Tradeoffs In Active Networking*

Seth Proctor {stp@cs.brown.edu}
Department of Computer Science
Brown University
Undergraduate Honors Thesis

May 6, 1999

Abstract

Active Networking is a relatively new idea in the field of networks and systems. The goal is to create a dynamic and extensible communications environment which can adapt to future needs and individual problems. While this is in keeping with recent trends, some people have overlooked many of the serious problems that arise in such a system. In particular, there are security and performance issues which could make Active Networks far too impractical ever to deploy in the real world. In addition to this, there are serious questions about whether such an environment actually is useful or even has any advantages over existing networks. This paper will discuss some of these issues, and then present a new system geared specifically toward these problems. In addition, several novel protocols will be explained which show some uses of an Active Networking environment.

1 Introduction

When the Internet was first being built, it was supposed to support a small number of sites. Today, the Internet is growing at an impressive rate, encouraged especially by environments like the web. As a result, the original network technology built to support the Internet is becoming outdated and inadequate. This has prompted many groups to start thinking about new approaches and new technologies for the "next generation Internet."

Some of these new technologies are targeted at routing and the hardware involved, while others are purely protocols and higher level software, aimed at improving services, security and convenience. ATM, for example, is a new routing technology that is aimed both at providing faster backbone switching and better QoS options at the hardware level. IPv6, the next version of the Internet Protocol, is a complete redesign of IPv4 built to meet the new needs of today's Internet. The protocol update provides a drastically larger address space, support for Multicasting, security extensions and hooks for future additions. This kind of flexibility is one of the major motivators for Active Networks.

With so much effort going into rethinking network technology, a recent new area of research has opened: Active Networking. Active Networking provides a more flexible and dynamic network environment by making the network elements (routers, switches, etc) programmable. In addition to programmablilty, caching schemes and other storage mechanisms are also introduced to allow faster data retrieval. The goal is to provide a system which can adapt to future needs, instead of one which needs to be rebuilt periodically.

*This represents a year long project under the advisement of Tom Doeppner, twd@cs.brown.edu

Such a system also exports functionality unavailable in current networks, and allows programmers to build new classes of small, lightweight protocols.

Along with extreme flexibility, however, comes some serious potential problems. Such a system can cause significant slowdowns in networks which are now being rebuilt to move data at faster rates. In addition to performance questions, Active Networking opens up myriad security problems not present in current networks. On top of these problems are questions about domains of control and administration of resources which aren't shared in current network environments.

This said, Active Networks have the potential to provide faster, safer and more flexible environments. Caching mechanisms allow for network-internal storage devices, creating smaller network paths and fewer single point bottlenecks. These caches also provide the framework for global shared object systems. In addition to storing data, new secure protocols can be plugged into pieces of networks where appropriate, and different versions of protocols can live in different domains providing varying levels of security as needed. Routers can work together towards providing better congestion detection and rerouting. Different protocols can work together inside of the network to provide better classes of service to applications at the edges. Finally, if nothing else, active environments make excellent test beds for new protocols and network algorithms.

Given these factors, the issue becomes one of tradeoffs. Is the general performance hit of such a system won back by the performance gains that can be made by intelligent caching, resource management and flow control? Are the security problems hopeless, or are there some solutions that make an active environment plausible? Perhaps most important is the question of need. Is there really any need for such a system, or are we just seeing some new features which should be "hard coded" into tomorrow's networks?

This paper talks about the tradeoffs seen in most active environments, citing specific implementations. Careful attention is paid to security and performance questions. The issue of what these systems can do that others cannot is also considered. Given these factors, a new system, PANTS, is introduced. It is an extremely lightweight Active Networking environment which includes its own language and a set of protocols which demonstrate its power, flexibility and simplicity. Written from scratch with no existing source base, it builds on ideas in current systems and provides new features and flexibility. Finally, a look at its performance helps to answer some questions about the realism of deploying active environments into live networks.

2 Active Networking

Before discussing the tradeoffs of an Active Network, it's important to understand what exactly such an environment is, and to understand what kinds of approaches have been taken in building them. There aren't many active environments out there, and there are no commercial systems, but even in the few systems that do exist there are some interesting and diverse approaches being taken.

2.1 Overview of Active Networks

The basic concept behind Active Networking is fairly straightforward. In today's environment, network elements know a certain number of standard protocols. They probably all speak IP, some routing protocol (often RIP or OSPF) and depending on what they're supporting, they may know other protocols built for routing, group management, data delivery, security, etc. Whatever set of protocols they do know, however, this is all they know. A network programmer cannot teach them new protocols, and can't use them as a node on which to run processes. They are a static, isolated piece of the network.

In an Active Network, all this changes. The essential idea is that a packet can carry not only data, but also code. This code can be loaded onto the interim components of the network and makes the previously static environment dynamic and customizable. In addition to this change, the routers and switches now become shared resources. Their processors, memory and network capabilities are now available for the programmer to use.

This central idea, programmability of network elements, is the common thread that ties all Active Networking systems together. That said, there are numerous approaches being taken towards supporting active environments.

2.2 Existing Active Networks

Most if not all of the well known Active Networking systems are still very much research projects. They are being built as parts of larger projects, or as test beds for future experiments. Many of the current systems were built to address specific issues related to caching, congestion control or code mobility. Regardless of their original purpose, however, there are now projects underway to start merging their functionality into standard environments, producing Active Networks with a wide range of features.

At MIT, the ANTS project [20, 19] is one example of an ongoing Active Networking research group. Originally an in depth look at adding activation flags to IP, ANTS has become a full simulation environment. Written entirely in Java, it provides a simple API for protocols to use and a simple packet format for moving data through the network. The idea of a "capsule," which encapsulates the active code and data in a normal network (IP) packet, allows for non-active nodes and active nodes to exist in the same environment. All that a custom protocol really does is define new forwarding/routing methods. Without this redefinition, standard IP forwarding and routing is used.

In addition to making simple routing choices, the active protocols have a second important role. As an example, they give a demonstration online auction service. The auction server can send a "price filter" to routers handling incoming client bids. This filter gets put in a local cache, and then bids coming through can be rejected without actually going all the way to the server, thus lowering congestion. The emphasis in this system is not on writing large dynamic protocols, but small, simple routines which can enhance performance and routing.

Another research network, the Switchware project [6, 5] at UPenn is approaching the problem from a slightly different angle. Again, they use a verifiable and strongly typed language, but the language of choice is a custom one built for the Switchware system (Programming Language for Active Networks). In addition to this, Caml is used both to support some of the more demanding protocols and to drive the underlying operating environment. The protocols are delivered in a similar capsule packet, and the protocols are used only with the data in the capsule. Small and simple, the protocols are meant to help guide the packet to an end destination.

In addition to being simple, the PLAN protocols are strongly typed and guaranteed to terminate. This is just one of the security features in the system. In addition to code safety, there are performance and communications protections too. Protocols are not allowed to install cached data or keep state between runs, and they are given little time/space to complete their execution. Domains of nodes may limit who can install protocols, or what kinds of features can be used. Optional authentication and encryption between nodes allows for a better security model than in systems which base all security simply on sandboxing or bytecode verification (which is also done in PLAN).

A simpler system is BBN's Smartpackets [4]. Again, they evaluated existing languages and decided to write their own for this project. Their goal, however, was not security of the language but compactness. They wanted to be able to provide bytecode representation that would fit within a single packet. They also use the notion of a capsule, but in addition to this they are using the IPv6 routing options to embed support for their system directly into IPv6 implementations.

Unlike the Switchware packets, BBN's Smartpackets are supposed to keep a fair amount of state between runs. To this end, the smartpackets system takes advantage of SNMP MIBs. All (or almost all) persistent data is stored in a series of these, and is accessed through a simple authentication scheme. They also limit the users who are actually allowed to inject protocols, requiring authentication before the protocol will be accepted. Given this model, the system is meant to be dynamic only to a few protocols and programmers who are running in the environment. The example application they suggest (and are currently working on) is an extension to RSVP to provide dynamic QoS support.

Not quite an Active Network, but a mobile and very active environment, the Liquid Software system [8] from the University of Arizona is a larger project that encompasses Active Networking as a subsystem. The idea behind liquid software is that it's "code that flows easily from one environment to another." In other words, mobile code. They use Java as the language to provide code mobility, and use a customized JIT to provide optimized performance. The operating environment they have built works by connecting streams and small chunks of mobile code together to provide more complicated processing.

The operating system itself is actually a user OS, complete with APIs, Windowing environment and filesystems. Underneath the operating system are several small "liquid" protocols chained together by input and output streams. The Active Networks that the group has proposed and modeled build on these relationships. Packets may contain small new java protocols which can chain into these "module graphs," thus providing new support, changing the way in which data is processed or updating old protocols. The system supports extensive caching and standard protocols (protocols such as IP, TCP, DNS, ARP, etc are supported as liquid pieces in the OS' module graph) which can be used by incoming data.

2.3 Different Approaches

The 4 different systems discussed above are the best known Active networking projects, and while there are others underway, most approaches being taken are covered by the listed systems. These approaches can be broken down into a few major ideas:

- **Activate the nodes.** Clearly the underlying feature in all Active Networks is that nodes are programmable. All of the systems discussed above employ some mechanism for providing a generic "execution environment" on all or some of the nodes in the network. Not all the nodes need be active, but at least some need the ability to accept incoming messages, filter code and data and execute protocols in a platform independent manner.

