BBN Technical Memorandum No. 1219

Overview of Languages for the Smart Packets Project

Job No. 11822000

September 27, 1999

Prepared for:
Defense Advanced Research Projects Agency
Computer Systems and Technology Office
3701 North Fairfax Drive
Arlington, VA 22203-1714

Prepared by:
GTE
10 Moulton Street
Cambridge, MA 02138
Overview of Languages for the Smart Packets Project
Beverly L. Schwartz and Alden W. Jackson

1. Introduction

The goal of the Smart Packets project is to place executable code in packets which can help with network management tasks. The Smart Packets project will need a programming language which is compact, has constrained operation such that it will lessen the security risk, and has functions geared for the IP environment.

In this phase of the project, different programming languages are being surveyed to find what already exists that can help with the development of a programming language for Smart Packets. This paper discusses what the issues are for a Smart Packets language, and then gives details about how each of the languages studied can or can’t serve the needs of this project. The languages studied are: Perl 5, Scheme and Emacs Lisp, Java, and the CAML family of languages. Comments will also be made about the methodology used by SNMP; Active IP Option work using Tcl done by David Wetherall and David Tenenhouse of the Telemedia Networks and Systems Group at the Laboratory for Computer Science at MIT; and Python.

Ideally, the Smart Packets language will have the following characteristics:

- it can be represented in a compact form
- it can be used to write functions for the Smart Packets project
- it can dynamically link to a library of Smart Packets functions
- it can be constrained in such a way that it is safe
- it has a publicly available compiler which can be modified for Smart Packets’ use
- it has a publicly available interpreter, virtual machine or just-in-time compiler
- it has a foreign function interface which can call C or C++
- it has a reasonable development environment
- it has a debugging environment
- it is easy to learn, its syntax is familiar to the people most likely to use it, or its syntax is similar to an existing language that people want to learn
- it allows programmers to write documentable, maintainable code
2. Compact Representation

Ideally, a Smart Packets program should fit into a single packet without fragmentation. Although RFC 1122 recommends that networks in the internet be able to handle an MTU of at least 576 bytes, in today’s internet, the most likely constraint is Ethernet with an MTU of 1500 bytes or IEEE 802.3/802.2 with an MTU of 1492 bytes. Subtracting 40 bytes for IPv6 header size (20 bytes for IPv4), Smart Packets should plan for at most 1452 bytes. Depending on how Smart Packets are encoded, the amount of space available could be less.

Given the small amount of space available, there is great need to compress a Smart Packets program into the smallest space possible. Some possible schemes are:

- use a lossless compression algorithm
- compile the program into byte codes
- put the burden on the programmer to use minimal white space, shortest possible symbol names, shortest possible programming constructs, and no comments
- use a translation/compression/byte code scheme which takes ascii and removes white space, shortens symbol names, and translates key words to a space-efficient field delimited format
- create our own compact, intermediate, op-code language

2.1 Compression

A compression algorithm such as Lempel-Ziv can be used as the only compression technique, or in conjunction with one of the other techniques. Lossless compression rates for standard text can be as good as 2 to 1. However, there is also the overhead of compression information, which can make the result larger than the original text. Given how small Smart Packets programs must be, it is likely that compression will provide little benefit.

2.2 Byte Codes

In theory, translating ascii words to byte codes can save space. However, in practice, those languages which have byte code compilers produced results larger than the original ascii. The reason for the larger output was that the byte codes did not simply reflect a translation of the source. The byte code included other library functions, symbol tables, or byte codes that look like assembly language.

2.3 Programmer Burden

Requiring the programmer to compact code to least amount of white space, shortest symbol names, etc. is sufficient for proof-of-concept where the only people writing code are the developers of the system and the programs aren’t expected to survive longer than the proof-of-concept stage. Programs written with no formatting, comments and meaningless symbol names are unmaintainable. It is unreasonable to expect a programmer to program with this style.

2.4 Homespun Cruncher

This technique would require writing a new compiler or modification of the backend of an existing compiler. It would also require translation back to original code at the target machine, or writing a virtual machine, interpreter or just-in-time compiler to execute the homespun byte codes. This new backend could remove all unnecessary white space, shorten symbol names, and translate key words into byte codes. Special care will have to be taken in the debugging environment. Translated symbol names will have no meaning to the programmer, thus the source machine will have to save the symbol table so that errors reported to the programmer will have
meaningful symbol names in them. Likewise, the source machine will have to remember what line number code occurred at so that locations of errors can also be meaningful to the programmer.

