This chapter, though brief, might be one of the most important in this book. It surveys the handful of technologies that have together become fundamental building blocks for expanding applications to Internet scale.

In the following pages, this book reaches its turning point. The previous chapters have explored the sockets API and how Python can use the primitive IP network operations to build communication channels. All of the subsequent chapters, as you will see if you peek ahead, are about very particular protocols built atop sockets—about how to fetch web documents, send e-mails, and connect to server command lines.

What sets apart the tools that we will be looking at here? They have several characteristics:

- Each of these technologies is popular because it is a powerful tool. The point of using Memcached or a message queue is that it is a very well-written service that will solve a particular problem for you—not because it implements an interesting protocol that different organizations are likely to use to communicate.

- The problems solved by these tools tend to be internal to an organization. You often cannot tell from outside which caches, queues, and load distribution tools are being used to power a particular web site.

- While protocols like HTTP and SMTP were built with specific payloads in mind—hypertext documents and e-mail messages, respectively—caches and message queues tend to be completely agnostic about the data that they carry for you.

This chapter is not intended to be a manual for any of these technologies, nor will code examples be plentiful. Ample documentation for each of the libraries mentioned exists online, and for the more popular ones, you can even find entire books that have been written about them. Instead, this chapter’s purpose is to introduce you to the problem that each tool solves; explain how to use the service to address that issue; and give a few hints about using the tool from Python.

After all, the greatest challenge that a programmer often faces—aside from the basic, lifelong process of learning to program itself—is knowing that a solution exists. We are inveterate inventors of wheels that already exist, had we only known it. Think of this chapter as offering you a few wheels in the hopes that you can avoid hewing them yourself.
Using Memcached

Memcached is the “memory cache daemon.” Its impact on many large Internet services has been, by all accounts, revolutionary. After glancing at how to use it from Python, we will discuss its implementation, which will teach us about a very important modern network concept called *sharding*.

The actual procedures for using Memcached are designed to be very simple:

- You run a Memcached daemon on every server with some spare memory.
- You make a list of the IP address and port numbers of your new Memcached daemons, and distribute this list to all of the clients that will be using the cache.
- Your client programs now have access to an organization-wide blazing-fast key-value cache that acts something like a big Python dictionary that all of your servers can share. The cache operates on an LRU (least-recently-used) basis, dropping old items that have not been accessed for a while so that it has room to both accept new entries and keep records that are being frequently accessed.

Enough Python clients are currently listed for Memcached that I had better just send you to the page that lists them, rather than try to review them here: http://code.google.com/p/memcached/wiki/Clients.

The client that they list first is written in pure Python, and therefore will not need to compile against any libraries. It should install quite cleanly into a virtual environment (see Chapter 1), thanks to being available on the Python Package Index:

```
$ pip install python-memcached
```

The interface is straightforward. Though you might have expected an interface that more strongly resembles a Python dictionary with native methods like `__getitem__`, the author of python-memcached chose instead to use the same method names as are used in other languages supported by Memcached—which I think was a good decision, since it makes it easier to translate Memcached examples into Python:

```python
>>> import memcache
>>> mc = memcache.Client(['127.0.0.1:11211'])
>>> mc.set('user:19', '{name: "Lancelot", quest: "Grail"}')
True
>>> mc.get('user:19')
'{name: "Lancelot", quest: "Grail"}'
```

The basic pattern by which Memcached is used from Python is shown in Listing 8–1. Before embarking on an (artificially) expensive operation, it checks Memcached to see whether the answer is already present. If so, then the answer can be returned immediately; if not, then it is computed and stored in the cache before being returned.

**Listing 8–1. Constants and Functions for the Lancelot Protocol**

```
#!/usr/bin/env python
# Foundations of Python Network Programming - Chapter 8 - squares.py
# Using memcached to cache expensive results.

import memcache, random, time, timeit
mc = memcache.Client(['127.0.0.1:11211'])

def compute_square(n):
    value = mc.get('sq:%d' % n)
    if value is None:
```