- **Encapsulate data.** The data and code being passed on the network need to be represented in one form or another. Especially in a network which contains inactive nodes, the packets need some flags or signals about how to process the contents. Most systems employ some kind of encapsulation method (replacing the term packet with capsule) which allows active traffic to flow through arbitrary networks and be processed correctly. This can be used to name a preferred protocol and protocols to fall back on if the desired one is unavailable (for example adaptive IP routing or simple IP forwarding).

- **Provide caching.** Most of these systems provide some caching mechanism. It can be at the nodes or on an external host, and the cache can be local to an instance of a protocol, local to all instances of a protocol or globally available to everything running through this node. Some caches provide long term storage for protocol state or protocol communications, while others are short term or volatile. Whether for temporary storage or global use, however, some kind of caching scheme is employed.

- **Interoperability.** Because of the dynamic nature of these systems, interoperability is heavily stressed. This can be in the form of different protocols being able to communicate, creating clear paths of data flow, interconnecting different active systems or creating generic mechanisms for network communications (as discussed above). The ability to work with existing systems, other active environments and different protocols is clearly desirable.

- **Language Safety.** Some systems have clear security designed into the communications and protocol installation processes, while others allow arbitrary communications and management. The common thread in security issues is the language being used [14] . Some kind of strongly typed, verifiable language needs to be employed to provide some base guarantees about the actions being taken. Without this, higher level security really is pointless.

- **Simplicity.** These are distributed systems, living in pieces across many nodes in a network. They are communicating using different protocols, and shipping small pieces of code around constantly. Keeping them as simple and lightweight as possible is a clear goal. This way, it's easy to move them to new environments, and the emphasis can be placed on the protocols being written.

While there is no one system that people are starting to use, or common language, feature set or caching strategy that is dominant, there are efforts underway to help connect these different systems and provide some common method for interaction. Different systems are being designed for different tasks, but there's still the desire to keep networks communicating with each other. To this end, an "ABone" [2] (backbone for Active Networking) has been established. It's fairly small, and has limited functionality, but it does define a common capsule format which supports using several systems such as ANTS and Switchware on the same network. Hopefully, as more work is done in this field, standard methods for connecting these systems will be formed and we will start to see more than just academic interest in future development.

3 Benefits of Active Networking

Given this background about some Active Networking projects in progress, the question is one of which features are really important or useful, and which are extraneous or even problematic. In other words, which are the real selling points? Clearly all the features described above aren't needed in all systems, and some feature choices are highly dependent on the target project. In general, however, there are some features which are desirable across all systems. These break down into a few major aspects of Active Networking which are particularly attractive.

3.1 Dynamic Environment

Certainly the most important feature of these systems is their dynamic nature. This means a few important things for an active system. Primarily, this means that new protocols can be introduced to a network, and old protocols can be updated dynamically. From a network administration viewpoint, this is a huge win. Instead of having to schedule down times to upload new firmware or replace entire boxes (potentially causing longer delays than expected), new code can be added without disrupting network usage. If the protocols have some kind of intelligent versioning system, when a new protocol is added the old version can be left on the network, thus allowing packets that used previous features and those that use new features to interoperate (for example, moving from IPv4 to IPv6).

This ability to seamlessly add new protocols also means that new protocols can be integrated quickly. A serious problem today with routing technology is the lack of modern routers with recent protocols to drive data flow. Multicasting, for example, is something that only up to date environments can take advantage of, since the code that drives it (IGMP, for example) is not available in older switches and routers. Only recently has multicast-enabled routing hardware become standard. This delay is in large part due to the complexity of adding new protocols into a network (which often involves buying new hardware). In an Active Network, new protocols could be added at any point, allowing for the widespread use of new technology, ease of updating, lower costs and better support for older protocols living side by side with new ones.

In addition to being able to add new protocols easily and quickly, Active Networks provide an excellent test bed environment for new code under development, and research projects on any area of networking. Perhaps this is the real value of an Active Networking system. The need to buy expensive test hardware would lessen if initial runs could be made in a localized, private active system. Especially in academic environments and labs, this would mean that interesting new algorithms, protocol ideas and system simulators could be built simply by writing some active protocols and pushing them into the test network. This certainly seems attractive as a first pass towards testing new network ideas, and has the added benefit that the protocols, running in a real network, could be exercised by everyone during standard day to day activities.

Finally, this dynamic nature means that resources can easily change as needed. For example, a router could act as a firewalling device at times and a public gateway at others. This wouldn't require rebooting, adding new protocols or reprogramming. A single packet could make the change. This change could even happen dynamically, as in [18, 10], so that a router that allows all traffic into a site, at the first suspicious packet it sees, could become a firewall with varying degrees of security. This takes the burden off a network administrator to watch the traffic flow through a site, and allows the systems themselves to monitor and protect the site.

3.2 Shared Resources

Another major change between passive and Active Networks is the way in which the routers and switches are viewed from the edges of the network. As was already discussed, in today's passive environments, the routers are black boxes somewhere in the network. They can be seen, but nothing can be done with them, or when a packet reaches them. In an Active Network this obviously changes.

When the router becomes a shared resource, new possibilities open up to the programmer (this can cause problems too–see the section on security for more). By making the routers a shared resource, caches can be put inside the network, thus allowing for network-internal caching of flow data and (hopefully) improved speed and congestion. For example, consider an extremely popular web site. It is likely to have some static data, and many people hitting it from diverse locations. When a request arrives, the data is sent back along the full path in the network. If another client somewhere along this path wants to get the same page, they have to travel along the length of the path to the server, get the page and bring it back. In addition to contention for network bandwidth, there will be a serious bottleneck at the server.

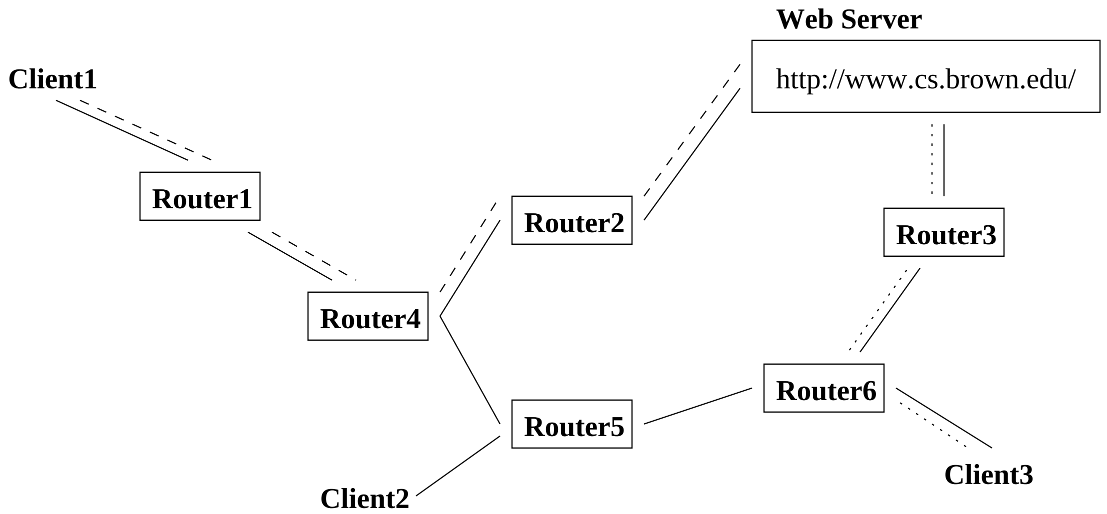

Figure 1: *Nodes 1 and 3 have both requested the page, so no matter which path Node 2 chooses, it will only take 1 hop to find the data.*

Now, suppose that as the data was returning to the first client, the page was cached along the network path. When the second client went to get the data, it would soon encounter a cached copy, and could return quickly with the desired data. This provides a serious speedup for the client, cuts down on overall network usage and bandwidth demands and removes the bottleneck from the server, which is now free to serve new pages much faster. This propagation would quickly spread cached copies across the network, to the point where data could be found locally almost anywhere. Periodic updates from the server, either by invalidating cached copies or pushing new cached versions into the network takes care of the problem that the data might well be dynamic. While this scheme isn't likely to be too efficient if every web page is included, it could certainly use some volume metric to choose popular pages, and cache them as long as they remain popular.

In addition to caching web data (or auction data, as in the earlier example), this caching system can be extended to support systems like shared objects. By creating a named object in a cache, a protocol has effectively created a shared object which other resources in the network can use. Within a local network, it would make sense to distribute some data about the environment, and perhaps even provide shared views of resources or more intelligent MIBs (as in SNMP). Within high speed, low traffic networks, it might even make sense to setup real shared object mechanisms, which could be used as communications mechanisms between programs, or hints about how to most efficiently communicate with other systems on the network.

Finally, this shared resource concept means that the router's processing power is turned into a shared cpu. This could be used, as in the previous DSM example, to get more cycles into a simulation. It can also mean that some simple calculations, stream choices, protocol filtering or other network tasks normally done at the edges of the network can now be done internal to the network. Some choices, such as how to setup paths or how to get accurate delay and traversal times can be difficult to do outside the routing elements of a network. Being allowed to calculate these things inside a network could allow for much more efficient protocol setup.

3.3 Network Internal

Being inside the network doesn't just mean that better performance numbers and more accurate flow choices can be discovered. It also means that agents can monitor what's going on much more closely. This fact can be used for many different gains. One that was already mentioned is the ability to switch purposes and protocols on potential security warning signs. In addition, a fair amount of work is already being done to understand what kind of flow control and congestion avoidance [13] can be done in a dynamic network. Having the ability to watch congestion and latency rates, even on specific streams or interfaces, internal to the network could help to create new control schemes to speedup communications and better use existing resources.

