To appear in IEEE OPENARCH'98, San Francisco, CA, April 1998

ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols

David J. Wetherall, John V. Guttag and David L. Tennenhouse*

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

Abstract

We present a novel approach to building and deploying network protocols. The approach is based on mobile code, demand loading, and caching techniques. The architecture of our system allows new protocols to be dynamically deployed at both routers and end systems, without the need for coordination and without unwanted interaction between co-existing protocols.

In this paper, we describe our architecture and its realization in a prototype implementation. To demonstrate how to exploit our architecture, we present two simple protocols that operate within our prototype to introduce multicast and mobility services into a network that initially lacks them.

1 Introduction

The performance of modern distributed computing is heavily dependent upon the network services used to move information among machines. Curiously, however, the evolution of these services has been much slower than the evolution of almost any other part of the environment on which computing systems are built.

The slow evolution is attributable neither to a lack of need nor to a lack of innovative ideas. In the case of IP, for example, changes are underway to better support multimedia applications, as well as to accommodate a larger number of potentially mobile hosts [4, 2, 3, 14]. Unfortunately, though agreement on the need for these changes was reached many years ago, they are still not fully deployed.

The problem is that the current process of changing network protocols is both lengthy and difficult. It requires standardization, since internetworking protocols are the basis of interoperability. This means that years may elapse between the time the need becomes apparent and the time consensus is reached on how to address that need. Furthermore, once the new protocol has been accepted, deployment is difficult. It must be done manually and in a backwards compatible fashion, since there is no automatic mechanism for upgrading functionality or dealing automatically with multiple protocols.

This paper presents a new approach to network service innovation that addresses all of these problems. The essence of our approach is to standardize a communication model (rather than individual communication protocols) that allows uncoordinated deployment of co-existing protocols. We have developed an active network [16, 17] toolkit, called ANTS[1], in which new protocols are automatically deployed at both intermediate nodes and end systems by using mobile code techniques. Our architecture views the network as a (somewhat restricted) distributed programming system, and provides a programming language-like model for expressing new protocols in terms of operations at nodes. Compared with alternative systems in which new protocols may be formed by selecting from a library of components, e.g., the x-kernel [9], ANTS provides the greater flexibility that accompanies a programming language and the convenience of dynamic deployment.

In the next section of this paper, we present the ANTS protocol architecture. We then demonstrate how the architecture can be exploited by presenting simple protocols that support multicast and mobility, two directions in which IP is currently being extended. This is followed by a discussion of our prototype implementation of the the ANTS architecture. We then contrast our system with related work, and offer conclusions and suggestions for further work.

*djw@lcs.mit.edu. http://www.sds.lcs.mit.edu/. This work was supported by DARPA, monitored by the Office of Naval Research under contract No. N66001-96-C-8522, and by seed funding from Sun Microsystems Inc.

[1] See http://www.sds.lcs.mit.edu/activeware.

2 ANTS Protocol Architecture

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

Unlike IP, the network service provided by ANTS is not fixed – it is flexible. Different applications are able to introduce new protocols into the network by specifying the routines to be executed at network nodes that forward their messages. Applications may customize network processing to suit their needs by pushing processing into the network – either processing that is traditionally performed at end-systems or novel kinds of processing that only make sense in the context of active networks.

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

- The nodes of the network must simultaneously support a variety of network protocols providing different services.
- The architecture must support the construction of new protocols by mutual agreement among interested parties, rather than requiring new protocols to be registered in a centralized manner. We do not expect all users to construct new protocols directly, but rather to choose between protocols offered by third party software vendors.
- The architecture must support the dynamic deployment of new protocols, since it is unreasonable to take portions of the network "off-line" in order to configure nodes to support new protocols – especially as the scale of the network increases.

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

- The packets found in traditional networks are replaced by *capsules* that refer to the processing to be performed on their behalf.
- Routers and end nodes are replaced by *active nodes* that execute capsule processing routines and maintain their associated state.
- A *code distribution* mechanism ensures that processing routines are automatically and dynamically transfered to those nodes where they are needed.

