Design Review, August 1997

Developing Network Protocols with the ANTS Toolkit

David Wetherall*

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

1 Introduction

ANTS, for Active Node Transfer System, is an active networking toolkit for building and dynamically deploying network protocols. It is intended to help answer the question "What kind of new network services can be readily deployed?" It provides a programming model that allows some kinds of protocol processing to be expressed, a code distribution system for loading new protocols into the network, and node runtime for executing them.

This paper outlines the programming model supported by ANTS. It includes information on the way the model and components behind it were designed, and the manner in which they are intended to be used. It is best read in conjunction with a concrete example of the API in use, which is provided as an appendix for convenience. All descriptions correspond to a revised version of the initial design that is intended for an August distribution and not yet fully implemented; descriptions of features that require further implementation work are flagged with the keyword "planned".

Since there are many possible conceptions of an "active network", we begin the paper at a high level by describing where ANTS fits in the design space and the specific goals that it sets out to achieve. The next section then provides an overview of the programming model, and the sections following it specific details of each of the key classes. This paper is concluded with a list of open design issues that impact the API.

1.1 Design Goals

ANTS is intended to support novel network services for routing, caching, transcoding, combining, filtering, regulating, and otherwise processing packets within the network itself. This includes the notion of "application-specific" protocols where portions of the application processing are "pushed" into strategic nodes of the network. No artificial restrictions have been placed on the kinds of network protocols that can be expressed because part of the goal of the system is to gain a better understanding of the kinds of applications which stand to benefit most from a flexible network.

Three simple goals for protocol innovation guided the design of ANTS. Each represents a more flexible form of innovation than is currently achieved in the Internet.

- The nodes of the network should support a variety of different network protocols being used simultaneously.
- New protocols should be supported by mutual agreement among interested parties, rather than requiring centralized registration or agreement between all parties.
- New protocols should be deployed dynamically and without the need to take portions of the network "off-line".

*djw@lcs.mit.edu. http://www.sds.lcs.mit.edu/.

1.2 Other Kinds of Active Network

There are many different approaches for designing a programmable network that meets these goals. We first digress to describe three alternative kinds of active network with the intent of better defining ANTS by emphasizing what is it not.

Unix in Every Router

Lampson has described an active network as one in which there is "unix in every router". This high-level view implies the flexibility of a general-purpose distributed programming system. Presumably large globs of new functionality can be downloaded into routers, perhaps as distributed components, and be run as long-lived processes communicating by the exchange of messages. At this level of abstraction, however, the model does not constrain new network services sufficiently to make any useful guarantees about the protection, allocation, or performance of shared resources. Without further restrictions, it is most appropriate for environments within which there is already a basis for trust, e.g., single-user end systems or single-administration networks.

ANTS does constrain the development of network services in the hope of making such guarantees for a shared network with untrusted users.

The Intelligent Firewall

An different approach is to allow the behavior of specialized network devices to be altered by users. The classic example of this genre is perhaps the "intelligent firewall". Firewalls already perform filtering based on application data while they process packets. Rather than being fixed, this filtering could be controlled by users within the protected domain to allow fine-grained control over the services that are exported. The Active Bridge [1] is an example of this approach.

From the point of view of introducing new network services, this approach leaves two questions unanswered. First, it is not clear how to extend the model of programming a particular device at a particular location to a model for programming network behavior as a whole, especially in the face of unknown routes and devices. Second, it is not clear who should be able to reprogram what functionality. In cases where the programmable device affects many users, access is essentially a management function and can be permitted by authorized users only.

ANTS does provide a model to allow all users to customize the network as a whole.

A Signaling Facility

A third view is one in which network programming is available for signaling rather than data transfer purposes. In addition to traditional connection establishment, this can be used as a network probe facility for users to query the status of the network, as may occasionally be useful for resource discovery, diagnostics, network monitoring, etc.

The significance of a "signaling" approach is that programming is infrequent compared to data transfer events. This removes the onus from data transfer performance and allows simplifications such as carrying program code along with data. ACTIVE IP [6], PLAN [3], Braden's Active Signaling project [2] and BBN's SmartPackets/Sprocket [4] are examples of this approach.

