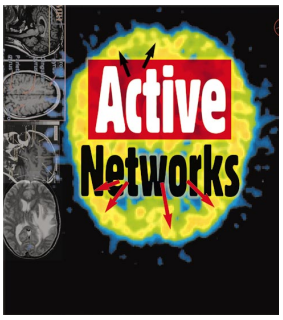

ANTS: Network Services Without the Red Tape

ANTS, a new approach to deploying network services, bases interoperability on a programmable network model, not on individual networking protocols. The promise is automatic protocol upgrades, which can hasten progress toward a more responsive Internet.

David Wetherall
University of Washington

John Guttag
Massachusetts Institute of Technology

David Tennenhouse
US Defense Advanced Research Projects Agency

How well distributed computing systems perform depends a great deal on the network services used to move information among their machines. Yet despite this close correspondence, network services have evolved much more slowly than any other part of the distributed system environment. It's not that the networking community lacks innovative ideas: Internet Protocol version 6, Mobile IP, IP Multicast, and Integrated/Differentiated Services aim to support multimedia applications more effectively and to accommodate more hosts, many of them mobile. Unfortunately, however, progress in implementing these solutions lags far behind the identified need.

The main problem is the way network protocols must change. First, network protocols are the main vehicle for achieving interoperability, so any candidate internetworking protocol has to become a standard. This means possibly years between the time someone identifies a need and the time everyone agrees on how to address it. Once the new protocol has been accepted, more delays occur because it has to be deployed manually and in a way that is compatible with the existing protocols.

At the Massachusetts Institute of Technology, we have developed an approach that will let the networking community largely bypass this long and tedious process. The idea behind the ANTS (Active Node Transfer System) architecture, which is based on the concept of *active networks*,[1] is to standardize on a communication model rather than individual communication protocols. The model is designed to support many protocols simultaneously, in such a way that only the parties that use a protocol—not the entire networking community—need agree on how it is built and used.

In an ANTS-based network, applications introduce new protocols into the network by specifying the routines to be executed at the nodes that forward their packets. Users will probably not build protocols directly, but will select from among vendor offerings. As applications send their packets into the network, the associated routines are automatically deployed to the necessary nodes through mobile code techniques. The advantage of this is that parts of the network need not go offline to reconfigure the nodes for a particular application.

The ANTS toolkit is a prototype implementation of the ANTS architecture that we created to validate our approach to creating and deploying protocols.[2] We and other members of our group have used the toolkit to study an auction service, a reliable multicast protocol, a Web caching service, multicast routing, a network-level packet cache, and a defense against TCP SYN-flooding. The toolkit's latest release and descriptions of these experiments are on our Web site (http://www.sds.lcs.mit.edu/activeware), which also has pointers to related work. We have also deployed ANTS nodes at different sites as part of the DARPA-sponsored ABONE, an experimental active network.

ARCHITECTURE

An ANTS-based network can be thought of as a conventional wide area network in which some of the IP routers have been replaced by active nodes, which execute the ANTS runtime. Applications communicate over the network by exchanging special kinds of packets called capsules. What distinguishes an ANTS network from today's Internet is the processing of capsules when they pass through active nodes. Figure 1 illustrates this approach. By defining new types of capsules and their processing at nodes, application developers can create new network services.

ANTS has three main components. *Capsules*, which take the place of traditional packets, contain application data and describe the processing they require within the network. *Active nodes*, which replace select routers and end nodes, perform this processing and

