaving programs from monopolizing resources across a group of nodes. This is analogous to the way that the current Internet does not prevent misbehaving users from monopolizing bandwidth. Both problems require further network mechanisms to be developed (see, for example, [8]), though the active network version of the problem is more complicated; we describe how it differs. To deal with this problem in the immediate future, we have fallen back on certification by a trusted authority, a measure not in keeping with [41] in that it will slow the rate of change. Nonetheless, we argue that the result is a system that can still evolve much more quickly than the Internet.

Applications. We have found capsules most useful for experimenting with and deploying new services that are routing variants, such as multicast, that make use of network layer state. Capsule code tends to be "glue" that acts as a flexible means for composing the capabilities made available by active nodes, rather than application-specific computation that is migrated to within the network as suggested in [41]. We expect that there will be relatively few active services, but believe that the case for using active networks is still compelling because of the great difficulties introducing, changing and experimenting with new services in the Internet today. In particular, the automatic code deployment of capsules provides a fundamentally new and valuable model for systematic change across wide-area network paths.

In sum, we believe that we have made substantial progress towards building the kind of active network envisioned in [41]. We find it feasible to add a degree of programmability across the network layer that provides clearly useful new flexibility. At the same time, the mechanisms we have developed and the restrictions we have adopted go a long way towards resolving performance and security concerns. On the other hand, it is clear that greater application experience is needed and open issues remain in areas such as resource management. We observe that programmability is rapidly being incorporated into network elements at essentially all locations where it is viable. Without further research this programmability will by default take the form of piecemeal solutions that allow individual network elements to be upgraded, but no more than that. If we are not careful, the network may well be "activated" without providing the benefit of a systematic means of upgrading services across the Internet.

The rest of this paper is organized as follows. We begin with background on active network research, and then summarize the essential features of ANTS needed to understand our results. In the main sections, we present our findings for each of the three topic areas. Finally, we discuss some more speculative observations in terms of network architecture.

2 Background

There are many approaches by which programmability can be deeply embedded into the network infrastructure, at either routers or individual packets, in the fashion of active networks. We briefly consider three styles to help place ANTS in context.

Some active network systems provide extensibility to individual devices, increasing their flexibility well beyond the level currently supported by router configuration mechanisms, such as Cisco's IOS. Two examples are the active bridge [1] and router plugins [11]. Such systems are well-suited to the task of imposing policy or functionality at a particular network location in the manner of a firewall or other edge boundary device. As such, they are typically meant for use by network administrators or other privileged users.

A second style of system provides programmability across multiple nodes for control tasks rather than new data transfer services. For example, BBN's SmartPackets [39] provides a programming environment that caters to management tasks; ISI's active signaling and the Tempest [43] target call and virtual network setup; and ACTIVE IP [47] supports measurement and discovery tasks. Netscript [48] is an example of a system that combines both of these styles by allowing management channels to program new data transfer services.

These two styles of system essentially restrict either where programs can be run or who can cause them to be run. Doing so limits their applicability, but is useful in practice precisely because it makes the design problem more tractable by specializing it to a particular domain. Single network element schemes, for example, are not intended to introduce services that are spread over the wide area and for that reason do not have to coordinate code distribution across an entire network. Similarly, control tasks are not executed at the granularity of packet forwarding and so have less stringent performance requirements.

In contrast, ANTS belongs to a third style of systems that do not *a priori* restrict who can program which nodes. Instead, ANTS aims to allow each user to construct new data transfer services across the wide-area, such as routing for host mobility, by controlling the handling of their own packets within the network. This is analogous to extensible operating system approaches [21, 5] that aim to offer untrusted applications as much control over the way system resources are managed as possible while still being able to protect the underlying resources and arbitrate between competing demands.

ANTS is based on an aggressive "capsule" approach in which code is associated with packets and run at selected IP routers that are extensible. We sketch its design in the following section. Two other capsule-based systems, PAN [30] and PLAN [16], are similar in spirit to our own, and experience with both, on the whole, supports the conclusions drawn here. We mention specific results in the text as appropriate. PAN is modeled on ANTS and differs principally in that its capsules transfer unsafe binary forwarding programs and execute them directly, trading security for performance. PLAN capsules carry short programs in a specially-designed language that is the basis for system security. In contrast, the

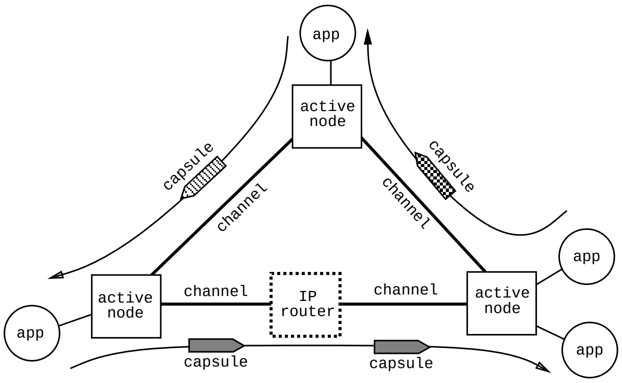

Figure 1. Entities in an ANTS active network

ANTS toolkit builds on the safety properties of Java byte-codes.

A final strategy that does not fit into these three styles is worth noting. Active services [2] seek to gain the advantages of active networks without disturbing the network layer by relying on domain-specific proxies to support "value-added" network services such as transcoding. We consider domain-specific solutions to be a useful tool, but the use of proxies to be largely orthogonal to many active network design problems. Clearly, proxies are valuable for incremental deployment. (They are not the only option though. We prefer firewall-style interception at selected routers because of the late-binding that it provides, and the potential for access to routing and load information that it retains.) However, the use of proxies rather than extensible routers does not resolve design issues such as the extension API, code distribution and global resource management, all of which must be tackled in any real system.

3 The essentials of ANTS

In this section, we summarize the details of ANTS that are needed to understand the subsequent discussion. We focus on "bootstrapping the reader" by explaining how it works; we defer arguments about why it works in this manner to the following sections. The summary is based on the reference version of ANTS detailed in a dissertation [45], which supersedes an earlier exposition [46].

We describe ANTS along two lines: the interface it presents to users at the edge of the network, and its implementation within the network. In addition, we comment on the ANTS toolkit, a reference implementation written in Java, and how ANTS may be deployed incrementally in the Internet.

3.1 Interface

The entities in an ANTS network are shown in Figure 1. Applications obtain customized network services by sending and receiving special types of packets called *capsules* via a programmable router referred to as an *active node*. Each active node is connected to its neighbors, some of which can be conventional IP routers, by link layer channels. The innovative properties of an ANTS network stem from the interaction of capsules and active nodes; the application and channel components are simply modeled on those of conventional networks.

The format of a capsule, shown in Figure 2, is an extension of the IP packet format. Capsules are like mobile agents in that they direct themselves through the network by using a custom forwarding routine. The type of forwarding is indicated by the value of the type field and is selected by end-user software when it injects a capsule into the network. The corresponding forwarding routine is transferred using mobile code techniques to active nodes that the capsule visits as it passes through the network. The routine is executed at each active node that is encountered along the forwarding path. At conventional nodes, IP forwarding occurs using the IP header fields.

Any party can develop a new network service and make it available for widespread use. The first step is to write a new set of forwarding routines that implement the desired behavior. This is done in a subset of Java in our reference implementation, the ANTS toolkit. Each different forwarding routine corresponds to a different type of capsule and can carry different header fields (the type-dependent header fields in Figure 2). The kinds of forwarding routines that can be constructed depend on the capabilities of the active node; routines are further restricted in the amounts of node resources they can consume. The ANTS toolkit provides a core API, listed in Table 1, that grew out of experience with a predecessor system [47] and consists of the smallest set of operations with which we were able to develop many different services. It provides three categories of calls that: query the node environment; manipulate a *soft-store* of service-defined objects that are cached for a short time and then expired; and route capsules towards other nodes or applications in terms of shortest paths. These calls allow novel routing services to be expressed by querying network characteristics, maintaining route information in the soft-store, and following it during forwarding. Additional API calls will likely be added with further development and experience. Loss information, for example, is clearly useful for congestion-related services, yet absent from the list because it is inconvenient in our current user-level Java implementation.

Once the code is written, it is signed by a trusted authority (an IETF-equivalent) to certify that the service makes use of overall network resources in a "reasonable" fashion. Certification reflects global resource management concerns that we have not otherwise resolved in the general case. This issue is discussed in Section 5. Finally, the code is registered with a directory service using a human-readable name (such as "Drop Priority") to make it available to other network users.

