Making ANTS Suitable for Per-Protocol Resource Control

Godmar Back
gback@cs.utah.edu

Flux Research Group
University of Utah
http://www.cs.utah.edu/projects/flux/

June 1, 1998

Abstract

ANTS 1.2 provides a basis for experimentation with active protocols in a user-mode JVM. We have decided to use ANTS as a component of our "JANOS" active network effort (a Java-based Active Network Operating System). One of the objectives of JANOS is to provide resource control over active protocols, for which ANTS 1.2's execution model was not well suited.

Therefore, we have begun to change the structure and execution model of ANTS to allow for better control over the resources used by a particular protocol, and also made some infrastructure improvements. These notes describe our changes and their rationale. A snapshot release of ANTS containing our changes is available; MIT expects to integrate the changes as they get time.

1 Summary of changes

This is a summary of the changes on which the following sections will elaborate.

- **Separation of ANTS run-time system and Node OS services.** We have started to separate the actual ANTS run-time system from underlying services that will be provided by a Node OS. This separation was made explicit by putting the services in a different package. This should allow for different implementations. At this time, the node services include a simple dispatch framework for incoming packets and a node OS buffer type.

- **Introduction of a dispatch framework for incoming packets.** We implemented a dispatch framework in Java that allows early packet demultiplexing. Our framework may be extended by active protocols.

- **Providing separate name spaces and threads for each protocol.** While ANTS 1.2 used one global name space for all remotely loaded protocols, we now run each protocol in its own name space. ANTS 1.2 used one thread per channel; we now use one thread per protocol, opening the possibility of per-protocol resource control.

- **Additions, updates, and fixes.** We added a simple remote console that allows one to connect to a running router to obtain and set status information. We updated the code base to JDK 1.1 and fixed a number of minor things. These changes are detailed in the Appendix.

ANTS' author, Dave Wetherall of MIT, has reacted positively to a general outline of our changes and intends to incorporate them into the main distribution when his time allows. Our ANTS snapshot is available from http://www.cs.utah.edu/projects/flux/janos/.

2 Separation of ANTS run-time system and Node OS services

ANTS 1.2 already provided for the use of different implementations of services upon which the ANTS run-time system relies. For instance, subclasses of the abstract base class `Channel` provided implementations of `send()` and `receive()` that were specific to a particular transport, such as UDP or Ethernet. We believe we improved upon that already existing separation.

We made the separation between ANTS run-time and the services it uses explicit by putting the services in a separate package, called `nodeos`. It should be noted that this separation is nowhere near complete: at this time, only two services have been extracted. These services are the dispatching of incoming packets and the scheduling of outgoing packets.[1] In addition, the Node OS package provides a buffer type upon which ANTS bases the Xdr streams it uses for encoding and decoding.

In the future, this package will be enriched with services to control the CPU and memory usage of different protocols. These services are currently under development.

The current implementation is in user mode, but all kinds of implementations are conceivable—essentially, the Node OS package is responsible for wrapping the abstractions provided by the actual Node OS and presenting them to the ANTS run-time. For instance, a Scout-based implementation might use message trees to implement buffers and paths and their demultiplexing capabilities to implement the dispatch of packets.

3 Dispatch framework for incoming packets.

In ANTS 1.2, a *channel* was the entity responsible for both sending and receiving capsules. The Channel base class would implement encoding and decoding, and subclasses would provide send and receive methods specific to an underlying transport. A ChannelThread would loop receiving, decoding, and executing packets for a channel. In our opinion, this approach mixed three or four functions that we believe require separate attention. These functions are:

- Sending of packets, which requires the scheduling of outgoing packets.
- Dispatching incoming packets.
- The encoding and decoding of the packet content.

For this reason, we split the Channel class in two: The new Channel is only responsible for sending and encoding; a class called InChannel is responsible for executing packets. The InChannel class implements Runnable and has a similar function as ChannelThread. Packet dispatching is provided by the `nodeos` package and the ANTS run-time system combined, as described below. The following subsection explains our rationale for our dispatch model, and why we believe that it is more appropriate for ANTS; we do not claim that it is novel. Its main focus is on resource control.

