Dynamic Languages for Networked Computing

Brian D. Carlstrom '95

1 Introduction

Recently dynamic languages have seen a rise in popularity. There are many reasons behind this such as advances in scripting languages, increased speed from the hardware as well as implementation techniques, and a more extensible language design. Simultaneously, interest has grown in breaking out of the traditional data-oriented client-server model and moving toward a more computationally based model. This new model raises many different issues such as limiting access to computational resources, restricting access to primitives, and accountability. Different languages favor different solutions to these problems.

1.1 Scripting Languages

There has been an evolution from primitive scripting languages such as provided by typical Unix shells to kitchen sink monsters such as Perl that provide complete system interfaces. With Perl's success have come other languages each with their own flavor. TCL sees itself as more of an embedded language. Python emphasizes that it is object oriented. The Scheme Shell tries to integrate the Unix computational model with the functional programming style.

Increased usability from performance improvements has been another reason for increased popularity. Part of this increased performance is just a side effect of the continuing increase in the power of the desktop box. Use of more advanced compiler and virtual machine technologies in scripting languages has also helped. Fast prototyping has always been listed as a benefit of dynamic languages, but there was always a tradeoff with the efficiency of static language implementations. Todays computing resources means that for many applications, a simple dynamic language implementation may well serve the needs of the user.

Finally, today's dynamic languages are designed to allow easy extension via more traditional languages to allow static implementation technologies to be used when speed is a primary concern, but ease of development is still desired.

1.2 Computation over the Network

There are many issues associated with running potentially dangerous code from another source. Restrictions on computational resources in both time and space are necessary to prevent buggy or malicious programs from wasting system resources. Some code might only have access to a subset of the full system services or through special interfaces to limit foreign access to the system. In any real system, accounting becomes a serious component along with authentication and encryption.

2 Current Implementations

There are several current implementations of network enabled languages. Two proposals are based on popular freely available scripting languages: Safe-TCL and Safe-Python. There are also two commercially backed languages: Telescript and Java. None of these languages are completely specified. All are still in the development process.

2.1 Safe-TCL

TCL is the Tool Command Language by Dr. John Ousterhout. TCL is designed to be a common extension language for applications. It provides a simple interpreter that focuses on strings as a data type, making it easy to interface with a wide variety of traditional programming languages.

Safe-TCL is a version of TCL designed by Nathaniel S. Borenstein. It's original purpose was as an MIME extension for writing an enabled mail application. Safe-TCL will be merged back in with the core TCL system and used for general Internet scripting applications. The Safe-TCL system actually uses two TCL interpreters: one for interpreting trusted code and another for interpreting untrusted code. The untrusted interpreted can call primitives that are in fact implemented in the trusted interpreted as well as defined in C and other languages.

The Safe-TCL system represents programs as textual scripts. Although there are TCL compiler efforts, it is not clear if there will ever be a bytecode wire line representation for Safe-TCL. Safe-TCL seems to suit its original purpose for enabled mail, but it is unclear whether or not it makes a good general purpose language for doing agent like computation.

2.2 Safe-Python

Python is a popular interpreted language by Guido van Rossum. In many ways it is similar to TCL. For instance it provides relatively painless integration with C code and can serve as an embedded application language. On the other hand, Python has features one would only expect find in a real programming language such as objects, a package system, and exceptions. In general, Python does seem to be more of a general purpose programming language than TCL, but on the other hand TCL may have more momentum behind it.

Python is implemented with a bytecode interpreter. This means the script code is compiled into a assemble language like form which is then interpreted by a virtual machine. Python does not seem to have a way to store scripts in predigested form, so each time a script is executed it is compiled, although the system could probable be extended to support static compilation with dynamic linking.

Safe-Python is a proposal similar to Safe-TCL. It is still not generally available but is under development. Since Python also seems to use text based scripts it is probably safe to assume that Safe-Python will be using a text based wire line representation. Like the system itself, it would seem possible to extend the system with a bytecode wire line representation. This would allow remove the compile step from the machine performing the computation but possibly require some sandbox analysis as well as a link phase before running.

2.3 Telescript

Telescript is a language designed by General Magic explicitly for the purpose of networked agent based communication. Although not much is known about the details of the language, it sounds similar to Python in some respects. It apparently has a high level object oriented form known as High Telescript. This version of the language is compiled into a Postscript like form called Low Telescript. In the Telescript system, the interpreter is a stack style virtual machine.

Unfortunately, at this time not much is publicly available about Telescript. It has been proven to work in AT&T's PersonaLink service, which provides electronic mail, information services, and an electronic marketplace. From what is available, it seems like Telescript is on the right track technically but whether it will succeed in the market place remains to be seen.

2.4 Java

Java is a language system from Sun Microsystems. Although also from a corporate environment, it is more publicly documented than Telescript. It is also a serious programmer's language that is object oriented and multithreaded and also provides the portability, performance, and security needed for demanding network environments. Java is compiled to a bytecode interpreted virtual machine similar to Python and Telescript. In addition, the Java virtual machine language is designed for easy translation into native code which is supposedly comparable to that generated by a C/C++ compiler.

Java's primary application so far has been the HotJava World Wide Web browser. This has been an impressive application showing the functionality of Java language. The browser is extendable dynamicly to support new protocols and services as implemented through the Java language. It will be interesting to see what direction this project takes. Of the current implementations, it seems to be on the forefront in terms of demonstrated technology. The Java system seems like it would also provide a good test bed for agent oriented computation.

3 Analysis

Applying programming languages to problems in networked environment combines aspects from many computer science disciplines such as languages, compilers, networks, authentication, and encryption. Some of the interesting issues in looking at the current technology are the effects of wire line representation on security concerns and the benefits of the virtual machine architecture.