2.5 Op-code Language

Create a CISC-style op-code language which is very compact. This would require writing a backend to a compiler, and a virtual machine to run the new op-code language. Since the code would be compiled to this intermediate format on the source machine, all compilation errors can be reported locally. The symbol table would stay at the source machine, so at runtime, special care would have to be taken for the debugging environment.

3. Safety

Smart Packets programs must not do damage or cause damage to the host they run on. This means they cannot write to the file system; get access to data they are not authorized to look at; damage the operating system; damage the run time environment; use too many resources; leave threads of execution lying around; spawn another program that does any of the above behaviors; or do any other anti-social behaviors. This means the language should be constrained so that it doesn't allow unsafe operations, and the interpreter or virtual machine that runs the program must be able to kill a program that has run too long or uses too many resources. Users of Smart Packets programs must also be authenticated.

4. Functions, Dynamic Linking and Foreign Function Interface

Smart Packets programs will do certain operations with regularity, such as "create IP packet", "send IP packet", "get MIB variable", etc. Rather than forcing the programmer to rewrite these functions every time, a Smart Packets development environment can provide a library for these common functions. Because packet manipulation functions involve moving bytes around and insuring correct byte order, the Smart Packets programming language must be able to handle byte manipulation easily.

Because of program space limitations, ideally, the functions a Smart Packets program calls should be installed in a Smart Packets aware router. A Smart Packets program must be able to dynamically link with the installed library. A Smart Packets program should fail if the program is not compatible with the version of the installed library.

Installed functions will be trusted code. They may have access to the file system and operating system functions. Any function that accesses the file system or OS is considered privileged. There will be access controls on installed privileged functions that prevent unauthorized programs from accessing sensitive information.

Some of the Smart Packets functions will need direct access to UNIX system functions that the language doesn't provide for. In these cases, a foreign function interface to C or C++ code will be needed and the efficiency of the language's foreign function interface becomes an important concern. Foreign function interface calls will be expensive for languages that provide threads and/or garbage collection, because the language will have to marshall/unmarshall heap objects and parameters.

Convenience functions can be dynamically loaded over the network. For safety, these functions cannot do any privileged operations. These functions must be written entirely in the Smart Packets language.
5. Compilers, Virtual Machines, Debuggers

Any compilers, interpreters, virtual machines, debuggers, and other programming tools that are publicly available for the Smart Packets language will speed up the development process. If the language that Smart Packets is based on is more mature, so will the programming tools that come with it. It is to the advantage of the Smart Packets project to be able to use mature tools from the public domain.

6. Learnability and Maintainability

The most likely audience for a Smart Packets tools is system and network administrators who are most likely familiar with scripting languages. Technical personnel are often interested in learning something hot or in demand such as a new language or utility that is highly marketable. If the Smart Packets language is too hard to learn or the motivation to learn it isn’t there, a Smart Packets tool may not be used to its full potential.

Additionally, Smart Packets programs should be usable over the years. As technical personnel change in an organization, the new people should be able to read a Smart Packets program and figure out what it does. If the needs for network management tools change, existing programs should be adaptable to the new needs. If the Smart Packets language is obtuse, long term maintainability of programs will be difficult.

7. Languages

The rest of this paper discusses the languages that were studied for the Smart Packets project. For those languages which provide byte code compilers, a function for computing fibonacci numbers was written, and compiled into byte codes. All implementations used the same algorithm. The size of the resulting byte code files are discussed with each of the languages. The size of the Scheme ascii representation is 131 bytes. Here is the Scheme implementation, taken from "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Sussman:

```scheme
(define (fib n) (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
    b
    (fib-iter (+ a b) a (- count 1))))
```

7.1 Perl 5

Perl is a scripting language which allows simple tasks to be implemented easily, yet contains the complexity to accomplish difficult tasks. Perl's syntax is succinct; Perl programs can have a great deal of functionality and be very terse. Perl has mechanisms for building objects which can be used for writing Smart Packets functions. Perl’s just-in-time compiler can easily link in existing Perl functions.

Perl does not provide a byte code compiler, nor a virtual machine. Its just-in-time compiler is in the public domain, so can be modified for Smart Packets’ use.