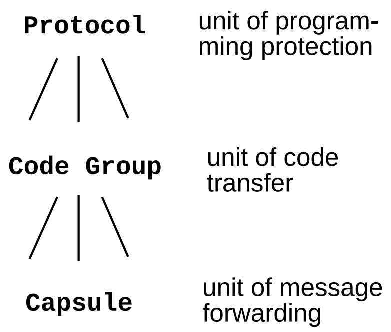

Figure 1: Capsule Composition Hierarchy

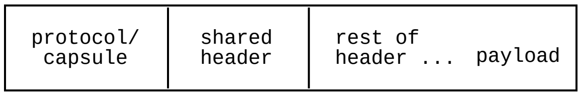

Figure 2: Capsule Format

2.1 Protocols and Capsules

To make use of programmable network elements, we require a model for combining forwarding routines at individual nodes into a pattern of behavior – a protocol – that defines the processing to occur across the network as a whole. Further, the model must separate patterns of behavior from each other.

In ANTS, we do this using capsules, code groups, and protocols. The relationships between these entities is illustrated in Figure 1.

- A *capsule* is a generalized replacement for a packet. Its most important architectural function is to include a reference to the forwarding routine to be used to process the capsule at each active node. Some forwarding routines are "well-known" in that they are guaranteed to be available at every active node. These primarily include routines for common case processing, i.e., unreliable data transfer with standard routing, and for bootstrapping network services, such as the code distribution scheme to be described shortly. Other routines are "application-specific." Typically, they will not reside at every node, but must be transfered to a node by the code distribution scheme before capsules of that type can be processed for the first time.
- A *code group* is a collection of related capsule types whose forwarding routines are transferred as a unit by the code distribution system.
- A *protocol* is a collection of related code groups that are treated as a single unit of protection by the ac-

tive nodes. Thus protocols are the units by which the network as a whole is customized by applications. Capsules belonging to the same protocol will typically manipulate shared information within the network.

Capsule Format

The format of capsules as they are carried across link-layer channels is sketched in Figure 2. Each capsule carries an identifier for its protocol and particular capsule type within that protocol. The identifier is based on a fingerprint (e.g., the MD5 message digest) of the protocol code. It is used for demultiplexing to a forwarding routine in the same sense as the Ethernet type and IP version and protocol fields.

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

- It greatly reduces the danger of protocol spoofing. When a node receives code that purports to correspond to a particular capsule type, it can easily verify for itself (without trusting external parties) that the code is indeed what it purports to be.
- It allows protocols and capsule types to be allocated quickly and in a decentralized fashion, since their identifier depends only on a fingerprint of the protocol code. One need only choose a hash function with a sufficiently large range to make the probability of a collision extremely low.

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

2.2 Active Nodes

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

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

Node Primitives

We chose an initial set of primitives based on our experience with a predecessor system [20]. This work suggests that a relatively small set of primitives is sufficient to express a number of different and useful forwarding routines. We support the categories of node primitives listed below. There are also some obvious additions (namely authentication, fingerprinting, compression, etc.) that we have not had the time to experiment with yet.

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

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

Execution Model

Our execution model is based on the assumption that the primary purpose of the computation done within an active network is facilitating communication. Consequently, our model is optimized to support a generalized form of packet forwarding rather than more general computation. More specifically, it has the following characteristics:

- The forwarding routine of a capsule is set at the sender and may not change as it traverses the network; nor may capsules belonging to one protocol create capsules belonging to a different protocol within the network. Given this, one user may not control the processing of another user's capsules in unintended ways.
- Not all nodes of the network need execute a particular forwarding routine. Some nodes may elect not to, depending on their available resources and security policies, in which case they perform "default" IP-like forwarding on these capsules instead. Additionally, forwarding routines may self-select nodes at which it is useful to perform their specialized processing depending on the location of the node and its capabilities.
- Since forwarding routines may be defined by untrusted users, they are limited in their capabilities. In particular, like traditional forwarding routines, they are expected to run to completion locally and within a short time, and their memory and bandwidth consumption is bounded by a TTL-like scheme.
- The data that a capsule may access while in the network is determined by the protocol to which it belongs. By default, only capsules belonging to the same protocol may share state. Further, once created, a protocol is closed in that new types of capsule that purport to belong to it in order to manipulate its data in a different manner are disallowed.

When a capsule arrives at a node, its associated processing routine is run to completion (unless it exceeds its resource limit). The routine processes the payload of the capsule and initiates any further actions, e.g., forwarding, that are necessary. Unlike more general mobile agent systems, the node provides no support for migrating computations at arbitrary points during execution. Instead, processing routines may update capsule fields and enter application-defined information into the shared node soft-store. Together, these mechanisms allow the construction of computations that evolve their behavior as they traverse the network.