0018-9162/99/$10.00 © 1999 IEEE

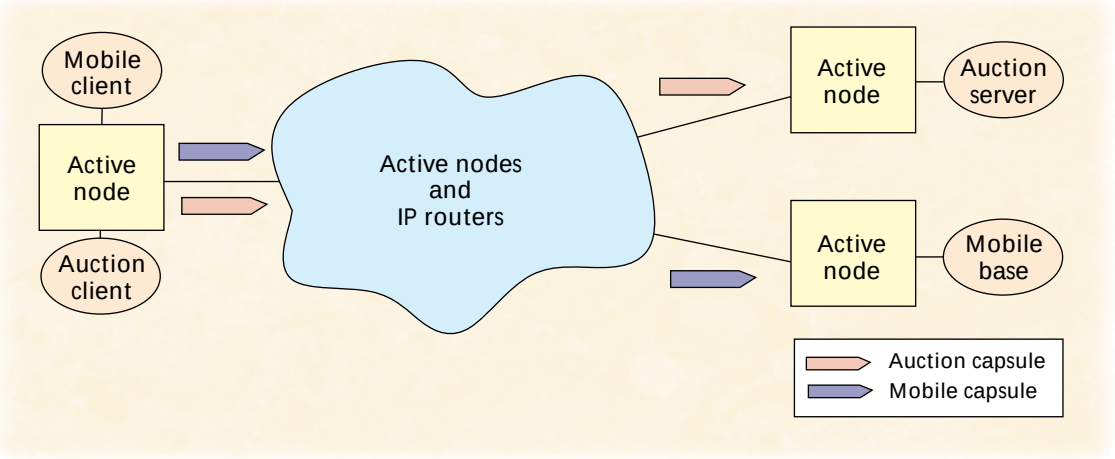

Figure 1. How applications use an ANTS network. Different applications inject different types of capsules into the network (represented by different colors in the figure)—in this case mobility and auction capsules. Each capsule type is associated with a different forwarding routine. As a capsule traverses the network, it passes through active nodes and conventional IP routers until it reaches its destination. At IP routers, capsules are forwarded in the normal manner. At active nodes, capsules are forwarded by executing the associated forwarding routine. For example, some types of mobile capsules might store forwarding pointers at active nodes and other types might follow these pointers. In this manner, new network services can be implemented.

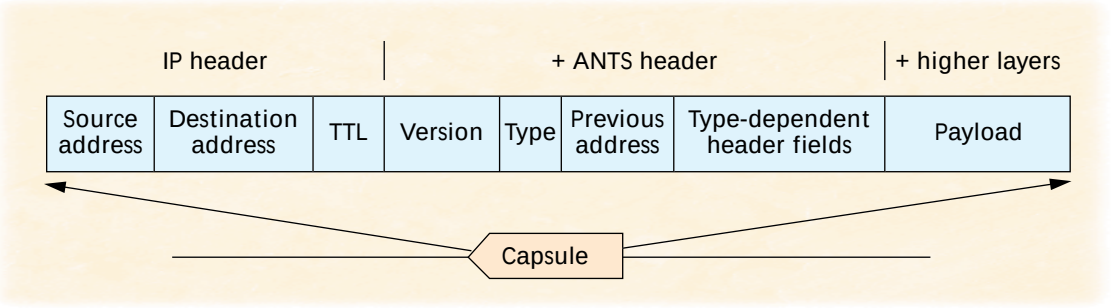

Figure 2. Format of an ANTS capsule. Each capsule begins with the regular IP header fields. The ANTS header then contains information about the capsule itself, including the protocol and forwarding routine (in the type field) and custom fields for use by the forwarding routine that vary with the kind of capsule (the type-dependent header fields). It also contains fields all active nodes need, such as the version and previous address fields. Finally, the payload of the capsule contains higher layer headers and data that are carried across the network.

maintain the state associated with network services in a way that keeps services from damaging the network or interfering with each other. A *code distribution system* ensures that the processing descriptions are automatically transferred to the nodes that require them.

Capsules

In ANTS, application developers build a network service by combining related types of capsules to create a *protocol*. Within each capsule is a reference to a specific *forwarding routine*—directions for processing that capsule at each active node. Capsules within a protocol can communicate with each other through state that is shared at active nodes. For example, one capsule type can set up location information at active nodes that other capsule types will use to reach a mobile host. They can also spawn other capsules that belong to the same protocol. For example, an *auction bid* capsule may dynamically create a *bid-too-low* capsule when its forwarding routine runs. The active node provides protection by ensuring that capsule types that belong to different protocols cannot interfere with each other.

Figure 2 shows the contents of each capsule. The capsule carries the regular IP header fields plus extension fields specific to ANTS. "Type" (fifth block from

A key difficulty in designing a programmable network is how to safely run the forwarding routines specified by applications that use the network.

the left) is an identifier that tells the associated protocol and forwarding routine. It is based on a secure hash of the forwarding routines in the protocol, which provides a fingerprint of the code. In the ANTS toolkit, we use MD5 as the hash function.

