PANTS whitepaper, v0.2*

Seth Proctor, stp@cs.brown.edu

February 22, 1999

1 Overview

The Programmable, Active Network Toolkit and Simulator (PANTS) is a package both for simulating and implementing active networking environments. A programmable or active network is one in which the network elements (ie, routers) are dynamic. A packet not only carries data, but also code. This code can be installed on the underlying network structure to be used in current and/or future communications. In this way, new protocols can be introduced as needed.

Because of the inherent portability, performance and security concerns associated with such a system, a certain amount of support needs to be in place to provide this type of environment. PANTS provides such a framework through a combination of real-time operating system, virtual machine and programming language. These work together to provide the active network.

The operating system and virtual machine would both be run on the routers themselves in a real environment, but in the typical scenerio, these are run as simple processes. This means that it's easy to simulate your new routines without actually pushing them onto a real network. The codebase for the entire project is written in ANSI C, so that it should easily compile on almost any system.

This paper will describe the entire system. This includes the high level language provided to the programmer, the instruction set that supports the bytecode (compiled) version of programs, the virtual machine and its properties, the operating system and functionality it provides and how all these components work together. This paper focuses on the technical aspects of the system, so things like bytecodes and memory representations will be discussed here. For a less technical and more conceptual overview of this system, see the companion paper "A proposal for a new active network."

*This is a year long thesis project for the academic year '98-'99. The academic advisor for the project is Tom Doeppner, twd@cs.brown.edu.

2 The Language

2.1 Introduction

The high level language provided to work with this system (ORPL) is the programmers interface to the active network. The language is C-like in syntax, and compiles to a bytecode that can be sent out to the network. There is a simple API that then wraps around the functionality needed to load this bytecode and install it on the active network.

2.2 Syntax

The syntax is, as has already been explained, very close to that of C. This is done for a number of reasons. Primarily, C is a familiar language to most network programmers, and so most people using this system won't have to learn a new or unfamiliar syntax. Also, the C syntax provides most of the functionality and flexability needed for programming to this system.

The syntax of the language does include every language feature of C. Memory is slightly different, as are pointers and pointer arithmetic. Also, accessing global data is done differently. On the whole, however, it should look very familiar.

2.3 The Mainline

The entry point for any program in this language is a function that takes a few parameters, including incoming data, information from the operating system, etc. [more on exactly what this is, and how it's provided]

2.4 References and Pointers

Because of security and verification concerns, true pointers are not allowed by the instruction set, and are therefore not available in the language. The declaration and allocation of a pointer as in C (ie, `char *data`) will actually yield a reference. The important difference between a pointer and a reference is that you are not allowed to arbitrarily access into the memory space (ie, pointer arithmetic).

Like in C, references are `NULL` before they're allocated. The `new` operator must be called to actually allocate memory of some given size. This size cannot be reallocated. When you're finished with the reference, the `delete` operator should be used to dispose of the space in memory being used.

A reference must be created as a particular type. This could be a base type, an array, or a new `struct`. This is decided by a lookup table that the virtual machine maintains.

2.5 Structures

Structures are the same as in C. They can be defined for a set of routines, and then allocated and referenced via the `.` or `->` operator.

2.6 Global Data and Functions

One feature exported by the operating system is a global store of data. This can be used to export/import data, structures and functions. [Some syntax needed for this kind of reference]

2.7 Interaction APIs

In order to use the programs that are written for the active networking environment, there are a set of simple libraries for loading bytecode, exporting it to the network, and installing at certain points. This interface also provides ways to send messages that use the header and language information associated with the system.

3 The Virtual Machine

3.1 Introduction

The virtual machine is obviously an important component of the system. It executes the bytecode language, and allows the system to be portable and migrate code across a network. There is also a great deal of security that is only available because of execution through a virtual machine. The VM is provided as an embedable library. In this particular project the VM is embedded into a small operating system, but the VM could be put into a testing system, a coordinating process for an active network region or any other program that needs a simple virtual machine.

The VM works with a simple set of structures, representing base types, instructions and different running processes. It can handle an arbitrary number of running programs, and is flexable enough to allow these programs to switch on and off. Security is provided such that string data types are enforced, code is verified (optionally), null dereferencing is prevented and array bounds checking is done on all accesses.

3.2 Operating With the Instruction Set

The instruction set is listed in the next section. It is a complete language for interacting with the virtual machine. The compiler for this system takes the programmer's high level code and translates it to an assembly-like representation in the VM's instruction set, and then into a bytecode that can quickly be shipped across the network. Alternately, an assembler is provided to take straight assembly and generate bytecode.

The bytecode is small and simple, and can be understood by a VM on any platform. The instruction set provides support for store/load commands, (un)conditional branching, function call/return, dynamic (de)allocation of memory, arithmetic, structure and array management and many other basic functions.

3.3 Layout of the Bytecode

The bytecode that is compiled from the instruction set takes a simple form. The first nine bytes give some information about the incoming data:

- The first byte contains special flags, such as whether this packet contains data or code.
- The next 8 bytes name the routine provided by the code.

The next section defines the constant pool:

- [give definition of the constant pool section]

The next section defines custom structures:

- 1 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 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 code follows as word long entries (see the next section on the instruction set for how these are laid out).

3.4 Handling Multiple Bytecodes

The VM will execute one instruction in one block of code at a time. Each block of bytecode is kept in a separate structure, along with information about its state, global and local variables, condition codes, etc. Switching between running routines is simply a matter of asking the VM to use a different code block. All associated state will stay consistent. The finest granularity for this is a single instruction.