During capsule processing, active nodes are responsible for the integrity of the network and handle any errors that arise. Since capsule processing resembles a distributed programming system in which there are many legitimate users with small tasks, authentication and other traditional security schemes are likely to be too heavyweight to be used for common-case forwarding

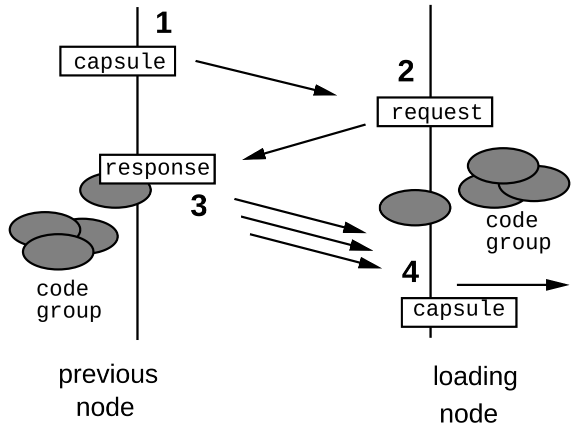

Figure 3: Demand Loading of Code Groups

programs. Instead, we rely on the safety mechanisms of mobile code technologies (e.g., sandboxing and Java bytecode verification) to execute untrusted routines efficiently in a contained manner. Conversely, the occasional use of primitives that manipulate shared logical resources, e.g., updates to the default routing tables, must be authenticated.

This model is not yet sufficient, however, to ensure that the network is robust, nor that its resources are allocated in an intended manner. For these purposes, we incorporate additional mechanisms to limit the physical resources consumed by capsule programs, both at individual nodes and across many nodes.

We associate with each capsule a resource limit that functions as a generalized TTL (Time-To-Live) field. This limit is carried with the capsule and decremented by nodes as resources are consumed; only nodes may alter this field, and nodes discard capsules when their limit reaches zero. In order to reason about total resource bounds, care must be taken to transfer resources when one capsule creates another inside the network: the resources allocated to each created capsule must be strictly less than those of the creating capsule.

It is straightforward to charge for resources as they are consumed. Processing time and link bandwidth are allocated by time and capsule quanta, respectively; node memory is allocated by cached objects, since caching converts memory into a renewable resource. We hope, however, that it will prove feasible to enforce static limits at nodes with a scheme similar to [5] or by using proof-carrying code techniques [12].

2.3 Code Distribution

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

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

Instead, our approach has been to couple the transfer of code with the transfer of data as an in-band function. We believe this has several advantages. It limits the distribution of code to where it is needed, while adapting to node and connectivity failures. It improves startup performance and facilitates short-lived protocols by overlapping code distribution with its execution. It further suits our research goals by allowing customized processing to be expressed at a fine granularity, i.e., per capsule rather than per application session.

We have designed a scheme that loads code on demand and caches it to improve performance in the expected common case of flows, i.e., sequences of capsules that follow the same path and require the same processing. At end-systems, applications may begin to use a new protocol at any time by registering the code definition at their local node. Capsules of the new type may then be injected into the network and received from it. As capsules travel through nodes of the network, a lightweight protocol is used to transfer the capsule programs incrementally from one node to the next, where they are cached for future use. For this purpose, capsules must be organized into code groups according to their dependencies. If one type of capsule refers to another type, their definitions are grouped for joint transfer.

A sequence of events that illustrates the operation of this demand loading protocol is listed below and shown in Figure 3.

1. Capsules identify their type and the protocol to which they belong as they travel. This information is immutable for a given instance of a capsule.
2. When a capsule arrives at a node, a cache of protocol code is checked. If the required code is not all present, a load request for the missing portion based on the capsule type and protocol is sent to the "previous" node, i.e., the node from which the capsule arrived. The capsule execution is suspended, awaiting the code, for a finite time.
3. When a node receives a load request that it can answer, it does so immediately. It sends load responses that contain the portion of protocol code that is implicated.
4. When a node receives a load response, it incorporates the code into its cache. If the required code is now all present, it wakes sleeping capsules. If the required responses are not forthcoming, sleeping capsules are discarded without further action.

