PANTS
- PYTHON ACTIVE NODE TRANSFER SYSTEMS

TECHNICAL REPORT NUMBER 525

MARCH 2000

Amila Fernanado, Bob Kummerfeld, Alan Fekete, Michael Hitchens
(Basser Department of Computer Science, Faculty of Science)
PANTS - Python Active Node Transfer System

Amila Fernando, Bob Kummerfeld, Alan Fekete, Michael Hitchens

\{amila, bob, fekete, michaelh\}@es.usyd.edu.au

Technical Report
Basser Department of Computer Science
University of Sydney
New South Wales, Australia

Abstract

Existing active network architectures depend on using static languages to improve performance. Unfortunately this limits some of the more dynamic features of an active network. In this paper we present PANTS - a dynamically extensible active network architecture. We describe a new architecture for an active node which does not depend on language based security, but provides the flexibility to perform dynamic changes to the node, and to the capsules. An implementation conforming to this architecture is described and compared with existing active network architectures.
1 Introduction

Active Networks [1, 2] is a new framework which allows programming to be taken into the network layer. Applications can inject packets into the network which have code associated with them. This code is then executed at the routers, or switches, as the packet propagates through the network.

These executable packets (called capsules) offer considerable flexibility over traditional networks, as they can examine a node and adjust to current conditions as they move from node to node.

Apart from allowing programmable packets, active networks allow for an elegant solution to deploying and extending new protocols. Current traditional networks suffer in this regard as deploying and implementing a new protocol takes a very long time (eg making the transition to IPv6 [3] from IPv4 in the Internet). By allowing the nodes themselves to be programmable or dynamically upgradable, we can shorten the time needed to use newer and more efficient protocols.

This Report presents the internal design and implementation of Pants, a dynamically extensible active network simulator. Pants was originally inspired by ANTS [4], a prototype built at MIT by David Wetherall et al.

We first briefly describe the active architecture reflected in previous work (eg ANTS [4]), then we present the PANTS architecture which takes a more dynamic approach to active Networks (Section 3) and then describe an implementation which conforms to this architecture (section 4). We show some of the advantages that can be gained by the use of a dynamic active architecture (Section 8) by use of examples. We then compare our architecture (Section 6) with some currently established active architectures.

2 ANTS

ANTS [4] is a toolkit developed at the Massachusetts Institute of Technology, which allows experimentation with active network protocols. An ANTS based network consists of a group of ANTS nodes connected together. A capsule is a generalization for a packet, which includes a reference to a forwarding routine which is used to process the capsule at each node.

Capsules are grouped into code groups, which are a collection of capsule types, and these code groups in turn are grouped into protocols. A protocol is treated as a single unit of protection, and capsules within a protocol can access shared information.

The ANTS architecture uses Java [5] as the language in which capsules are programmed. Java (through compilation into bytecodes and their verification) allows capsules to be executed in a safe sandboxed environment. An ANTS node exports a small set of primitives into the restricted environment that capsules execute in. These primitives supply a capsule with an interface to access node resources and perform more optimized common tasks (such as forwarding).
3 PANTS Architecture

PANTS was initially inspired by ANTS. Many features are inherited from ANTS, but the architecture is much more dynamic. PANTS also offers additional features which are of interest for Routing and Bandwidth Sharing. The Pants model is equivalent to an execution environment as described by the AN Working Group in [1].

After experimentation with the Ants prototype a few improvements to the overall architecture were designed. First an abstraction of a Link between two nodes was integrated into pants. This would allow a more cleaner platform to build and experiment with Bandwidth sharing policies.

Secondly in Ants, there is a distinction between the code fault handling and the transportation of Capsules. The capsules get placed in a buffer upon request to be transported, but the code handling is done at different level and not on the same buffer or medium as the capsules. In Pants this is removed and both code fault handling and capsules are subjected to the same policy management issues.

The method of code distribution should be variable, as it is not always advisable to demand load code all the time(because of the latency associated with the first time a capsule is sent over the network). It should be possible for the capsule/application itself to have control over the method of code distribution so more intelligent choices can be made.

To address routing issues in an active network, it should be possible to examine, and change the routing tables dynamically. It should be possible for nodes to start with no routing information, but be able to build up a full routing table with the help of some active protocol without resorting to calculated static routing tables.

The system should provide some mechanism whereby the node can protect itself from runaway or malicious capsules.