The fingerprint is important for two reasons. First, since inverting the hash function is not feasible, the code can't be spoofed. When a node receives code that purports to match a particular capsule type, it can easily verify the correspondence by using the hash function. There is no need to trust external parties. Second, because the identifier is automatically generated from the protocol's contents, there is no need to have a central authority to allocate type identifiers. This differs from the standards-based approach currently used.

The rest of the capsule contains a shared header with fields common to all capsules, including version information and addresses used by the code distribution system. Next are header fields that the forwarding routines use directly, which vary with the capsule type. For example, a congestion notification capsule might carry an extra header field that the forwarding routine uses to indicate whether congestion has been encountered so far. Finally, the payload contains higher layer information that is carried across the network and exchanged between applications.

Active nodes

A key difficulty in designing a programmable network is how to safely run the forwarding routines specified by applications that use the network. Not only must the network protect itself from rogue protocols, it must also ensure that each protocol has an independent view of the network and efficiently allocate resources among them.

We have found that the best approach to satisfying these goals is to execute protocols within a restricted environment that limits their access to shared resources. In ANTS, active nodes are this environment. Each node provides a set of primitives that are used to construct forwarding routines. Each node runs forwarding routines within an execution model that controls the resources they can access.

Node primitives. We selected 10 primitives according to our experience with an earlier system. This number seems small, but with the right choice of primitives it has proved sufficient to express dozens of useful forwarding routines. The choice of primitives is important because it determines the kinds of routines that applications can deploy. For example, without the ability to store and access state at nodes, different capsules would not be able to communicate with each other. The primitives must also be a good match for the task at hand if the capsule's program is to be compact and run efficiently. For example, a forwarding routine can find a node's neighbors either by walking the entire routing table looking for adjacent nodes or by querying the node directly. The direct query can be represented compactly and executed efficiently as a built-in node primitive; the other program cannot.

The node primitives in ANTS roughly fall into three categories.

- *Environment calls* return information about the local node environment, such as its address. We have provided few calls at present, though it would be straightforward to add more calls as needed—for example, calls that examine forwarding statistics.
- *Storage calls* manipulate a soft-store of application-defined objects, including other capsules. The store is "soft" because objects placed in it are not guaranteed to be present after the forwarding routine completes. To improve performance, however, the node caches regularly used objects. It also removes an object from the store when an application-defined interval passes, thereby ensuring that the network does not retain stale information.
- *Control operations* direct the flow of capsule processing to other parts of the network. They are used to route capsules toward other nodes using default (shortest-path) routing or deliver them to local applications. New routing services typically express their routes as default routes, since this is a robust way to navigate the network. If no control operation is called, the capsule is discarded.

At some point, we may experiment with additional categories (authentication, scheduling, and transcoding are excellent candidates), but so far we have been concerned primarily with defining enough operations to demonstrate the validity of our approach.

Execution model. Because we assume that computation at active nodes is mainly to support communication tasks, we optimized the execution model to support generalized packet forwarding rather than distributed computing. The model has four main characteristics:

- *Fixed forwarding routine.* The forwarding routine is fixed at the sender, when the capsule is injected into the network, and may not change as the capsule traverses the network. This ensures that a rogue application cannot control the handling of another application's capsules.
- *Selective execution.* Not all nodes must execute a particular forwarding routine. Some may elect not

to, depending on their available resources and security policies. In this case, the node acts as a regular IP router for the corresponding capsule type. A nice by-product of this mechanism is that it naturally supports ANTS-based networks that contain unmodified IP routers—the IP routers simply elect not to run any ANTS forwarding routines.

- *Resource limits*. Because forwarding routines are not trusted, nodes limit the resources they consume when run. Nodes ensure that forwarding routines complete quickly and bound how much memory and bandwidth the routines can use.
- *Protocol-based protection*. The protocol that a capsule belongs to determines what data that capsule can access while in the network—only capsules belonging to the same protocol can share state. This protection scheme keeps capsules belonging to other protocols from accidentally or maliciously interfering with each other.