If the option of shortening symbol names is used for compact representation, then Perl’s eval statement will not work properly. Eval takes a string, and evaluates it. If the program puts together a statement based on an original symbol name, performing an eval on the old symbol name will not get the desired result since the program only knows about the translated symbol names. Thus, if symbol names are modified, the eval statement cannot be allowed.

Perl allows C code to be embedded into the Perl environment. Calling C from Perl will not be an expensive operation. A Perl interpreter can also be embedded in C code such as a daemon or the
Perl provides a "tainted" mode that doesn't allow code to harm the file system or execute external programs that could harm the file system. Perl in tainted mode can interface with trusted code that does privileged functions. Perl provides a "Safe" module which sets up special compartments for executing suspect code in which all system operations are trapped, namespace access is carefully controlled and limits operations available to the suspect code. Perl does not provide a mechanism for detecting a program that has run too long and/or used too many system resources.

Penguin, a Perl module on the public domain, provides authentication facilities. Penguin can send encrypted, digitally signed Perl code to a remote machine.

Perl provides a debugger for testing Smart Packets programs locally.

Perl is a mature scripting language with a large international community of users. Perl is popular among its users, and Perl programmers are currently in demand.

### 7.2 Scheme and Emacs Lisp

Scheme and Emacs Lisp are dialects of Lisp. Lisp is a symbolic processing language. There is a lot of power in Lisp and its derivatives, but the power tends to be hidden in subtleties and nuances. Lisp is difficult to learn and is not well known among system and network administrators. Lisp is loved by some and hated by others.

Scheme and Emacs Lisp both provide byte code compilers on the public domain. Scheme48, the Scheme byte code compiler, produced a byte code file of close to a megabyte for the fibonacci function illustrated above. Most of this massive size can be attributed to Scheme48 dumping all of its library functions. The compiler would have to be modified to dump only the desired functions.

Emacs Lisp produced a byte code file of 617 bytes. This file contained comments and code which checked for the correct version of emacs. After editing the comments and extra code, the resulting file was 139 bytes. A simple script can be written to automatically remove the comments and extra code. Emacs Lisp does not translate any symbols, so extra bytes can be saved by translating long, frequently used symbol names to something shorter. Like Perl, symbols constructed at run time can be evaluated, so unless certain operations are restricted, changing symbol names may cause programs to fail.

Functions can be built from Scheme and Emacs Lisp. Scheme stresses conceptual elegance and simplicity; it is very small. It is often used in computer science curricula and programming language research due to its ability to represent many programming abstractions with its simple primitives. However, it lacks "real world" usefulness due to its lack of utility functions. It will take significant effort to build Smart Packets functions in Scheme. Emacs Lisp has more real world utility functions and would allow a higher starting point for building functions. However, the Lisp model is one of symbolic processing and doesn't easily handle data manipulation at the byte level.

Scheme's and Emacs Lisp's interpreters both provide debugging environments for programs run in the local environment.

Scheme's lack of "real world" functionality does provide language constraint for Smart Packets' safety needs. Because Lisp can write code at run time that can be executed, it is unclear how to enforce safety at compile time. Scheme and Emacs Lisp both run in public domain interpreters which can be modified to limit how long a program runs and how many system resources the program is allowed to use.
7.3 Java

Java is a full featured, strongly typed, object oriented language, with easy to use exception handling. It provides classes and methods for more complex operations such as file I/O.

Java provides memory management. Programs must allocate space for the objects they use, but after they finish using them, Java will take care of doing garbage collection. Java makes no guarantees about when memory will be collected. It is unclear if program execution continues without interruptions or delays during garbage collection.

Java can be compiled into byte codes. Java's byte code compiler is not on the public domain, so if any modifications to the compiler are needed, we would have to write our own compiler or provide some sort of kludgy wrapper. The fibonacci function, written in Java (199 bytes of ascii), produces 370 bytes of byte codes. Part of this bulk is attributed to a symbol table included in the byte code file. Java does translate symbol names; changing a frequently used symbol from a name of 3 bytes to a name of 12 bytes results in only 9 extra bytes in the byte code file. The byte codes produced by Java are platform independent.

There is a Java virtual machine on the public domain called kaffe. It is not yet mature and does have bugs. Kaffe provides a debugging environment for locally executed code. Kaffe can be modified as needed.