More radically, the architecture should allow a node to change dynamically. This would give a node administrator freedom to change the way a node behaves, and so allows a method by which nodes can be upgraded dynamically or allow installation of new protocol support at a node, without it being taken down.

Figure 1 shows a simplified layout of a pants node. The node OS is responsible for maintaining the resources of the node. These resources are fully available to the interface object. It is the task of the interface object to share and restrict access to these resources among capsules.

The nodes are connected to each other via Links (figure 2). Capsules are injected into the network by applications running at some node. The capsule then routes itself through the nodes, accomplishing some task.

After experiences with implementing bandwidth sharing policies in ANTS [6, 7], it was decided to include abstractions for links between nodes in PANTS. These links in turn can have policies which describe how to transmit on that link.

A capsule after injection, first goes through some security checks by the node, and is then allowed to execute in a capsule environment. The capsule environments are created dynamically and discarded after use.
3.1 Application

An application is a process which runs on a node, and provide a means for end to end communication. Capsules are initially injected into the network by applications, and capsules may be delivered from the network to an application.

3.2 Capsules

A PANTS capsule is conceptualized as an object that can distribute itself over an active network and execute at those intermediate nodes. When transmitting capsules, the state of the capsule object (its variables, and corresponding values) are extracted (serialized) and transmitted. The code for a capsule need not be transmitted if the code is cached at the receiving node. Code handling is discussed in more detail in section 5.1.

A capsule does not concern itself with its serialization. This is handled automatically by the node when a request is made to send a capsule. The details of how the object is to be serialized, and distributed is left to the node OS. A transport layer exists underneath
the Capsule layer which guarantees that the capsule will be delivered free of errors (if the destination is reachable).

This allows for small clean capsules to be written which concentrate solely on the task they perform (see section 5.1 for an example).

A capsule is given several attributes to allow it to be identified by any node that processes it (table 3.2). In ANTS, there is also an extra identifier which specifies what group (or protocol) the capsule belongs to. A PANTS capsule does not carry such information but provides a mechanism to group capsules dynamically (see section 4.4).

<table>
<thead>
<tr>
<th>run id</th>
<th>This is a globally unique identifier for a distributed instance of some capsule type. It is envisaged that the run id will be formed in a hierarchical manner. (e.g. proc-id.networkaddr).</th>
</tr>
</thead>
<tbody>
<tr>
<td>type id</td>
<td>This is a globally unique identifier for the code type the capsule belongs to. This is used for code handling (see section 5.1).</td>
</tr>
</tbody>
</table>

Table 1: Special Capsule Attributes

3.3 Capsule Environment

Capsules for the most part contain code that is untrusted. The node builds an Execution Environment for the capsule, to try and keep malicious capsules at bay, and maintains tight control over the environment.

Once a capsule arrives at a node, the node OS creates an execution environment for the capsule. The capsule is loaded into the environment by the node OS and the capsule is invoked.

The capsule is invoked with 2 arguments, a reference to the capsule itself and a reference to the interface object. These are the only two objects that the capsule can see initially. and unless the capsule has the authority to load in extra objects these will be the only 2 objects that it has access to.

The capsule environment created for the capsule is restricted in that the capsule has access only to those objects within it, and the capsule can do no system tasks other than computation without the help of extra modules that may be loaded through the interface (see section 3.4).

3.4 Capsule Interface

Since a capsule cannot do any useful work (other than number crunch) it is effectively sand-boxed. Those capsules wishing to access the resources at a node must first load services into their execution environment (provided that they have the authority to do so) and may then make use of such services. The authorization and loading is done by the interface object that the capsule is given. This is useful for several purposes:

- the capsule should only load in services that it needs for the task it was designed for. Thus the execution environment of the capsule does not get cluttered with unnecessary services.
- The node can enforce some security mechanisms such as authentication and authorization when a service is requested to be loaded.

The interface object provides a very minimal set of primitives which the capsule can work with (Table 2).

<table>
<thead>
<tr>
<th>addt()</th>
<th>Returns the address of the current node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(service, secinfo)</td>
<td>Allows the capsule to load services provided by the node into the capsule's environment. The capsule provides the name of the service and some security information.</td>
</tr>
</tbody>
</table>

Table 2: Interface Methods

A node should provide a minimum set of services that have been named according to some convention. Apart from these services the node may also provide extra services that are node implementation specific (e.g. a node can provide a service which allows a capsule to modify the node runtime).

Note that apart from the most basic primitive of finding out where it is, a capsule can do almost nothing useful without loading in some service (for example services must be loaded even for routing itself to another node and delivering itself to an application). This model allows us to concentrate our efforts into making the interface secure.