Being inside the network would also allow for better performance and statics data gathering. Protocols could be built simply to exchange information and watch what the network is doing. This information can be used to understand what causes problems and which pieces of a network may need updating or redesigning. Protocols could also be built which are designed specifically to limit and control how shared resources are leased to data passing through the network. Protocols like RSVP, which is slow to setup and can't easily adapt to network path changes could be replaced with a small set of dynamic protocols which are watching the network, what resources are available, and are able to continuously move data through the network at an optimal rate. These kinds of protocols can't be built today because of the static nature of our networks. The ability to run within the network opens a new class of protocols.

3.4 Generic Interfacing

One final feature that these systems provide is the ability to create generic interfaces to networked devices. This kind of functionality is provided today by systems like SNMP. It requires writing extensive client and device support, however, and doesn't naturally communicate with other devices on the network. Sun's Jini System [15] attempts to deal with this by giving each device on the network a simple standard interface, and then allowing other devices and a global registry the ability to query the object. The object can in turn send a small piece of bytecode which yields information about the device, sets up drivers, etc. (note that this effectively makes Jini an Active networking environment). This allows all devices on a network to communicate with all other devices, share information and customize features based on the network configurations.

Instead of setting up a custom environment such as this, however, an Active Network would provide all of this for free. A simple protocol could define an interface to generic devices, and then other devices could send queries, set values and use this information to configure their own settings. In this way, the devices on the network could take advantage of simple exchanges of data to configure complex settings. Such a system would greatly simplify the special purpose software that exists today for dealing with these problems.

4 Problems with Active Networking

It's clear that there are some good reasons for exploring what an Active network can do. The flexibility that it provides allows for applications and control which doesn't currently exist. The above examples, however, don't discuss any tradeoffs. There are also plenty of problems that they introduce into the network. In order to understand whether these tradeoffs are worthwhile, and to get a sense of which of the above are really valuable and which are simply too much trouble, it's important to understand where the potential problems are, and what kinds of problems may be opened up by deploying an Active Network.

4.1 Security

The most obvious problem with these systems are the security problems involved. True, today's networks are static, which means that there is little ability to customize the way in which the network operates. It also means that attacks on routers are minimized. Simple DoS and more complicated exploits in buggy code are still very possible today, but that's the extent of what can be done. In an Active Networking environment, a whole new class of attacks and security holes are opened.

Perhaps the most fundamental question becomes one of who can execute code where. In an environment where remote programs can be loaded onto arbitrary machines, how do you limit what actions can be taken? There is the option of adding signatures or even encryption to all packets containing code, and then providing some set of users who are allowed to put code onto a device. This requires a fair amount of setup, however, and also increases the processing overhead for incoming packets. It also may not provide a flexible enough solution, since a network may need varying levels of permissions, and this may need to change based on network need. In a strict system all ability to upload code could be limited to console-only access, or something equally paranoid, but this would defeat the purpose of an Active Network.

In addition to who is allowed to send code to the routers, there is a secondary problem of who is allowed to execute the code. It may be that some routines are used for maintenance, device configuration (as described above) or or administration. These routines and services shouldn't be available to any user. In a typical administrative system today, you might log in, supplying account name and password, before you are allowed to access or change certain information. In an active system, however, this notion of being logged into a service may not exist. A protocol may well need to differentiate between authorized and unauthorized users, but this may require a high overhead in terms of space and time for every packet (authentication) or some more clever internal scheme.

There are also regional considerations to be made. Suppose a packet needs a certain custom protocol to travel between two nodes. Part of the path is through locally administered routers which contain the protocol in question, but part is through a backbone which is run by a different group. The owners of the backbone won't give you permission to run custom protocols, so is the packet never received? This is a specific case, but in general the issue of domain control could fragment Active networks into small subregions, each locally run and each containing their own protocols which no one else is running. The politics of data flow are already a complicated issue without this. Perhaps protocols could be written to tackle this problem, but then they would have to be running throughout the network, which comes back to the same problem.

Given that the code has somehow been allowed onto the system, the security problems get more interesting. As was discussed earlier, most systems use some strongly typed, verifiable language to protect the systems that are running the protocols. This is to provide some guarantees about what the protocols are doing, and to make sure that the code can't crash the execution environment or the node itself. But how much is a protocol allowed to do? Depending on what's provided, a protocol may not have enough to do what it needs, or it may have too much, and be able to wreck havoc on the host. Again, the choice of language, language features and APIs is very important. In addition, the system that is running the code has to be solid. Assuming that it's possible for malicious code to load itself onto the host, that piece of code can try to exploit holes or break the VM. The code that's executing the protocol must be able to handle any such attack. This brings back the point of simplicity. The simpler a system, the fewer unknown security holes are likely to be lurking in the code.

Finally, there is the issue of resource consumption. Even if a protocol seems to be allowed to run on a node, and even if the language guarantees that the protocol can't do anything too damaging to the system, there's still the problem of protecting the resources on the host node. A protocol could drop into a tight loop for ever, start using obscene amounts of memory, fill local caches with bogus data and flood the interfaces going out. How does the host operating environment make sure that the node is protected from actions like this? To an extent, careful scheduling and memory pool allocation can protect the processor and memory.

Limitations on sending messages and caching data could protect the rest, but then limitations start getting placed on what a protocol can do, and this can severely limit its potential productivity. All of these security problems pose some serious threats to the feasibility of an Active Networking system.

4.2 Performance

As was just discussed, a malicious piece of code could drop into a tight loop, monopolizing the processor and hindering the system's ability to work for other protocols and incoming or outgoing data. But this isn't just a problem with evil code. innocent code could easily do the same thing. A careless programmer could create a protocol which is extremely resource hungry or an efficient implementation could simply be solving a very intensive problem. Add to this the layers of filtering and processing, and the fact that the protocols are being run through some sort of virtual machine, and it quickly becomes clear that even if the protocols aren't taking up too many resources, the potential for performance degradation is huge[1].

As an initial hit for a protocol and the system, there is the time that's taken to load the protocol. This probably requires not only loading the protocol code into memory, but also verifying it, checking certain security requirements and setting up any data structures the execution environment needs to execute the protocol. Verification is a potentially very expensive process, and even if that's the only significant performance hit happening when a protocol is loaded, it's probably enough to slow down the system and cause momentary performance loss.

Once the protocol has been loaded, there's now the time to process incoming data. Again, there may be security restrictions on the data, and assuming that it maps into a known protocol, structures may need to be built and data verified before the code can start executing with the given data. After the data has been loaded, it needs to get to the given protocol. This is probably a negligible cost, but depending on the security built into the system, this could still be expensive.

After factoring in the cost of setting everything up to run, there's the problem of running the code. Because the code is platform independent, it must be run through some kind of virtual machine or translator. The Liquid Software project uses a modified Jit and additional translators to build roughly native code that's optimized and should perform well, but not all systems will be able to perform this action. Running unoptimized code through a VM can incur a serious performance hit on every instruction. Even given a JIT of some kind there's still a hit on every instruction, and using a JIT degrades the security available in the system (the VM can enforce types, certain access restrictions, etc) so it may not always be an option. Regardless, executing the protocol's code is going to be slower than running data through native router code.

Finally, one of the huge performance problems will be in the caching requests. As already discussed, one of the common threads through all these systems is some notion of caching. This ranges from very limited volatile entries to long term, global data. This could include caching web pages, auction data, stock quotes, etc. Allowing multiple protocols to cache data on this scale will require a large amount of space to store the data. The number of protocols that will then take advantage of these caches, potentially for every packet that comes through the system (as in the case of web data), could produce a huge processing overhead. The routers and switches that drive our network need to be as fast as possible, typically taking extremely small amounts of time to read in a packet, processes it, and send it back out. A system which slows this process down for all packets could potentially cause the whole network to slow down, especially as the caches get larger. While the hope is to improve performance by minimizing network traversal, these caches will certainly generate a huge overhead in packet processing.

Given all of the previously suggested benefits of an Active Network and now these problems with an active environment, combining these with the ideas generated by other systems, what can be learned and applied toward a new system? What features should this new system have, and what are the motivating

[1] refer to the *Observations* section for a better sense of what these degradations look like

factors behind this design? In the next section, a new system is presented that draws on all that's been discussed to this point, and attempts to provide some novel features.

5 PANTS

Shifting focus, a new active system will now be discussed. PANTS[2] (Programmable, Active Network Toolkit and Simulator) was written by the author of this paper after considering the already discussed advantages and problems inherent in Active Networks. I wrote this system from scratch, building on ideas and features from other systems and adding some new ideas from areas in networking and distributed computing.

PANTS provides most of the basic features that have been discussed so far, including local and global caching at all nodes, various levels of security, a generic execution environment and the ability to interoperate passive and active networks through generic encapsulation methods. The basic idea is still to provide a dynamic network environment through programmable nodes. The entire codebase is written in standard C, and is therefore highly portable (it has also been carefully kept quite small, ~9,000 lines of C source code). It currently is being simulated on top of a host operating system using a network/router simulation package, but requires little of the host OS, and has been designed to easily port to native routing environments. The PANTS system provides several important features:

- A custom language and compiler to generate bytecode (~1,000 lines C code)
- A virtual machine which executes the custom bytecode (~2,700 lines C code)
- An operating system which embeds the VM (~3,500 lines C code)
- A simulation environment for testing and debugging protocols (~1,400 lines C code)
- Custom protocols for adaptive routing, multicasting, directory services, mobile computing and network management (see the protocols section)

5.1 Project Goals

The project, started in October '98, had three major goals. The first major goal was to understand what pieces of an Active Network, if any, really make sense. This initially meant considering whether or not Active Networks themselves actually seem useful. Does an Active Network really provide something new and powerful, or would it be better simply to adapt routers to provide new features? After careful consideration of some of the benefits of an Active Network, it seems that some applications (adaptive routing for congestion control on individual streams, for example) really do benefit from an active system. Simple caching, protocols which rely on dynamic data and basic adaptive protocols simply don't provide enough flexibility to create some of the desired functionality.