This scheme has some important properties. First, the reliance of a node on the "previous" node is designed to draw code from a source node along the network paths where it is needed. As many capsules are transferred, a region is grown, within which the same processing is invoked repeatedly and code transfer is no longer necessary. If network paths change, then code transfer will resume in order to adapt to the new connectivity.

Second, the connectionless nature of the scheme is designed to provide rapid loading without concern for reliability. There are two reasons a load request might fail:

1. The requested code might not be available at the node to which the load request is directed, or
2. Network congestion might cause the loss of capsules.

Requests for code are caused by the arrival of a capsule. Since these requests are always directed at the node that sent the capsule to the requesting node, it is highly probable that the code is in the code cache of the node receiving the request. The main source of failure is, therefore, likely to be network congestion. This is the reason that we have not elected to use a higher-level connection, e.g., TCP, for code delivery. In the unlikely event that a load request fails, the capsule that provoked the request is lost. In this case, protocol-specific higher-level processing at the end-systems is used to provide whatever level of reliability is appropriate.

3 Programming with ANTS

To demonstrate how we intend our architecture to be used, we describe two simple protocols that introduce multicast and mobility services into an ANTS network that initially lacks them. We chose these examples because they represent two areas of widespread interest in which the Internet community is currently dealing with the difficulties of innovating protocols.

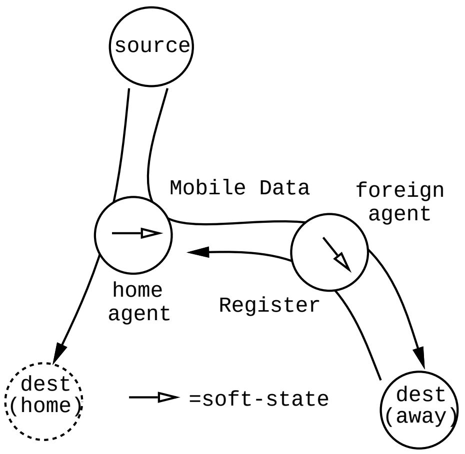

Figure 4: Mobile Capsule Paths

```
// on entry:
//   home = home agent
//   next = node at which to register
//   forward = address to be registered

// go to foreign and then home agent
if (n.address() != next) {
  n.routefornode(this, next);
  return;
}

// insert a forwarding address
n.put(src, new W_N(forward), IDLE);

// after doing foreign, do home
if (n.address() != home) {
  forward = next; next = home;
  n.routefornode(this, next);
}
```

Figure 5: Mobile `Register` Capsule

The presented protocols were written and tested using the Java-based prototype implementation of ANTS described in Section 4. In developing the protocols, it was not our intent to present new and better solutions to these particular problems. Our goal was merely to demonstrate how our approach may be used to write protocols that address these kinds of problems in a number of different ways, depending on application requirements rather than relying on a "one size fits all" solution. That is, we wish to show that ANTS facilitates protocol construction and deployment, not that the particular problems of mobility and multicast are straightforward.

3.1 Mobile Hosts

We introduce support for mobile hosts into an ANTS network with a `Mobile` protocol composed of two cooperating capsule types. One type of capsule is sent by the

```
// look up forwarding record
W_N f = (W_N)n.get(dst);

// if found, update our route
if (f != null) next = f.node;

// and continue on our way
if (n.address() != next)
  n.routefornode(this, next);
if (n.address() == dst)
  n.delivertoapp(this, dpt);
```

Figure 6: `Mobile Data` Capsule

mobile host to register forwarding information while it is roaming. The second is used by other hosts to send messages to the mobile host. To be consistent with mobility schemes, we use the notion of home and foreign agents. The home agent is used to intercept messages at the base location of the mobile host. The foreign agent is used as a "care of" address to reach the mobile host while it is away from its base. The paths of these two types of capsule is shown in Figure 4 and their code in Figures 5 and 6.

Mobile hosts that are roaming periodically send `Register` capsules to their home agent via a local foreign agent. The program carried by this capsule updates forwarding addresses cached at the home and foreign agents. In each, an updated forwarding pointer is entered into the node cache; the home agent forwards to the foreign agent, and the foreign agent to the current mobile location. At the home agent, `Register` capsules are silently discarded, having established their forwarding pointers within the network. As the mobile moves, old forwarding pointers will either be supplanted by fresh information or evicted from the cache after a brief interval.