3.5 The Dynamic Nature of Nodes and Capsules

One of the key advantages that active networks have over traditional networks is the ability to remotely and dynamically change how a node behaves. This would allow new features or protocols to be implemented and deployed more efficiently.

Current active network architectures allow programmers to develop capsules, but do not allow the node runtime to change dynamically.

For example it should be possible for a capsule to carry and install a new service at a node (providing it passes authentication and authorization), or even perhaps modify the node’s default routing algorithm if desired. To achieve the former flexibility, services are built as loadable objects (section 4.2). To change the routing algorithm we depend on the dynamic nature of the language of the PANTS system (section 4).

Capsules in other active network architectures are only allowed to be programmed in a strongly typed language [4, 8] such as Java. Using a strongly typed language allows for easier and more efficient security checks, but sacrifices flexibility that a capsule may achieve with a more dynamic language. For example it would be extremely difficult to write self-modifying capsules in an architecture like [4]. In allowing this flexibility we sacrifice some performance, as runtime checks now need to be made as the code executes.

3.6 Security Considerations

A general security architecture for active networks was proposed by the Active Networks working group [9]. A more detailed security architecture for PANTS has been developed, building upon the Active Networks working groups architecture. The security
architecture relies heavily on cryptographic methods. We refer the reader to [10] for an in-depth discussion of our security architecture. We describe here briefly, some of the issues involved.

The security architecture defines a number of principle entities that are the responsible entities in the network. These can be a user, node or service which cause network activity. We will concentrate on what measures are used against active capsules that these principles introduce into the network.

First a capsule needs to be authenticated upon receipt by a node to ensure they are from a trustworthy source. This involves the capsule containing a certificate and a digital signature. The certificate details the necessary information to determine who the responsible entity for the capsule is.

The creating, issuing and management of certificates is handled by an entity known as the Certification Authority (CA). This is a trusted application running on well-known nodes. Principles requests a certificate from the CA, and uses this certificate when performing service requests or transmitting capsules.

Authentication by itself is not sufficient to determine if the capsule is allowed to perform the relevant task. The responsible entity must also be authorised after authentication, to make sure it indeed does possess the required privileges. Authorisation is performed dynamically as follows.

A capsule once authenticated, is allowed to run in a safe sand boxed execution environment. The environment blocks direct access to any node resource other than a single interface object given to the capsule at startup. Access control is enforced by runtime checks against an access control list whenever a capsule requests a service to be loaded into its execution environment.

Some performance is lost due to the overhead of dynamic authorisation of node services. This loss is regrettable but must be in place due to the dynamic nature of PANTS capsules. However these checks only need be done once, when a capsule requests the relevant service to be loaded.

One more issue in the PANTS architecture that deserves mention, is the facility to dynamically modify the node infrastructure. While it is very desirable to allow such modifications to the node runtime, it also provides an opening for malicious capsules to abuse this power. The most stringent and highest level of authentication and authorisation must occur before allowing a capsule to change the node runtime.
4 Implementation

The PANTS architecture has as one of its main design goals, the ability to dynamically change the node runtime. To achieve this flexibility we needed a dynamic language which would still give us the ability to achieve security. After surveying several languages, Python [11, ?] was chosen as the language to implement PANTS.

Python is a dynamically typed interpreted language that has no static type checks. Python code is compiled into python byte-codes and executed in a virtual machine. The language is very dynamic and has an extensive Library (which includes serialization and Restricted Execution Environments), and was ideally suited to build our architecture. The Capsules and Applications running on a node are also written in Python, and as some examples will show, it allows us to write interesting code which takes advantage of the dynamic nature of PANTS.

Figure 3 shows a simplified view of some of the internal components interacting within a PANTS Node.

![Figure 3: Internal View of a PANTS Node](image)

4.1 Restricted Execution Environment

Each PANTS node upon start-up initializes a python restricted execution environment [11](henceforth referred to as REE). A new module is created within the execution environment for each capsule that arrives. This allows the capsules to execute in isolation to other capsules ¹.

The REE protects the node by denying certain attributes and methods to the capsule. To do useful work a capsule must Load services. A capsule would first ask the interface

¹In the Panta implementation an optimization is made so that only 1 execution environment per capsule type is created. The capsules within this environment are isolated as they do not see each other
to load a service, providing authorization data in the process. The interface would then check with the corresponding service if the capsule passed Authorization. If it does then the Service is allowed to be loaded into the REE.