A second major goal of this project was to understand, assuming that Active Networks provide a valuable environment, what kinds of protocols should be built to take advantage of these new systems. Some of the standard examples include online auctions, web caching and stock data (the gold standard of networking demo systems). But each of these relies on little more than a cache and a single filter through which incoming data is passed. For the most part, these applications could be facilitated by adding some disks to routers (although an Active Network provides a much better environment for implementing flexible versions of these systems). To really sell an Active Networking system, there need to be some applications that are unique

[2] The C source code, protocols, and other supplemental information can be found at *http://www.cs.brown.edu/people/stp/an/*

and really useful. In other words, an important goal of this project was to think about what new kinds of applications can be built given this flexibility, and what these protocols offer.

Finally, the last goal of the project was to understand what the performance problems really are. There has already, in this paper, been a lengthy discussion on potential slowdowns, but without actually building a codebase and testing some real world scenarios, it's hard to understand where the real issues are. Most other papers don't discuss this point very much, writing it off as something that would be addressed were they building something other than a research prototype, or claiming that the performance increase due to caching (or some other feature) outweighs any potential performance hit incurred by the system. This doesn't answer any questions about performance, however, and most claims about caching or path adaption aren't backed up by numbers or test results. Therefore, a primary goal of this project was to understand what the real performance problems are, and whether this limits the usefulness of an Active Network, or the applications that should be built for it.

5.2 Goals of the System

Along with general project goals, some clear goals emerged for the system itself. The first of these was the need to make the supporting code as small and simple as possible. In order to provide a network-oriented environment, which is by definition highly dynamic and therefore highly susceptible to attacks, it needs to be as tight and bug free as possible. A simpler system will have less overhead, and this will improve performance. In addition to a simple system, it needs to be highly portable. In order to provide an Active Network, the support OS needs to run on the internal routers. These could consist of very different hardware, and therefore the code that's going to run on each of these needs to be transportable.

The second system goal was to make as much of the environment as fast as possible. Clearly this is an important goal in all projects, but this is an especially important goal for this one. Since one of the major points of the project is to understand and critique the performance problems and speedups in Active Networks, it's important to strive for a fast implementation, which can help to define an upper bound and provide some insight into the end limitations of these systems. The issue of speed is addressed by both caching strategies and the actual processing time. Coordinating the execution time and security overhead is a key component of this.

Third and finally, the system needed to make certain security provisions. Clearly, without a base amount of security, no one would even consider running an inherently dangerous system like this in a real world environment. This baseline of security comes from a combination of the language and the execution environment. The language needed to be something with strong types that can be compiled into a compact form that is quickly verifiable, and the environment executing the code needs the ability to watch all running protocols, protecting system resources and the host itself. Beyond this, additional security considerations can be made to protect data, limit who can install code or share information outside domains. These types of features, however, are secondary to the need for base security. Once the support system is safe, other features can be layered.

5.3 Overview of the System

Given these system requirements, the project evolved in the form of three different pieces, each of which works together to provide the complete system. They are a custom language, a virtual machine and an operating system. The operating system takes care of initializing caches, handling incoming and outgoing network traffic, verifying and storing code and data, scheduling protocol time and managing memory usage. The OS essentially has one input format, which is that it accepts incoming data from a network. This data is classified as either input data to an existing protocol, a new protocol to install or internal data from some other node on the network (ie, data generated by another OS). The OS also provides all resources to the running protocols and subsystems. This includes memory allocation, process creation and destruction, switching running threads on the cpu and a number of other related features. Embedded in the OS is the virtual machine. The VM provides structures to the embedding application for loading protocols, initializing protocol instances, and stepping through executions. The VM executes code that has been written in a custom language, and then built using the provided compilers. The language looks similar to the Java instruction set, but has some subtle and important differences in terms of the functionality that's provided, and the bytecode that is generated.

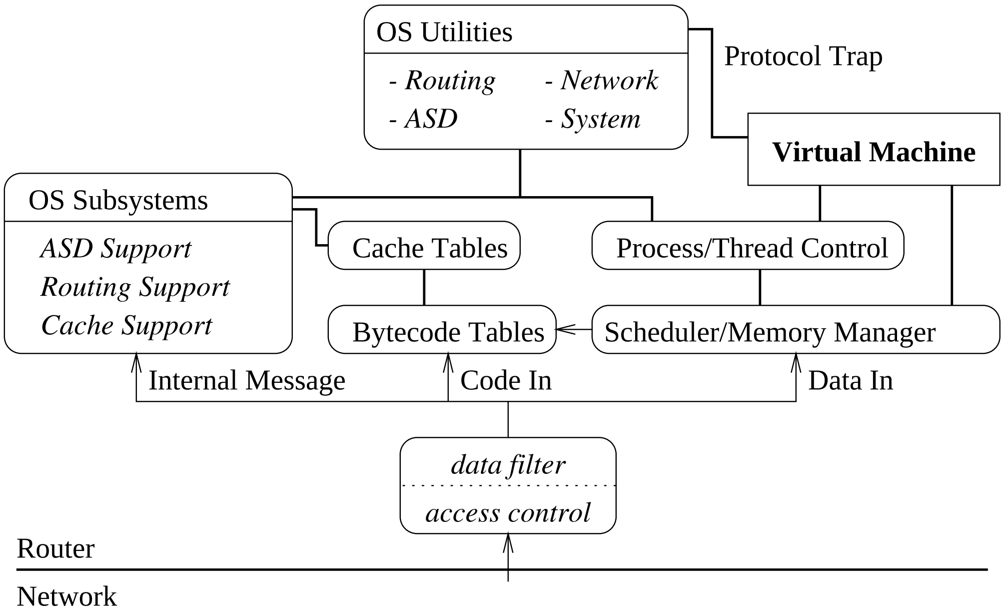

Figure 2: *Overview of the Operating System*

The protocols themselves are all named. That is, they have some unique 64-bit value associated with them. This acts both as a way of identifying a protocol to use, and as a security check to make sure that malicious versions of well known protocols aren't introduced into the network. A typical packet contains a flag, specifying whether code, data, or system data (data initiated by this or other execution environments, as opposed to a client using some protocol) is being carried in the capsule, the protocol identifier, and either bytecode or input data. This information is processed, resulting in code being loaded and stored in the OS, or data being moved to the appropriate protocol or OS subsystem. Once a protocol has been installed, it can be used by any data that flows through that node.

In addition to the instruction set provided by the VM, protocol bytecode can take advantage of other features provided by the operating system through a trap instruction. One such set of features is provided by the caching subsystem. This can be used to store and share data in a number of ways. In addition to simply storing and loading variable data during a run, protocols can use the cache to provide data to all instances of that protocol. For example, a routing algorithm could start constructing tables which can be seen and modified only by that algorithm, but these tables would be accessible from any running instance of that protocol. The caching system also provides space to store global data. This is data that is accessible by all protocols on the host. This is one way to communicate large amounts of data between protocols. For instance, in the web caching example, one protocol might be using the global cache to store the web pages, while a second protocol could be responsible for deciding what data should be kept, and what data should be expunged.

Along with the caching subsystem, another system that's available to the code is a simple set of routing protocols. Built into the system, this routing code keeps track of simple vector-based tables for reasonable routing paths. If a developer doesn't want to worry about writing a set of routing routines, or if they don't want to rely on a certain custom routing protocol existing at each node in a network, they can always use this subsystem to resolve paths.

One final and important subsystem is an implementation of the Arrowed Distributed Directory Protocol [3]. This unique directory service is geared towards providing lightweight indexing for shared object systems. The basic idea is that when an object is created, every node in the system learns about the object, and constructs an "arrow" pointing to the location of the object. This arrow is analogous to an entry in a RIP routing table. Given a certain node and an object to find, there is an arrow at that node which doesn't tell where the object lives, but where to go next to look for it. In this manner, a request to find a named object in the group is facilitated by following the arrows from the start node to the node which has the object. For a request to lock the object (ie, find the object and bring it back to the originating node so the data can be rewritten), again the path of arrows is followed, but they are all switched so that they now point back to the site which initiated the locking request. If another node also wants to lock the same object, its request will eventually follow the new path back to the node which already requested a lock on the object, where it will stay queued until the object is released. In this manner, objects can be found quickly, and when multiple requests are made to lock the same object, the requests are queued up in a defined manner. Support for creating, joining and leaving arrowed groups, creating objects, locking and unlocking objects and reading and writing these objects is provided by the operating system. This creates some interesting features unavailable in other systems, and as explained in the protocols section, this provides for some interesting new applications.