End-user software can use a new service developed according to this model in a simple manner. First, the service code is obtained via the directory service, which is simply the local filesystem in our prototype. In a large-scale network, this step can be made automatic (without burdening applications) with a process analogous to DNS host resolu-

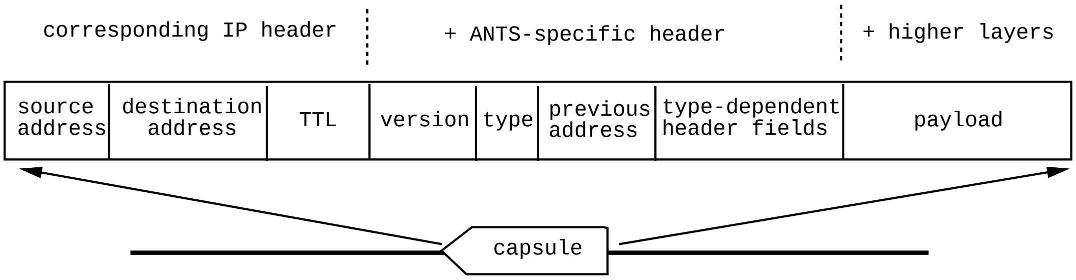

Figure 2. Key features of the capsule format

Method	Description
`int getAddress()`	Get local node address
`ChannelObject getChannel()`	Get receive channel
`Extension findExtension(String ext)`	Locate extended services
`long time()`	Get local time
`Object put(Object key, Object val, int age)`	Put object in soft-store
`Object get(Object key)`	Get object from soft-store
`Object remove(Object key)`	Remove object from soft-store
`void routeForNode(Capsule c, int n)`	Send capsule towards node
`void deliverToApp(Capsule c, int a)`	Deliver capsule to local application
`void log(String msg)`	Log a debugging message

Table 1. Active node API

tion. Second, the service is registered with the local active node. This provides the node with a copy of the service code to bootstrap code distribution within the network, and allows the node to compute the type values that will be placed on corresponding capsules. Once these steps are complete, applications are free to send and receive capsules that belong to the new service, and these capsules will be forwarded within the network by executing the corresponding routines.

3.2 Implementation

To process capsules efficiently for the expected traffic patterns of our target applications, ANTS separates the transfer of service code from that of the rest of the capsule and caches code at active nodes. Each capsule carries a type field as shown in Figure 2 that refers to its customized forwarding routine. The value of this type field is based on an MD5 [36] fingerprint of the associated service code. Code is provided to nodes at the edges of the network by end-user software. Within the network, a lightweight code distribution system transfers the code along the path the capsule follows when the code is not already cached.

Our code distribution system is designed to provide rapid but unreliable transfer of short routines between adjacent active nodes; ANTS places a limit of 16 KB on the code of each service to limit the impact of code transfers on the network. The protocol works as follows (Figure 3). When the code needed to forward a capsule is not found in the cache, a request is sent to the previous active node that the capsule visited. The "previous address" header field (Figure 2) is maintained by active nodes for this purpose. If this node has the required code, which is likely because it executed the code moments before, it sends the code. The code is then cached for later use and executed to forward the capsule. If messages are lost or the code is unavailable, the capsule is discarded and considered lost. Thus code transfer will either succeed quickly, in which case the interruption to forwarding will be close to minimal, or occasionally fail, in which case applications must recover the lost capsule in the normal manner.

Once the code has been distributed, capsule processing at an active node is straightforward: capsules are received, they are demultiplexed using their type field to the associated forwarding routine, the routine is safely executed within a sandbox, and the process repeats. We refer to this model as *extensible packet forwarding*, since it generalizes the IP forwarding model in use today. The sandbox prevents untrusted code from corrupting the node and the state of other services that are being run concurrently. Capsules may only effect externally visible behavior through node API "system calls". The sandbox also enforces constraints that facilitate the protection and resource management mechanisms discussed in Section 5: overall runtime is limited; access to the capsule itself is limited so that the source address and type are constant, while the previous address and TTL are maintained by the node; creation of other capsules is restricted so that only related types can create each other; and freshly manufactured capsules obey invariants such as having a smaller TTL and the source address of the parent to facilitate distributed debugging. The additional packet overhead of this model is

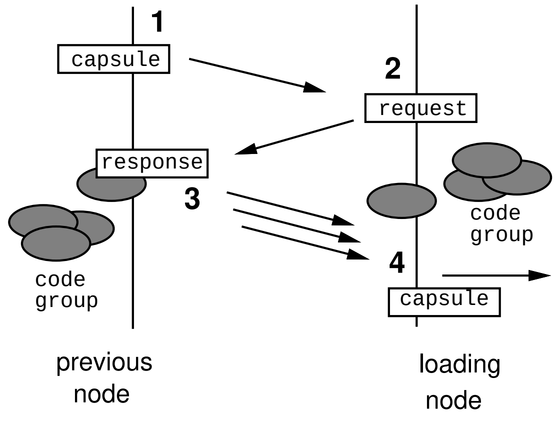

Figure 3. Demand loading of capsule code

modest; type is the largest field in our implementation, and at 128 bits it is the size of a single IPv6 address. The additional runtime steps are also modest, as is reported in Section 4.2, because we are able to leverage the restrictions placed on forwarding routines to simplify the implementation.

3.3 ANTS Toolkit

The ANTS toolkit [45] is a reference implementation of the architecture described above. It incorporates everything described in this paper with the exception of signing code and checking the signatures, since this has no effect once code is loaded. The toolkit also carries capsules within UDP datagrams that are organized into an overlay, rather than as a direct extension of the IP format. This is convenient as the toolkit runs at user-level on end-systems.

Both the reference platform itself and the capsule forwarding routines (which are supplied by application developers) are written in 100% Java. This has been a valuable design choice in that it has allowed us to rapidly develop a portable, compact (approximately 10000 lines) and flexible experimental platform that is accessible to application developers. The active node sandbox, for example, builds directly on Java's type-safety and security manager framework, while code distribution is facilitated by dynamic loading mechanisms. As a result, the ANTS toolkit has been widely used within the research community.

The cost of this design choice has primarily been performance, though the lack of fine-grained control over resources, for example, the size of objects and garbage collection, has also been a limitation. We expect much of these costs to eventually be recovered, since neither the reference platform nor the forwarding routines need to be written in Java in a different implementation. In the short term, the platform can be statically compiled and, with a suitable Java runtime, reside in the operating system along with other networking code. Ultimately, safe execution techniques applicable to binaries (namely software-based fault isolation (SFI) [44] or proof-carrying-code (PCC) [29]) could be used in higher performance implementations. Alternatively, specialized runtimes would improve performance by exploiting the subset of Java (or other language) that is used to write forwarding routines. For example, ANTS requires that forwarding routines be single-threaded.

3.4 Deployment

ANTS is designed to be deployed incrementally within the Internet today. Since not all nodes are required to be active, a straightforward step is to selectively activate nodes in strategic locations. This is likely to begin with end-systems, along with routers connected to bandwidth-limited wireless and access links, and followed by increasingly high performance routers at ISPs within the network. Activating such routers can be straightforward because ANTS nodes can be embedded in the network and intercept capsules that pass through them.

At a finer granularity, nodes can be active for selected services, but perform IP forwarding for other services. The ANTS architecture allows this decision to be made locally by each node. This strategy can extend the performance range of active nodes: compute-intensive services that are not dynamically loaded because they do not run at the line packet rate can instead be statically loaded if the capsules that use them arrive infrequently. This is analogous to the way ICMP and other services are handled separately from the IP "fast path" in high-bandwidth routers. This line of reasoning suggests that hybrid active node implementations will be useful for combining the performance of high-end routers with the flexibility of programmable services. For example, an IP router could be extended with an attached PC that receives packets via a classifier.

4 Are capsules feasible?

We now switch gears and turn to the main contributions of this paper. The first question we consider is essentially one of performance: are capsules a feasible mechanism on which to build active network programs?

We argue that when capsules are implemented as described in Section 3.2, they can be a competitive forwarding mechanism wherever software-based routers are viable. We argue this in two steps. First, that capsule code can indeed be carried by reference and loaded on demand. Fingerprints have proved an effective design technique here. Second, that the intrinsic overhead of capsule processing is low and so adds little to the cost of IP forwarding when both are implemented in software.