ANTS is intended to be used to introduce new data transfer services.

1.3 An ANTS-style Network

The ANTS model can be described as generalized packet forwarding. It has the following characteristics:

- Packets include a reference to the routine to be used to forward them at network nodes. This allows network users to control the forwarding of packets that they send, but not packets that they receive.
- The forwarding routines may be defined by untrusted users and so are limited in their capabilities. In particular, like traditional forwarding routines, they are expected to run to completion locally and within a short time.
- Individual forwarding routines will do different things at different nodes depending on the local environment. In particular, only a subset of nodes may be appropriate for particular routines, with other nodes performing default forwarding instead.

This approach is a step up from the fixed packet forwarding in use today. It is intended to be flexible enough to support novel protocols, while being restrictive enough to guarantee some level of protection, allocation and performance of shared resources in an untrusted environment. Protection is based on the ability to specify processing for your packets only. Allocation is based on the limited resources that will be granted to each packet by nodes. Performance is based on the simplicity of the processing model and the ability to tailor processing to the diversity of heterogeneous networks, including those in which only "strategic" nodes may be active.

The ANTS model is also connectionless, in that every packet is an independent entity and may refer to its specialized forwarding routine at any time and any place. This is intended to support our research goals in two ways. First, it allows us to explore the use of many new protocols at fine granularities, including application-specific protocols and network (rather then end-system) protocols. Second, it forces us to examine the difficulties of network programming in the face of unreliability (due to changes in connectivity, packet loss, duplication and reordering) rather than assume that these issues will be handled "elsewhere".

2 Programming Model Overview

The ANTS architecture is described in [5], and a Java-based prototype of it that defines the programming API has been implemented. All implementations must transfer protocol definitions across the network using mobile code technology. In the prototype, however, applications and the active node itself are also written in Java. Further, they run within a single address space on top of the user-level network services of the local operating system. These choices were made for the convenience of rapidly prototyping and evolving the design.

In the model, an ANTS network is customized by its users in terms of protocols. New protocols (and applications that use them) may be developed by users at any time by subclassing virtual classes provided specifically for the purpose. New protocols (and applications) are then used by manipulating instances of these new subclasses in conjunction with an executing ANTS runtime; the runtime provides the network services of a single node in the network. In more detail:

- A new protocol is developed by subclassing the virtual class `Protocol`. This requires identifying all of the different types of packet that will enter the network by their different forwarding routines. Each type of packet and its forwarding routine is specified by subclassing the virtual class `Capsule`.
- A new application is developed by subclassing the virtual class `Application`.
- An instance of the class `Node` represents the local ANTS runtime.
- A new protocol and application are used by creating instances of their classes and attaching them to node. The application is connected to the node in order to send and receive capsules from the network. The protocol is registered with the node in order for the network to be able to obtain its code when it is needed.

This model allows customization by the network users assuming that a network of active nodes already exists and is up and running. The active nodes, however, will often be part of the system under study, particularly for experimentation with different topologies, application workloads, and node services. For these purposes, ANTS provides two facilities:

- Configuration tools allow a network topology complete with applications to be managed. This includes the calculation of routes and the initialization of local node configurations.
- A node extension architecture allows different nodes to support different service components, e.g., multicast, caching, RSVP, transcoding, etc., as appropriate. Extensions are developed by subclassing the virtual class `Extension`.

This concludes the overview of the ANTS programming model. The following sections present more detail in a bottom-up fashion, looking at each of the key classes just identified in turn. First we detail the forwarding routines themselves as visible through the `Capsule` class, then what they are able to do by using the `Node` and `Extension` classes. Next we describe `Protocol` and `Application`, and finally the configuration of entire networks.

3 Capsule

A capsule is a combination of a packet and its forwarding routine; the forwarding routine is executed at every active node the capsule visits while in the network.

New types of capsule, with different forwarding routines, are developed by subclassing the virtual class `Capsule`. This class must satisfy conflicting roles: it must be flexible and exposed for easy subclassing, yet it must protect header fields that are key to the integrity of the system.