When a capsule arrives at a node, its associated forwarding routine runs to completion. The routine can manipulate the capsule's header fields and payload as it runs. It initiates any further needs, such as sending the capsule on to a remote node. The routine must also enter into the soft-store any information that should be kept at the node for subsequent capsules belonging to the same protocol.

While the forwarding routines are being run, the nodes monitor their integrity and handle any errors that arise. To keep a faulty routine from corrupting an entire node, we rely on the safety mechanisms of mobile code technologies. Our prototype implementation uses Java, but we could have also used software-based fault isolation[3] or proof-carrying code.[4] Traditional operating system protection schemes, on the other hand, are too heavyweight for this task because they cannot run efficiently at the forwarding rate.

Finally, the node limits the resources that a forwarding routine can consume when run. This prevents one protocol from starving another of network resources and improves the network's robustness. At each node, forwarding routines are monitored as they run to prevent them from running too long, sending too many capsules, or using too much of the soft-store. Across nodes, infinite forwarding loops are broken using the IP Time-To-Live (TTL) field. The TTL is used in the same way it is currently used to break routing loops in the Internet.

A more difficult problem is how to keep a protocol from consuming a bounded but unreasonably large portion of the network's resources. For example, a poorly written protocol might saturate a link if it causes a capsule to ping-pong across it many times. In ANTS, we address this problem by having a trusted authority certify forwarding routines. This mechanism differs from the rest of the node design, which can work with untrusted code, and is an important area for further research.

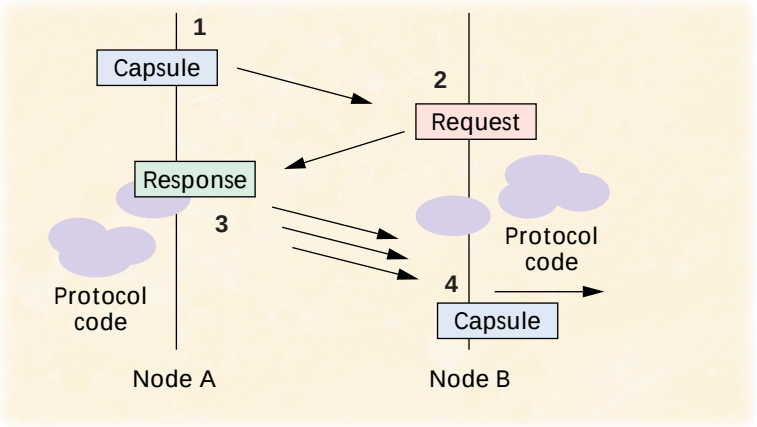

Figure 3. Sequence of events in loading protocol code. (1) Node A forwards a capsule to Node B, the next active node, and becomes the previous node. (2) Node B, the loading node, generates a load request to Node A to get code that matches that capsule's type if the required code is not already there. (3) When Node A receives the request, it generates a series of load-response capsules that carry the requested protocol code in parts to Node B. (4) As Node B receives the load responses, it assembles them to form the requested protocol code and checks the result. When the assembly is complete, Node B uses the transferred code to forward the capsule that triggered the load sequence in (1).

Code distribution system

The code distribution system sends forwarding routines to nodes where they must be run. Although ANTS will accommodate many new protocols over time, we expect that—as is true in the Internet today—only a few protocols will account for virtually all traffic at any instant. We do not believe this will change, even in a more dynamic network, because of higher layer flows and a dependence on third-party software.

To implement capsules efficiently for this operating regime, ANTS transfers forwarding routines separately from the capsules they are associated with and caches the forwarding routines at nodes. At the network's edges, applications provide forwarding routines to their local node before sending the corresponding capsules. Within the network, a lightweight protocol is used to transfer the code along the path that the capsule follows, if it is not already cached. The result is that applications can send capsules at any time, without first setting up a connection.

Our code distribution protocol provides rapid but unreliable transfer of short routines between adjacent active nodes: There is a limit of 16 Kbytes on each protocol's code to limit the effects of code transfer on the rest of the network. Most services we have built easily fit within this limit.