To communicate with the mobile host, other hosts send `Mobile Data` capsules that make use of this forwarding information. This capsule program is directed by default routing towards the base location of the mobile. If the mobile is at home, the capsule will reach it and be delivered. If the mobile is roaming, the capsule will discover a forwarding pointer as it traverses the home agent, and follow it to the foreign agent. There, it will find a further pointer to the current mobile location and so be delivered.

Despite the simplicity of this scheme, it provides the essential feature of mobility: hosts may be reached as they move without introducing another layer of addressing. There are also some interesting comparisons with Mobile IP [14]. First, it is not necessary to confine mobile forwarding information to the edges of the network. To facilitate shortcut routing, mobile updates may enter for-

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

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

// or make a new one if necessary
if (m == null) {
  m = new W_JAN();
  n.put(group, sender, m, IDLE);
}

// are we at an intermediate node?
add: if (reverse != 0) {
  if (m.nodes == null) {

    // start a new list
    m.nodes = new W_N[1];
    m.nodes[0] = reverse;
  } else {

    // does it contain our info?
    for (int i = 0; i < m.nodes.length; i++)
      if (m.nodes[i] == reverse) break add;

    // if not, add it
    int len =  m.nodes.length;
    W_N[] nn = new W_N[len+1];
    System.arraycopy(m.nodes,0,nn,0,len);
    nn[len] = reverse; m.nodes = nn;
  }
}

// need to refresh upstream entry?
long time = n.time();
if (time - m.time < RATE) return;
m.time = time;

// if so, update route and continue
if (n.address() != sender) {
  reverse = n.address();
  n.routefornode(this, sender);
}
```

Figure 7: Multicast `Subscribe` Capsule

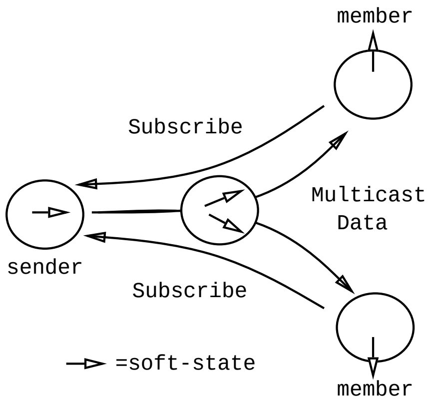

Figure 8: Multicast Capsule Paths

```
// look up forwarding record
W_JAN m = (W_JAN)n.get(group, sender);

// must find it to continue
if (m != null) {
  if (m.nodes != null) {

    // send a copy every way
    for (int i = 0; i < m.nodes.length; i++)
      n.routefornode(this, m.nodes[i]);
  } else