Note that while the REE will stop capsules from doing anything harmful (like writing to files) it will not stop a capsule from executing forever. This could be fixed by having a maximum run time for the capsule, and then killing the capsule after it fails to finish execution within this time. However the current Python implementation does not support thread cancellation, and is therefore not implemented in pants).

4.2 Services

A PANTS node has a set of services that it offers to capsules. These may include optimized code for special algorithms (e.g., compression) or provide access to the resources at a node.

Each service can specify the method of authorization that is to be used when a capsule requests it. This allows for different levels of authorization to be applied depending on the service.

The services are each independent and it is possible to load new services without stopping the node. These new services can be loaded in at the node (by an Application) or even by a capsule with the right authority. The new Service is dynamically compiled and loaded into the PANTS runtime, then a change is made to the table in the interface object to reflect the new Service. This same method can also be used to upgrade existing services.

4.3 Routing Services

The primary concern of most capsules will be simply to go to some node. Pants maintains a Route Table object which contains all the information about routes that are known to that node. This object also provides some services which allow capsules to calculate their own customized routes.

The most common service which uses the RouteTable is the RouteTo service. This service directly looks up the nodes’ routing table to see if the destination is reachable, and if so the next hop to take, and pushes the capsule towards the next hop.

For capsules which decide to apply their own routing mechanism (or for other reasons) the Neighbours Service provides a quick list of neighbouring nodes to the current node. The RouteTable object will also record the time at which changes have been made to the routing table, and provides a set of call back services, where a capsule or Application can respond to route table events:

- `WaitTillNeighChange`
- `WaitTillRouteTableChange`

Pants allows a node to start with a minimal routing table or a full static routing table as in ANTS. The minimal table will just contain the knowledge about the neighbors a node has, and will depend on some active routing protocol to determine paths to non-neighbor nodes. In fact due to the protocol described in section 5.1 A node can start
with no routing information, and it can gradually build up a routing table as nodes connect to it.

The one unusual service, MarkUnreachable marks a neighbor as being an infinite amount of hops away. It also goes through the table, looking for destinations which have the same 'nexthop' field set as the unreachable neighbor, and marks these as unreachable as well.

Capsules can also grab a copy of the whole Routing table itself via the RouteTable service.

4.4 Node Soft State

A capsule is allowed (if it has permission to) to store and retrieve information at a node. The information can be as simple as a stream of bytes or as complex as executable objects.

A node administrator may want to put restrictions on who can create and access state information. However capsules will also want to impose some restrictions on who can access the information they create (with regards to other capsules). For example a capsule may insist that only capsules originating from its source node have access to its soft state object, while some capsules may want to make their state objects readable by any capsule.

Instead of setting up distributed groups or protocols as in ANTS, PANTS employs a more active approach. When a capsule makes a request to create a state object it also has the option of specifying an auth method to use as the authorization method when other capsules request access to the state object. The method given can a be default or one that the capsule itself created.

Therefore any capsule that makes a request to access some soft state object must first pass the node’s authorization, and then the creating capsule’s authorization to access some state object.

As well as giving arbitrary flexibility to a capsule in performing access control, this offers an elegant way to group capsules. A capsule can publish itself on a node, and capsules who can pass the authorization mechanism have access to the saved capsule. This allows programmers to write new distributed applications using multiple capsules by dynamically grouping them.

The MakeStateObj Service provides capsules with the ability to create such a state object. The capsule must provide a name for the object, and can optionally provide an auth method

GetStateObj Tries to get a state object with a given name. The capsule must pass the relevant arguments to any auth method as a tuple to the Load method, if the the object is protected by an auth method.

4.5 Active Routing

When a node first attaches itself to a network, it should in essence only know the location of its neighbors. Other nodes that are in the network, are at first unknown to the new node. It is the task of the node to try and find the most efficient and shortest paths to other nodes.
In a traditional network, each node (router) must agree on a routing protocol, and use this routing protocol to develop a routing table. There are many such protocols /cite-Blah/. In active networks, protocols can now make use of the new flexibility offered.

4.6 Active RIP

One of the oldest and simplest routing protocols developed was RIP ?? - a member of the distance vector family. It was a fairly trivial task to emulate the behavior of RIP using pants capsules. Unfortunately this does not take too much advantage of an active network.

