Viability of Twisted and Python for Application Server Herd Implementations

Viet T. Nguyen
University of California, Los Angeles

Abstract

This paper provides a cursory look into the use of the Twisted framework, built on top of the Python language, to build application server herds with particular interest to language considerations such as type checking, memory management, and multithreading.

1 Introduction

The motivation behind this look into the development of an application server herd framework comes from a desire at our company to build a "new Wikimedia-style service designed for news." Due to differing requirements we sought an alternative to Wikimedia's LAMP platform (GNU/Linux, Apache, MySQL, and PHP). Our service intends to provide much more frequent updates, access via a variety of protocols, and a robustness towards increasing mobile traffic share.

The requirements outlined for the service result in a number of high level technical desires. To avoid application server bottlenecks we intend to distribute the application servers. With distributed servers we require rapid diffusion of unstructured data via a flooding algorithm. Beyond data diffusion there are a number of "-ility's" we hope to achieve: maintainability, reliability, scalability, parallelizability. Of course from a cost perspective we would also like ease of implementation.

2 Twisted

One of the many options available to implement such a framework is Twisted. Twisted is an event-driven networking engine written in Python [2]. Twisted shows promise because of its event driven, multi-process nature.

3 This is Another Section

Some embedded literal typset code might look like the following : [1]

```
int wrap_fact(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1o]);
    result = fact(arg0);
    sprintf(interp->result,"%d",result);
    return TCL_OK;
}
```

Now we're going to cite somebody. Watch for the cite tag. Here it comes [**?**, **?**]. The tilde character (~) in the source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

4 This Section has SubSections

4.1 First SubSection

Here's a typical figure reference. The figure is centered at the top of the column. It's scaled. It's explicitly placed. You'll have to tweak the numbers to get what you want.

This text came after the figure, so we'll casually refer to Figure 1 as we go on our merry way.

Figure 1: Wonderful Flowchart

4.2 New Subsection

It can get tricky typesetting Tcl and C code in LaTeX because they share a lot of mystical feelings about certain magic characters. You will have to do a lot of escaping to typeset curly braces and percent signs, for example, like this: "The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{, %}` block is copied directly into the output. allowing the inclusion of header files and additional C code."

Sometimes you want to really call attention to a piece of text. You can center it in the column like this:

`_1008e614_Vector_p`

and people will really notice it.

The noindent at the start of this paragraph makes it clear that it's a continuation of the preceding text, not a new para in its own right.

Now this is an ingenious way to get a forced space. `Real *` and `double *` are equivalent.

Now here is another way to call attention to a line of code, but instead of centering it, we noindent and bold it.

```
size_t :  fread ptr size nobj stream
```

And here we have made an indented para like a definition tag (dt) in HTML. You don't need a surrounding list macro pair.

> `fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read.

This concludes the definitions tag.

4.3 How to Build Your Paper

You have to run `latex` once to prepare your references for munging. Then run `bibtex` to build your bibliography metadata. Then run `latex` twice to ensure all references have been resolved. If your source file is called `usenixTemplate.tex` and your `bibtex` file is called `usenixTemplate.bib`, here's what you do:

```
latex usenixTemplate
bibtex usenixTemplate
latex usenixTemplate
latex usenixTemplate
```

4.4 Last SubSection

Well, it's getting boring isn't it. This is the last subsection before we wrap it up.

5 Acknowledgments

A polite author always includes acknowledgments. Thank everyone, especially those who funded the work.

6 Availability

It's great when this section says that MyWonderfulApp is free software, available via anonymous FTP from

`ftp.site.dom/pub/myname/Wonderful`

Also, it's even greater when you can write that information is also available on the Wonderful homepage at

`http://www.site.dom/~myname/SWIG`

Now we get serious and fill in those references. Remember you will have to run latex twice on the document in order to resolve those cite tags you met earlier. This is where they get resolved. We've preserved some real ones in addition to the template-speak. After the bibliography you are DONE.

References

[1] INC., F. Phage lambda: description & restriction map. `http://www.example.com`, November 2008.

[2] LABS, T. M. Twisted. `http://twistedmatrix.com/trac/`, November 2012.