    // or deliver to application
    n.delivertoapp(this, dpt);
}
```

Figure 9: `Multicast Data` Capsule

3.2 Multicast

We introduce a basic `Multicast` protocol, resembling IP multicast [4], composed of two cooperating capsule types[2]. One type of capsule is sent to subscribe to a group, and the other carries the multicast message itself.

Applications that wish to receive messages sent to a given group by a particular sender periodically send `Subscribe` capsules towards the sender. The program carried by this capsule installs (or refreshes) forwarding pointers that are cached in each router it traverses; forwarding information sent by different nodes is merged to form a distribution tree. To multicast to the group, the sender node sends a `Multicast Data` capsule that simply routes itself along the distribution tree. The paths of these capsules are shown in Figure 8 and the code in Figures 7 and 9.

The `Subscribe` program begins by looking up the for-

warding pointers at any node, and messages from other nodes will follow them like a trail of crumbs once their paths cross. Second, because our approach is based on innovation rather than backwards-compatibility, a different protocol is used to reach stationary and potentially mobile hosts. This poses no bootstrapping problem however: it is straightforward to select which protocol to use in the first place with a directory service, e.g., in the same manner that the selection of IPv4 versus IPv6 is being incorporated into the DNS [7] for the IPv6 transition.

[2] Readers familiar with IP multicast will note that our multicast provides a somewhat different service. This is discussed after the scheme is presented.

warding record for the group in the node cache, creating a fresh record if none is found. To separate this forwarding record from other multicast session information in the cache, the record is stored under a key that is the combination of the group and sender addresses. Once the forwarding record is located, a "reverse" pointer in the direction of the subscriber is merged into the forwarding record. Leaf subscriber nodes are indicated by empty forwarding lists.

The forwarding method of the `Multicast Data` capsule makes use of the forwarding records it finds at nodes, sending a copy of itself along every indicated "reverse" path found in the record at each node. If no forwarding information can be found, the capsule is discarded. At end-systems, where there is an empty forwarding record, the capsule delivers itself to an application.

Together, these capsule programs implement an unreliable multicast protocol with the central property of network-based multicast: efficient use of bandwidth. The service differs from IP multicast in two significant respects. First, the scheme is localized to the nodes using the protocol, and does not require that multicast-capable routers 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 protocol developers. If multiple senders are needed, then multiple distribution trees may be formed by having members subscribe to each of the senders. Alternatively, the sender may be considered the root of a core-based tree [1], with messages routed up the tree towards the root and down other branches.

4 Prototype Implementation

We have been experimenting with a prototype implementation of the ANTS architecture[3]. The implementation was designed primarily to allow us to evaluate the suitability of our approach to creating and deploying protocols. We have used it to test and debug the `Mobile` and `Multicast` protocols discussed in this paper, as well as a number of other protocols developed in our group, e.g.,a high performance reliable multicast developed by Lehman [10], a TCP SYN-flooding defense protocol developed by Van [19], and an auction service developed by Legedza. Recently, ANTS nodes have also been deployed at different sites as one of several technologies within the DARPA-sponsored ABONE, an experimental active network in which nodes communicate by tunneling through the Internet using UDP.

[3]Our toolkit is publicly available as a source distribution at http://www.sds.lcs.mit.edu/activeware.

Class	Key Methods
`Node`	`address`, `get`, `put`, `routefornode`, `delivertoapp`
`Channel`	`send`, `receive`, `node`
`Application`	`send`, `receive` (upcall), `node`
`Capsule`	`evaluate`, `length`, `encode`, `decode`

Table 1: Key Classes and Methods

The current implementation is written in Java and runs as a user-level process under Linux. The code distribution protocol transfers processing routines in Java classfile format. We chose Java because of its support for safety and mobility (through bytecodes and their verification) and the likely emergence of higher performance compilers and runtimes. Its flexibility as a high-level language and support of dynamic linking/loading, multithreading, and standard libraries has allowed us to evolve our design while maintaining a small code base (≈10000 lines).

The major components of the ANTS architecture are implemented using the classes listed in Table 1. When each ANTS runtime is started, its root thread instantiates a single `Node` object, one `Channel` object for each local network interface, and one `Application` object for each local distributed application. Applications may then communicate by exchanging capsules, sending them via the local node, which transmits them as packets using the link layer services of the local channels. Conversely, when packets are received from the link layer, the channel attempts to convert them to instances of the appropriate `Capsule` subclass. If the required code is not present at the node, then the packet is retained by the node while the code is fetched using the code distribution protocol. Once a capsule instance is created, the thread calls its `evaluate` method, passing the node instance as a parameter. As it is evaluated, the capsule code has access to the private soft state of the associated protocol as well as the public state (e.g., routing tables) of the node.

`Node` Class

The `Node` class represents the runtime of a single network node, including its code and soft-storage caches and code distribution protocol. It provides a set of node primitives that can be invoked by capsule programs. These primitives allow access to the state at the node and enforce various security constraints.