3.1 Rationale on Dispatchers and Flows

The processing of incoming packets poses a special problem for the management of resources for two reasons. First, the time needed to process the interrupt in which the packet is received cannot be charged to any application since the destination of the packet is not known beforehand. Finding the packet's destination, however, requires examining the content of the packet. Secondly, the memory in which the packet is received cannot be immediately accounted for. Druschel et al. have shown in their work on lazy receiver processing

[1] The scheduling currently works only in a simulator and is not part of this distribution.

(LRP) that overall system throughput suffers significantly if too much time is spent on processing packets that must ultimately be discarded, especially in an overload situation.

The traditional solution for this problem is to minimize the amount of processing that is not accounted for. The packet processing is divided into two steps: packet classification and subsequent protocol processing. The packet classification step determines the receiver of the packet. The subsequent protocol processing occurs later in the context of the receiver, subject to the receiver's resource limits.

The packet classification process usually reflects the layering of protocols: a layer in the protocol stack uses a field in this layer's header to determine the higher layer to which the packet is destined. For instance, ethernet packets use a type field to identify IP packets, the IP header contains a protocol field to identify UDP packets, and so on. Other packet receivers use different mechanisms: for instance, the packet may be run through interpreted or dynamically compiled packet filter routines. Active protocols executing on a node must have the ability to extend the system's classification mechanisms in a safe, yet flexible way.

Flow dispatchers. Our system uses a chain of dispatchers to identify incoming packets. A dispatcher is a Java interface that consists of a single routine that accepts incoming packets. Depending on its implementation, a dispatcher may buffer a packet or identify another dispatcher to which the packet must be presented, possibly after skipping a header. Since the dispatch routines are executed at non-accounted interrupt time, their execution time must be individually and globally bounded. They should not make copies of the buffer itself. In the general case, guaranteeing these properties requires worst-case execution time (WCET) analysis. We make the following simplifying assumptions: first, we assume that trusted dispatchers adhere to this restriction. Second, we assume that untrusted dispatchers—such as those used by untrusted active protocols extending the dispatch chain—are only implemented as subclasses of trusted dispatcher base classes, except for one single untrusted method that examines the actual buffer content and makes a yes/no decision as to whether to accept this packet, akin to traditional packet filters. We expect to be able to restrict the code of these filter routines in a way that will allow a simple verifier to assure their safety, i.e., their boundedness. If we had more general verifier available to us that would allow us to ensure the safety of the more general dispatch function, we could extend the trust perimeter even further.

Flow dispatchers that demultiplex packets to higher layers act as as partial classifiers. Defining these dispatchers in our framework allows them to be written in a modular fashion, which allows different combinations of protocols to be assembled.

Flow handles. A flow handle is a special kind of dispatcher that is associated with two queues: a queue of free buffers and a queue of already received packets. A flow handle, which is implemented as a trusted system class, checks in its dispatch routine whether the queue of free buffers is empty. An empty free queue means that the buffer space for this flow is exhausted. The flow handle was not supplied with enough memory and no further action is taken. Otherwise, a buffer is dequeued and "exchanged" with the system buffer that is presented to the dispatch routine. The buffer containing the data is then enqueued in the handle's received queue. In this way, a receiving flow is directly charged for the incoming packet when it is demultiplexed.

Flow handles must be supplied with preallocated buffer space upfront. For this purpose, they provide methods to allocate buffers which are charged to the caller's budget, or to recycle buffers handed to them by a caller.

Flow handles are also used to "receive" a packet, just as one would use sockets to receive a packet. Receiving a packet simply involves dequeuing a packet from the handle's receive queue. As usual, receivers that are blocked on an empty receive queue are awoken when a packet is enqueued.

Dispatching packets might involve traversing internal data structures; for instance, an ANTS node will try to look up the method associated with an incoming packet in its code cache. To avoid duplicating this work once the packet is ready to be processed, the dispatcher may set buffer-specific data at dispatch time which can be read at packet processing time.