Figure 3 shows the sequence of events in code loading. This scheme has two important properties. First, it is adaptive and scalable. The reliance of a node on the previous node draws code along the network paths with nodes that need it. As more and more capsules are transferred, a region emerges in which the same processing is invoked repeatedly. Code transfer then

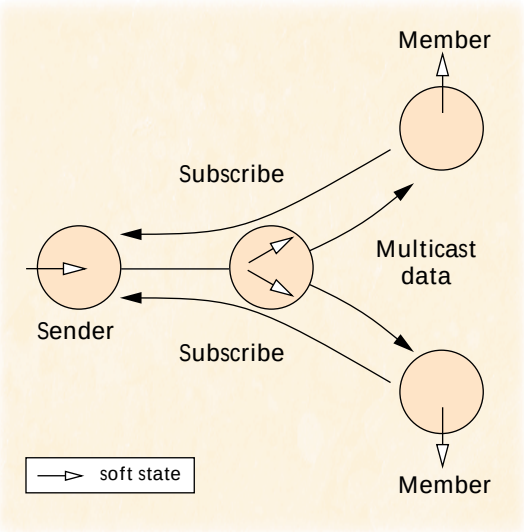

Figure 4. Paths of multicast capsules. Members of the multicast group send subscribe capsules to the sender to establish soft-state forwarding pointers at active nodes in the network. Multicast data capsules sent by the sender then follow these forwarding pointers to deliver a copy of the message to each member.

becomes unnecessary. If network paths change, code transfer simply resumes accordingly.

Second, the scheme is optimistic. It is intended to succeed rapidly so that it does not cause significant delays to applications. Code loading does occasionally fail, most often because network congestion causes a code distribution message to be lost. In this case, ANTS simply treats the capsule that triggered the load as lost. Higher level software running at end systems, such as TCP, then provides the appropriate level of reliability.

PROGRAMMING

Because capsules specify new forwarding routines, ANTS protocols are typically variations on IP forwarding. Unlike services that work above IP, they can directly exploit the network's topology and load information. In this way, service designers can construct novel types of routing, flow setup, congestion handling, and measurement protocols.

Our multicast protocol demonstrates how ANTS can be used. Multicast, which lets one host communicate with many other hosts by replicating messages, has been the subject of much research. Indeed, the Internet is being upgraded to permit multicast. The example we present here is necessarily simple, but works under realistic conditions, such as networks with a mix of active nodes and IP routers and asymmetric routes.

Our multicast protocol is a single-source version of IP multicast,[5] built with two cooperating capsule types. Figure 4 illustrates their interaction. The subscribe capsule is sent periodically by applications that want to receive messages sent to the group. It travels across the network to a particular sender. Along the way it refreshes soft-state forwarding pointers that are kept at nodes to form a distribution tree. The multicast data capsule delivers a copy of a message to each group member. It simply routes itself along the distribution tree by using the pointers it finds in the soft store, spawning new copies of the message as needed.

Figures 5 and 6 show the forwarding routines for these two capsules, respectively. In Figure 5, the subscribe capsule begins by looking up the forwarding pointers for the group in the node's soft store, creating a new pointer if none are found. To separate the forwarding pointers of different groups in use simultaneously, our protocol stores the pointers under a key that combines the group and sender addresses. Once the capsule locates the pointers, it adds a new entry that points in the direction the subscribe capsule has come from, if such a pointer is not already there. The capsule then continues toward the sender if it is time to refresh upstream pointers.

In Figure 6, the multicast data capsule travels through the network by following the forwarding pointers. When a copy of the capsule reaches its destination, an empty set of forwarding pointers signifies that it is to be delivered to a local application that is a member of the group.

Together, these capsules provide a service with the central property of network-based multicast: efficient use of bandwidth. The service differs from IP multicast in two significant respects. First, because it is local to the nodes that use the protocol, multicast-capable routers need not be separately identified or organize themselves into a tree. Second, it provides a different multicast primitive, since members subscribe to the combination of a group and sender. This choice is typical of the flexibility that ANTS offers. If applications need multiple senders, they can form multiple distribution trees by having members subscribe to each sender. Alternatively, they can consider the sender the root of a core-based tree,[6] and route capsules up the tree toward the root and then down other branches.