All capsules inherit the following methods to access header fields that are part of the `Capsule` base class.

- `getSrc()` returns the source address of the capsule as a 32 bit integer. The class `NodeAddress` contains routines for formating such integers as IPv4 style-addresses. The source address is set implicitly by the node that injects the capsule into the network, and never changed.
- `getDst()` returns the destination address of the capsule as a 32 bit integer. At non-active nodes, this address is planned to be used for default forwarding; the special value 0.0.0.0 represents a "statically indeterminate" address that cannot be forwarded without evaluating the capsule.
- `setDst()` mutates the destination address.
- `getPrevious()` is planned to return the address of the previous active node encountered by this capsule. It is used by the demand loading system.
- `getType()` is planned to return the type of the capsule.
- `getResources()` is planned to return the available resource limit.

New subclasses of `Capsule` will typically include additional header fields as appropriate for the application. They must also implement the following key methods so that the node runtime is able to manipulate them.

- `evaluate()` is the forwarding routine itself. It is passed the local node as a parameter. It returns an indication (which is currently unused) of whether forwarding was successful.
- `serialize()` initializes a linear representation of the capsule object for network transmission.
- `deserialize()` initializes a capsule object from a buffer received from the network.
- `length()` calculates the length of the serialized capsule.

The `evaluate` method is the heart of the capsule. It is written using Java constructs and node and capsule methods. It should not access other Java classes, and it should terminate quickly. The first restriction is partly enforced by the capsule code loading system and the node security manager; full enforcement may require additional load-time checks. The second restriction is planned to be enforced at a coarse granularity by a watchdog thread.

The serialization methods are used by the node to transmit and receive capsules. As far as applications are concerned, capsule objects are injected into and received from the network via the local node. Within the network, however, communication between nodes requires that the capsule objects be converted to and from a serial representation, i.e., a linear sequence of bytes. The methods are written in a highly stereotyped fashion using an Xdr (external data representation) library. Ideally, they can one day be written automatically by using the names and types of the additional header fields. Besides relieving drudgery, this would be advantageous because serialization and deserialization are security sensitive functions, i.e., they manipulate the raw capsule bits, including fields which may not be changed. Currently, we rely on them being "well-formed".

Finally, subclasses should provide some boilerplate for standard constructors. A default constructor (having no arguments) is needed by the system to provide a template for deserialization. It should call the superclass constructor, passing it the type identifier of the capsule. The type identifier itself should be statically initialized using the `findID()` method provided for this purpose.

Other convenience constructors may also be provided. They must call the superclass constructor to ensure that the type identifier and other base capsule fields are set correctly. These constructors typically exist in two forms. The first takes some number of arguments and first calls the superclass constructor passing the capsule type identifier. It is for use by applications. The second takes a type identifier as well and passes it to the superclass constructor. It is for use by more specialized subclasses.

4 Node

The `Node` class implements a single active node. It exports its services to:

- capsules, to execute their forwarding routines and implicitly transfer their code
- applications, to register protocols and send and receive from the network
- node extensions, to attach to and augment local node services.

These activities need to be separated, since they have different protection requirements, e.g., capsule activity must be contained, while extensions benefit from more relaxed access to the node. In this section, we describe only the node services available to capsules. Other services provided to support applications, protocols, and extensions are described in their respective sections.

Capsules access node services from their `evaluate()` method. It is passed a reference to the local node. The following services are available:

- `address()` Get the address of the local node.
- `routeForNode()` Send a capsule towards a given node, using the default routing table.
- `sendToNeighbors()` Send a capsule towards all immediate neighbors of the node, using the default routing table.
- `deliverToApp()` Deliver a capsule to a given (local) application
- `time()` Get the current time.
- `softstate.get()` Lookup an object in the softstate cache.
- `softstate.put()` Put an object into the softstate cache.
- `softstate.remove()` Remove an object from the softstate cache.