Dependent buffers. Copying packets is one of the operations dispatcher routines cannot do. However, a packet may be presented to—and accepted by—multiple flow handles, each requiring its own writable copy of the data. As mentioned above, the first accepting flow handle exchanges its free buffer with the system buffer in which the packet was received. The second and all subsequent handles will also dequeue buffers off their free queues. These buffers, however, are then declared *dependent* on the original buffer containing the data. Each dependent buffer keeps track of its original buffer, and each buffer containing an original copy of the data keeps track of its dependents in a singly linked list. Before a write or map operation can be performed on the buffer, it must be made "independent." There are two cases: for a dependent buffer, this involves copying the data from the master copy into its own memory and removing this buffer from the list of dependents. For a master copy with dependents, it involves copying the data to the first dependent, which is then declared the master for the remaining dependents. It can be seen that copying the data can always be postponed to a context where the time spent copying is accounted for. The scheme is slightly unfair in that it benefits the thread picking up the last copy from a set of dependent buffers. It avoids, however, the problems associated with computing the costs of sharing buffers. Read operations on the buffer do not require dependent buffers to be made independent because the data can be copied directly from the master source.

3.2 Use of dispatchers in ANTS

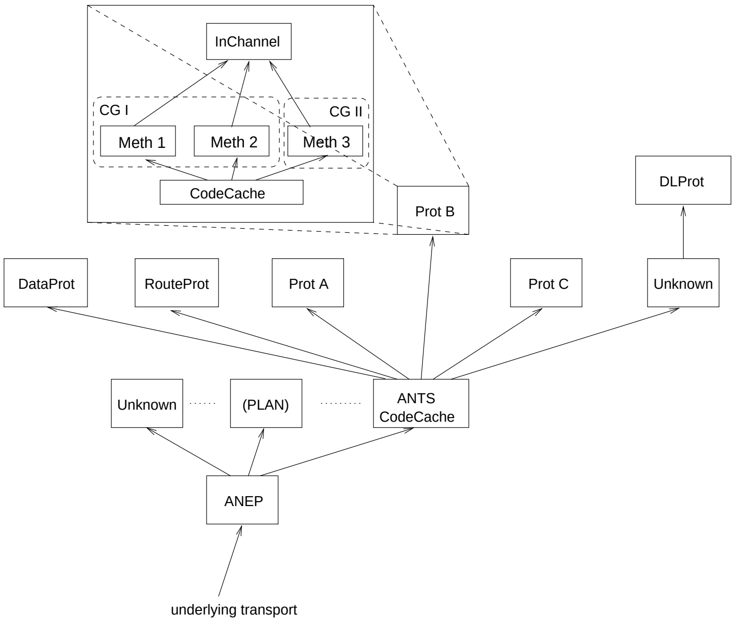

Figure 1: The new ANTS dispatch tree

Figure 1 shows the dispatch of packets in ANTS. Each box represents an implementation of the dispatch interface. The Node OS package provides a dispatcher for ANEP packets. This dispatcher accepts registrations for different ANEP types, to which incoming packets are dispatched. A special flow handle may be registered to which packets are dispatched whose types are unknown. A dedicated thread will pick up packets from this flow and provide the default behavior that ANEP demands.

In ANTS, packet types correspond to method IDs. The code cache maps these IDs to Methods. Therefore, the `CodeCache` class implements the dispatch interface to dispatch packets to the various methods. We changed the `Method` class to implement the dispatcher interface as well. There is initially one flow handle, i.e., one actual queue per protocol. However, one protocol may contain multiple code groups, each of which may contain multiple capsule types, corresponding to multiple methods. Currently, all methods dispatch incoming packets to their protocol's flow handle.

We include both built-in and (locally or remotely) loaded protocols in this scheme. Unknown packets are dispatched to a flow handle that is served by the internal protocol that initiates the loading of code on demand. In this way, the node can decide how much of its resources it wants to dedicate to route updates, built-in data capsules, and the loading of new protocols.