Of course, software-based forwarding will not be viable at all network locations, even in router designs with a processor per port. At one extreme, current generation PC-based routers provide a flexible processing environment capable of forwarding at least 70,000 packets/sec, easily reaching 100 Mbps for typical packet sizes [28]. They are poorly-suited to this task as they are I/O limited. At the other extreme, modern high-end commercial routers can forward 70 byte packets at wire speed for OC-48 (2.4Gbps) line rates, which requires forwarding rates approaching 4,000,000 packets/sec [42]. The difference is a factor of 50. It is difficult, however, to evaluate the potential of extensible routers because of the changing underlying technologies; for example, reconfigurable hardware such as RaPiD [10] is appearing. Ultimately, the applications of a programmable node will be constrained by its capabilities and position in the network. For example, processor rates of 1 GHz and line rates of 1 Gbps imply an average processing budget of 1000 cycles, if all packets are to processed and the average size is 128 bytes. This figure varies by orders of magnitude, reaching 100,000 cycles if 10% of the packets are processed and the line rate is 100 Mbps, for example. Thus a network in which node capabilities are heterogeneous seems fundamental to us. It is for this reason that our architecture supports partially active networks in which some routers, such as those in the core of the Internet, can implement IP forwarding only and are not slowed by capsule processing.

4.1 Implementing capsules

One of the first changes we made in moving from an earlier active network system to ANTS was to carry capsule code by reference rather than by value. Besides the obvious space and time overheads of carrying code and converting it to an efficient executable representation, we observed that most of the applications we were exploring exhibited significant locality — the same code was executed along the same network path. This is particularly so for network layer evolution where, although many new services might be tried over time, a small number of services account for virtually all of the traffic at any given instant. We believe that this is unlikely to change, even in a much more dynamic network, because of the existence of higher layer flows and a dependence on third party software. Most other active network systems that emphasize performance also cache code, for example PAN [30] and router plugins [11, 12]. PLAN [16] does carry a small amount of code directly, but similarly distinguishes between forwarding fragments and extension code that is cached[1].

To provide the same behavior as the capsules described in [41], we must be able to distribute the right code to the right place, where "right" is determined by a model in which the code is actually present in each capsule. Each mechanism can potentially cause difficulties.

To ensure the right code, we compute the type identifiers that are stamped on each ANTS capsule from an MD5 [36] fingerprint of the corresponding code[2]. This is analogous to the way fingerprints are used to name the types of network objects [7]. We originally chose this method as a distributed naming scheme to eliminate the need for standardized protocol identifiers, but quickly came to value its security properties. It is secure because a belief that the fingerprint function is one-way implies that the association between a capsule and its forwarding code is unambiguous[3]. Because this correspondence can be verified locally, without trusting external parties or relying on external information, the danger of code spoofing is eliminated. SFS uses fingerprints as the basis of self-certifying pathnames [26] for essentially the same reasons.

A significant difference compared to conventional protocols is that the identifier names an implementation rather than an interface. This is potentially beneficial because it eliminates versioning problems: different versions of code are treated as different services within the network. However, we have found that this property typically means that a higher level of naming is soon introduced. In our system, a directory service is used by end-user software selecting protocols, and bootstrap capsules (such as those that transfer code between active nodes) carry well known names. Any higher level of naming must then be secured if it is not to negate the fingerprint properties that we value.

To transfer code to the right place, we have found the simple scheme described in Section 3.2 to be quite general and sufficient to our needs. In networking parlance, service code is "soft-state" [9] that is automatically replenished as it is needed. The scheme thus adapts to packet loss, node failures and changing routes, all without complicating capsule semantics with extra mechanisms such as an explicit "connection" setup phase. Because it interleaves capsule forwarding and code transfer hop-by-hop, it can be used to load new routing services.

There are also more subtle issues that we have uncovered while studying code distribution. For example, the code distribution system should not leave active nodes open to a denial of service attack. The hop-by-hop nature of our scheme may be of use here, since code messages are only exchanged between neighbors and can easily be authenticated, thus reducing the scope of attacks. Also, unless the latency of loading is small it may conflict with the end-system timeouts that are used to detect loss. For our system, a measurement-based analysis [45] predicts that loading delays will fall within the normal range of one-way transit variations (of 0.1 to 1 second) reported in [33], even for cross-country paths. Caching behavior under overload is also of concern.

For the most part, however, we have deferred serious analysis of the performance of code distribution mechanisms until there is greater experience with large scale active networks. We observe that, if code loading is rare enough, its performance will be of little consequence as long as it is adequate. We believe our scheme provides adequate performance, but further observe that ANTS could be used with other code distribution schemes in practice.

4.2 Forwarding performance

At active nodes, processing can be separated by the granularity at which it occurs:

- per capsule forwarding tasks;

[1] It is also interesting to note that PLAN forwards capsules roughly three times as fast as ANTS. As best we can tell, this is due more to the performance of the underlying language runtimes and better marshaling code than architectural choices.

[2] We first heard this technique suggested in the context of protocol code by Gary Minden. We pioneered its use at the same time as PAN [30].

[3] There is some evidence that the collision-resistance of MD5 may be broken in the foreseeable future [37], but to our knowledge no evidence that the one-way property on which we depend is in question.

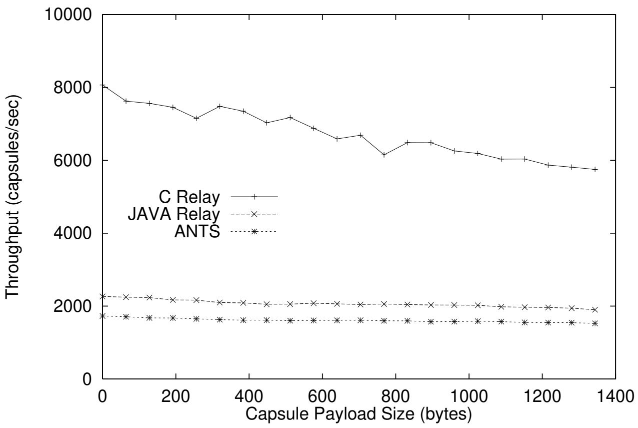

Figure 4. Active node throughput (pps)

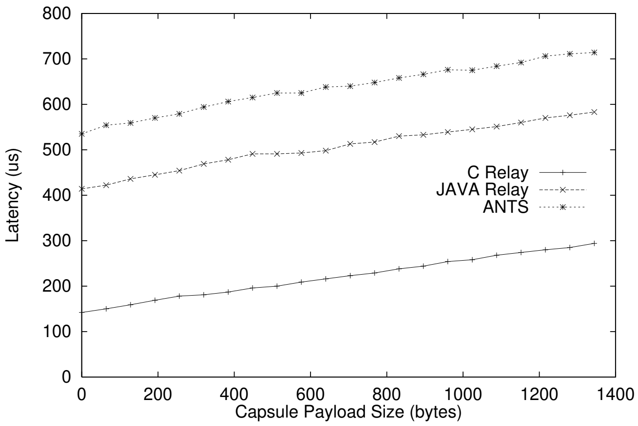

Figure 5. Active node latency (μs)

- per service code distribution; and
- periodic node management tasks.

Of these tasks, only capsule forwarding limits node performance in practice. For the traffic patterns of interest, code caching is effective and so code distribution is rarely needed. Measurements of the ANTS toolkit showed a processing cost of approximately 1 ms per code distribution message, with up to a maximum of 16 messages to transfer service code. This is approximately one order of magnitude more expensive than regular capsule forwarding, and requires that the cost of loading be amortized over around 1000 capsules before it has a negligible impact (1%) on network performance.

Periodic tasks in ANTS, such as route updates and cleaning of the soft-store, do not result in a noticeable slowdown either. This is because we are able to make them occur at a much coarser granularity than capsule forwarding (seconds rather than microseconds) and implement them in a manner that does not block forwarding for their duration.

We also note that capsule forwarding is readily parallelized to run on routers in which there is one processor per port. This is because no tight synchronization is required between the per port forwarding engines. An initial concern was that the soft-store would require synchronized access. We have found, however, that this potential problem can be eliminated by design, since it does not complicate active network programming to define the soft-store to be a per port resource.