To use the activness, a modified version of RIP is currently being implemented. it works as follows:

Upon initialization, a node will send sentinel capsules to each of its neighbors with a copy of its minimal routing table. Upon arrival, the capsules will compare the neighbors routing table with the routing table of its originating node. If any shorter routes or new routes to new nodes are found, it will send back a capsule with this information.

The originating node will broadcast update messages to each of its sentinels when the routing table changes. The sentinels will send alive messages back to the originating node regularly, and will examine the routing tables for changes regularly (by waiting on route table events). If such a message cannot be sent, then the link is assumed dead and the capsule stops execution.

The advantage to this approach is that given a node is running RIP, there is no limitation that, the other nodes also need to be running RIP. Assuming that the routing tables in all nodes are of a similar format, the nodes can use whatever routing protocol that is advantageous to itself.

5 The Node Operating System

While a capsule may only access node resources through one of the services, the services themselves use the underlying Node OS to perform its task. This section describes some of the main objects within the Pants OS. A capsule will never see these objects but instead interact with them through the loadable services.

5.1 Channel

The channel provides an interface to the underlying network. It manages the links to other nodes. The channel is responsible for serializing the state of some capsule and transferring this to another node upon request by a capsule. The channel is also responsible for invoking the loader when capsules are received from a link.

To a capsule the channel implements a transport protocol which guarantees that capsules will be delivered free of error if delivery is possible.

In Pants (as in Ants) if a node does not have the code for a capsule cached it is demand loaded from the adjoining node. This can lead to unnecessary delays for first time capsules, as they traverse the network. To circumvent this problem and to give a capsule more flexibility, a capsule can optionally request that its code be also sent across when the forwarding routine is invoked.
It is also important to note that when code is sent, it uses the same medium as the capsule thus code objects are subjected to the same policies that govern the link. This is all handled by the Channel object in Pants.

Upon initialization the Channel object is given a port number on which it starts a server which listens to connections from other pants nodes. It then tries to connect to other pants nodes to initiate links between them. Once a connection is established the Channel creates a Link object corresponding to that link in the network. As each Link object is created an entry is made in the routing table.

The channel uses the following handshaking protocol over TCP to establish links.

```python
s = connect(neighbor node)
s.send(our address)
reply = s.receive()
if reply == close:
    close connection.
else
    store connection and modify routing table
```

A reply may be a close message if there already exists a Link (Possible if both nodes try to initiate a link at the same time)

Once a Link is established 4 different types of messages can be sent across a Link.

- `StateMsg`
- `CodeMsg`
- `StateNCodemsg`
- `RequestCodeMsg`

When a capsule requests that it be sent across to another node, the Channel is called from the interface to deliver the capsule to its destination. The Channel then extracts the state from the capsule. Only this information is sent over the link initially (`StateMsg`). When a node receives this, it checks in its code cache whether the corresponding code is present at the current node. If it is the Loader is invoked to load and run the capsule. Otherwise the pickled state is stored and a request is sent back on the link, requesting a code update (`RequestCodeMsg`). When a code object is received it is used to update the code cache and any waiting capsule states.

The 3rd type of message (`StateNCodemsg`), includes both the state of the capsule and its associated code. This type of message always has precedence over the cached code and will replace existing code in the cache. This type of message is justified as demand loading code always has a latency penalty for capsules which are sent across first time. Providing this message type allows capsules and applications to use some intelligence when transmitting capsules. The `RouteTo` service has an optional boolean flag which indicates whether this type of messaging should be used.

- put in table with figures comparing `StateMsg` and `StateNCodemsg`
5.2 Link

- Bandwidth sharing
- latency/simulated bandwidth etc
- using TCP socks.
- Policies implemented at this layer

5.3 Loader

The loader is responsible for creating and invoking the capsules in a Restricted Execution Environment. The loader maintains the code cache (which contains the code of the most recently used capsules). If a message arrives with only capsule state, then this code cache is queried to see if the corresponding code is present.

Once the code and state is available the loader recreates the capsule. This capsule is then given to the scheduler which places it in a ready to run queue. Currently the scheduling is simple non preemptive first in first out. Any exceptions generated and not caught by the capsule signals an end to its execution. Due to the current limitation of Python, the Capsules cannot be preemptively canceled, so capsules are trusted to not execute forever.

5.4 Data Flow

Figure 4 shows how data flows from one pants node to another pants node. The capsules are first introduced into the network by an Application. These capsules are then pickled, and put into a send buffer according to some policy that is associated with the link on which they are sent. The policy is responsible for buffer management, and the servicing of capsules waiting to be sent.

