GOVERNMENT ARTS AND SCIENCE COLLEGE, KOMARAPALAYAM

DEPARTMENT OF COMPUTER SCIENCE

COURSE	: M.Sc (COMPUTER SCIENCE)
SEMESTER	: III
SUBJECT NAME	: OPEN SOURCE COMPUTING
SUBJECT CODE	: 19PCS09
HANDLED BY	: Dr.V.KARTHIKEYANI
	ASSISTANT PROFESSOR
	DEPARTMENT OF COMPUTER SCIENCE
	GOVERNMENT ARTS AND SCIENCE COLLEGE
	KOMARAPALAYAM – 638 183

Scanned with CamScanner

UNIT-V

Concurrency: Queues – Processes – Threads – Green Threads and gevent – twisted – Redis. **Networks:** Patterns – The Publish-Subscribe Model – TCP/IP – Sockets – ZeroMQ –Internet Services – Web Services and APIs – Remote Processing – Big Fat Data and MapReduce – Working in the Clouds.

Scanned with CamScanner

Concurrency and Networks

UNIT 5

> Time is nature's way of keeping everything from happening at once. Space is what prevents everything from happening to me.
>
> — Quotes about Time

So far, most of the programs that you've written run in one place (a single machine) and one line at a time (*sequential*). But, we can do more than one thing at a time (*concurrency*) and in more than one place (*distributed computing* or *networking*). There are many good reasons to challenge time and space:

Performance
: Your goal is to keep fast components busy, not waiting for slow ones.

Robustness
: There's safety in numbers, so you want to duplicate tasks to work around hardware and software failures.

Simplicity
: It's best practice to break complex tasks into many little ones that are easier to create, understand, and fix.

Communication
: It's just plain fun to send your footloose bytes to distant places, and bring friends back with them.

We'll start with concurrency, first building on the non-networking techniques that are described in Chapter 10—processes and threads. Then we'll look at other approaches, such as callbacks, green threads, and coroutines. Finally, we'll arrive at networking, initially as a concurrency technique, and then spreading outward.

Scanned with CamScanner
Scanned with CamScanner

Some Python packages discussed in this chapter were not yet ported to Python 3 when this was written. In many cases, I'll show example code that would need to be run with a Python 2 interpreter, which we're calling python2.

Concurrency

The official Python site discusses concurrency in general and in the standard library. Those pages have many links to various packages and techniques; we'll show the most useful ones in this chapter.

In computers, if you're waiting for something, it's usually for one of two reasons:

I/O bound
: This is by far more common. Computer CPUs are ridiculously fast—hundreds of times faster than computer memory and many thousands of times faster than disks or networks.

CPU bound
: This happens with *number crunching* tasks such as scientific or graphic calculations.

Two more terms are related to concurrency:

synchronous
: One thing follows the other, like a funeral procession.

asynchronous
: Tasks are independent, like party-goers dropping in and tearing off in separate cars.

As you progress from simple systems and tasks to real-life problems, you'll need at some point to deal with concurrency. Consider a website, for example. You can usually provide static and dynamic pages to web clients fairly quickly. A fraction of a second is considered interactive, but if the display or interaction takes longer, people become impatient. Tests by companies such as Google and Amazon showed that traffic drops off quickly if the page loads even a little slower.

But what if you can't help it when something takes a long time, such as uploading a file, resizing an image, or querying a database? You can't do it within your synchronous web server code anymore, because someone's waiting.

On a single machine, if you want to perform multiple tasks as fast as possible, you want to make them independent. Slow tasks shouldn't block all the others.

"Programs and Processes" on page 247 demonstrates how multiprocessing can be used to overlap work on a single machine. If you needed to resize an image, your web server code could call a separate, dedicated image-resizing process to run asynchronously and

Scanned with CamScanner
Scanned with CamScanner

concurrently. It could scale your application horizontally by invoking multiple resizing processes.

The trick is getting them all to work with one another. Any shared control or state means that there will be bottlenecks. An even bigger trick is dealing with failures, because concurrent computing is harder than regular computing. Many more things can go wrong, and your odds of end-to-end success are lower.

All right. What methods can help you to deal with these complexities? Let's begin with a good way to manage multiple tasks: *queues*.

Queues

A queue is like a list: things are added at one end and taken away from the other. The most common is referred to as *FIFO* (first in, first out).

Suppose that you're washing dishes. If you're stuck with the entire job, you need to wash each dish, dry it, and put it away. You can do this in a number of ways. You might wash the first dish, dry it, and then put it away. You then repeat with the second dish, and so on. Or, you might *batch* operations and wash all the dishes, dry them all, and then put them away; this assumes you have space in your sink and drainer for all the dishes that accumulate at each step. These are all synchronous approaches—one worker, one thing at a time.

As an alternative, you could get a helper or two. If you're the washer, you can hand each cleaned dish to the dryer, who hands each dried dish to the put-away-er (look it up; it's absolutely a real word!). As long as each of you works at the same pace, you should finish much faster than by yourself.

However, what if you wash faster than the dryer dries? Wet dishes either fall on the floor, or you pile them up between you and the dryer, or you just whistle off-key until the dryer is ready. And if the last person is slower than the dryer, dry dishes can end up falling on the floor, or piling up, or the dryer does the whistling. You have multiple workers, but the overall task is still synchronous and can proceed only as fast as the slowest worker.