To understand the costs of capsule processing, we measured the forwarding performance of a single ANTS node while running a basic service that is the equivalent of IP, yet implemented within the ANTS framework rather than as a built-in capability. For comparison, the performance of user level relays written in Java and C, but otherwise running on the same system, are also shown; these blindly receive and then transmit packets, without the intervening processing required by ANTS. The experimental platform used was a Sun Ultra 1 running Solaris 2.6 and an early access release of Sun's JDK1.2. It has an estimated performance of 3.5 on the SPECJVM98 benchmark, while the hardware and operating system alone have publicly available performance results of approximately 6 on the SPECINT95 benchmark[4].

Throughput results are shown in Figure 4, and latency results in Figure 5. We see that ANTS forwards approximately 1700 capsules/sec for small packet sizes and 16 Mbps for large (Ethernet) packet sizes, with corresponding latencies that range from approximately 500 to 700 μs. ANTS and Java relay throughput is similar, while C relay throughput is greater by between a factor of three and four. For latency, the result is similar though less pronounced, with ANTS adding roughly half of the difference between Java relay latency and C relay latency; we have not tuned our system for latency.

While these results are unimpressive by production standards, they do serve to demonstrate three points. First, even at face value, they show a level of performance that is adequate to deploy active nodes across access links up to T1 (1.5 Mbps) rates immediately. Second, ANTS captures most of the potential of Java relay, the minimal Java-based user-level system running on the same platform. Third, the similar slopes of the latency lines indicate that capsule processing does not contain hidden data-dependent costs (extra copies) compared to a minimal relay. The conclusion we draw is that the performance of our prototype is limited by both user-level and Java-based operation, neither of which is required by our architecture. The user-level C relay is faster by a factor of four, and it is likely that an in-kernel implementation is faster by a factor of at least two again. These observations suggest that substantially faster implementations can be constructed without resorting to custom hardware. PAN [30] is one step in this direction. It implements an architecture that is modeled on ANTS, but with in-kernel binary forwarding routines that forego protection. By doing so, it is able to saturate 100 Mbps Ethernet with 1 KB packets, almost an order of magnitude improvement.

To understand the fundamental costs of capsule processing in more detail, we profiled an ANTS node running on the same platform. We did this for 512 byte capsules with

[4]The SPECJVM98 result is an estimate because one of the seven tests we ran produced invalid results due to bugs in the runtime and benchmark. The publicly available results come from the SPEC website at www.spec.org.

Operation	IP?	Time (μs)	(%)
1. Packet Receive	no	180	29
2. Header Processing	yes	30	5
3. Type Demultiplex	no	20	3
4. Capsule Decode	no	110	18
5. Capsule Evaluate	no	10	2
6. Route Lookup	yes	30	5
7. Capsule Encode	no	90	14
8. Header Processing	yes	40	7
9. Packet Transmit	no	80	13
Other	n/a	25	4
Total	n/a	615	100

Table 2. Profile of basic capsule processing

built-in Java runtime facilities, and normalized the result using the previously reported latency to account for profiling overhead. The results are shown in Table 2. In the profile results, ANTS processing is divided into nine steps, some of which are analogous to the steps that occur in IP and some of which are unique to ANTS. This is noted in the "IP?" column. For this comparison, IP processing steps are abstracted from RFC 1812 [3], which defines the standard IP processing that must be performed by all IP routers.

1. A message is received from the incoming network interface via the operating system. IP in the kernel does not require this step, and neither would ANTS if running in the kernel.
2. The structure of the message is checked to ensure that it is a valid capsule. The checks are simple, and IP performs analogous checks.
3. The capsule type is mapped to the appropriate forwarding routine. This is done with a hash table lookup. There is no corresponding requirement for IP.
4. The capsule is decoded from an external packet representation to an internal object representation. This step requires object allocation and member initialization by copying. It is an artifact of the Java-based implementation that is not intrinsic to the ANTS architecture.
5. A forwarding routine is invoked. To implement IP service, the forwarding routine simply invokes default forwarding via the active node. While default forwarding is the same as IP forwarding, expressing it as a service within ANTS generates some overhead to set up the call in a safe and generic manner.
6. A routing table lookup is performed. In the prototype, a simple hash table lookup is used, while in IP and a production ANTS system a more expensive longest matching prefix operation is needed. (The greater relative cost of this step in practice will result in a more favorable comparison than is made here.)
7. The capsule is encoded to an external packet representation from an internal object representation. Again, this step is not intrinsic to ANTS.
8. Header fields are updated, for example, by decrementing the TTL and setting the previous node address. The updates are simple, and IP requires analogous updates, though fewer in number.
9. A message is sent to the outgoing network interface via the operating system. IP in the kernel does not require this step, and neither would ANTS if running in the kernel.

Clearly, our user-level profiling and analysis can project only approximate results for a kernel-level implementation. However, what is significant is how few complex processing steps are required for capsule forwarding. In particular, a number of steps are not required in the current model, but easily could be in alternative designs: code distribution, because it has been separated; authentication, because state protection is built on the fingerprint identifiers as described in the next section; and resource reservation and accounting, because we follow the basic connectionless forwarding model.

To contrast ANTS and IP processing using the results in Table 2, we first exclude the receive and transmit times, since they would not be substantial in a kernel-based implementation. Note that this will increase the overhead attributed to ANTS. Of the remaining ANTS-only steps, the largest components by far are for capsule encoding and decoding, both of which are artifacts of our Java-based prototype rather than intrinsic costs of capsule processing. This difficulty is well known but not satisfactorily addressed without language support such as the `view` construct in SPIN's version of Modula-3 [18]. We therefore exclude these costs.

The remaining processing steps correspond to intrinsic costs. The profile data shows that, for these steps, ANTS adds 30 μs to the 100 μs needed for IP. That is, ANTS adds an overhead of roughly 30% to IP to perform type demultiplexing and safe evaluation. Both of these operations are known to run quickly. Demultiplexing is efficient because it requires only a hash lookup. This is facilitated in ANTS because the fingerprint provides pseudo-random bits that can be used directly as an index. Safe evaluation can be performed efficiently with binary forwarding routines in a high performance implementation: SFI [44] is known to add an overhead of approximately 4% for fault isolation (write protection) and approximately 20% for general (read/write) protection, while PCC [29] adds no runtime overhead after the proof is checked when the code is loaded. Other costs will of course depend on the forwarding routine itself, but these costs are deliberately incurred when a new service provides a performance advantage. The forwarding times for some routines are reported in Section 6. We note here that all of the node API operations can be implemented to run quickly since all are lightweight; for example, none require disk access or network roundtrips.

5 Who can introduce new services?

The vision presented in [41] is that all users should be able to customize processing within the network. This would foster third party developers and a marketplace for new services that would accelerate the pace of innovation. Mobile code technologies are suggested as a mechanism that will enable programs to be run safely in the context of shared resources.

We have partly succeeded in reaching this goal, finding security concerns to be more challenging than those of performance. After all, it is clear that the speed at which capsule forwarding can be implemented will dictate where it is applied. But it is not clear that the resource management and security difficulties can be resolved in a large wide-area system without restrictions that negate the original vision.

In ANTS, we are able to isolate the code and state of different services in a manner that is equivalent to that of static and trusted protocols today. This is despite the fact that service code is mobile and untrusted. We view this as substantial progress. On the other hand, we currently handle the problem of global resource allocation with a certification mechanism that slows the rate of change compared to the active network vision, though we argue that the resulting system can still evolve much more quickly than the Internet.

The rest of this section elaborates on these findings along the lines of the protection and resource management threats that exist, and how they are handled in ANTS compared to the Internet. We consider protection threats to be those that directly impact the execution of a service within the network so that it is no longer isolated from other code and hence no longer guaranteed to behave correctly. We consider resource management threats to be those that affect the performance of a service, rather than its correctness, by consuming shared resources in an unreasonable fashion that starves other legitimate network users. That is, we are more concerned with network robustness than fairness.

5.1 Protection

The ultimate goal of ANTS is to allow untrusted users to control the handling of their own packets within the network, yet to ensure that the code they provide can do no harm to the users of other services even if it is designed poorly or used maliciously. Early in our design effort, we realized that the capsule model provides a clear basis for simplifying interactions within the network: each capsule explicitly specifies its own handling via its type field, which is carried with the capsule through the network as shown in Figure 2. To extend this scheme into a protection model that is equivalent to the operation of conventional protocols, we added the stipulation that a capsule cannot change its type to that of another service (or equivalently create capsules of another service) within the network. This is enforced by the ANTS runtime sandbox described in Section 3.3. The result is that the processing a capsule can undergo is fixed by the value of its type field at the moment it is injected into the network, in the same manner that an IPv4 packet is forwarded with IPv4 processing only. This means that, for example, it is not possible to construct a service that arbitrarily searches the network and then interferes with capsules belonging to another service.