It is worth noting that all types of objects sent across the network are subjected to the policy governing that link. From experimentation this has revealed that simple policy schemes like FIFO are not suitable in an active network, as important objects like those that respond to code faults, can get thrown out of send buffers. One solution (that is employed in pants) would be to use weighted fair queuing as the policy, in which code objects are given a high priority user class.

The policy layer is not symmetrical. It is only enforced when capsules are sent across the network. Capsules that arrive are subjected to a different policy - namely the scheduling.

5.5 Typical Life of a Capsule

A Capsule would initially be created by an Application and passed to the Loader. In PANTS an injected capsule always starts execution at the node it is injected. The loader creates a module within the REE for the capsule to run, and invokes the capsule.

The capsule would now perform some calculation and request for some services to be loaded into the REE. Once the capsule has decided which node to send itself (either by its own routing mechanism or by using the node's default routing algorithm), the capsule makes a request to route itself using the RouteTo service.
The capsule now gets passed to the channel which extracts state information from a capsule and packages this into a PANTS message. This is then queued for sending on the relevant link.

The message would be received on the remote node, and the channel would again be invoked to process the message. The node now would check if the code corresponding to the state is cached (if not a request is sent back to get the code), and the capsule is recreated from the state information. The loader would now be invoked from the channel to execute the capsule again. This cycle would continue until the capsule decides to not propagate.

5.6 Dynamic changes to the Pants Runtime

A PANTS node has the ability to be dynamically altered by a capsule, if the capsule contains the correct privileges. This is achieved by getting access to the relevant object within PANTS and overwriting its methods (or creating new methods). The details of the implementation are discussed in greater detail in [12].

One example of the use of node upgrading is where we may want to introduce a new service into a node. A capsule can for example transport and install a new compression algorithm, or routing algorithm at a node, and other capsules may load in this service as normal.
As well as changing the services (which capsules can see) we may also wish to alter the behaviour of the PANTS node itself. For example a PANTS node has the ability to implement different bandwidth sharing policies over links. The default policies can be either FIFO or Weighted fair queueing (See [7, 6, 13] for an in depth discussion about bandwidth sharing and reservation in an active network).

A Capsule can carry code for a new bandwidth sharing policy within it, and then compile and dynamically load this new code into the PANTS runtime. The capsule can then examine the link object and its current policy and replace the old policy with the new policy.

The changes are not limited to bandwidth sharing policies, and almost any of the PANTS internal modules can be remotely upgraded in this manner. The next section describes the process by which one may upgrade the Scheduler within PANTS. However as one would expect dynamically changing the run time of a node is a rather extreme step, and must be attempted with great care. Security is paramount and capsules should only be allowed to take such actions once they have been strongly authenticated and authorised. Our security architecture [10] describes in detail some of the implications of upgrading a node.

The main idea is that pants objects are organized in a hierarchical manner (Figure 5). A privileged capsule can load in a service - TopLevelExecute which executes code at the top of this hierarchy. This code can then traverse the hierarchy and overwrite the contents of the objects.

![Diagram of Pants Object hierarchy]

Figure 5: Pants Object hierarchy
5.7 Upgrading the Scheduler

The PANTS system is implemented as a set of modules which interact. The modules are organised in a hierarchical manner with a toplevel module which loads each sub module. Given access to this toplevel module, any other modules can be changed by traversing the hierarchy and overwriting methods or classes or even whole modules.

As an example we shall consider the upgrading of the scheduler within PANTS. The default scheduler has a method `ReadyToRun` which maintains a simple FIFO queue of capsules which are ready to be executed. We wish to upgrade the queueing scheme so that special network management capsules (which belong to a certain user class) are placed at the front of the queue so they may be processed quicker.

```
def ReadyToRun(self, cap):
    self.lock.acquire()
    self.queue.append(cap)
    self.qlen = self.qlen + 1
    self.qcond.signal()
    self.lock.release()
```

**Figure 6: ReadyToRun method of Pants Scheduler**

Figure 6 shows the default `ReadyToRun` method implemented in the PANTS Scheduler. Figure 7 shows the code for a new method transported in a capsule. Figure 8 contains the code that a capsule would execute in the toplevel module in order to upgrade the Scheduler.

```
def ReadyToRun2(self, cap):
    self.lock.acquire()
    # Capsules of netmanager user class gets priority
    if cap.userClass == UC_NETMAN:
        self.queue.insert(0, cap)
    else:
        self.queue.append(cap)
    self.qlen = self.qlen + 1
    self.qcond.signal()
    self.lock.release()
```