Our scheme will allow protocols to subdivide their resources: by creating an additional flow handle and spawning off a thread that serves it, a protocol can in the example shown in the figure dedicate some part of its resources to the processing of packets belonging to code group I, and another part to those belonging to code group II. If the packet types in code group I represented multicast subscriber capsules, and the packet types in code group II multicast data capsules, it can ensure that a flurry of subscriber messages does not interfere with an ongoing multicast, or that subscriber capsules will always processed with a higher priority than data capsules, depending on the specific requirements of the protocol. The next step will be to define the interfaces protocols can use to do this subdivision.

Overhead. The user-mode implementation uses an additional thread to actually perform the receive and invoke the dispatcher(s). The thread will then signal the thread(s) waiting on the flow handle(s) to which the packets were dispatched. This involves an additional context switch. However, to be frank, we believe that user-mode implementations without OS support for the dispatch will remain experimental prototypes; so we do not consider this additional overhead a problem. With OS support, the interrupt handler will directly awake the Java thread to which the packet was dispatched.

Summary. The way packet dispatching is done is by defining an interface called `FlowDispatcher`. Implementations of this interface are expected to finish quickly. Each node in the dispatch tree implements this interface in a way that is specific to the dispatch process at that level. Dispatched packets are buffered in a queue when dispatched to a flow handle that is directly associated with the receiver, i.e., the unit of resource consumption.

4 Separate name spaces and threads for each protocol.

ANTS 1.2 shared one name space between all capsules. This prevents protocol developers from using the same names for otherwise completely unrelated protocols, and requires the restart of one node when capsule code had to be changed, but retained its name.

We now create one class loader per protocol. The capsules of that protocol live in that name space. Note that only the capsules themselves are instantiated in this new name space. Applications, for instance, are not. This means that two unrelated applications running on one node may not use the same class name for their (unrelated) capsules. (This is one of the next things to fix.)

Secondly, ANTS 1.2 used one thread per channel to process packets, independent of which protocol they belonged to. This does not allow the assignment of CPU time on a per-protocol basis. In addition, since ANTS 1.2 lacked timers to kill off longer-running capsule evaluation, a malicious capsule could stop the system, not just the protocol to which it belonged. While the latter problem could have been solved by adding timers and a scavenger thread, as alluded to in the documentation, we believe that firing up one thread per protocol is the right approach. We believe that the startup costs will be amortized, just like the costs for the code-on-demand loading protocol are expected to be amortized.

Unknown capsule types are first picked up by the DLProtocol, and a DLRequestCapsule is sent to where the capsule came from. At the same time, a new Method object is created and an entry in the code cache associated with it. This method creates a new flow handle. The "suspended" capsule is enqueued on the flow handle. Once the first DLResponse capsule is received, a new CodeGroup is created with its own flow handle. The flow handles of all methods belonging to that code group are merged with that flow handle, i.e., their packet queues are concatenated. If the first code group is complete, a new InChannel is created. This InChannel is provided with the code group whose completion caused its creation. During initialization, it will create a new class loader and use that class loader to load the classes into the JVM. The new InChannel will obtain the code group's flow handle and start evaluating the capsules queued on it. It can be seen that flow handles allowed us to get rid of the additional `Waiter` objects in ANTS 1.2.

It should be noted that there is a potential for a denial-of-service attack as well as for an accounting problem here: the memory used by the DLProtocol to store the byte arrays containing the bytecode cannot by charged to any protocol before the protocol is known. We consider this unavoidable; out of band methods must be used to deal with an attack where nodes abuse the DLProtocol to deplete a router's memory resources. Once the protocol is known, the memory allocated for it by the DLProtocol must be "transferred" to the newly created InChannel, since it must be charged to its budget. (This is not done yet.)

As mentioned above, flow handles are associated with free queues. At dispatch time, packets are discarded if the free queue is empty. Therefore, the InChannel must ensure that the flow handle it serves has at least one buffer in its free queue. Normally, the buffer used by a capsule can be reused immediately after its evaluation. In some circumstances, for instance, when the capsule was delivered to an application, this is not possible and a new buffer must be allocated to replenish the free queue.