This model is straightforward to understand, and has worked well for us once we extended it to allow for the controlled sharing of state between related types of capsules (as described shortly). It provides an authentication-free foundation on which other security mechanisms can be constructed as they are needed. We have found that, while it does limit the kind of services that can be constructed in ANTS, the benefits of having this form of protection model far outweigh the costs. A typical example of a service that is difficult to realize is a firewall, since one type of capsule cannot directly control the forwarding of another type. (However, these kind of services are not targeted by ANTS.)

Protection threats that would break this model must stem from the transfer and execution of code within the network. It appears to us that there are only three kinds of threats that could result in capsules of one service being handled in an unintended manner, whether accidental or malicious, that cannot occur in the Internet except in response to a faulty implementation:

- the node runtime may be corrupted by service code;
- service code distributed to an active node may be corrupted or spoofed; and
- the state cached at an active node on behalf of one service may be inadvertently manipulated by another service.

The first threat, corrupting the active node itself, is met through the use of safe evaluation techniques for executing service code. While the ANTS toolkit relies on the properties of Java, other implementations could be based on proof-carrying code (PCC) or software-based fault isolation (SFI). The net result is that the sound implementation of the node runtime implies that it cannot be crashed or otherwise corrupted by arbitrary service code in the same sense that traditional operating systems protect themselves from applications.

The second threat, code spoofing or the transfer of corrupted code, is met through the use of fingerprint-based capsule types, as was previously discussed.

A third means of interfering with a service is to corrupt the state that it maintains at an active node. This is prevented in ANTS by a restricted node API. Access to state shared across services is guarded. For example, the default shortest-path routes can only be read by services, so that one service cannot alter them for another. Only two forms of service-specific state are retained by a node after a capsule is forwarded: service code itself and data placed in the soft-store. The code is readily protected because it is read-only. The data is protected by building on the fingerprint-based capsule types to partition the soft-store by service. This guarantees that state maintained on behalf of one service cannot be manipulated by code corresponding to another service. Conversely, sharing between the sessions of a service is controlled by the service code.

A significant complication that we faced to make this model useful is the need for read and write sharing between different types of capsules. To implement many services, a set of related forwarding routines must be able to share state within the network. For example, one type of capsule may establish custom routes by placing information in the soft-store of nodes, and another type of capsule follow those routes by accessing the information. We accomplish one level of sharing in a secure manner by using a hierarchical fingerprint scheme for computing the capsule type identifiers. Essentially, if capsules with forwarding routines

A and B are to share state within the network and X_H is the fingerprint of X, they are marked with the type identifiers $(A, (A, B)_H)_H$ and $(B, (A, B)_H)_H$.[5] Code distribution then transfers the routines A and B when they are required so that the type identifier can be verified. In this manner, capsules of type A and B can be recognized as belonging to a single service that is identified as $(A, B)_H$ and state can be shared between them. This scheme also prevents another routine, C, from later claiming to belong to the service and manipulate its state.

In sum, ANTS provides measures that defeat all of these kind of threats so that the protection provided by ANTS is as good as the protection afforded to protocols in the Internet today. This is so despite the fact that ANTS services are implemented with mobile and untrusted code (certification is not required for basic protection) while conventional protocols are implemented with static trusted code. Further, these measures are robust because they are implemented by each active node without relying on any external parties.

In the larger context, Internet security is in the process of being extended because it is known to be weak. Both ANTS and the Internet protect different protocols from interfering, but not different users of a protocol from interfering with each other. An interesting use of ANTS is to introduce new security services that add authentication and encryption at selected points. This could easily be done by extending the node API with authentication and encryption calls that services could combine to meet their own security needs.

5.2 Resource management

We now provide a taxonomy of resource management threats. In ANTS, one service can potentially interfere with the performance of another in three ways:

- a capsule may consume a large, possibly unbounded, amount of resources at a single node;
- a capsule and other capsules it creates within the network may consume a large, possibly unbounded, amount of resources across multiple nodes; and
- an application running on an end-system may be used to inject a large, possibly unbounded, number of capsules into the network.

This classification is useful because it highlights resource management tasks that can readily be addressed, remain open in both ANTS and the Internet, and the difference between the two.

The first threat, too many resources used at a node during the forwarding of a single capsule, is able to be met in ANTS with current technology because the programming model is restricted. In the Internet, the resources consumed during packet forwarding are implicitly bounded by the design and correct implementation of IP and other network protocols. In ANTS, the node runtime enforces simple resource bounds. Long running forwarding routines are broken with a watchdog timer. Termination is simplified by unloading all state associated with forwarding routines that trigger violations. Access to memory and bandwidth is limited by decrementing the TTL field to prevent a capsule from sending a large number of outgoing capsules (as defined by the TTL) or placing a large number of objects in the soft-store. Alternative local policies, such as static limits on the fanout of a capsule, could easily be enforced. Other resources, such as the stack, can be similarly limited, though this is not done in the prototype because it requires the Java runtime to be modified. This simple scheme has been sufficient to our purposes because we are more concerned with preventing gross errors in the set of cached services than accounting for all resource consumption or enforcing fairness between capsules. Other research efforts, such as RCANE [27], are exploring active network environments that support more fine-grained control of resources.

The third threat, that misbehaving applications may starve well-behaved ones exists in both ANTS and the Internet and is not well addressed in either. The root of this problem is that the Internet currently lacks network-based resource allocation mechanisms; instead, it relies on the cooperation of users[6]. This is increasingly recognized as problematic, and efficient network-based mechanisms are fundamentally needed to improve fairness and penalize non-responsive flows [8] in the same manner that operating systems, not applications, must arbitrate between competing resource demands and protect the system. These mechanisms will benefit an active network in the same ways they benefit the current Internet.

In-between these extremes lies the more difficult task of controlling the resource consumption of a capsule and its descendents across a group of nodes. This task differentiates active networks from the Internet. In the Internet, the resources that are consumed as a packet is forwarded from source to destination are relatively well understood in terms of a static model that limits bandwidth, memory and processing time. This, of course, assumes correct design and implementation, which is not always the case.[7] In an active network, however, resource consumption is driven by programs and cannot be finessed by arguments of a restricted set of developers. It will be much more dynamic in nature, and must be restricted in its form to ensure that the effects of one service on others or a region of the network are reasonable.

One useful technique is a per capsule hop limit. In both IP and ANTS, packets and capsules would consume unbounded resources if blindly forwarded around a routing loop. The TTL field is used to break such loops in IP, and ANTS uses the same mechanism to detect and stop an analogous class of infinite loop program errors. PLAN [16] goes further by dividing the TTL between a capsule and its de-

[5] Actually, type calculation is more complex because there is an intermediate level of naming between individual capsules and aggregate services that is used to minimize code distribution. See [45] for details.

[6] It is often mentioned that the access bandwidth of a user limits the impact they can have on the network. However, this is a poor allocation mechanism at best. It does not prevent, for example, equally capable users from starving one another of bandwidth to a given destination.

[7] For an example flaw see CERT Advisory CA98-01, *Smurf IP Denial-of-Service Attacks*, January 1998.

scendents so that the original TTL bounds the total amount of resources consumed within the network. We consider TTL-based mechanisms insufficient, however, to prevent capsules from consuming an excessive (though bounded) amount of node resources if directed to do so by a poorly designed service. TTLs cannot provide a tight bound on activity when multicast must be accommodated. This is because the number of "capsule-hops" that can legitimately result from a single capsule grows with the multicast group size, and so can be large. More fundamentally, the property that we want to hold is that capsule forwarding does not result in activity that is concentrated in a small region of the network, swamping nodes in the process. Activity that is significant but not concentrated, as occurs when multicast is implemented correctly, can be an efficient use of resources and should be allowed. However, since TTLs are not related to topology they cannot readily restrict the location of activity. Even without multicast this is problematic. For example, consider a service that "ping-pongs" a capsule between adjacent nodes until its TTL is exhausted. With a maximum TTL of 255 (8 bits as in IP), a node sending one capsule with a buggy or malicious forwarding routine might inflict 100 capsules worth of forwarding work on other nodes.