**Figure 7: Extract from capsule code to upgrade Scheduler**

The code searches the Scheduler module and installs the new method in the Scheduler module. It then overwrites (or rebinds) the `ReadyToRun` method to the newly installed method.
exec newschedcode in Scheduler._dict_
# Overwrite method binding
Scheduler.Scheduler.ReadyToRun = Scheduler._ReadyToRun2
# cleanup namespace
del Scheduler.ReadyToRun2
del newschedcode

Figure 8: Extract from capsule code to upgrade Scheduler

new_policy = ""
class WFQPolicy():
    ....
""

exec new_policy in Channel.Link._dict_

# update all links in the node
for l in channel.links.keys():
    # lock the link, so it stops processing new capsules
    channel.links[l].lock()

    # remove old policy.
    del channel.links[l].policy

    # add new policy
    channel.links[l].policy = Channel.Link.WFQPolicy()
    channel.links[l].unlock()

Figure 9: An extract to dynamically install WFQ

5.8 Dynamic Upgrading of Policies

Figure 9 shows a code extract from a capsule that updates a nodes links to the WFQ policy. Variations of the above capsule can be sent throughout the network to enable WFQ, or to selectively add it to a stream of links.

Note in the above example, that we discard all buffered capsules stored to be sent on a particular link. It would be a simple matter to add code, that will extract the buffered capsules and insert them into the new policy.

Once WFQ was implemented and available as a policy to be installed on nodes, more complex applications could be developed. [25] describes a novel bandwidth reservation scheme using WFQ as a basis for an end to end reservation scheme.

5.9 EXPERIMENTAL DATA

A series of experiments were conducted in PANTS to examine the behaviour of dynamic policy changes.

Figure 10 shows the percentage of bandwidth used, by 3 differing applications.
As can be seen, the graph shows the bandwidth being shared somewhat randomly and erratically, depending purely on when a capsule arrives at a node. The FCFS scheme does not care who the data belongs to. After 200 seconds, we introduce a capsule, which carries with it a new policy - WFQ. App1 has 50%, App2 30%, and App3 has 20% of the bandwidth allocated to them. The capsule installs the new policy on the node, which then treats the differing user classes according to their allocated share.

Within a very small time frame, the node shares the bandwidth according to the weighted fair queuing policy.
6 Comparison with Other Active Architectures

The PANTS architecture was initially based on the ANTS architecture built at MIT [4]. While resembling the general ANTS architecture, PANTS differs from ANTS (and in fact all other active network architectures) in several major ways.

A PANTS node is not static, and can be changed during the lifetime of the node. This allows us to dynamically load and change the behavior of the node itself. While ANTS has the concept of soft state, PANTS makes it more extreme by making the whole node customizable in principle.

Capsule grouping in Pants is dynamic. ANTS provide a scheme protocols which allows capsules to be grouped. In Pants all capsules are isolated and run in their own environment. However capsules can communicate with each other using soft state at nodes. By restricting access to all but the capsules they wish to group with, capsules can form dynamic groups without the need for some external authority.

The general node architecture does not depend on any language based security. ANTS uses the Java security model to verify byte-codes. The Switchware architecture [8, 14] takes the approach of using restricted semantics to keep mischievous capsules in check. This is achieved by using a strongly typed language in which compile time checks can be made. In this architecture, a proof is carried by the code, and thus guarantees (once the proof is verified) that the code can run without the need of security checks. A PANTS capsule in contrast, executes in an execution environment which initially has no services. When and if a capsule needs a service, they need to be loaded in dynamically. As this load is taking place, authentication and authorization checks can be made to see whether the capsule is allowed to load that particular service. The disadvantage of our system is that we sacrifice some performance for all run time checks that are made when capsules load in services.

Unlike the Smart Packet architecture [15], PANTS does not limit the size of a capsule. The transportation of the capsule is handled by the node OS, which hides details such as error checking and flow control from a capsule (at the link layer). We removed a size restriction in part because the PANTS implementation depends on transporting python byte-codes which are not optimized to manage space, and it is unrealistic to expect some capsules to fit into 1 packet (for example a capsule which may transport routing table entries as part of its state).
7 Conclusion

In this paper we have presented a new architecture for an active network. Our architecture allows capsules to dynamically rewrite their code, it also allows all the features of the node to be modified (Subject of course to security checks).