Table 1 lists some key methods, including `routefornode`, which forwards a copy of a capsule towards a given destination, and `get` and `put`, which are used to manipulate that part of the node soft-store that can be directly accessed by the executing protocol. The soft-store is managed in a least-recently-used order, and additionally removes entries with coarse-grain timeouts. This prevents the network from retaining stale state, e.g., old session identifiers, and further allows us to stress test protocols by shortening the value of the timeout. The code cache is also managed in a least-recently-used order, and does not require timeouts because the protocol naming scheme obviates versioning problems.

`Channel` Class

The `Channel` class provides the interface to the link layer, connecting nodes via point-to-point or shared medium channels. At present, either Ethernet or UDP "tunnels" may be used to transfer capsules. These choices allow small networks to be constructed by running one node per machine and connecting the nodes with Ethernet channels. Larger networks are emulated by running many nodes per machine and connecting the nodes with UDP channels.

`Capsule` Class

The `Capsule` class is a virtual class that can be specialized to create the capsule types that comprise protocols. During capsule processing at nodes, each packet received from the link layer is manipulated as an instance of its corresponding `Capsule` subclass. In our prototype implementation, if an error occurs, execution of the capsule is terminated and the state associated with that execution of the capsule is released. It would be straightforward to extend this recovery process with an error message scheme analogous to ICMP.

In addition to providing the base class for new protocols, our current implementation provides several built-in subclasses. The class `DataCapsule` allows applications to transfer data using default (i.e., shortest path) routes. The system classes `DLRequestCapsule` and `DLResponseCapsule` are used by the code distribution protocol. They provide the bootstrapping capability needed to install other protocols.

`Application` Class

Programs that use the ANTS service are constructed by specializing the `Application` class. This is a container for end-system processing that provides a small API for registering protocols, injecting capsules into the network and receiving capsules from the network. It runs within the same address space as the node to which it is directly attached. At nodes internal to the network, it can be used to implement SNMP-like node management applications. At end-systems, it provides a bridge to the end user.

Runtime	Latency (us)	Throughput (capsules/sec)
JDK1.0.2	1050	1080
JDK1.1.4	800	1510
JDK1.1.4 + JIT	680	1680
C user-level relay	225	6350

Table 2: Node Baseline Performance

Capsule	Size (bytes)	Latency (us)	With Load (us)
Data	1429	680	4700
Mobile	2154	720	5900
Update	2029	700	6800
Multicast	2255	740	5800
Subscribe	2850	780	6100

Table 3: Capsule Program Measurements

Measurement and Evaluation

Though our prototype implementation was not built for performance, we did run a small number of performance tests. The goal of these was to gain some insight into the performance impact of various architectural decisions. All tests reported in this section were performed on a Sun Ultrasparc 1 (167MHz) running Solaris 2.5 and connected with 100 Mbps Ethernet.

Table 2 shows the baseline performance of our node runtime. We measured the throughput of a single node in capsules per second by using an external traffic generation system written in C. We measured the latency in microseconds across a single node by using a passive `tcpdump`-based monitor that recorded cycle-counts on packet arrival within a modified Linux kernel. In both cases, we used minimal length capsules running a minimal forwarding program – this is our equivalent of a "null RPC" that is intended to show the costs of our architecture and its implementation. The progression down the table shows how the performance varies with Java VM runtime, given identical code, hardware, and operating system. At the bottom of the table, measurements for a C packet relay running at user level place the other measurements in context by reporting on the raw hardware and operating system performance.

We find that the base performance of our node is reasonable for a high-level prototype, especially given the current early state of Java development tools and runtimes. At over 1000 capsules/second, the system is usable for experimenting with distributed applications. There has been significant improvement with successive generations of Java runtimes, though our system falls well short of the raw machine performance. We anticipate further improvements from: Java development environments that combine statically compiled native code with dynamically loaded bytecodes, next generation "just-in-time" compilers that perform adaptive inlining, and of course the profile-driven tuning that we have not yet undertaken.

Two additional experiments would assist in calibrating our node performance. First, throughput measurements in bytes/second would expose the costs of our data paths. We believe that this comparison would be favorable since, like IP, our implementation does not require payloads to be copied. Second, measurements of IP processing would determine how much of the raw hardware performance can be obtained for a well understood forwarding model.

To describe the costs of executing user-defined processing routines, Table 3 lists measurements for the capsules defined in our example protocols. We provide three measurements: program size, latency when the code is already loaded, and latency when the code must be loaded for the first time. Latency is measured as described previously and using the best runtime of Table 2. Repeated code loading is simulated by using an artificially small code cache and causing swapping to occur; however, the cost of bytecode verification is not included since we are unable to cause classes to be removed from our Java runtime.

The measurements suggest that the overhead of user-defined processing routines can be low. Despite the fact that the size of capsule programs is considerably larger than is necessary because our implementation uses the Java classfile format directly, the example routines are short, and can be transferred without consuming much bandwidth. Additionally, the latency over simple forwarding is small and the latency attributable to demand loading seems quite reasonable in that it is comparable to that associated with establishing a connection on conventional networks.

5 Related Work

We believe our approach is novel in its application of mobile code, demand loading, and caching techniques within the network layer.