PROTOTYPE IMPLEMENTATION

The ANTS toolkit is written entirely in Java, as a stand-alone network system rather than an extension of an existing IP implementation, and transfers forwarding routines in Java's classfile format. We typically run the toolkit as a user-level process under Unix, though it can run in any standard Java environment. We chose Java because of its support for safety and mobility (through bytecodes and their verification). Java's flexibility as a high-level language and its support for dynamic linking/loading, multithreading, and

standard libraries have also let us evolve our design while maintaining a small code base (approximately 10,000 lines).

Java classes

In the toolkit, nodes, capsules, applications, and other entities are each implemented as a corresponding Java class. When each ANTS runtime is started, its root thread creates a single `node` object, one channel object for each local network interface, and one application object for each local distributed application. Applications then communicate by exchanging capsules, sending them via the local node, which transmits them as packets using the services of the local channels. Conversely, when the channel receives packets from the network, it attempts to convert them to objects of the corresponding capsule class. If the required forwarding routine is not at the node, the node retains the packet and uses the code distribution protocol to fetch the routine. The node then runs the routine to forward the capsule, and the process repeats.

Measurement and performance

Although we did not build the toolkit for performance, we did run a few tests to gain insight into how architectural decisions affect performance. We measured the performance of the toolkit on a Sun UltraSparc 1 (167 MHz) running Solaris 2.6 and connected with a 100-Mbps Ethernet.

We found that ANTS delivered roughly 1,700 capsules per second for small capsules and 16 Mbps for large (Ethernet-size) capsules. This performance level is good enough to deploy ANTS nodes in parts of the Internet today—for example, wireless links and access links through T1 (1.5-Mbps) rates. To place these measurements in context, we measured the performance of relays, which blindly forward packets, written in Java and C and running at user level. We found that ANTS achieves most of the throughput of a Java relay, adding approximately 35 percent overhead for 512-byte capsules, of which only five percent is fundamental. The C relay is faster by a factor of four, and it is likely that an in-kernel implementation would be faster by a factor of at least two again. Absolute system performance is *thus* limited by both user-level and Java-based operation, neither of which ANTS requires.

These observations suggest that we could build much faster ANTS implementations without resorting to custom hardware and operating system support. In fact, PAN[7] implements substantially the same architecture as ANTS with in-kernel binary forwarding routines and can saturate a 100-Mbps Ethernet with 1-Kbyte packets.

We also measured the latency of capsule forwarding and code loading, and found both to be reasonable. ANTS adds a constant overhead of approximately 400

```
// on entry:
// group = multicast group
// sender = multicast sender
// reverse = last visited node

// look up forwarding record
Object[] m = (Object[])n.get(group, sender);

// or make a new empty one if necessary
if (m == null) {
        m = new Object[1];
        m[0] = new Long(0);
        n.put(group, sender, m, IDLE);
}

// are we at an intermediate node?
add: if (reverse != 0) {
        // does it contain our info?
        for (int i = 1; i < m.length; i++) {
                Integer r = (Integer)(m[i]);
                if (r.intValue() == reverse) break add;
        }

        // if not, add it
        int len = m.length;
        Object[] nn = new Object[len+1];
        System.arraycopy(m,0,nn,0,len);
        nn[len] = new Integer(reverse);
        m = nn;
        n.put(group, sender, m, IDLE);
}

// need to refresh upstream entry?
long time = n.time();
long last = ((Long)(m[0])).longValue();
if (time - last < RATE) return;

// if so, update route and continue
m[0] = new Long(time);
if (n.getAddress() != sender) {
        reverse = n.getAddress();
        n.routeForNode(this, sender);
}
```

Figure 5. Forwarding routine for the* subscribe *capsules depicted in Figure 4. The routine's main goal is to establish pointers for subsequent capsules to follow to reach the members of the group.

```
// look up forwarding record
Object[] m = (Object[])n.get(group, sender);
// must find it to continue
if (m != null) {
        if (m.length > 1) {
                // send a copy every way
                for (int i = 1; i < m.length; i++) {
                        Integer r = (Integer)m[i];
                        n.routeForNode(this, r.intValue());
                }
        } else {
                // or deliver to application
                n.deliverToApp(this, dpt);
        }
}
```

Figure 6. Forwarding routine for* multicast data *capsules in Figure 4. The routine's main goal is to follow the pointers established in Figure 5's routine so that a copy of the capsule reaches each member of the group.