With these few services, a number of different styles of forwarding routine can be programmed. Special purpose routes may be constructed atop the default routing table by transporting node addresses and storing them in the cache for later use. Communication between packets can occur via the cache, including fusion in which a capsule places itself in the cache and waits to be collected. Features of the Java language assist in combining these services too. For example, Java's synchronization primitives are useful for dealing with the concurrency issues that arise when multiple capsules access the same cache entry. Note that the services provide no way of suspending and later resuming the execution of the forwarding routine. This means the time-dependent actions, such as the timeout interval for fusion, must be driven by the external clock of capsule arrivals.

This model is deficient in two respects that are planned to be addressed and will affect the API.

- Access to the cache is not partitioned between protocols, so direct interference is possible. With partitioning, calls to manipulate the cache of softstate may implicitly access the current protocol state only, with further calls provided to access state from another protocol.
- Resource limits are not yet in place. Since these limits will be enforced across a capsule and all of its descendents, calls to `routeForNode()`, etc., will need to indicate how much of the resource quota should be transferred to the outbound capsules.

5 Extension

Node extensions augment the built-in services of the node with large components that are not readily transferred on a per capsule basis or are too specialized to be needed at all nodes within the network. The first extension is the `GroupsManager`, supporting multicast primitives. Other candidates are the RSVP, transcoding routines, and so on.

Extensions are manipulated in two ways. First, capsule forwarding routines must locate an extension at the local node to use its services. Second, nodes must provide a means for extensions to attach themselves and securely export their services to capsules.

The interface seen by capsules consists of two methods. They separate the location of extensions into a more expensive load-time portion, and a cheaper per capsule portion. Once the extension is located, capsules may invoke its services directly.

- `findExtension()` This is called, typically via a static initializer on the capsule class, to map an extension name to its local index.
- `getExtension()` This is called on the node to get a reference to a particular extension, or null if the extension is not present.

New extensions are developed by subclassing the virtual class `Extension`. They should be installed on the node as part of local configuration. The interface provided by the node to assist extensions is as follows.

- `attachExtension()` This informs the node of the extension and allows capsules to find it.
- `exportClass()` This allows extensions to export classes, e.g., themselves, for use by capsules. Without this call, the node would deny capsule code access to the classes.
- `addMediator()` This adds a class to the list of classes that can successfully mediate a `checkCapsule()` invocation (as described next). It may needed if the implementation of the extension services calls sensitive node services directly.
- `checkCapsule()` This check throws a security exception if the current execution point has been reached by a method on a capsule class without passing through a method on a mediating class, such as `Node`. It is used as part of the implementation of sensitive methods in the node that must be public but should not be invoked directly by capsules, e.g., `attachExtension()` is public because it must be accessible

by any extension, yet it is not for use by capsule forwarding routines. It is similarly available for use within extension implementations.

6 Protocol

Protocols are the unit in which the network is customized. They serve to group related capsule types for use by the network. In doing so, they must divide capsules into code groups, where a code group consists of the capsule types which invoke each others code within the network layer (but not at end applications). This forms the basis for propagating capsule code by growing the network neighborhood within which a code group is used.

New protocols are developed by subclassing the virtual class `Protocol`. Each protocol should provide a default constructor that groups capsules as appropriate. The following methods are inherited for this purpose:

- `startProtocolDefn()` Begin the protocol grouping.
- `startGroupDefn()` Begin a new code group.
- `addCapsule()` Add a type of capsule to the current code group.
- `endGroupDefn()` End the current code group.
- `endProtocolDefn()` End the protocol grouping.

Once a new protocol is instantiated with the appropriate grouping, it must be registered with the node before an application can send or receive any of its capsules. It should be unregistered when the application is done. The following methods on the node are provided for this purpose.

- `register()` Register a protocol with the node.
- `unregister()` Unregister a protocol with the node.

7 Application

Applications are the entities that make use of the network to send and receive capsules as well as run independent activities. New applications are developed by subclassing the virtual class `Application`. It provides access to the node and its services via the following methods.

- `send()` Injects a capsule into the network via the local node.
- `receive()` (Upcall) Extracts a capsule from the network via the local node.
- `thisNode()` Returns a reference to the current node.
- `port()` Returns port on which the application is connected to the local node. Ports are used for application addressing, and are analogous to TCP/UDP ports.

