Rapid Development Using Python

SkipWare is a series of extensions to standard Internet transport protocols that, when installed on a network device, maximizes link utilization and reliability in a stressed transmission environment. It has been developed collaboratively between GST and Comtech: GST provides the software and Comtech provides the hardware. Although SkipWare is an implementation of SCPS (Space Communication Protocol Standards), a graphical user interface is required to specify configuration settings that impact overall system performance. Given that the product forwards IP (Internet) traffic, a web-based client was the most feasible interface for our customers.

Because the interface requirements were soft and time was short, we approached the interface with a rapid development mentality centered around constant customer feedback. A language decision had to be made, and we had the following choices (primarily due to our experience): C, PHP, Perl, Java, Python. Our selection was based upon the following criteria:

- We planned to prototype on a remote device and anticipated numerous changes. We needed a language that was designed with change in mind.
- We wanted to avoid the added step of code compilation in order to minimize the overhead associated with a change. An interpreted language seemed very pragmatic.
- We wanted a language with good introspection capability.
- We needed to do lots of string manipulation and file I/O. Whatever language we chose had to excel in both of these areas.

Disk space was a concern but not a driving force. The hardware provided to us was an 800Mhz processor with 64Mb RAM and a 32Mb root filesystem. Of those characteristics, the disk space was clearly the most precious. Because of the high clock speed and availability of RAM, interface performance was not a driving force.

Rapid Development

Our development environment focused on customer interaction. Because time was short, the traditional approach (formal requirements gathering followed by independent development and testing) was not practical. We anticipated frequently making changes on a remote device with a customer representative viewing the interface and providing instant feedback.

Python allowed us to achieve this environment primarily because it is easy to use interactively. Prototyping through an interactive interpreter is an effective mechanism for exploring different approaches to solving a problem. With a customer representative providing real-time feedback, we needed a language where we could instantly prototype a function through an interactive interpreter and then copy the working function to the remote device for approval. Perl did not provide an interactive interpreter that enabled this behavior.

Compilation
Our rapid development environment meant that changes had to be visible immediately by both developer and customer representative. Coding sessions would frequently involve work on a remote device during which time changes would be made and feedback would be gathered. Use of a compiled language inhibited our ability to prototype on a remote device because it required maintaining a build environment. In the scenario where we changed one or two lines of code on the remote device during an interactive feedback session, we did not want to have to wait for compilation before gathering the next round of feedback. IDEs were not feasible as a solution to our problem because much of our development occurred on the remote device.

Introspection
We wanted to dispatch web requests to code in a very direct fashion. An architecture based upon a central controller that used introspection to determine where to route requests seemed clean and simple. Because our controller-based architecture was abstract and generic, we had concerns regarding safety. Python protects the programmer from buffer overflows and has excellent reflection capabilities that can be explored at run-time. Both characteristics satisfied our needs for introspection and safety. Moreover, an inspectable run-time is nearly equivalent to running an application under a debugger while make changes. This characteristic appealed to us.

String Manipulation and File I/O
All user input arrived in string format. All output was stored in files or returned to the user in HTML format. Both of these concepts are well supported in Python. Iterating over the content of a file takes 2 lines of Python. In comparison, Java requires 4 or 5 instantiations followed by 2 or 3 lines to read. Once read into memory, content must be tokenized before iteration. While iterating, casting is required. The Java approach exemplifies the importance of selecting the appropriate language for your development environment – it would not have been a good language choice for the tasks we were to perform (primarily string and file manipulation). With all of its built in types and excellent string and file manipulation, Python provided us with the tools we needed to accomplish our tasks quickly and easily (more on this later).

Narrowing down the language choices was not particularly difficult. C requires compilation, is easy to exploit via buffer overflows (and thus poses a security risk), and does not handle strings and file I/O particularly well. PHP was also dismissed because we were not accessing a database and hence many of the features of the language would not be used. Java is too bloated, requires compilation, and has horrible file I/O and string manipulation.

Python survived our language elimination process because it satisfied all of our requirements. In addition to satisfying our requirements, we gravitated towards Python due to its modularity and support for objects. This allows structured code growth and accommodates change over time. We also welcomed the safety (over C) that the interpreter provides. However, above all else, we selected Python as our language

because it supported rapid development (and our specific requirements) extremely well and no other language came close in comparison. While other languages fulfilled a subset of our requirements, no language we considered fulfilled all of the requirements better than Python.

Development

Our hardware vendor provided a 32Mb solid state device to use as a root filesystem. With the kernel, operating system, and the SkipWare binaries occupying approximately 16Mb, we had to be crafty in determining our runtime environment. We initially looked to Boa as a lightweight web server, but were concerned about our ability to scale it *down* to fit within 10Mb. We feared that we would end up having to convert our Python code to C for deployment on the production systems, and none of us looked forward to the port. None-the-less, the quick progress and huge strides we made as a result of using Python convinced us to move forward with the language until such time that we were forced to recode.

We looked to "freeze" as an option to avoid a re-write. Freeze seemed to offer us the best of both worlds: it allowed us to develop our interface in Python, statically link it against the interpreter, and produce an executable binary we could load onto the production hardware. Unfortunately, scope creep forced us to eliminate "freeze" as a viable option. During the development process, unforeseen requirements were created at the last minute. In the end, we ended up building an interface well beyond the original set of requirements. Our architecture supported the unanticipated creep, but class organization became difficult.