µs per capsule to the latency of the C relay. This means that there are no hidden data-dependent costs (extra copies) in implementing our forwarding model. The cost of code loading at a node was approximately 1 ms per code distribution message, with up to a maximum of 16 messages to transfer protocol code. However, the code is typically transferred only once and then forwards many thousands of capsules thereafter, so this overhead should have only a negligible effect on network performance.

Finally, to describe the costs of forwarding routines themselves, we measured the performance of a multicast service. To be realistic, we used an ANTS version of the sparse-mode Protocol Independent Multicast (PIM-SM) service, which is based on current Internet specifications,[8] rather than our simple example. We found that the protocol code for this service totaled 9.5 Kbytes, and sending multicast data capsules slowed forwarding by approximately 25 percent compared to blind forwarding. Again, this data shows that ANTS is a viable way to introduce useful protocols.

Our experience with ANTS suggests that active networks can cut through the red tape that impedes innovation in networking. A fully deployed ANTS network would foster innovation by allowing applications to freely select services, whether new or old, that best meet their individual needs. There are two obstacles to such a deployment. On the technical side, ANTS must be made more secure and its performance enhanced. We are working on both of these tasks. On the economic side, there is a large installed base of conventional routers and networking software. Because we designed ANTS to be compatible with the existing infrastructure, the networking community should be able to see many of its benefits relatively quickly as our ideas are gradually incorporated into the Internet. ❖

Acknowledgments

We thank the members of the Software Devices and Systems Group at MIT's Laboratory for Computer Science. This work was supported by DARPA, monitored by the Office of Naval Research, under contract N66001-96-C-8522, and by seed funding from Sun Microsystems.

References

1. D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Computer Comm. Rev.*, Apr. 1996, pp. 5-18.
2. D. Wetherall, "Service Introduction in an Active Network," PhD dissertation, Massachusetts Inst. of Tech., Cambridge, Mass., Feb. 1999.
3. R. Wahbe et al., "Efficient Software-Based Fault Isolation," *Proc. 14th Symp. Operating Systems Principles*, ACM Press, New York, 1993, pp. 203-216.
4. G. Necula and P. Lee, "The Design and Implementation of a Certifying Compiler," *Proc. Programming Language Design and Implementation 98*, ACM Press, New York, 1998, pp. 333-344.
5. S. Deering, "Host Extensions for IP Multicasting," *Request for Comments 1112*, Aug. 1989; http://www.ietf.org.
6. A. Ballardie, P. Francis, and J. Crowcroft, "Core Based Trees," *Proc. SIGCOMM 93*, ACM Press, New York, 1993, pp. 85-95.
7. E. Nygren, S. Garland, and M.F. Kaashoek, "PAN: A High-Performance Active Network Node Supporting Multiple Code Systems," *Proc. Open Architectures and Network Programming 99*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 100-109.
8. D. Estrin et al., "Protocol Independent Multicast—Sparse Mode (PIM-SM): Protocol Specification," *Request for Comments 2362*, June 1998; http://www.ietf.org.

David Wetherall is an assistant professor of computer science at the University of Washington (as of June 1999), where his primary research interests are distributed systems, programming languages, compilers, mobile code, and operating systems. He received a PhD in computer science from the Massachusetts Institute of Technology, where he did pioneering work on active networks, including the development of ANTS. Contact him at djw@cs.washington.edu.

John Guttag is head of the Electrical Engineering and Computer Science Department at the Massachusetts Institute of Technology. He also leads the Software Devices and Systems Group at MIT's Laboratory for Computer Science, a group that does research in networking, distributed computing, computer and communications security, and wireless communications. He received a PhD in computer science from the University of Toronto. Contact him at guttag@lcs.mit.edu.

David Tennenhouse is director of the Information Technology Office of the US Defense Advanced Research Projects Agency, where he is responsible for information technology issues of strategic concern. He is temporarily assigned to DARPA from MIT's Laboratory for Computer Science and the Sloan School of Management, where he conducted research on systems issues related to networks, software radio, distributed computing, media processing, and the effect of information technology on organizations. Tennenhouse received a PhD in computer science from the University of Cambridge. Contact him at dtennenhouse@darpa.mil.