CHAPTER 7

Server Architecture

This chapter explores how network programming intersects with the general tools and techniques that Python developers use to write long-running daemons that can perform significant amounts of work by keeping a computer and its processors busy.

Instead of making you read through this entire chapter to learn the landscape of design options that I will explore, let me outline them quickly.

Most of the network programs in this book—and certainly all of the ones you have seen so far—use a single sequence of Python instructions to serve one network client at a time, operating in lockstep as requests come in and responses go out. This, as we will see, will usually leave the system CPU mostly idle.

There are two changes you can make to a network program to improve this situation, and then a third, big change that you can make outside your program that will allow it to scale even further.

The two changes you can make to your program are either to rewrite it in an event-driven style that can accept several client connections at once and then answer whichever one is ready for an answer next, or to run several copies of your single-client server in separate threads or processes. An event-driven style does not impose the expense of operating system context switches, but, on the other hand, it can saturate at most only one CPU, whereas multiple threads or processes—and, with Python, especially processes—can keep all of your CPU cores busy handling client requests.

But once you have crafted your server so that it keeps a single machine perfectly busy answering clients, the only direction in which you can expand is to balance the load of incoming connections across several different machines, or even across data centers. Some large Internet services do this with proxy devices sitting in front of their server racks; others use DNS round-robin, or nameservers that direct clients to servers in the same geographic location; and we will briefly discuss both approaches later in this chapter.

Daemons and Logging

Part of the task of writing a network daemon is, obviously, the part where you write the program as a daemon rather than as an interactive or command-line tool. Although this chapter will focus heavily on the “network” part of the task, a few words about general daemon programming seem to be in order.

First, you should realize that creating a daemon is a bit tricky and can involve a dozen or so lines of code to get completely correct. And that estimate assumes a POSIX operating system; under Windows, to judge from the code I have seen, it is even more difficult to write what is called a “Windows service” that has to be listed in the system registry before it can even run.

On POSIX systems, rather than cutting and pasting code from a web site, I encourage you to use a good Python library to make your server a daemon. The official purpose of becoming a daemon, by the way, is so that your server can run independently of the terminal window and user session that were used to launch it. One approach toward running a service as a daemon—the one, in fact, that I myself prefer—is to write a completely normal Python program and then use Chris McDonough’s supervisord daemon to start and monitor your service. It can even do things like re-start your program if it should die, but then give up if several re-starts happen too quickly; it is a powerful tool, and worth a good long look: http://supervisord.org/.
You can also install python-daemon from the Package Index (a module named daemon will become part of the Standard Library in Python 3.2), and its code will let your server program become a daemon entirely on its own power.

If you are running under supervisord, then your standard output and error can be saved as rotated log files, but otherwise you will have to make some provision of your own for writing logs. The most important piece of advice that I can give in that case is to avoid the ancient syslog Python module, and use the modern logging module, which can write to syslog, files, network sockets, or anything in between. The simplest pattern is to place something like this at the top of each of your daemon’s source files:

```python
import logging
log = logging.getLogger(__name__)
```

Then your code can generate messages very simply:

```python
log.error('the system is down')
```

This will, for example, induce a module that you have written that is named serv.inet to produce log messages under its own name, which users can filter either by writing a specific serv.inet handler, or a broader serv handler, or simply by writing a top-level rule for what happens to all log messages. And if you use the logger module method named fileConfig() to optionally read in a logging.conf provided by your users, then you can leave the choice up to them about which messages they want recorded where. Providing a file with reasonable defaults is a good way to get them started.

For information on how to get your network server program to start automatically when the system comes up and shut down cleanly when your computer halts, check your operating system documentation; on POSIX systems, start by reading the documentation surrounding your operating system’s chosen implementation of the “init scripts” subsystem.

### Our Example: Sir Launcelot

I have designed a very simple network service to illustrate this chapter so that the details of the actual protocol do not get in the way of explaining the server architectures. In this minimalist protocol, the client opens a socket, sends across one of the three questions asked of Sir Launcelot at the Bridge of Death in Monty Python’s *Holy Grail* movie, and then terminates the message with a question mark:

```plaintext
What is your name?
```

The server replies by sending back the appropriate answer, which always ends with a period:

```plaintext
My name is Sir Launcelot of Camelot.
```

Both question and answer are encoded as ASCII.

Listing 7–1 defines two constants and two functions that will be very helpful in keeping our subsequent program listings short. It defines the port number we will be using; a list of question-answer pairs; a recv_until() function that keeps reading data from a network socket until it sees a particular piece of punctuation (or any character, really, but we will always use it with either the '. ' or '?' character); and a setup() function that creates the server socket.

**Listing 7–1. Constants and Functions for the Launcelot Protocol**

```python
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 7 - launcelot.py
# Constants and routines for supporting a certain network conversation.

import socket, sys
```