Applications provide a fairly low-level veneer over the node services. They are trusted by the node, receiving capsules using a node thread and sending them with their own. They support only a single port.

The node provides the following methods to connect applications:

- `attachApplication()` Connects the application to a node on a specified port. Should probably be connect, and there should also be disconnect.

8 Configuration

As a practical matter, experimenting with an active network requires that a network topology be set up. ANTS provides some tools and infrastructure conventions to automate this process.

First, entire network configurations, complete with node addresses, applications and all initialization parameters, are specified as files in a simple textual format. The `makeroutes` utility program then processes these files to compute shortest path routes between all nodes; ANTS uses static offline routing tables. The routes are output them in a format that can later be parsed by nodes to initialize their default routing table.

Second, a `ConfigurationManager` class can read the files and start one of its nodes locally. This includes creating the node runtime, applications, and extensions, connecting them to each other, and starting their operation. For this purpose, all of these entities are subclasses of `ManagedObject` and support several conventions.

- `setArgs()` is used to process command line arguments
- `defaults` is used to hold the default command line arguments
- `start()` is called by the system when all entities are connected and ready to commence operation.

The entities also accumulate and filter informational, warning, and error messages. For this purpose, they are named and connected to a managing entity. The current system is somewhat ad hoc.

9 Open Issues

To conclude this paper, we present a list of outstanding design issues that may impact the programming model.

Overloading of Capsule Classes

The implementation of capsule subclasses fulfills two different roles. First and foremost, it provides functionality needed by the node runtime, e.g., the forwarding and serialization routines. Second, it provides convenience functions for use by applications, e.g., constructors. It may be wise to separate this code, since the former must be carried over the network and is subject to security checks, while the latter can be run locally with application privilege. However, it is not clear how to accomplish this, and at a minimum it is likely to require a further class structure.

Node Addresses

It would be convenient to reuse the implementation of some types as other data types, especially primitive types for performance reasons. An extreme case is the treatment of node addresses as 32-bit integers. However, Java has no typedef facility nor preprocessor macros to express this mapping. The Java-style alternative is to create a wrapper class that simply contains the primitive type, e.g., class `NodeAddress` with one 32-bit integer. Unfortunately, this results in multiple and otherwise unnecessary object allocations and initializations per capsule send or receive. To avoid this, the node address type is inlined as an integer, thus reusing types at the cost of exposing the implementation.

Load-time versus Run-time Forwarding Work

The split at a node between per capsule type work and per capsule instance work is poor. There is the potential to increase performance by evaluating some code only once per node (when it is loaded) because the result does not vary on a per capsule basis, e.g., locating a local extension. However, there is no convenient place on the capsule class to store these partial results in a type-specific manner. Static fields are a possibility, but do not allow multiple nodes per Java runtime, as may be the case when emulating large networks. Other mechanisms, e.g, passing precomputed values as parameters or retrieving them from the cache, would complicate the programming interface and may actually hurt performance.

A different form of this issue occurs when one type of capsule repeatedly accesses information maintained by another, e.g., custom routes that are occasionally refreshed. Rather than the maintainer storing the information in the cache for frequent lookup by the accessor, it would be more efficient to pass the information to the accessor directly. Again, this requires a place to hold these partial results.

Inter-Capsule Calls

Calls between capsules occur in a variety of situations (e.g., when one type of capsule creates another type) and require that the name of the class be known. At coding time and for applications, capsule classes are known only by a local name, e.g, `PingCapsule`, and using this name is the only convenient coding option. At runtime and within the network, however, capsules should only be known by a type identifier – local names may cause collisions. It is difficult to implement this distinction without altering the Java runtime, and so the current implementation fails to hide local names within the network.

One solution may be a pre-loading phase that re-writes compiled code; type identifiers cannot be embedded in the capsule code before compilation because they are based on a fingerprint of the compiled code. This is complicated by the need for both node and application capsule names to exist at end systems. If this solution is unworkable, then the API may need to identify and handle inter-capsule calls explicitly.