Java's security model provides restricted name space, restricted operations, no access to the file system. The Java language has no pointers or global variables. Java does not protect from denial of service attacks. Java applets can create never ending threads, or threads which kill other Java threads. Kaffe can easily run a thread to kill never ending threads, but it would have to be modified to handle rogue thread killer applets.

Smart Packets functions can be written using Java objects and methods, and put together in a name space protected package. This package can be loaded into Smart Packets aware routers so that Smart Packets programs only have to carry references to the package. Java provides methods for dynamic loading over the net as well.

Java provides a networking package for sending and receiving TCP and UDP packets and HTML. It does not provide access to lower level IP networking calls. Java provides a foreign function interface to C, so that low level networking code can be done in C. Unfortunately, the foreign function interface is inefficient and calls to it will be compute intensive.

The development environment for Java is somewhat klunky. Java packages can be hierarchical, but the source and compiled code must be placed in a directory structure with the same hierarchy as the package definitions. Public Java classes that are accessible to code in files other than the one the class is in must be in a file with the same name as the class. Thus, if there is a class called Foo which is referenced from some other Java source file, then the source code for Foo must be in a file called Foo.java. The compiled Foo class will be in a file called Foo.class. If there is another class called Bar, and Bar is accessed only by Foo, then the source code for Bar can also be in Foo.java; the compiled Bar class will be in Bar.class. Class files can be pulled together in something called a zip file so that a single file can be used for installation.

Java syntax is similar to C++. For someone who is already familiar with C++, Java is not difficult to learn. For someone not familiar with object oriented programming, Java will have a steep learning curve. Because Java is new, there is not a large community of Java programmers; however, Java programmers are currently in great demand.

7.4 The Caml Family of Languages

Caml or "Categorical Abstract Machine Language" is a family of functional programming languages, i.e., the basic units of programs in the languages are functions. The Caml family, related to the ML programming language, includes Caml, Caml Light, and Objective Caml (previously
known as Caml Special Light). It is developed and distributed by INRIA. Any reference to Caml alone is assumed to refer to the entire family, unless stated so.

Caml is a strongly typed language. Types are static and annotations are handled by the compiler. Caml also features automatic memory management. There are no "new" or "free" primitives. Allocation and deallocation of data structures is kept implicit and handled by the compiler. The memory manager is designed to work in parallel with the application, so that there is no noticeable stop in the Caml program while the garbage collector is running. Caml also provides an exception mechanism and libraries.

The strongest differences are between Caml and Caml Light/Objective Caml. The latter two languages were designed to be smaller, faster, more portable and more efficient than Caml. While Caml Light produces byte codes, Caml only produces native machine code. Objective Caml can produce byte codes for all supported platforms and native machine code for a subset of the supported platforms. The module system that was designed for Caml Light becomes support for objects and classes in Objective Caml.

Caml byte codes can run on multiple platforms without modification. The fibonacci program written in Caml Light and Object Caml produces byte code files of 342 and 294 bytes, respectively. There is no direct relation between length of symbol names and size of resultant file. When a symbol that was used 3 times was made 5 bytes longer, the resultant byte code files were 350 and 298 bytes, respectively.

Caml Light and Objective Caml both support lexer and parser generators similar to lex and yacc. This would allow the project to take advantage of the byte code output, advanced memory management and strong typing, while giving us the freedom to design our own primitives and parser. Both the languages also support user-defined functions written in C to be linked with Caml, and called from Caml functions. Yet, the facility appears to be awkward to use.

Caml Light provides an interpretive system, a compiler, a standalone system for bytecode, lexer and parser generators, a debugger, and profiler. Objective Caml supports all the previous with the exception of a debugger. Objective Caml can also compile to native code on selected platforms. Both languages have libraries that support data structures, I/O, parsing, system interfaces including Unix system calls and threads (Objective Caml only).

Caml provides a simple means to constrain the language at compile time, but it does not provide any means to protect from run time problems such as a program that runs too long or uses too many resources.

Caml does not have a large user community, nor are Caml programmers in great demand.

7.5 SNMP Methodology

SNMP is not a full language - it is a protocol and a method for getting and storing information from and to a management information base (MIB). Most of Smart Packets needs cannot be met with SNMP.

SNMP is wordy and variable names in an SNMP MIB can be extremely long. SNMP uses clear text passwords for security. By convention, the password is almost always set to "public" for getting information. SNMP provides minimal security which is useful for keeping a naive user from accidentally breaking things. It provides no security for an experienced attacker.