5.4 Further Technical Information

The first major hurdle to building a system like this is figuring out where it's going to run. Ideally, the OS and VM should be running natively on a set of routers, which are attached to a live network. Unfortunately, there were no routers available for this project, so in order to test any code, a simulation environment was needed. The simulator written for PANTS provides a simple router-like environment. It allows for multiple interfaces, and there are operations to write to and read from these interfaces. Latency and packet dropping/garbling are provided as tunable parameters on all links. A single configuration file also allows for the network topology to be defined on startup. This base functionality allows for higher level code to be written on top of the same type of system software running on a router sees. In this way, the PANTS OS is already well suited for a router-based environment. With a test environment built, the OS could be targeted to run in this simulation environment, and clients were connected to this testbed environment to test and tune protocols and the OS itself. Given a well defined environment for testing and simulating, the effort turned towards the language issues and the VM which is executing the language bytecode.

Since much of the base security and some of the performance questions come from the language and how it's executed, it's important to understand how it works. First, however, it's important to understand why a custom language instead of something well known like Java was chosen. Primarily, the JVM doesn't give enough control to the application using it, in this case the operating system. In order to make certain QoS claims, and provide resource security, the time that each protocol gets for execution, and the amount of memory each protocol instance is allowed to use needs to be monitored and carefully guarded. There is no convenient way to do this, so in order to get this kind of functionality from a JVM, the source would have to be hacked up to support these features. Also, many of the features in Java far exceed the functionality needed in an Active Networking system. The object oriented nature of Java code, and the bulk inherent in using support classes makes that environment a little too heavyweight. Finally, since most routers probably don't already support some JVM, custom code bases need to be written regardless. If this is the case, it makes more sense to build a new, small VM with features specifically designed to work with Active Networks then to hack around existing source bases which are geared toward significantly different systems.

These specific features are tailored to provide the set of functionality that a protocol in this environment needs, without providing too complex an execution environment. Many of the instructions are similar to those in the Java VM[3]. Instructions that deal with memory (load/store), math, comparisons and generally different data types are prefaced by the data type expected. For example, there is a different addition call for integers, floats, doubles and longs. This provides some fundamental guarantees about the execution, and allows the host to verify and validate portions of the code before running it. Also, dynamic data is provided through references. This means that nothing like pointer arithmetic is allowed, addresses are known to be valid, and null addresses can't be dereferenced. This strong typing is the basis for the security provided in the system. In addition to this, the VM is stack based to provide slightly better performance for some operations. There are still, however, special purpose registers for temporary storage of variables, as well as larger caching options for storing more values. The VM is written as a simple library, so that any application can embed it to build test environment or custom programs which will work with existing systems through this common code base.

```
; load a structure refernece we had ealier
ALOAD_0
; load the struct's second field, an integer
GETFIELD        1
; increment the value by 32
LDC             32
IADD
; pass this as input to a function...return replaces this
; on the stack
JSR             func
; send this value out on the interface listed in
; the third field of the original structure
ALOAD_0
GETFIELD        2
LDC             TR_SEND_INTERFACE
TRAP
; finish off the routine
RET
```

Figure 3: *An example protocol*

[3] For a complete instruction list, as well as a technical description of the bytecode format and other VM features, see the appendices

When writing custom protocols, the language looks much like standard assembly code. Instructions are one per line, and some of them take parameters, while others operate exclusively with what's on the stack. When compiled, the bytecode is broken into two main sections. The first half contains constant data and custom structures, while the second portion contains the code itself. A programmer has the ability to define custom data structures for use throughout the protocol. These structures are also well defined, so they can be allocated dynamically, passed between protocol instances and stored into the cache for other protocols to look at or modify. In addition to using these structures as a convenience in a protocol, one of these structures can be defined as the input type. That is, when a protocol instance is being created and started up, this happened because of some input data from the network. If the programmer wants, a structure can be defined as the input type, and the incoming data will be safely mapped into the given structure. If no input structure is provided, then the data will appear as a char array. In either case, the data is the lone element in the stack when the protocol instance starts running.

5.5 PANTS Protocols

Having defined a set of goals, and having used these goals in part to build a prototype system, the next step is to produce some custom protocols to demonstrate the features of this system, and some of the power of an Active Networking environment. As has been discussed previously, the goal is not to create the same protocols that already exist elsewhere[4], but to come up with some new and creative approaches. In addition, test protocols should take advantage of most if not all the features of this system to give some real sense of the overall performance. The protocols discussed here aren't the only ones which have been written for this system (first passes for the auction filter and the web cacher have both been written), but they are particularly interesting and point out some unique features of this system.

A simple protocol written early in the testing phase of the system provides somewhat adaptive routing [12]. It starts out by using the default routing system to move data through the network. When a packet is sent to a neighboring node, the neighbor also uses the default routing system to pass the packet on. The neighbor also responds to the sending node (on every 20th packet to cut down on network congestion) that the data has been received. In this manner, the sending node can keep tabs on how fast that particular link is running. If the protocol decides that a link is moving too slowly, it can talk to its neighbors to negotiate a faster path to the destination. At this point, it will stop using the default routing for that one hop, and start using this custom path. The protocol also makes sure not to move everything onto this new path, but to spread data between a few paths. The hope is that under heavy loads better paths will quickly be discovered and traffic will thin out as it is passed along different interfaces. This protocol takes advantage of the routing and caching systems, and is only about 300 instructions in total length.

Another new application relies on the arrowed directory service, as described above. The idea of the new protocol is to use the arrow subsystem to create a group, and then use this membership group to define a multicast group which provides ordered transmissions. Most common multicast environments have a single sender which is sending data out to many receivers (ie, streaming any type of audio/video media). In these cases, ordering can be achieved by a simple sequence number, which the sender starts at zero (or some known value), and then increments with each transmission. There are other situations, however, where every multicast group member is both a sender and a receiver (ie, online conferencing). In these cases, a simple sequence number can't be used for ordering. When a message arrives from node A, it's not enough to know that this really is the next message from node A. You also need to know that no message was sent before node A sent this message. In order to achieve this, there effectively needs to be a single sequence number which the entire group uses, and is somehow logically shared amongst all the members. Providing

[4]unfortunately, I couldn't resist writing the stock ticker protocol, since it's such an important feature of any networked system these days :)

this functionality in current multicast systems requires a transport level and some complex acknowledgment schemes[5].

Providing ordering for multiple sender multicast in the PANTS environment, however, can be done quite easily using the arrowed directory system. When a node first creates a group, it also creates a single object to share among the group. This object is the write token. A node will only be allowed to send a message to the group if it has the token. The value of the token will start at zero, and will increment by one with each message (ie, the token is the shared sequence number, as well as write lock). This value gets included in each message sent to the group as the sequence number, so that every receiver can look at the last sequencing number, and decide whether or not this is really the next message from the group. This is extremely lightweight, adding almost no time to the normal process of multicasting, and gaining ordering (almost) for free. The entire protocol is implemented in just under 120 instructions. This includes functionality for creating and joining a group, sending messages to the group and leaving the group.

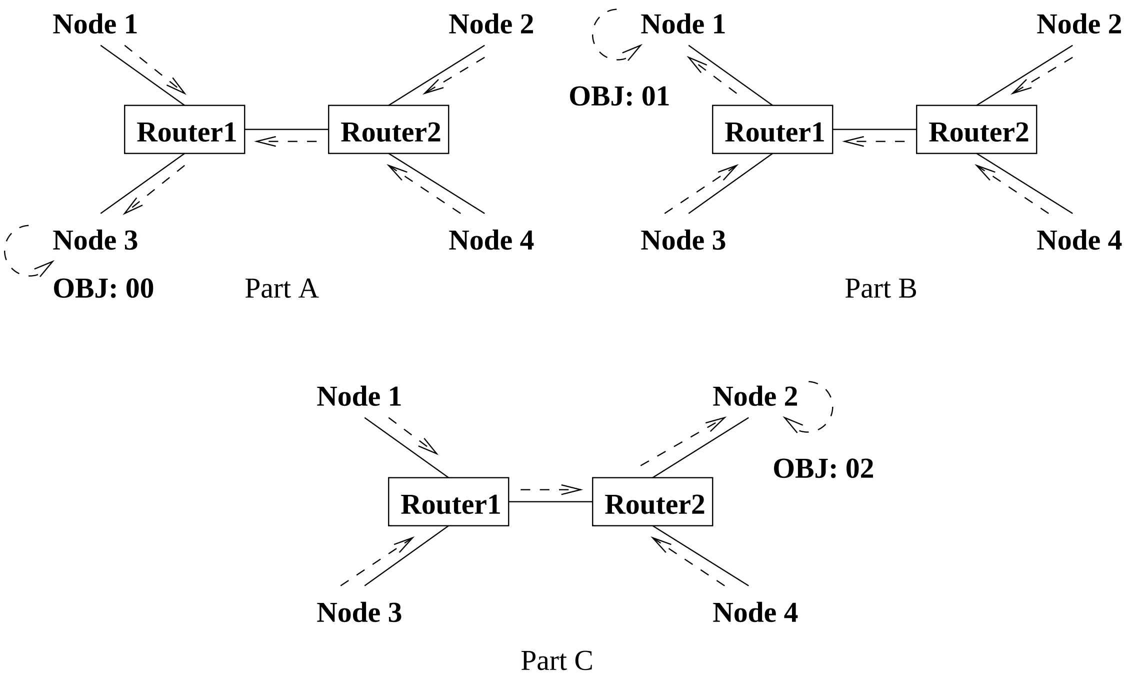

Figure 4: *In part A, Node 3 can send a multicast message with ID 0. Part B shows Node 1 sending with ID 1, and in part C Node 2 sends with ID 2. No node will finish processing an incoming multicast message unless the group sequence number is correct. The arrows represent the arrows in the directory system pointing to the object.*