Demand Loading is Implicit

The current API makes demand loading implicit: it occurs in response to the arrival of a capsule of unrecognized type. Directives for pre-loading a path or carrying code with some capsules may be useful to reduce startup latency. One possibility is to provide alternate forms of `routeForNode()`, etc., such as `fatRouteForNode()`. It seemed premature to do this without first measuring the costs of demand loading in detail.

Helper Classes and the Cache

The interface for manipulating the cache is based on generic objects. It may therefore be useful to carry helper classes (i.e., non-capsule classes) within a code group to support particular objects for particular applications. This is not well addressed by the current situation in which only object wrapper classes available at the node can be used to manipulate the cache. On the other hand, the need for helper classes may be considered to be a shortcoming of the API or its implementation.

Load-Time Checks

The integrity of the system relies on several checks that are not currently implemented. All appear to be feasible with a single load-time pass over Java bytecodes, but if they are not then some restructuring may be necessary.

- `serialize()`, `deserialize()` and `length()` must begin with a call to the superclass method. They must call no other methods except those in the Xdr library.
- The type identifier of the capsule must be statically initialized. The default constructor must consist of a single call to the superclass constructor that passes the type identifier of the capsule.
- No methods should catch security exceptions, generic exceptions, errors or throwables. No methods should throw errors or throwables.

Class Documentation

Finally, much of this paper should have been able to be generated on-the-spot from comments left in the source files. It was not that simple.

A Sample Code Appendix

To provide a concrete example of the API in use, this appendix provides the code of a simple ANTS test application: ping. It is taken directly from the source repository and unedited. It consists of several files that define capsule, protocol, and application classes, respectively.

A.1 Ping Capsule Class

```
package apps;

import ants.*;

/**
 * ping capsule processing
 *
 * @author David Wetherall
 */

public class PingCapsule
  extends DataCapsule
{
  final static byte[] MID = findID("apps.PingCapsule");

  public boolean ping = false;

  public int length() {
    return super.length() + Xdr.BOOLEAN;
  }

  public Xdr serialize() {
    Xdr xdr = super.serialize();
    xdr.PUT(ping);

    return xdr.trim();
  }

  public Xdr deserialize() {
    Xdr xdr = super.deserialize();
    ping = xdr.BOOLEAN();

    return xdr.trim();
  }


  public boolean evaluate(Node n) {
    if ( n.address == dst) {
      ping = true;
    } else if (ping!=true) {
      return n.routeForNode(this, dst);
    }

    if (n.address == src) {
      return n.deliverToApp(this, dpt);
    } else if (ping) {
      return n.routeForNode(this, src);
```

```
    }
    return false;
  }

  public PingCapsule() { super(MID); }
  public PingCapsule(byte[] mid) { super(mid); }

  public PingCapsule(short sa, short da, int na, ByteArray d) {
    this(MID, sa, da, na, d);
  }

  public PingCapsule(byte[] mid, short sa, short da, int na, ByteArray d) {
    super(mid, sa, da, na, d);
  }

}
```

A.2 Ping Protocol Class

```
package apps;

import ants.*;

/**
 * Ping protocol definition
 *
 * @author David Wetherall
 */

final public class PingProtocol extends Protocol {

  public PingProtocol() throws Exception {
    startProtocolDefn();

    startGroupDefn();
    addCapsule("apps.PingCapsule");
    endGroupDefn();

    endProtocolDefn();
  }
}
```