SNMP has a minimum of operations - get value, get next value and set value. SNMPv2 also provides a get bulk operation, which gets multiple contiguous values. Any actions that happen in SNMP occur because of side effects - if a certain MIB variable is set, then the host knows to perform an operation.

Smart Packets could use the "SNMP way" for some of its operations. A Smart Packets MIB could
have variables that signify actions. When a Smart Packets aware router notices one of these variables is set, it can perform the intended action. For example, there could be a "ping host" variable whose value takes the "octetstring" type. When the "ping host" variable is set to anything other than 0.0.0.0, an ICMP echo request packet can be sent to the IP address specified in the value.

The simplicity of this mechanism is enticing. Very little language would be needed and all of the operations can be in the router. However, this allows little flexibility and only very simple operations can occur. More complex tasks can be broken into smaller parts, but difficulties will occur because operations happen asynchronously. There is no guarantee that operations will occur in the order that MIB variables are set. Operations occur when the host machine notices a variable has been set. Proper order of operations can be assured by having the source of a Smart Packets program wait for a response after each step; however, this will make execution of a larger and more complex task painfully slow.

### 7.6 Tcl and The Active IP Option

Tcl is an interpreted scripting language. It has no data types other than strings, although various Tcl extensions make it look like it has classes and structures. Tcl can easily embed C code, and can easily be embedded in C code. Tcl operations can easily be constrained by rebuilding the Tcl interpreter without unwanted functionality. Tcl is easy to learn and has quickly become very popular.

David Wetherall and David Tennenhouse from the Telemedia Networks and Systems Group at the Laboratory for Computer Science at MIT developed a proof-of-concept active networks system using Tcl code in the options field of the IP header.

Active IP Option functions were written in C and built into a stripped down Tcl interpreter. The interpreter was embedded within the kernel, and a restricted environment similar to that of Safe-Tcl was provided for Active Nets programs. It provides for no run time security; memory is allocated from the shared kernel pool, and execution time is not limited.

Active Nets programs are written in Tcl, and sent as ascii text. Compact representation is achieved by convenience functions in the interpreter and programmer’s burden. Short names are used for Active Nets functions.

Errors in the script are logged on the machine executing the code; no error message is sent back to the source machine.

### 7.7 Python

Python is an interpreted, object oriented scripting language. It uses white space and indentation style to delimit blocks of code. Python can be embedded in C and C++, and Python can be extended with C or C++ code.

Python is translated into byte codes and stored in byte code files. This was done to speed up Python’s start up time with a program. It is not clear if the resultant byte code file is platform independent.

Because of Python’s dependence on white space, deeper investigation of this language was not pursued.

### 8. Conclusions

Given sufficient time and money, tools can be developed so that any of the languages discussed could be used for the Smart Packets project. Given real world constraints, Perl 5 and Object/Light Caml seem to have the highest jumping off point with the least potential for future intractable problems.
9. References

[Ab+]

[Ch+]

[Flan]

[Ross]

[Stal]

[Tunes]

[W3C]

[Wall+]

[Weth]
## CONTENTS

1. Introduction ............................................................................................................. 1

2. Compact Representation .......................................................................................... 2
   2.1 Compression ....................................................................................................... 2
   2.2 Byte Codes ........................................................................................................ 2
   2.3 Programmer Burden ............................................................................................ 2
   2.4 Homespun Cruncher ............................................................................................ 2
   2.5 Op-code Language ............................................................................................... 3

3. Safety .......................................................................................................................... 3

4. Functions, Dynamic Linking and Foreign Function Interface ................................ 3

5. Compilers, Virtual Machines, Debuggers ............................................................... 4

6. Learnability and Maintainability ............................................................................... 4

7. Languages .................................................................................................................. 4
   7.1 Perl 5 .................................................................................................................. 4
   7.2 Scheme and Emacs Lisp ...................................................................................... 5
   7.3 Java ................................................................................................................... 6
   7.4 The Caml Family of Languages .......................................................................... 6
   7.5 SNMP Methodology ............................................................................................ 7
   7.6 Tcl and The Active IP Option ............................................................................. 8
   7.7 Python ................................................................................................................ 8

8. Conclusions ............................................................................................................... 8

9. References ............................................................................................................... 9