5 Where this is going.

Ultimately, we want to explore what mechanisms are best suited for ANTS protocols and applications, and write a `nodeos` package and JVM that best supports them. We hope that other applications with similar requirements will benefit from that, such as servlets. We will explore the applicability to protocols of a more concrete "process" model in Java, such as the "Alta" nested process model (http://www.cs.utah.edu/projects/flux/java/alta/). Such a process would include the resource controls mentioned or alluded to, as well as the capabilities needed to submultiplex resources, but would also provide features such as a separate name space.

A Appendix

A.1 Remote console

We added a tiny remote console to allow administrative access to a running ANTS node via telnet.

ANTS 1.2 uses the class `ConfigurationManager` to read a configuration file that describes the channels, applications and other options for a particular host. We added an option `-consoleport n`. If it is specified to the configuration manager (on the line starting with 'manager'), the configuration manager will start listening for telnet connections on port n.

Currently, only three commands are supported:

1. `who` gives a list of all protocols currently running.
2. `shutdown` shuts down node.
3. `application` followed by node address, application name and parameters (just as in the config file) starts up a new application on this node.
4. `quit` terminates the connection.

To be added: connect keyword to add routes, channel keyword to add outgoing channels, password protected authorization. A way to actually stop or kill protocols on demand. Dumping of more information, such as routing table etc.

A.2 Update to JDK 1.1

The main reason we deemed this necessary was the awkward dependency on `java.awt` incurred by the `ants` package due to the silly way in which AWT 1.0 worked. ANTS 1.2's "base class", `Entity`, was derived from `java.awt.Frame`, so that subclasses could override `handleEvent`. We eliminated this dependency, making the `ants` package independent of `java.awt` (except for the configuration manager). Applications have to create their own windows, if they want to use any.

Unfortunately, applications will have to be changed. A typical change would be to insert:

```
Frame window = FrameCreator.createFrame(getName());
```

in the application's start method, and then to prefix all calls to AWT-related functions such as `pack()`, `show()`, etc. with "`window.`" The FrameCreator class is an utility class that creates windows that will exit the application when closed, just like ANTS 1.2 did.

Secondly, functions such as `handleEvent` will have to be replaced with the proper 1.1 methods. For instance,

```
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);
  }
```

must be replaced with

```
public void actionPerformed(ActionEvent evt) {
  if (evt.getSource() == pinger) {
    new Thread(this).start();
  }
}
```

The application must implement the `ActionListener` interface, and register itself:

```
pinger = new Button("ping!");
pinger.addActionListener(this);
```

Changing existing applications is really straightforward and took a few minutes. We believe that the added cleanliness is well worth that effort.

In addition, we eliminated all deprecated functions. These were mainly functions dealing with String to byte array conversion.

While stopping at JDK 1.0.2 might have had some advantages, we believe that all relevant JVMs already or in the near future support JDK 1.1.

A.3 An implementation note on `findMID`.

The Capsule class has a static method that is called in the static initializer of a capsule, i.e.:

```
final private static byte[] MID = findMID("ants.DataCapsule");
```

This looks up the name in a table and finds the method id associated with that name. Since these names are now local to one particular InChannel, we must make sure that we consult the table of the InChannel whose class loader loaded this class. To be more exact, since the static initializers are executed in the course of the class loader's defineClass method, it must consult the table of the InChannel that is currently loading that class. Unfortunately, there is no way in Java to identify the current class loader in a static method: in a non-static method, `getClass().getClassLoader()` could be used. To circumvent this problem, we simply map the current thread to its InChannel, and maintain initializers tables per InChannel. We assume that the same thread that call defineClass will also execute the static initializers of the class being defined.

A.4 GNU make and preprocessor

We changed the build environment to support ".Java" files, which are preprocessed by the cpp preprocessor into ".java" files.

This might be a controversial undertaking; C and C++ have often been criticized for encouraging programmers to make overly liberal use of the preprocessor, which can decrease the readability of code. We believe that the preprocessor should be used sparingly, in the spirit of Java.

We use it to avoid having to hardcode package and import names, for assertions (since the compiler will not eliminate function calls if debug is disabled), and for conversion macros (such as extracting an integer value from an array of bytes) since the Java compiler does not inline these functions.