This allows for several different programs to run on the VM at once. If more than one copy of the same program is running, only the core set of data particular to each running instance of code need be duplicated. The description of the program itself can be loaded once and pointed to by many different copies.

3.5 Security

A fair amount of security is provided by the VM. This consists mostly of verification, typechecking, and memory monitoring.

On each instruction, type checking is done to make sure that the right data is being provided. For instance, before dereferencing a data structure, the VM verifies that the provided value is indeed a reference. Or, when indexing into an array, the VM verifies that the data structure is an array, and that the indexing value is an integer.

Before actually performing any data dereferences, the addresses being read or written are checked. Whether accessing a field in an struct, a dynamically allocated base type, or an index into an array, the addresses are all carefully checked. Index values are verified in the valid range of an array, assignment types have to match up, and all memory locations must be valid.

3.6 Running Independently

Because the VM is built as a simple library, it can be run as a simple process, embedded into other programs for testing or execution. This means that in addition to running the operating system (which embeds the VM), any number of other applications can be written around the VM.

Testing or debugging environments could be built to simulate the actual system, using the VM as a way to work with the code being built. New operating systems can be built tailored to the individual's needs. Simple client/server systems could be constructed to interact with an active network, or to connect disjoint systems.

The VM provides, as one of its instructions, a trap call. This will hook into a programmer function, and provide the embedding program a trap value (integer) from the assembly call and the state of the running program. The OS exports a standard set of trap values, but the embedding program could define a new set of values for custom routines.

4 The Instruction Set

[NOTE: This is the set of currently supported instructions. The "complete" set of instructions is in the instructions document. As more instructions are available, they will be moved into this section.]

4.1 Introduction

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).

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).

4.2 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 remainder 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

4.3 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 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

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 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

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 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

ixor		none	none	none

Pops the top 2 entries off the stack, bitwise xor 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

4.4 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

4.5 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 0

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)

4.6 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

4.7 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

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 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

4.8 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

4.9 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.

4.10 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

4.11 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

4.12 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

4.13 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

5 The Operating System

5.1 Introduction

The operating system is simple wrapper around both the virtual machine and the network interface. It provides layers of security and QoS, as well as certain platform dependent functionality. It is also responsible for timesharing the different routines fairly, and monitoring things like cpu time and memory usage.

Trap calls from the VM are handled by the OS, such that certain custom support is provided. Caching, storage, globalization of functions and structures and communication to other systems is also provided through custom routines in the OS.

5.2 Tasks Handled

The operating system handles a few specific tasks. When data initially arrives, it needs to be preprocessed, to see if there is attached code, or if the data is supposed to be processed by a currently installed routine. If the code requires a routine that's not currently installed, that may need to be fetched first [explain process of sharing exported data between OSes].

If there is attached code, the operating system will run the code through a verifier, making sure that certain conditions are met, or the code will be rejected. This is similar to what java does to remote bytecode. Once the code checks out, it is installed, and ready for incoming data.

Incoming data has its header info stripped to determine in which routine it needs to be run, and if it has any special concerns (timing issues, etc). Checks are made to ensure that it's ok to pass this data on, and then it is put in a queue for the given protocol. [explain what the criteria are]

At this point, the operating system is responsible for making sure that the various installed routines all get fair cpu time based on certain admin defined requirements or QoS concerns. It also coordinates moving data from the incoming queues into a state of execution on the virtual machine.

5.3 Interacting With the VM

The operating system drives the virtual machine, switching the routines that are getting run time. At the same time, the virtual machine makes requests of the operating system. These requests could be for something platform dependent that the operating system is providing or access to the network. They could also be reads from or writes to the system database. More can be found on this in section 5.6.

The OS also calls on VM utility routines to set up new process that are about to run. Process specific data is initialized for the new process, but global information (like the code for the routine) is kept global (but protected).

5.4 Interacting With the Network

Very little interaction with the network is actually needed. A routine does not get called until there is some input data for it. The running routine may never explicitly write any data back out onto the network, may may just return a block of data, which in turn the operating system will write out to the network. [more on how data moves back out, including simple forwarding, and command bit flags]

5.5 Security

[This subsection needs to be a whole section to itself, explaining some of the QoS, security, resource management and protection schemes that are available to the admin...of course, these need to be defined first, which is semester #2]

5.6 Exporting and Storing Data and Routines

The operating system provides a mechanism like a shared object repository. In it programs can store data (including code and function declarations) that can then be accessed not only by other routines on the same system, but remote systems as well. In this manner, global state about a network can be tracked and shared, different routines can communicate with each other, and routines can remember information from run to run.

There are, of course, security domains placed around this data sharing.

6 Putting It All Together: PANTS

Given this framework, The PANTS system can be described.

Each node in a network runs the operating system at some low level. It has an interface for network incoming/outgoing data, which can be the actual network, or another wrapper call from another interface. The OS works as a filter in this manner, accepting data meant for special processing, and returning data to pass back out onto the network.

The programmer writes a program in either the C-variant language or in direct assembly, and generates from this a bytecode file that can be installed in the OS. Using the libraries provided, the code can be bundled into IP packets, and sent onto the network. Using the same libraries, regular data can be sent that has explicit code dependencies.

7 Conclusion

Not much yet...some of this is very doable, while I'm still pretty sceptical on other points. At any rate, the VM is built and has been embedded in the first version of the OS. A testbed is in progress which will allow simulation, on a large scale, of a real environment using this system. Stay tuned for more information!