A third protocol that is particularly well suited to this system is a modernized version of SNMP. The Simple Network Management Protocol is anything but. It is overly complicated, comprised of elaborate formatting rules, interfaces and variable names. The current protocol ranges across multiple RFCs, requires extensive custom work to get communications working with each device and is quite host dependent. A

[5] For more on providing support for multicasting via active networks, see [11, 16]

simpler way of providing this service is to extend the Active Network to include *all* devices[6] which are communicating over the network. Each one produces a protocol-local cache with all important management data (name, location, interfaces, statistics, etc). When the device boots up, it sends a simple message to the network, advertising itself. This message is cached in the routers for ease of lookup later by manager nodes and can be resent by the device on request. In addition to caching the advertisement, the routers also cache the device info (like MIBs) for persistence should one device go down or become unreachable. There's no need to format values in specific ways, since the system takes care of translation (ie, the protocol doesn't care about endianness since the VM does all the necessary translations), and requests are made simply by naming the piece(s) of data in question. Authorization is achieved by requiring an incoming request to be signed as a valid administrator. On the whole, the entire process is vastly easier. Communicating with any device is the same, there are no weird specifications about formatting data, devices announce themselves, authentication is simple and the whole process is provided as a basic protocol file. The devices themselves can keep the same interface, but customize the basic protocol to provide their own processing and management functionality. This kind of setup simply isn't possible with a standard passive network, and demonstrates the kind of flexibility and simplicity that can be achieved through activating the nodes in a network.

[6] Simulation of this protocol involved a simulated set of devices, but would work the same way once the PANTS OS was adapted to run natively on the host hardware (printers, etc)

```
; The input structure has field 0 the address of the machine
; which sent the request, field 1 an integer representing the
; action to take, and field 2 the name of object that is being
; requested

; store the input structure...the input has been authorized
ASTORE          10
ALOAD           10
; get the comand flag, to see what action we're taking
GETFIELD        1
ICMP_0
BNE             NEXT1
; the command number was zero, a request to get a
; device variable...get the variable name
ALOAD           10
GETFIELD        2
LDC             TR_GET_PROTO_CACHE
TRAP
; send back the cached value
ALOAD           10
GETFIELD        0
LDC             TR_SEND
TRAP
; we're finished!
BA
```

Figure 5: *A small piece of the base protocol. This code example shows a message requesting a particular named object.*

6 Observations

Some existing systems were considered and the "best" features were pulled out of them. A careful exploration into security and performance problems yielded a checklist of things to avoid, and a sense of which features were critical. Given all of this, a prototype system was built and custom protocols were written to test all aspects of the system and answer some questions about what an Active Network offers. With all of this done, the final question can be addressed: what about performance? To what degree can performance gains be made that outweigh the penalties and what situations are simply too slow to consider practical?

The simple answer is that PANTS, and all of its associated pieces performs decently. The language provides a relatively efficient means for executing typical network functionality, which means that most actions require few instructions. The Actual hits that are taken by installing and running protocols are slim, and they are in specific places. Installing a protocol is a one time action, and despite what it involves (security checks, cache writing, etc) requires little cpu overhead. Starting a protocol running given some input data also requires security checks, and mapping the data into the correct input type, but again this doesn't require too much of the host. The security, aside from the initial checking, is largely provided by the VM or very simple stack checks. On the whole, this is a small amount of time compared to total execution time, and is well worth the small effort to gain the heightened level of safety.

The protocols described (both those 3 discussed in detail and the other ones mentioned throughout this paper) each exercise slightly different feature sets of the system, and overall give a good sense of average protocol performance. Certainly from a user perspective, running these protocols on a (simulated) high-load network, they perform decently. Running these test protocols in parallel without much degradation shows that the language and OS interface really is flexible enough to provide what a network programmer needs without detracting from security features.

6.1 Performance Data

Given these vague assertions about the performance of the system, what were the real numbers like, and do they provide any useful information? Well, unfortunately, it's hard to get anything close to a real performance data set for this system. The OS is supposed to be running on routers in a real, user flooded network. In order to get performance data, therefore, which can be accurately compared to real protocol numbers, a physical test network needs to be built to start logging resulting data. Without these resources available, the best thing is to make some approximate performance measurements, and try to conjecture about the reality of the situation. In this case, the OS is being run in a network simulator which provides a router's interface.

The general points where performance degradation is caused have already been discussed throughout this paper. Starting here, an attempt can be made to understand what kinds of performance problems will arise:

- **The Network.** On the whole, there will be the same number or fewer messages passed on the network. There isn't much space overhead in the packet format for node-to-node messages, so moving from a passive to an active format shouldn't increase the network traffic. Of course, one of the goals of caching is to *decrease* the amount of traffic, so this should be a net win for the system. That leaves two other points of potential performance degradation.

- **The Operating System.** Here there are several problematic points. Calculating and verifying signatures is expensive, and if the option is required, will be done for all incoming data. The protocol switching, data mapping and process management is relatively fast (since it happens constantly), and should incur a minimal speed hit. In testing the protocols described in this paper, less than 5% of the OS' processing time was spent in these sections, and they use a very small amount of memory ($< 20Kbytes$). In the same protocol testing, the caching systems performed decently. They are currently unoptimized, simple sets of hashtables which should be fine tuned (in future versions). This is a clear area where drastic performance increases could be achieved. In addition, the process management code could be written to take advantage of the threading model more efficiently, and could work with waking/sleeping protocols to better move the flow of control.

- **The Virtual Machine.** The VM is the biggest performance hit at the moment. On average, a VM instruction goes through 11-12 lines of C code, averaging about 40 actual machine instructions (on the SPARC architecture). Trapping adds an additional cost, and depending on the trap call being invoked, this can slow the action further. The trap call to join an ASD group, for example, is quite expensive. The calling protocol instance, however, is put to sleep until the join command has completed, so other protocols have a chance to work in the mean time. As has already been discussed, the performance of the entire system could be greatly enhanced by adding JIT features to the code processing routines, but this in turn has the potential to remove certain security features. Altering the code processing routines to JIT compile sections of code, while retaining certain security checking, would undoubtedly provide a significant overall speedup to the system.

```
case iTRAP:
  /* make sure that we've installed a trap handler */
  if (trap_handler_) {
     /* a trap needs to be passed an integer */
     if (stack_top_type(code->proc_stack) != BT_INT) {
        code->pc++;
        break;
     }
     /* we're ok to trap...get the trap value */
     stk_val = stack_pop(code->proc_stack);
     trap_handler_(stk_val->data, code);
  }
  code->pc++;
  break;
```

Figure 6: *As an example of what the VM code looks like, this is the trap instruction. It makes sure that an integer value was passed as the trap number, pops this value and passes it to the trap handler along with the state of the program. The trap handler is installed by the embedding application.*

These descriptions aren't really enough, unfortunately, to give an accurate sense of the performance of the system. They do, however, illustrate that given some work on a few basic features of the network (not outside the realm of the possible, merely beyond the scope of this project), the overall performance would be reasonable. What does reasonable mean? Testing of the protocols discussed in this paper suggests that from a user point of view, things work at about the same rate[7]. Some applications (such as web browsing and stock monitoring) sped up considerably, but only when several test clients were looking at the same data. Other protocols (for example multicast) showed a slight slow down over the expected rate, due to the cost of getting a lock on the shared "write token." Of course, this is a slight loss over regular, unordered IP Multicast, so the comparison is somewhat unfair. In all, the protocols demonstrate decent enough performance that pursuing this system further seems worthwhile. Moving to a real network, running the OS on routers and watching the overall network reaction would be the next step towards understanding how well the system is really working.

7 Conclusion

The final question that is left to answer is the question of realism. Given all that has been discussed, how practical are these systems? Are the tradeoffs worthwhile? Do they provide a significantly different set of features that makes them uniquely valuable? The answer, for the time being, seems to be yes to a

[7] Tests run in the active environment were compared to the same types of actions in a real network. Simulation time for the Active Network was factored into the side-by-side comparisons

certain extent. There is definitely a certain amount of value gained by constructing an active environment (demonstrated especially well by the second and third protocols discussed above). The features really do provide for a new class of protocols which can't be built in a static network. As for the tradeoffs, these seem reasonable. Yes, there is some slowdown on some applications, and not everything will perform as well in an Active Network, but not everything is meant to run in this new environment. Some applications (simple IP forwarding, for example) simply don't need a dynamic system, and will continue to perform faster building on the static nature of the network. In addition, today's environments may not be well suited for widespread deployment of Active networks. It seems clear that modern routers are built for doing a small class of tasks extremely fast, and the overhead of an active system might be too much. That said, it's an area that is definitely worth pursuing further. Given where networks and network applications are headed, some of the applications and protocols discussed in this document seem particularly valuable and interesting. The future of these systems is somewhat unclear, but certainly expect to see them in the research spotlight for some years to come.

References

[1] Hagit Attiya and Jennifer Welch. *Distributed Computing*. McGraw Hill, 1998.

[2] SRI Computer Science Labratory. http://www.csl.sri.com/ancors/abone/. Web Site.

[3] Michael J Demmer and Maurice P Herlihy. The arrow distributed directory protocol. Undergraduate Thesis, Brown University, May 1998.

[4] Beverly Schwartz et al. Smart packets for active networks. BBN Internetworking Research Department, 1999.

[5] D Scott Alexander et al. A secure active network environment architecture: Realization in switchware. *IEEE Network*, 12(3):37–45, May/June 1998.