Better mechanisms are therefore needed. In particular cases, it may be practical to locally check that programs match a restricted form. For example, forwarding routines that route a capsule towards a fixed destination and do not create other capsules satisfy most definitions of "safe." Though this class may sound restrictive it includes selective discard, congestion notification and network merging services. Fairness mechanisms that limit the impact of misbehaving users, such as fair queuing, will also mitigate the damage that can be inflicted by a poorly designed program. However, they cannot prevent such programs from consuming more resources than is appropriate.

Until better solutions are available, we have fallen back on the mechanism of requiring that service code be certified with a digital signature by a trusted authority (an IETF equivalent) before it is freely executed at nodes. This runs counter to the other mechanisms of the node runtime in that it depends on external parties. Nonetheless, we argue that it results in a system that is capable of evolving much more rapidly than is possible today. This is because certification differs from standardization in that it does not seek to define a single preferred behavior. Rather, no consensus is needed and certification only seeks to establish that a service makes reasonable use of overall network resources, regardless of whether it is considered an effective means of accomplishing a particular task. To simplify the potentially difficult task of inspecting programs for certification, we note that the latter need not be foolproof since it is not necessary to hold all programs to the same level of accountability. To support rapid experimentation, for example, it would be possible to modify the ANTS runtime to execute code that is newly certified (or even not certified) at a reduced rate, so that it consumes no more than 1% (say) of the available resources. Fresh code could similarly be certified for a restricted period in which a trial can be run. Together with a revocation mechanism, such techniques would allow new services to be gradually accepted as experience is gained with them. We also note that the availability of executable code raises the possibility of automatically testing services against common failure cases.

Our current dependence on certification begs the question of whether it is still useful to incur the costs of other security mechanisms. We believe that protection and resource allocation mechanisms are invaluable because they defend against accidental as well as malicious errors; we seek to expand our mechanisms to cover the ground nominally protected by certification. Even if this can be accomplished, however, it is likely that certification has a role to play in a practical active network. Signed code provides a trail to follow when problems arise. It also enables a spectrum of trust, whereby more trusted users are allowed access to wider and more powerful APIs. We have deferred the study of these tradeoffs and associated issues for which we can anticipate a need, such as revocation, to future research.

6 What services can be introduced?

We have found the most compelling use of capsules to be as a means of rapidly upgrading the services of large, wide-area networks such as the Internet. There is a clear need for such a mechanism because of difficulties that exist today. For example, IPv6 (the next version of the IP) is proving slow and difficult to deploy; if the Internet continues to be successful IPv6 will not be the last evolutionary step that is needed. In the meantime, backwards-compatible solutions such as Network Address Translation (NAT) boxes (which map a large set of internal addresses to a small set of external ones) are gaining ground because they are more readily deployed, despite the architectural difficulties they present [40]. Furthermore, today's Internet hampers experimentation: it was not possible to test the IPv6 candidates under real conditions before selecting a standard. Thus active networks would be of tremendous value if they facilitated the widespread deployment of even a small number of services such as multicast, mobility or IPv6.

In the remainder of this section we characterize the kind of services that ANTS is able to introduce and discuss this characterization using examples. We have found that ANTS is able to introduce a class of new services that are otherwise difficult to deploy. We have also been struck by the number of service variations that are enabled by using a programming language to combine a small set of node operations. This leads us to conclude that a viable active network could be extremely useful for the experimentation necessary to design new protocols. Despite this, it is clear that greater application experience is needed to judge the utility of active networks. No single application ("killer app") that proves the value of extensibility has emerged, and none is likely to because support for such an application can always be built-in to the base system. Rather, the advantages of extensibility accrue over time. Active networks are no different in this regard than extensible operating systems [21, 5], databases or language implementations [22].

Service	Capsule	Code (bytes)	Latency (us)	Slowdown
ANTS–PIM	MulticastData	3496	670	1.25
	JoinPrune	4325	975	1.82
	RegisterStop	1690	560	1.05
WebRoute	Query	1844	615	1.15
	Bind	1821	685	1.28
	Redirect	1843	600	1.12
	Activate	1991	620	1.16

Table 3. Latency and code size for sample ANTS capsule forwarding routines

6.1 Characterization of services

The reason capsules provide a powerful means for effecting change is that they deploy processing along network paths in a clean manner that is not dependent on the details of the path itself. This is in contrast with the existing mechanisms that are essentially administrative and proceed one router at a time. New services that can be deployed in ANTS but are difficult to introduce into the Internet today are thus those whose implementation is spread across multiple network elements. They are often variations on routing and forwarding. Unlike services constructed above the network layer, they can make direct use of topology, loss and load information to construct novel routing, flow setup, congestion handling and measurement services.

By analyzing our architecture we have identified the characteristics that a service must possess to be able to be readily introduced in ANTS:

- *Expressible.* The service code must be constructed using the restricted node API listed in Table 1.
- *Compact.* The service code must be smaller than a self-imposed limit of 16 KB to limit the impact of code distribution on the rest of the network.
- *Fast.* Each routine must run to completion at the forwarding rate, or it will be aborted by the node.
- *Incrementally deployable.* The service must not fail in the case that not all nodes are active.

We believe that many useful services can be constructed to have these characteristics. We have implemented at least five services (host mobility, source-specific multicast [46], path MTU queries, PIM-style multicast, and Web cache routing [45]), the last two of which were case studies of services that are intended to be realistic. Others have used ANTS to study a number of novel services: an auction service [23]; a reliable multicast protocol [24]; and a network-level packet cache [20].

As evidence of the utility of capsules, the strongest argument we can presently make is to provide examples of services that: have been promoted by others; are meritorious enough that they have received serious consideration by the Internet community; are difficult to introduce in the Internet today because they rely on network layer state and their implementation is spread across the network; and can be introduced in an equivalent form with ANTS-style capsules. We have identified a number of such services:

- multicast (PIM-SM [13], CBT [4], Express [17]);
- reliable multicast support ([31, 25], Cisco's PGM);
- explicit congestion notification (ECN) [34];
- PIP [14], a former IPv6 candidate; and
- anycasting [32].

We identified these services as candidates only gradually, as we gained experience constructing forwarding routines that possessed the required characteristics. It is not obvious how all of these services can be designed to meet the required criteria, and in the remainder of this section we discuss the criteria in light of these examples.

6.2 Discussion

6.2.1 Expressible

The small API available in our prototype has proved sufficient to express a variety of services. This is partly because its capabilities, such as the ability to place application-defined objects into the soft-store, are widely applicable, and partly because the forwarding routines themselves tend to act as "glue" that binds together these capabilities. It quickly became apparent as we wrote services that this code is a much more flexible form of glue than the traditional means of composition in networked systems, layering models, even compared to micro-protocol systems such as the *x*-kernel [19]. We note that Click [28], an experimental router infrastructure containing much finer-grain routines (such as queue manipulation), has recently emerged. We view systems such as Click as synergistic with active networks, since they provide a router with a native set of fine-grain APIs that ANTS capsules could compose into novel functions.

Many variations on a forwarding routine are often possible, such as multicast protocols that support reliable multicast, or changes in routing between Web caches that implement different search policies. This is significant because in ANTS a slightly different service is straightforward to use since its deployment is automatic. This could stimulate the real-world experimentation that is necessary to design new and improved protocols. Code also provides a wider interface with which to discover network properties than the "black-box" model that is used today, where properties are inferred from observations of traffic. For example, path MTU estimates can be obtained by recording the minimum MTU as a capsule is forwarded along a path (at connection setup time too) rather than relying on heuristics and error messages. This too enriches the variety of services that can be expressed.

The API does not readily allow two kinds of services to be constructed, both of which fall outside the design space of the ANTS programming model. It is difficult to embed long-lived processes in the network, such as a Web cache or transcoding engine. This is because the API does not support reliable storage or timers. Such capabilities could be added to the API by other means, for example an administrator upgrading an active node, but they cannot be extended at the capsule level. However, ANTS works in synergy with such complementary evolution. This is because new services can be written to take advantage of new capabilities as they are embedded in the network: forwarding routines simply query nodes for the availability of a particular feature and act accordingly. We speculate that one of the most valuable uses of ANTS may be new services that connect applications with network-embedded resources (caches, transcoding engines) in a more flexible way than the fixed proxies or transparent interception that is used today.

The second kind of service that is difficult to construct is that which cuts across many flows, such as firewall filtering. (But recall that one service can be composed of several cooperating types of capsule.) This is because network processing is explicitly selected for each capsule as it is injected into the network to support our security model. Instead, this kind of service is well-served by extending routers to be programmable by the administrator, as is done in [11]. This is because processing that cuts across many flows is often used to enforce policy at a point.