3.1 Safety At All Levels

The two primary wire line representation used have been plain text scripts and bytecode representations. For the former, presumably the interpreter's symbol table will only allow access to safe primitives and there is no way of describing unsafe computations as a result of the language design.

For the bytecode representation there are many issues similar to verifying the safety of native code. In general, direct access to system calls needs to be prevented. Pointer operations must be bounds checked. Function calls must be restricted to user defined functions and standard library functions. In general, because more semantic information is preserved in the bytecode, the verification of the bytecode representation is simplified. After the bytecode passes inspection, it can be translated into native code assuring both safety and efficiency.

One other important issue in either system is the evolution of interfaces. While interfaces do not evolve as rapidly as applications, they are by no means static. A versioning system is probably necessary to preserve backwards compatibility over time. That way applications and interfaces can continue to evolve and still interact in an predictable manner.

3.2 Virtual Machine Architecture

The virtual machine architecture is a unique implementation technique that provides some of the speed of compilation with the portability of an interpreter. Another virtual machine based system to take a look at that is commonly used is Scheme 48, an implementation of the R4RS Scheme standard with extensions such as a package system, threads, and objects. Daniel W.

Connolly of the World Wide Web Consortium made the following observation about the Scheme 48 virtual machine:

> "The [Scheme 48 virtual machine] seems to be very fast, and since it supports full continuations, it should be able to support any of the popular interpreted programming languages – Python, Perl, Icon, TCL, Smalltalk, etc. (as well as ML, Prolog, and other functional/logic languages), with suitable bytecode compilers and runtime libraries."

This provides an interesting perspective. Instead of a one language, one virtual machine philosophy, the virtual machine becomes the abstract interface for several languages. With appropriate native code translators on common platforms, the virtual machine architecture resembles the hourglass shape from the networking protocol world. We have one common piece that supports a large variety of software functionality on a wide variety of hardware platforms.

3.3 Future Directions

There are several directions that further research work can proceed from here. One possible area is to study the requirements of a virtual machines to efficiently support a wide variety of programming languages while still allowing straight forward translation to both CISC and RISC architectures. Another direction would be to take an existing system and add support for a new language above the virtual machine level. Alternatively, an existing compiler's back end could be modified to generate another system's virtual machine instruction set.

The Scheme 48 system seems like a good platform for exploring these ideas. There is already an effort to back end the existing Scheme 48 system to both CISC and RISC style assembly languages. Perhaps someone could meet Connolly's challenge and implement another language on top of the Scheme 48 virtual machine. Another interesting project would be to modify the Scheme 48 compiler to generate code for the Java runtime system.

The Scheme Shell already provides extensions to the basic Scheme 48 implementation for systems programming. A network toolkit has already been written in the Scheme Shell to provide many common network protocols and applications including an World Wide Web browser for the Apple Newton as well as a NCSA compatible HTTP server. While some preliminary work on agent protocols has been undertaken, there is still a lot of potential avenues for exploration.

4 Conclusion

There is a lot of interesting and varied work in the area of networked languages. On the one end of the spectrum is the simple text based freely available interpreter while on the other is a corporate project that supports a compiled environment with native code support. However, there is still room for further development. The virtual machine architecture favored by the more serious language implementations lends itself toward a multitude of languages and platforms, few of which have been explored. This topic should remain an active research area for some time to come.

5 References

For more information on TCL:

- "Tcl: An Embeddable Command Language" John K. Ousterhout, 1990 Winter USENIX Conference Proceedings.
- "EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail" Nathaniel S. Borenstein, submitted to ULPAA '94.

For more information on Python:

- "FAQ: Python – an object-oriented language" `ftp://ftp.cwi.nl/pub/python/doc/FAQ`
- Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283-303.
- "Distributed Programming: Agentware, Componentware, and Distributed Objects" by Steven D. Majewski `http://minsky.med.virginia.edu/sdm7g/Projects/Python/`

For more information on Telescript:

- “The world of Telescript technology”, General Magic, 1994.
- Personal communication with Warren Harris `<warren@genmagic.com>`
- Telescript info originally from Safe-TCL mailing list on Safe-Python WWW Page
  `http://minsky.med.virginia.edu/sdm7g/Projects/Python/TeleScript/`

For more information on Java:

- “The Java Language: A White Paper”
  `http://java.sun.com/1.0alpha2/doc/overview/java/`
- “The Java Language Specification”
  `http://java.sun.com/1.0alpha2/doc/javaspec/javaspec_1.html`
- “The Java Virtual Machine Specification”
  `http://java.sun.com/1.0alpha2/doc/vmspec/vmspec_1.html`
- “HotJava White Paper”
  `http://java.sun.com/1.0alpha2/doc/overview/hotjava/`

For more information on Scheme 48:

- “Scheme 48 Home Page”
  `http://www-swiss.ai.mit.edu/ jar/s48.html`
- “A Tractable Scheme Implementation”
  `http:/ftpdir/users/jar/lsc.ps`
- “Scheme48 For Collaborative Engineering?” by Daniel W. Connolly
  `http://www-swiss.ai.mit.edu/ jar/connolly-thoughts.html`

For more information on the Scheme Shell:

- “A Scheme shell” by Olin Shivers. To appear in the *Journal of Lisp and Symbolic Computation*. Also available as MIT LCS Technical Report TR-635.
  `ftp://clark.lcs.mit.edu/pub/shivers/scsh.ps`
- “The scsh manual” by Olin Shivers and Brian D. Carlstrom. December 1994, scsh release 0.3.
  `ftp://www-swiss.ai.mit.edu/pub/su/scsh/scsh-manual.ps`