[6] D Scott Alexander et al. The switchware active network architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[7] Dan S Decasper et al. A scalable high-performance active network node. *IEEE Network*, pages 8–19, January/February 1999.

[8] John J Hartman et al. Joust: A platform for liquid software. *IEEE Computer*, pages 50–56, April 1999.

[9] Jonathan M Smith et al. Activating networks: A progress report. *IEEE Computer*, pages 32–41, April 1999.

[10] Kirk A Bradley et al. Detecting disruptive routers: A distributed network monitoring approach. *IEEE Network*, pages 50–60, September/October 1998.

[11] Maria Calderon et al. Active network support for multicast applications. *IEEE Network*, 12(3):46–52, May/June 1998.

[12] Ramesh Govindan et al. An architecture for stable, analyzable routing. *IEEE Network*, pages 29–35, January/February 1999.

[13] Theodore Faber. Acc: Using active networking to enhance feedback congestion control mechanisms. *IEEE Network*, 12(3):61–65, May/June 1998.

[14] Stefanos Gritzalis and George Aggelis. Security issues surrounding programming languages for mobile code: Java vs. safe-tcl. *Operating Systems Review*, pages 16–32, 1999.

[15] Sun Labs. Jini technology architectural overview. Sun Microsystems Labs, 1999.

[16] Li-Wei H Lehman, Stephen J Garland, and David L Tennenhouse. Active reliable multicast. MIT Laboratory for Computer Science, 1997.

[17] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[18] Marina Thottan and Chuanyi Ji. Proactive anomaly detection using distributed intelligent agents. *IEEE Network*, 1998.

[19] David Wetherall, John Guttag, and David Tennenhouse. Ants: Network services without the red tape. *IEEE Computer*, pages 42–48, April 1999.

[20] David Wetherall, Ulana Legedza, and John Guttag. Introducing new internet services: Why and how. *IEEE Network*, 12(3):12–19, May/June 1998.

8 Appendix A: The PANTS Bytecode Specification

The bytecode is laid out into three distinct pieces: constants, structures and code.

Constants.

At the top of the bytecode is a section where constant values can be defined. This might include a large integer value or an array of characters (a string). The Layout is as follows:

- Two bytes give the number of constant items in the bytecode.
- For each item, the first byte gives the type (at present, this integer or string).
- For each item, the second byte gives the length of the object (in practice, most constant values are less than 256 bytes, so the length is limited by this size).
- For each item, the next N bytes (where N is the value in the second byte) contain the data of the object.

The ordering of the objects is also the way they are referenced, so the first object is #0, the second #1, etc. Instructions in the code can then refer to one of these constant values instead of having to construct them dynamically.

Structures.

The next section of the bytecode contains all of the custom structures that the program uses. This is the most complicated section of the bytecode, since structures can take on many forms. The layout is as follows:

- One Byte for the number of custom structures included
- For each structure, a byte listing the number of fields
- For each field, a byte giving the type
- If the type is a reference, another byte follows giving the ref type (base, array or struct)
- If the type is an array, another 2 bytes follow, with a number representing the length of a fixed size array, or zero representing that the array will be dynamically allocated.
- If the type is a reference, a final byte follows enumerating the variable type of the reference (either a base type, or indexing one of the custom structures defined in this block).
- Finally, a single byte follows which defines the input type for the routine. This is what type the routine expects at startup to be passed from the embedding code. In practice, this is the data received as input from the network. This type can be a base type, or one of the custom defined structures.

The type information is found in the VM's `rvm_types.h` header file. There are defines for base types, arrays, structures and custom types. Again, the ordering in this list defines the index number for referencing a custom structure. Custom structure reference numbers start at 20 and go up, since there are about 20 base types already defined by the VM.

Code.

The code follows last in the bytecode. It is simply a set of words (4-byte values) which each represent one instruction. These instructions are listed in the next section.

9 Appendix B: The PANTS Instruction Set

Each instruction is described like this:

instruction	op-code	operand1	operand2	operand3

- A word in the VM is 4 bytes (32 bits).
- Each instruction is 1 word, where the operator itself is 1 byte (8 bits) long, and the operands get 3 bytes (24 bits).
- The VM is stack based

When referring to position in the stack, the top-most item is labeled 0, and on down the stack the label number increases (so that the bottom most item is labeled 'height-1').

Data can be stored in one of two places during the run. There are a few named "registers" for local storage, and then all other data can be stored into memory. Entering a new function (jsr) clears all local variables, and then returning from the function (ret) restores the locals.

Unlike the the Java language, there are no boolean base types in this language, so truth is simply determined by zero or not zero (as in C). Also unlike Java, numeric types can be signed or unsigned. There are not, however, separate operators for these. A variable is declared as an int or an unsigned int, and then all operations will take this initial declaration into account.

A "char" is basically an 8-bit data type (occupying the top 8 bits of a word). The type can, like numeric types, be either signed or unsigned.

Integers, floats and references are single word data types.

Doubles and longs are double word data types. This means that they use two words on the stack. The most significant bits are in the upper word (n) and the least significant bits are in the lower word (n+1).

All numeric values in bytecode, as well as those values passed from VM to VM are represented in network byte-order.

Condition Codes are global (per process) values which are set by certain operations (mostly comparisons). These can then be used to understand relationships between values.

Structures are represented by references, and then indexed by order of fields (so that the first field is index 0, the second index 1, etc). Union style structs, or similar struct casts can happen using the map call.

All branch offsets are representing a number of instructions from the start of the code (ie, to branch to the start would be zero, branching to the second instruction would be one, etc).

This list represents the instructions which are currently implemented. The floating point operations aren't avilable yet, but work in the same manner as the integer versions. This core subset was implemented to test the complete system, and so the rest of the instruction set will be added soon.

9.1 Arithmetic Operations

iadd		none	none	none

Pops the top 2 entries off the stack, add their values, and then push the resulting value back onto the stack

PRE: There must be 2 words on the stack. They must both be integers

POST: A new stack top of type INT

isub		none	none	none

Pops the top 2 entries off the stack, subtract the value of the second from the value of the first, and then push the resulting value back onto the stack

PRE: There must be 2 words on the stack. They must both be integers

POST: A new stack top of type INT

imul		none	none	none

Pops the top 2 entries off the stack, multiply their values, and then push the resulting value back onto the stack

PRE: There must be 2 words on the stack. They must both be integers

POST: A new stack top of type INT

idiv		none	none	none

Pops the top 2 entries off the stack, divide the first by the second, and then push the resulting value back onto the stack

PRE: There must be 2 words on the stack. They must both be integers

POST: A new stack top of type INT

irem		none	none	none

Pops the top 2 entries off the stack, divide the first by the second, and then push the resulting remainder back onto the stack

PRE: There must be 2 words on the stack. They must both be integers

POST: A new stack top of type INT

ineg		none	none	none

Pops the top entry off the stack, negate its value (multiply by negative 1), and then push the resulting value back onto the stack.

PRE: There must be a word on the stack. It must be an integer

POST: A new stack top of type INT

9.2 Logical Operations

ishl		none	none	none

Pops the top 2 entries off the stack, shift the second value left by the low 5 bits of the first, and then push the resulting value back onto the stack
PRE: There must be 2 words on the stack. They must both be integers
POST: A new stack top of type INT

ishr		none	none	none

Pops the top 2 entries off the stack, shift the second value right by the low 5 bits of the first, and then push the resulting value back onto the stack
PRE: There must be 2 words on the stack. They must both be integers
POST: A new stack top of type INT

iand		none	none	none

Pops the top 2 entries off the stack, bitwise and the two values, and then push the resulting remainder back onto the stack
PRE: There must be 2 words on the stack. They must both be integers
POST: A new stack top of type INT

ior		none	none	none

Pops the top 2 entries off the stack, bitwise or the two values, and then push the resulting value back onto the stack
PRE: There must be 2 words on the stack. They must both be integers
POST: A new stack top of type INT

ixor		none	none	none

Pops the top 2 entries off the stack, bitwise xor the two values, and then push the resulting value back onto the stack
PRE: There must be 2 words on the stack. They must both be integers
POST: A new stack top of type INT

9.3 Numeric Conversions

i2c		none	none	none

Pops the top entry off the stack, convert the value to a character, and then push the new character back onto the stack.
PRE: There must be one word on the stack. It must be an integer
POST: A new stack top of type CHAR

9.4 Constants

iaconst_null		none	none	none

Pushes the constant reference to null onto the stack
PRE: none
POST: A new stack top of type REF and value 0

iconst_n1		none	none	none

Pushes the constant value -1 onto the stack
PRE: none
POST: A new stack top of type INT and value -1

iconst_0		none	none	none

Pushes the constant value 0 onto the stack
PRE: none
POST: A new stack top of type INT and value 0

iconst_1		none	none	none

Pushes the constant value 1 onto the stack
PRE: none
POST: A new stack top of type INT and value 1

lconst_0		none	none	none

Pushes the constant value 0 onto the stack
PRE: none
POST: A new stack top of type LONG and value 0

lconst_1		none	none	none

Pushes the constant value 1 onto the stack
PRE: none
POST: A new stack top of type LONG and value 1

fconst_n1		none	none	none

Pushes the constant value -1 onto the stack
PRE: none
POST: A new stack top of type FLOAT and value -1

fconst_0		none	none	none

Pushes the constant value 0 onto the stack
PRE: none
POST: A new stack top of type FLOAT and value 0

fconst_1		none	none	none

Pushes the constant value 1 onto the stack
PRE: none
POST: A new stack top of type FLOAT and value 1

dconst_n1		none	none	none

Pushes the constant value -1 onto the stack
PRE: none
POST: A new stack top of type DOUBLE and value -1

dconst_0		none	none	none

Pushes the constant value 0 onto the stack
PRE: none
POST: A new stack top of type DOUBLE and value 0

dconst_1		none	none	none

Pushes the constant value 0 onto the stack
PRE: none
POST: A new stack top of type DOUBLE and value 1

ldc		data_high_bits	data_mid_bits	data_low_bits

Pushes a constant value onto the stack.
PRE: none
POST: A new stack top of type INT and value of bits (high mid low)

9.5 Stack Manipulation

pop		none	none	none

Pops and discard the top value on the stack.
PRE: There must be a single-word type on top of the stack
POST: Previously topmost element gone

pop2		none	none	none

Pops and discard the top two words on the stack. This could represent two single-word types, or a single double-word type.