6.2.2 Compact and fast

Forwarding routines tend to be small because they tend to be "glue" code. We have not found it difficult to construct services that are smaller than our 16 KB limit, despite a naive transfer format that transfers Java classfiles directly. Other systems have similarly found that capsule-style code can be compact and often acts as glue [39, 16]. Inevitably though, this bound will limit the complexity of forwarding routines that can be constructed. We believe that a relatively small bound, on the order of the current 16 KB, will prove sufficient for the type of services that ANTS targets and that other deployment mechanisms will be more appropriate for more ambitious services, such as video transcoding.

The forwarding routines also tend to complete quickly. This is because all of the node API operations complete rapidly (none block on disk or network access) and the glue code typically combines them in a straightforward manner. The majority of the routines we have constructed have run within a factor of two of the basic capsule forwarding time.

As evidence to support these assertions, we provide in Table 3 the forwarding times and program code sizes for capsules of the two most realistic services we have implemented in ANTS: PIM-style multicast and Web cache routing. Forwarding time is given in microseconds across a single ANTS node, using the same experimental platform as described in Section 4.2. It is also given as a slowdown factor relative to results for the "null" ANTS capsule reported in Section 4.2. This shows the impact of executing a real routine that is more complex than IP forwarding. Each line represents one type of capsule; each service is comprised of multiple kinds of capsules that work together. Web `Query` capsules, for example, take 1.15 times as long to forward as the "null" ANTS capsule and have roughly 1.8 KB of associated code that is first demand loaded. We omit an explanation of the services themselves because it is not necessary to understand their operation for the purpose of this paper, and they are fairly involved. The interested reader is referred to a dissertation [45].

6.2.3 Incrementally deployable

Essentially all new Internet services must be able to be introduced incrementally or they cannot be deployed. In ANTS, this requires that services be able to function in a network in which not all nodes are active. Designing services for this case has proved to be a challenging but usually soluble task.

For example, some of the potential services we put forth, notably the multicast services such as PIM, are specified for a network region in which all routers participate in the implementation. It is not obvious how to define an equivalent service that works in a partially active network. Yet this can be done. To see how, consider that PIM can be run in an overlay, which is effectively a situation where only some of the network nodes implement the service — what is needed then is a dynamic means of establishing the tunnels of the overlay. We designed such a mechanism using capsules as part of our PIM-equivalent implementation in ANTS.

Similarly, active path MTU discovery requires that all nodes along the path be active or the result must be treated as an estimate. Technically, however, Internet path MTUs can only ever be estimates because they must allow for routes that change unexpectedly. To the extent that there are active nodes adjacent to bandwidth limited links, which typically have the smallest MTUs, the result of an active path MTU service will be a useful estimate.

6.3 Upgrading ANTS

An interesting class of services to consider are those that are not easily introduced without upgrading ANTS itself. These are primarily services that require changes to widely-held assumptions, of which we have identified the following examples:

- the way capsule types are computed and carried;
- the format of addresses;
- the code distribution mechanism;
- resource limiting via TTLs;
- and the node API.

These base assumptions are shared by active nodes throughout the network, and so are difficult to override locally. While it would perhaps be preferable to have an architecture that encoded no such assumptions, this position is untenable. The chief benefit of ANTS compared to the Internet is that the scope of node processing that is fixed at the outset in its function has been greatly reduced. When changes to widely-shared assumptions are necessary, they must be made by upgrading ANTS in the same way as protocols are upgraded in the Internet today, with backwards-compatibility, versioning or overlays. For this reason, ANTS capsules carry a version number.

We note, however, that there are often design workarounds that obviate the need to change the base assumptions. For example, though it is not possible to introduce IPv6 *per se* because the format of addresses is fixed, it still possible to increase the size of the address space and so solve the same problem. PIP [14] accomplishes this by using the existing addresses hierarchically with a technique known as source routing. As another example, while the node API is fixed, language-level introspection can be used to query whether a new feature is available at the local node and if so gain access to it. This is a powerful mechanism for supporting change, and ANTS includes the notion of extensions in its node API (Table 1) for this purpose.

7 Architectural observations

Active networks represent a different architectural perspective than traditional layered protocol stacks. In this section, we offer somewhat more speculative observations on network architecture that we have made in the course of our work and that have not appeared in the literature to the best of our knowledge.

7.1 Value of systematic change

As we have experimented with ANTS, we have realized the value of a systematic means of upgrading protocol processing within the network, as opposed to depending on backwards-compatibility. ANTS makes the new processing that a capsule can undergo within the network explicit, while depending on backwards-compatibility implicitly extends old processing.

It is quite remarkable how effectively backwards-compatibility has been leveraged to extend TCP/IP to new situations. However, an implicit means of extension ultimately has unexpected or adverse side-effects, while an explicit one has much cleaner semantics. This is already apparent in the case of NATs compared to IPv6, and we believe that many more situations will come to light. For example, consider network-embedded Web caches, such as Cisco's CacheDirector, that are currently being developed. For reasons of backwards-compatibility, they work in a manner that is transparent to their clients, spoofing the remote Web server. This may pose no direct problems today. However, research is now beginning to address how to share congestion information to improve the performance of short connections. In this context, transparent proxies have the potential to confuse hosts by mixing different congestion information under one name.

7.2 Dealing with heterogeneous nodes

Implementing active nodes has forced us to reckon with the differing capabilities of nodes at different network locations. A difference in style between ANTS and IP that has emerged is in how we accommodate this heterogeneity. IP essentially defines the minimal forwarding required for internetworking, with the expectation that this processing will run at all locations. On the other hand, in an active network we must confront the issue of different kinds and complexities of capsule processing running at different locations.

Our strategy has been to bind capsule routines at runtime to those nodes that have sufficient forwarding resources to execute them. We do this in a clean manner by: using capsule code to query the node environment and decide if there are sufficient resources; having active nodes protect themselves by unloading routines in the case that they take too long to run; and writing services that do not need to be run at all nodes to work correctly. IP routers then exist at the bottom of this organization as those nodes that are not active for any type of capsule. Architecturally, this strategy could be seen as the logical extension of the two-tier "edge and core" model recently in vogue with developments such as IETF Differentiated Services.

7.3 End-to-end argument

An initial concern of the active network approach was that it might conflict with the end-to-end argument [38, 6, 35]. With one exception, encryption, we have not found this to be the case, most likely because we view ANTS as simply a framework for expressing and deploying new services. The new services themselves can be well designed or poorly designed, depending on the skill of the developer. It is certainly possible to construct services that do not conflict with the end-to-end argument (such as the multicast variants described in this paper) and features of our architecture (such as the provision of soft-storage) are intended to encourage good design.

However, end-to-end encryption, such as that specified in IPv6 IPSEC, poses a challenge for the active network approach. When it is present, active code cannot readily operate on the packet payload. Two factors mitigate this problem. First, many ANTS services require access only to packet header fields, for example, routing variations. It is then possible to design encryption in a manner that exposes these fields. Second, end-to-end encryption that covers the entire payload, such as IPSEC, precludes many useful network-embedded services that are already deployed, such as firewalls, transparent proxies, and wireless boosters. For this reason, variant encryption standards that expose header fields are likely to emerge.

7.4 Localizing change

We have come to appreciate that changes in network services must be localized in their implementation if they are to be easily deployed. This observation holds regardless of the protocol layer of the change, and whether they are effected by means of an active network or not. Much of the value of the capsule model is that it allows changes to be made along an entire network path at one time, rather than at individual network locations. This new kind of locality greatly expands the scope of changes that can be made.

Arguably, IPv6 is proving slow and difficult to introduce because its changes are not easily localized. Since it replaces rather than extends existing addresses, it logically needs to be deployed at all network locations before it can be used. This is clearly not feasible, and an overlay and complicated transition plans are necessary to avoid long periods of restricted connectivity [15]. Conversely, NATs are rapidly gaining ground despite their drawbacks precisely because they are easy to deploy with only local changes. It is interesting to speculate whether incrementally deployed solutions to IPv6, such as PIP [14], would have enjoyed more success.