For simplicity, our architecture initially consisted of one module containing several classes (views). Over time, we found that separating the classes into modules enabled a plug-able overall architecture. Because it supports import-on-demand and module reloading through introspection, Python allowed us to change logic within external modules and have the changes immediately accessible by the central controller. Overall system architecture became significantly cleaner (and simpler) when we shifted to using multiple modules. Unfortunately, this meant that "freeze" was no longer a viable option because we would have to statically link every individual module and that would require significant disk space.

As described above, we witnessed scope creep during development. What started out as a simple interface to control IP Address, Subnet Mask, Routes, and Round Trip Time turned into a full fledged dynamic configuration wizard. Moreover, we found ourselves building an "Auto-Update" feature as well as a management interface that displayed uptime, traffic statistics, and intelligent displays of routes and acceleration. All this complex functionality would now have to be ported to another language at the last minute.

Recognizing the difficulty of a last minute port, we decided to load the Python interpreter onto the production hardware. This was a difficult and risky decision to make because we only had 16Mb of free space, and the interpreter consumed a fairly large percentage

of the remaining disk space. However, during the prototyping process, we often remarked to each other: “this code is *not* going to port well to *language X*”. We worried about the many times we uttered that phrase, and dreaded the upcoming port. The unknown risks associated with porting to another language compelled us to continue using Python with the assumption that it would be available on the production hardware.

In retrospect, installing Python turned out to be a wise *investment.* Even though the interpreter and libraries consumed over 5Mb of valuable disk resource, the flexibility and robustness of the language led to numerous speed-ups during development. The string manipulation and file I/O commands became trivial operations. Instead of wasting time fighting with the language, we spent time fulfilling customer requirements. Our rapid development environment blazed to life fueled by Python.

The Pythonic Approach

We found that looking at problems “from a pythonic standpoint” often led to simple and elegant solutions that addressed both functionality and portability. In on instance we needed to compare two configuration files for equality. Our first solution was to use the “diff” command. However, due to time and space limitations, we were unable to load the “diff” onto the production hardware. We looked to Python and implemented a solution within 10 minutes: create two dictionaries (one from each file) and compare them using the != operator. The “pythonic approach” to our problem was pragmatic and very easy to implement. This trend was common throughout the development process.

With Python available on the production hardware, our capabilities have grown vertically and horizontally. Our interface continues to evolve through re-factoring, and we have also written a guardian to perform sanity checks on routes, arp entries, and network connectivity. It does so by harvesting information from the proc filesystem but also includes non-proc oriented functionality that prevents software theft. Implementation of the nanny within another language (presumably C) would require significantly more code and would also be more difficult to maintain.

Python Evangelism

The appeal of Python goes well beyond its functionality and usability. The user community and external module support is superb. FTP transfers under Python are extremely easy via the ftplib module. Simple checksums are painless via the p2 module. We have used both of the above libraries within our application and have found them considerably more intuitive than their Java or C counterparts. After our experience with both syntax and external libraries, many of us frequently ask why other languages do it “any other way”.

Python’s built-in types immediately come to mind. File access and iteration is extremely easy:

```
for line in open('file.txt').read().split('\n'):
      for word in line.split():
            words[word] = words.get(word, 0) + 1
```

The above demonstrates iteration over a file followed by iteration of individual words per line. It concludes by showing how a bucket in a dictionary can be referenced explicitly via brackets or through a method. Conversion between built-in types is also easy. In this example, tuples are used to construct a dictionary representing the values in a configuration file:

```
for line in open('settings.conf').read().split('\n'):
      if line.strip() == '': continue
      (name, value) = line.split(delimiter)
      props[name] = value
```

The above illustrates how a list can automatically be converted to a tuple. The variables in the tuple can then be used as inputs to other variables. After executing this code, the developer will have a dictionary where the key is the name configuration setting and the bucket contains the value of the configuration setting.

We have also marveled at the simplicity of documenting our Python modules to conform to pydoc specifications. We were initially apprehensive about embedding our method documentation below the method signature. This seemed counter-intuitive to us (some of us came from a Java background), but after doing it a couple of times we became fanatical about it. Unlike Java (where comments are above the method signature), it is extremely easy to locate a method within a module because the function signature (what you are searching for) stands out and is not concealed by documentation above and code below. Here is an example of a JavaDoc comment and method:

```
/**
 * Returns true if a equals b, false otherwise
 * /
public boolean equals(Object a, Object b) {
  if (a instanceof b.getClass()) {
    …
  }
}
```

If you are a Java developer looking for the "equals" method that returns boolean and accepts two Objects, you are either going to have to use an IDE or visually scan for the signature. It will not be easy to find because the one line you are looking for is surrounded by a significant amount of noise: javadoc above and code below. PyDoc (below), keeps the noise ratio low by embedding the documentation below the function declaration:

```
def equals(a, b):
'''returns true if a equals b, false otherwise'''
if type(a) == type(b):
      …
```

Looking for the "equals" method under Python is significantly easier that in Java.

Conclusion

Our interest in Python was spawned from a prototyping standpoint, but we have quickly adopted it as the language of choice for our management interface as well as for ad-hoc

prototyping of future concepts. Numerous developers have quickly come up to speed with Python and have expressed pleasant surprise at the simplicity of the language and how quickly they are able to fulfill functional requirements. Moreover, Python's robust with string and file handling (as well as it's built in types) make it the language of choice for our user interface. We're quite excited about the potential that Python has given us and are happy to be using such a powerful language to solve customer problems.

In the future, Python will be looked upon favorably by both the development staff as well as the company at large because it as allowed us to respond to customer requests in a fast paced rapid development environment. The built in types, lack of compilation, interative interpreter, and usability of Python have made it the language of choice of many, and have raised the bar of excellence for others when evaluating other languages.