Our dynamic approach to active networks allows us to experiment with more interesting capsules and applications. Apart from the simplistic examples presented in this paper we have implemented more complex capsules which allow us to make bandwidth reservations and capsules which compute routing tables.

In contrast to other active network architectures, our system allows us to modify the node itself through capsules. This allows us to dynamically deploy newer and better protocols without the need to take nodes off-line. As more efficient protocols are developed, it will be important to be able to deploy and use them without the need to go through years of laborious administration, and is a viable possibility in an active network as our implementation has shown.
8 Appendix - Example Capsules

The following examples contain the full code of several test applications and capsules. They have been modified slightly from the distribution in that they do not use tk (the gui interface) to make the code clearer.

8.1 Traceroute

Traceroute in an IP network works by transmitting packets of increasing time to live (ttl) to the destination node. The route is traced by waiting for the icmp error messages received as each packet runs out of its ttl.

```python
class TraceApp:
    def __init__(self, inter):
        thread.start_new_thread(self.receive, (inter,))
        self.run-id = inter.MakeRunId()
        cap = inter.MakeCap('TraceCap', 'TraceCap.py', typeid=4,
                    self.run-id, useClass=2)

        cap.dst = 11.1.1.1
        inter.Inject(cap)

def receive(self, inter):
    while 1:
        cap = node.Receive(self, self.run-id)
        now = time.time()
        str = cap.lastlocation + 'now - cap.starttime'
        print 'Received capsule: ', str
```

Figure 11: Code for Trace Route Application
class TraceCap:
    def __init__(self, inter=None):
        self.starttime = inter.time()

    def run(self, inter):
        routeTo = inter.Load('routeTo', None)
        if inter.addr != self.dist:
            routeTo(self, self.dist)
        self.run = self.__back
        self.lastlocation = inter.addr
        routeTo(self, self.src)

    def __back(self, inter):
        if inter.addr == self.src:
            inter.Load('SendToApp', None)(self)
        else:
            inter.Load('routeTo', None)(self, self.src)

Figure 12: Code for a Trace Route Capsule

Figure 11 and figure 12 shows the code required to accomplish active Trace Route. The Application is simple, first it forks off a thread on which it listens for incoming capsules with the given run-id. It then creates and injects the capsule into the network. The call to inter.Receive() blocks until a capsule with the given run-id presents itself to be delivered.

The run-id is the unique identifier for the series of capsules that this application will generate. It is a combination of a number and the node address, which will be unique across the network. The UserClass specifies what high level user group the capsule belongs to. This is used for scheduling and bandwidth policy decisions.

The capsule works by sending itself forward towards the destination and sending itself backwards to the sending node informing it of the current node in its path. The application when it receives each backwards going capsule, prints out a message.

The above example also shows some of the power of using python. The capsule dynamically changes the run method, and passes itself back towards its originating node. But the returning capsule now has its run method bound to the method __back which is executed instead.

The capsule above could arguably be implemented using an if statement, but we kept the example simple to emphasize the flexibility. Much more complex and powerful capsules can be built using the above method, which may even involve capsules which can exchange code with each other at some intermediate node.
8.2 Random Hopping

```python
class RandApp:
    def _verb.init_verb_(self, inter):
        cap = inter.MakeCap('RandCap', 'RandCap.py', 2, 2, 1)
        node.Inject(cap, 100000)
```

Figure 13: Code for a Random Hop Application

```python
import rand

class RandCap:
    def run(self, inter):
        val = inter.Load('GetStateObj', None)('randcounter', None)
        if not val:
            msobj = node.Load('MakeStateObj', None)
            val = msobj('randcounter', None, None)
            val.count = 1
        print 'I have been here... ', val.count, 'times.'
        val.count = val.count + 1

        # route myself to a random location.
        neigh = inter.Load('Neighbours', None)()
        r = rand.rand() % len(neigh)
        print 'Going to ', neigh[r]
        inter.Load('RouteTo', None)(self, neigh[r])
```

Figure 14: Code for a Random Hop Capsule

The Code shown in Figures 13 and 14 implements a capsule which just hops from node to node in a random order. While the capsule does nothing greatly useful, it demonstrates the use of StateObjects and use of the routing table.

The Application is very simple, it merely injects one capsule and exits. The Capsule however first looks for a state object with name 'randcounter' in the current node (creating it if it does not exist), and increments this this counter. It then loads in the Neighbours service, and picks a random neighbor to visit next.

8.3 Python Extension Modules

References


ISBN 1 86487 333 7