A.3 Ping Application Class

```
package apps;

import java.awt.*;

import ants.*;
import ants.exception.*;
import utils.*;

/**
 * test GUI application with ping buttons
 *
 * @author David Wetherall
 */

public class PingApplication
  extends Application
  implements Runnable
```

```
{
  final public static String[] defaults = {};

  int target, rTotCount = 0, delay;
  long beginTime;
  Button pinger;
  Label latency, received, thruput;

  TextField iterations, interval;

  synchronized public void receive(Capsule cap) {
    super.receive(cap);

    switch ((rTotCount % 100)) {
      case 0:
        beginTime = thisNode().time();
        break;
      case 99:
        double diff =
          (((double)thisNode().time() - (double) beginTime) / 1000.0);
        double thru = 100.0 / diff;
        thruput.setText(String.valueOf(thru));

        if (cap instanceof PingCapsule) {
          PingCapsule pcap = (PingCapsule)cap;
          Xdr buf = new Xdr(pcap.getData(), 0);
          long lat = thisNode().time() - buf.LONG();
          latency.setText(Long.toString(lat) + " ms");
        }
        received.setText(Integer.toString(rTotCount + 1));
        break;
    }

    rTotCount++;
  }

  public boolean handleEvent(Event evt) {
    if (evt.id == Event.ACTION_EVENT && evt.target == pinger) {
      new Thread(this).start();
      return true;
    } else
      return super.handleEvent(evt);
  }

  public void run()
  {
    int iter = Integer.parseInt(iterations.getText()),
      interv = Integer.parseInt(interval.getText());

    for (int i=0; i < iter; i++) {
      Xdr buf = new Xdr(Xdr.LONG);
      buf.PUT(thisNode().time());
      PingCapsule c = new PingCapsule(port, port, target, buf);
      send(c);
      thisNode().sleep(interv);
    }
  }

  public void setArgs(KeyArgs k)
    throws Exception
  {
    k.merge(defaults);

    for (int i = 0; i < k.length(); i++) {
      if (k.key(i).equals("-target")) {
```

```
      target = NodeAddress.fromString(k.arg(i));
      k.strike(i);
    }
  }

  super.setArgs(k);
}

public void start()
  throws Exception
{
  thisNode().register(new PingProtocol());

  resize(400, 200);


  // 2 rows by 2 columns with 2 pixel point spaces in between them
  setLayout(new BorderLayout());

  pinger = new Button("ping " + NodeAddress.toString(target));

  Panel outputDisplay = new Panel();
  outputDisplay.setLayout(new GridLayout(6, 1, 1, 1));
  outputDisplay.add(new Label("Capsules Received", Label.CENTER));
  received = new Label("0", Label.CENTER);
  outputDisplay.add(received);
  latency = new Label("0", Label.CENTER);
  outputDisplay.add(new Label("Capsule Latency (ms)", Label.CENTER));
  outputDisplay.add(latency);
  thruput = new Label("0", Label.CENTER);
  outputDisplay.add(new Label("Throughput (cap/s)", Label.CENTER));
  outputDisplay.add(thruput);

  Panel iterDisplay = new Panel();
  iterDisplay.setLayout(new BorderLayout());
  iterDisplay.add("Center",
                  new Label("Num Iterations", Label.CENTER));
  iterations = new TextField("0");
  iterations.setEditable(true);
  iterDisplay.add("South", iterations);

  Panel intervalDisplay = new Panel();
  intervalDisplay.setLayout(new BorderLayout());
  intervalDisplay.add("Center",
                      new Label("Ping interval (in ms)", Label.CENTER));
  interval = new TextField("0");
  interval.setEditable(true);
  intervalDisplay.add("South", interval);

  Panel topPanel = new Panel(), botPanel = new Panel();
  topPanel.setLayout(new GridLayout(1, 2, 1, 1));
  botPanel.setLayout(new GridLayout(1, 2, 1, 1));

  topPanel.add(pinger);
  topPanel.add(outputDisplay);
  botPanel.add(iterDisplay);
  botPanel.add(intervalDisplay);

  add("South", botPanel);
  add("Center", topPanel);
  pack();
  show();
}

public PingApplication()
```

```
    throws Exception
  {
    super();
  }
}
```

References

[1] D. S. Alexanger et al. Active Bridging. In *SIGCOMM'97*, 1997.

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

[3] C. A. Gunter et al. PLAN: A Programming Language for Active Networks. Available from http://www.cis.upenn.edu/ switchware/PLAN, July 1997.

[4] A. Jackson et al. SmartPackets. See http://www.net-tech.bbn.com/smtpackets/smtpackets-index.html, 1997.

[5] D. Wetherall et al. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. Submitted to OPENARCH'98.

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