PRE: There must be 2 words on the stack. They must both be single-word or part of the same double-word.

POST: The previously top two words are gone

dup		none	none	none

Duplicates the topmost element on the stack.
PRE: There must be a single-word element on top of the stack
POST: The previously topmost element has been duplicated

swap		none	none	none

Pops the top 2 elements off the stack, and push them back in reverse order.
PRE: There must be 2 single-word elements on top of the stack
POST: The previous top 2 elements have been switched

9.6 Flow of Control Operation

icmp		none	none	none

Compares the values of the 2 top elements on the stack, and sets the condition codes appropriately.
PRE: There must be 2 words on the stack. They must both be integers
POST: The condition codes have been set

acmp		none	none	none

Compares the values of the 2 top elements on the stack, and sets the condition codes appropriately.
PRE: There must be 2 words on the stack. They must both be references
POST: The condition codes have been set

icmp_0		none	none	none

Compares the value of the top element on the stack and zero, and sets the condition codes appropriately.
PRE: There must be 1 word on the stack. It must be an integer
POST: The condition codes have been set

acmp_0		none	none	none

Compares the value of the top element on the stack and zero, and sets the condition codes appropriately.
PRE: There must be 1 word on the stack. It must be a reference
POST: The condition codes have been set

beq		off_high_addr	off_low_addr	none

If the condition code for equal is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

bne		off_high_addr	off_low_addr	none

If the condition code for not equal is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

blt		off_high_addr	off_low_addr	none

If the condition code for less than is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

bgt		off_high_addr	off_low_addr	none

If the condition code for greater than is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

bge		off_high_addr	off_low_addr	none

If the condition code for greater than and the condition code for equal to is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

ble		off_high_addr	off_low_addr	none

If the condition code for less than and the condition code for equal to is set, branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the next line or the branch offset

ba		off_high_addr	off_low_addr	none

Branch to the address offset given by the bits (off_high_addr off_low_addr). The offset must be valid (within the code of the program).
PRE: None
POST: The pc goes to the branch offset

9.7 Variable Management

iload		index_high	index_mid	index_low

Loads the integer stored in local variable with bits (index_high index_mid index_low). If there is no such variable, nothing happens.
PRE: none
POST: There is a new integer on the top of the stack.

fload		index_high	index_mid	index_low

Loads the float stored in local variable with bits (index_high index_mid index_low). If there is no such variable, nothing happens.
PRE: none
POST: There is a new float on the top of the stack.

aload		index_high	index_mid	index_low

Loads the reference stored in local variable with bits (index_high index_mid index_low). If there is no such variable, nothing happens.
PRE: none
POST: There is a new reference on the top of the stack.

iload_n		var_index	none	none

Loads the integer stored in special local variable var_index (where var_index is 0, 1, 2 or 3).
PRE: none
POST: There is a new integer on the top of the stack.

aload_n		var_index	none	none

Loads the reference stored in special local variable var_index (where var_index is 0, 1, 2 or 3).
PRE: none
POST: There is a new reference on the top of the stack.

iaload		none	none	none

Pops the top 2 elements off the stack, uses the first as in index into an array referenced by the second element, and then pushes back onto the stack the indexed value of type integer. If the types are wrong, or if the reference or index is invalid, noting happens.
PRE: There must be 2 words on the stack. The first must be an integer, and the second a reference
POST: The top of the stack contains the indexed value of type INT

aaload		none	none	none

Pops the top 2 elements off the stack, uses the first as in index into an array referenced by the second element, and then pushes back onto the stack the indexed value of type reference. If the types are wrong, or if the reference or index is invalid, noting happens.

PRE: There must be 2 words on the stack. The first must be an integer, and the second a reference

POST: The top of the stack contains the indexed value of type REF

caload		none	none	none

Pops the top 2 elements off the stack, uses the first as in index into an array referenced by the second element, and then pushes back onto the stack the indexed value of type char. If the types are wrong, or if the reference or index is invalid, noting happens.

PRE: There must be 2 words on the stack. The first must be an integer, and the second a reference

POST: The top of the stack contains the indexed value of type CHAR

istore		index_high	index_mid	index_low

Pops the top word off the stack, and stores this integer value at local variable (index_high index_mid index_low). If the top value is not an integer, nothing happens. The previous value at (index_high index_mid index_low) is overwritten.

PRE: There must be a word on the stack. It must be an integer

POST: The previous top stack element has been removed

astore		index_high	index_mid	index_low

Pops the top word off the stack, and stores this reference value at local variable (index_high index_mid index_low). If the top value is not a reference, nothing happens. The previous value at (index_high index_mid index_low) is overwritten.

PRE: There must be a word on the stack. It must be a reference

POST: The previous top stack element has been removed

istore_n		var_index	none	none

Pops the top word off the stack, and stores this integer value in special local variable var_index (where var_index is 0, 1, 2 or 3).

PRE: There must be a word on the stack. It must be an integer

POST: The previous topmost word has been removed

astore_n		var_index	none	none

Pops the top word off the stack, and stores this reference value in special local variable var_index (where var_index is 0, 1, 2 or 3).

PRE: There must be a word on the stack. It must be a reference

POST: The previous topmost word has been removed

iastore		none	none	none

Pops the top 3 words off the stack. The first is the integer value to store and the second is the index into an array referenced by the third value. If the types are wrong or the index or reference are invalid, nothing happens.

PRE: There must be three words on the stack. They must be (in order) an integer, an integer and a reference

POST: The array index is set to the given value

aastore		none	none	none

Pops the top 3 words off the stack. The first is the reference value to store and the second is the index into an array referenced by the third value. If the types are wrong or the index or reference are invalid, nothing happens.

PRE: There must be three words on the stack. They must be (in order) a reference, an integer and a reference

POST: The array index is set to the given value

castore		none	none	none

Pops the top 3 words off the stack. The first is the char value to store and the second is the index into an array referenced by the third value. If the types are wrong or the index or reference are invalid, nothing happens.

PRE: There must be three words on the stack. They must be (in order) a char, an integer and a reference

POST: The array index is set to the given value

9.8 Memory Management

new		data_index_high	data_index_low	none

Allocates a new block of memory of size and type indexed by (data_index_high data_index_low). Any index value below BT_UNDEFINED represents a base type, and any higher value is a structure.

PRE: none

POST: A new entry is on the top of the stack, pointing to the new memory block, of type REF

newarray		data_index_high	data_index_low	none

Allocates a new block of memory for an array. The length of the array is given by the integer value on the top of the stack. The type is defined by the variable index (data_index_high data_index_low). Any index value below BT_UNDEFINED represents a base type, and any higher value is a structure.

PRE: none

POST: A new entry is on the top of the stack, pointing to the new memory block, of type REF

arraylength		none	none	none

Pops a reference to an array off the stack, and pushes a new value that is the length of the referenced array.

PRE: The stack must have one word. The word must be a reference to an array.
POST: A new entry is on the top of the stack, representing the length of an array.

9.9 Memory Manipulation

getfield		field_high_index	field_low_index	none

Pops the top value off the stack, which is a reference to a structure. Pushes back onto the stack the value of the structure's field represented by (field_high_index field_low_index).
PRE: There must be a word on the stack. It must be a reference to a structure.
POST: A new entry is on the top of the stack, with the value of the structure's field.

setfield		field_high_index	field_low_index	none

Pops the top 2 words off the stack, the first of which is the data to set, and the second is a reference to a structure. The field (field_high_index field_low_index) of the structure must be of the same type as the data, and is set to the given data value.
PRE: There must be 2 words on the stack. The second must be a reference
POST: none

9.10 Function Call/Return

jsr		off_high_addr	off_low_addr	none

Jumps to a subroutine starting at (off_high_addr off_low_addr) instructions from the start of the program. All local variables (except the special named ones) are cleared. The index must be in the range of the program code.
PRE: none
POST: The pc moves to the new instruction value.

ret		none	none	none

Returns from a subroutine. The previous local variables are restored. If returning from the original (main) function, the program finishes execution, passing back the top value on the stack (if there is one).
PRE: none
POST: The pc and local variables from the calling routine are restored

9.11 Global DB Calls

trap		none	none	none

Moves out of the VM code, pops the integer from the top of the stack, and passes this value to the custom trap handler.

PRE: There must be a word on the stack. It must be of type integer
POST: none

9.12 Miscellaneous

nop	00000000	none	none	none

Does nothing
PRE: none
POST: none

iinc		none	none	none

Increments the integer on the top of the stack by one.
PRE: There must be a word on the stack. It must be of type integer
POST: none