The most similar recent work we are aware of is the messenger paradigm [6] and work on flexible protocol stacks that preceded it [18]. Like our system, this work allows new protocols to be deployed. The intent, however, is to investigate the structuring of communicating systems, including distributed operating systems and intelligent agents. As such, it lacks the network layer specializations, e.g., demand loading, that we have developed.

Some modern protocol architectures have been configurable, as opposed to programmable. The x-kernel [9] provides a collection of micro-protocols from which protocols (e.g., RPC) can be synthesized. Configurable systems can further increase their flexibility by deferring the selection of components until runtime, and so the x-kernel supports the dynamic composition of micro-protocols on a per packet basis. Although configurable systems are capable of expressing a range of protocols, their means of composition, e.g., layering, is less flexible than that of a programming language.

The earliest programmable network based on mobile code that we are aware of is Softnet [22], an experimental packet radio network constructed in the early 1980s. Its goal was similar to our own: to allow users to define their own high level services. As with our approach, packets were considered to be programs of a language, FORTH, and interpreted at nodes on arrival. Softnet is an intriguing example of a real programmable network that inspired a user community and workshops, but unfortunately fell into disuse with little documented about its successes and failures. We speculate that this was because of difficulties with safety and efficiency, problems that may now be more tractable, given the recent advances in mobile code and operating system technology.

End-to-end code shipping to improve performance has been studied in the context of RPC [15, 13]. Our approach offers a greater scope for customization by including intermediate nodes as well as end-systems.

Our work is complementary to several other active network efforts. The use of general-purpose Java bytecodes and VM has allowed us to evolve our architecture quickly, but at the cost of less control over resource usage and lower absolute performance. Research at the University of Arizona on Liquid Software [8] and Scout [11] enable a finer granularity of local resource management as well as competitive performance through the construction of a specialized node operating system. Research at the University of Pennsylvania on PLAN and BBN on Sprocket enables stronger resource management and security guarantees across the nodes of a network through the use of language design techniques. Finally, research on active signaling at USC ISI and NetScript at Columbia University [21] explore alternative models of active networks in which new services are introduced for control rather than data transfer purposes, or by network management agents rather than all users.

6 Conclusions

In this paper, we have presented an architecture that supports the construction and dynamic deployment of network protocols. In contrast to a standardization-based process, our approach:

- Allows new protocols to be automatically, dynamically, and rapidly deployed to exactly those nodes in the network they are needed; and,
- Requires no advanced consensus about the kinds or definitions of the protocols.

We achieved these results by treating the network as a distributed programming system and through the application of mobile code, demand loading and caching techniques.

Our Java-based prototype allowed us to experiment with the ANTS network programming model and test its code distribution system. In addition to the simplistic examples presented here, we have used ANTS to implement several more complex protocols, including a high performance reliable multicast.

Our early experience with ANTS strongly suggests that this use of mobile code technologies has considerable promise. It can provide the means for automatically upgrading network protocols. This, in turn, can remove barriers to innovation, stimulate experimentation, and hasten the arrival of new functionality. It is interesting to speculate how many other protocols would be deployed were it not for the barriers to innovation that we are addressing.

Acknowledgments

We thank the members of our research group for much useful feedback as the design of our system progressed. In particular, we wish to acknowledge David Murphy for his work on a separate node implementation and his contributions to an earlier version of this paper; Jonathan Santos for his assistance with the design and development of the current ANTS implementation; and Ulana Legedza and Li Lehman for proofreading.

References

[1] A. Ballardie et al. Core based Trees. In *SIGCOMM'93*, 1993.

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

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Request For Comments 1883, Dec 1995.

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

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

[6] G. Di Marzo et al. The Messenger Paradigm and its Impact on Distributed Systems. In *Workshop on Intelligent Computer Communication*, 1995.

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

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

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

[10] L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *INFOCOM '98*, 1998.

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

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

[13] C. Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment.* PhD thesis, Harvard University, 1992.

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

[15] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, Oct 1990.

[16] D. Tennenhouse et al. A Survey of Active Network Research. *IEEE Communications Magazine*, pages 80–86, Jan 1997.

[17] D. L. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. In *Multimedia Computing and Networking 96*, 1996.

[18] C. Tschudin. Flexible Protocol Stacks. In *SIGCOMM'91*, 1991.

[19] V. C. Van. A Defense Against Address Spoofing Using Active Networks. M.Eng Thesis, Massachusetts Institute of Technology, June 1997.

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

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

[22] J. Zander and R. Forchheimer. Softnet - An Approach to High-Level Packet Communication. In *ARRL 2nd Computer Networking Conference*, 1983.