8 Conclusions

In this paper, we have reported substantial progress towards *active networks* as envisioned in [41]. In order to build a real, working system, we have revised some of the positions originally put forth in [41]. Nevertheless, even after this healthy dose of reality, we find that a surprising amount of the original vision still holds sway:

- Capsules have proved a worthwhile model because they provide a clean means of upgrading processing along an entire network path. This model of deployment is considerably more powerful than the pointwise administrative upgrades that are the norm today. To implement capsules efficiently, we have come to depend on the demand loading of code and on traffic patterns for which code caching is effective.
- We have partly succeeded in designing a network that any untrusted user can freely customize. We have managed to isolate different services from each other without trust or centralized control, but not to protect the network as a whole from untrusted services. To accomplish the latter in the general case, we have fallen back on certification by a trusted authority until better solutions are found. This is a measure that runs counter to our vision but which still allows easy change relative to standards bodies today.
- We have found the most compelling application of capsules to be network layer service evolution, rather than the migration of application code to locations within the network. We have found capsule code to be well-suited to the task of introducing many variations of a service, and hence valuable for experimentation. We also speculate that capsule code will act in synergy with network embedded devices (caches and transcoders) that are deployed by other means, such that both will work more effectively together.

Acknowledgments

This work has benefited from the advice and assistance of many. We are grateful to the members of the SDS group at MIT, and in particular wish to thank Tom Anderson, Steve Garland, John Guttag, Hank Levy and David Tennenhouse for their encouragement and guidance. The SOSP referees and our shepherd, Jay Lepreau, have further helped us to improve this work with their careful feedback.

References

[1] D. S. Alexander et al. Active Bridging. In *Proc. SIGCOMM'97*, pages 101–111, Cannes, France, Sep 1997. ACM.

[2] E. Amir et al. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proc. SIGCOMM'98*, pages 178–189. ACM, Sep 1998.

[3] F. Baker. Requirements for IP Version 4 Routers. Request for Comments 1812, IETF, June 1995.

[4] A. Ballardie et al. Core based Trees. In *Proc. SIGCOMM'93*, pages 85–95, San Francisco, CA, 1993. ACM.

[5] B. Bershad et al. Extensibility, Safety and Performance in the SPIN Operating System. In *15th Symp. on Operating Systems Principles*, pages 267–283. ACM, Dec. 1995.

[6] S. Bhattacharjee et al. Active Networking and the End-to-End Argument. In *Intl. Conf. on Network Protocols (ICNP'97)*, pages 220–228, Atlanta, GA, Oct 1997. IEEE.

[7] A. Birrell et al. Network objects. In *14th Symp. on Operating Systems Principles (SOSP'93)*, pages 217–230, Ashville, NC, Dec. 1993. ACM.

[8] B. Braden et al. Recommendations on queue management and congestion avoidance in the internet. Request for Comments 2309, IETF, April 1998.

[9] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM'88*, pages 106–114, Stanford, CA, Aug. 1988. ACM.

[10] D. Cronquist et al. Architecture design of reconfigurable pipelined datapaths. In *Conf. on Advanced Research in VLSI*, pages 23–40, Atlanta, GA, March 1999.

[11] D. Decasper et al. Router Plugins: A Software Architecture for Next Generation Routers. In *Proc. SIGCOMM'98*, pages 229–240. ACM, Sep 1998.

[12] D. Decasper and B. Plattner. DAN: Distributed Code Caching for Active Networks. In *Conf. on Computer Communications (INFOCOM'98)*, San Francisco, CA, April 1998. IEEE.

[13] S. Deering et al. Protocol independent multicast-sparse mode (PIM-SM): Protocol Specification. Request For Comments 2362, IETF, June 1998.

[14] P. Francis and R. Gondivan. Flexible Routing and Addressing for a Next Generation IP. In *Proc. SIGCOMM'94*, pages 116–125. ACM, Sep 1994.

[15] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. Request For Comments 1933, IETF, April 1996.

[16] M. Hicks et al. PLANnet: An Active Internetwork. In *Conf. on Computer Communications (INFOCOM'99)*, pages 1124–1133, New York, NY, Mar. 1999. IEEE.

[17] H. Holbrook and D. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *Proc. SIGCOMM'99*, pages 65–78, Boston, MA, Sep 1999. ACM.

[18] W. Hsieh et al. Type Safe Casting. *Software Practice and Experience*, 28(7), Sep. 1998.

[19] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, Jan. 1991.

[20] E. Johnson. A Protocol for Network Level Caching. M.Eng Thesis, Massachusetts Institute of Technology, May 1998.

[21] M. F. Kaashoek et al. Application performance and flexibility on exokernel systems. In *Proc. of the 16th Symp. on Operating Systems Principles (SOSP'97)*, pages 52–65. ACM, Oct. 1997.

[22] G. Kiczales et al. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] U. Legedza et al. Improving the Performance of Distributed Applications Using Active Networks. In *Conf. on Computer Communications (INFOCOM'98)*, San Francisco, CA, April 1998. IEEE.

[24] L. Lehman et al. Active Reliable Multicast. In *Conf. on Computer Communications (INFOCOM'98)*, San Francisco, CA, April 1998. IEEE.

[25] B. N. Levine and J. Garcia-Luna-Aceves. Improving internet multicast with routing labels. In *Intl. Conf. on Network Protocols (ICNP'97)*, pages 241–250, Atlanta, GA, Oct. 1997. IEEE.

[26] D. Mazières et al. Separating Key Management from File System Security. In *17th Symp. on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, 1999. ACM.

[27] P. Menage. RCANE: A resource controlled framework for active network services. In *1st Intl. Working Conf. on Active Networks (IWAN'99)*, Berlin, Germany, Jul 1999.

[28] R. Morris et al. The Click modular router. In *17th Symp. on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, 1999. ACM.

[29] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proc. PLDI'98 Conf. on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[30] E. Nygren et al. PAN: A High-Performance Active Network Node Supporting Multiple Code Systems. In *2nd Conf. on Open Architectures and Network Programming (OPENARCH'99)*, pages 78–89, New York, NY, Mar. 1999. IEEE.

[31] C. Papadopoulos et al. An Error Control Scheme for Large-Scale Multicast Applications. In *Conf. on Computer Communications (INFOCOM'98)*, San Francisco, CA, April 1998.

[32] C. Partridge et al. Host Anycasting Service. Request For Comments 1546, IETF, Nov. 1993.

[33] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. SIGCOMM'97*, pages 139–152, Cannes, France, Sep 1997. ACM.

[34] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. Request For Comments 2481, IETF, Jan. 1999.

[35] D. P. Reed et al. Commentaries on the Active Networking and End-to-End Arguments. *IEEE Network Magazine*, 12(3):69–71, May/June 1998.

[36] R. Rivest. The MD5 Message-Digest Algorithm. Request For Comments 1321, IETF, April 1992.

[37] M. Robshaw. On Recent Results for MD2, MD4, and MD5. *RSA Laboratories Bulletin*, Nov. 1996.

[38] J. H. Saltzer et al. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[39] B. Schwartz et al. Smart Packets for Active Networks. In *2nd Conf. on Open Architectures and Network Programming (OPENARCH'99)*, New York, NY, Mar. 1999. IEEE.

[40] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations. Request For Comments 2663, IETF, Aug 1999.

[41] D. L. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. In *Multimedia Computing and Networking 96*, San Jose, CA, Jan. 1996. A revised version appears in CCR Vol. 26, No. 2 (April 1996).

[42] The Tolly Group. Cisco 12000 Series GSR POS: Performance Evaluation. No. 199128, Manasquan, NJ, Sep. 1999.

[43] J. van der Merwe et al. The Tempest – A Practical Framework for Network Programmability. *IEEE Network Magazine*, 12(3), May/June 1998.

[44] R. Wahbe et al. Efficient Software-based Fault Isolation. In *14th Symp. on Operating Systems Principles (SOSP'93)*, pages 203–216. ACM, Dec. 1993.

[45] D. Wetherall. *Service Introduction in an Active Network*. PhD thesis, Massachusetts Institute of Technology, Feb. 1999.

[46] D. Wetherall et al. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *1st Conf. on Open Architectures and Network Programming (OPENARCH'98)*, pages 117–129, San Francisco, CA, Apr 1998. IEEE.

[47] D. J. Wetherall and D. L. Tennenhouse. The ACTIVE IP Option. In *7th SIGOPS European Workshop*, Connemara, Ireland, Sep 1996. ACM.

[48] Y. Yemini and S. da Silva. Towards Programmable Networks. In *Intl. Work. on Dist. Systems Operations and Management*, Italy, Oct. 1996.