Many hands make light work, goes the old saying (I always thought it was Amish, because it makes me think of barn building). Adding workers can build a barn, or do the dishes, faster. This involves *queues*.

In general, queues transport *messages*, which can be any kind of information. In this case, we're interested in queues for distributed task management, also known as *work queues*, *job queues*, or *task queues*. Each dish in the sink is given to an available washer, who washes and hands it off to the first available dryer, who dries and hands it to a put-away-er. This can be synchronous (workers wait for a dish to handle and another worker to whom to give it), or asynchronous (dishes are stacked between workers with different

Scanned with CamScanner
Scanned with CamScanner

paces). As long as you have enough workers, and they keep up with the dishes, things move a lot faster.

Processes

You can implement queues in many ways. For a single machine, the standard library's `multiprocessing` module (which you can see in "Programs and Processes" on page 247) contains a Queue function. Let's simulate just a single washer and multiple dryer processes (someone can put the dishes away later) and an intermediate `dish_queue`. Call this program *dishes.py*:

```
import multiprocessing as mp

def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)

def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
        input.task_done()

dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Run your new program thusly:

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
Drying bread dish
Drying entree dish
Drying dessert dish
```

This queue looked a lot like a simple Python iterator, producing a series of dishes. It actually started up separate processes along with the communication between the washer and dryer. I used a `JoinableQueue` and the final `join()` method to let the washer know that all the dishes have been dried. There are other queue types in the `multiprocessing` module, and you can read the documentation for more examples.

Scanned with CamScanner
Scanned with CamScanner

Threads

A *thread* runs within a process with access to everything in the process, similar to a multiple personality. The `multiprocessing` module has a cousin called `threading` that uses threads instead of processes (actually, `multiprocessing` was designed later as its process-based counterpart). Let's redo our process example with threads:

```
import threading

def do_this(what):
    whoami(what)

def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

Here's what prints for me:

```
Thread <_MainThread(MainThread, started [illegible])> says: I'm the main
program
Thread <Thread(Thread-1, started [illegible])> says: I'm function 0
Thread <Thread(Thread-2, started [illegible])> says: I'm function 1
Thread <Thread(Thread-3, started [illegible])> says: I'm function 2
Thread <Thread(Thread-4, started [illegible])> says: I'm function 3
```

We can reproduce our process-based dish example by using threads:

```
import threading, queue
import time

def washer(dishes, dish_queue):
    for dish in dishes:
    print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)

def dryer(dish_queue):
    while True:
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
        dish_queue.task_done()


dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
```


Scanned with CamScanner
Scanned with CamScanner

```
        dryer_thread.start()

    dishes = ['salad', 'bread', 'entree', 'desert']
    washer(dishes, dish_queue)
    dish_queue.join()
```

One difference between `multiprocessing` and `threading` is that `threading` does not have a `terminate()` function. There's no easy way to terminate a running thread, because it can cause all sorts of problems in your code, and possibly in the space-time continuum itself.

Threads can be dangerous. Like manual memory management in languages such as C and C++, they can cause bugs that are extremely hard to find, let alone fix. To use threads, all the code in the program—and in external libraries that it uses—must be *thread-safe*. In the preceding example code, the threads didn't share any global variables, so they could run independently without breaking anything.

Imagine that you're a paranormal investigator in a haunted house. Ghosts roam the halls, but none are aware of the others, and at any time, any of them can view, add, remove, or move any of the house's contents.

You're walking apprehensively through the house, taking readings with your impressive instruments. Suddenly you notice that the candlestick you passed seconds ago is now missing.

The contents of the house are like the variables in a program. The ghosts are threads in a process (the house). If the ghosts only looked at the house's contents, there would be no problem. It's like a thread reading the value of a constant or variable without trying to change it.

Yet, some unseen entity could grab your flashlight, blow cold air down your neck, put marbles on the stairs, or make the fireplace come ablaze. The *really* subtle ghosts would change things in other rooms that you might never notice.

Despite your fancy instruments, you'd have a very hard time figuring out who did it, and how, and when.

If you used multiple processes instead of threads, it would be like having a number of houses but with only one (living) person in each. If you put your brandy in front of the fireplace, it would still be there an hour later. Some lost to evaporation, perhaps, but in the same place.

Threads can be useful and safe when global data is not involved. In particular, threads are useful for saving time while waiting for some I/O operation to complete. In these cases, they don't have to fight over data, because each has completely separate variables.

Scanned with CamScanner
Scanned with CamScanner

But threads do sometimes have good reasons to change global data. In fact, one common reason to launch multiple threads is to let them divide up the work on some data, so a certain degree of change to the data is expected.

The usual way to share data safely is to apply a software *lock* before modifying a variable in a thread. This keeps the other threads out while the change is made. It's like having a Ghostbuster guard the room you want to remain unhaunted. The trick, though, is that you need to remember to unlock it. Plus, locks can be nested—what if another Ghostbuster is also watching the same room, or the house itself? The use of locks is traditional but notoriously hard to get right.

In Python, threads do not speed up CPU-bound tasks because of an implementation detail in the standard Python system called the *Global Interpreter Lock* (GIL). This exists to avoid threading problems in the Python interpreter, and can actually make a multithreaded program slower than its single-threaded counterpart, or even a multi-process version.

So for Python, the recommendations are as follows:

- Use threads for I/O bound problems
- Use processes, networking, or events (discussed in the next section) for CPU-bound problems

Green Threads and gevent

As you've seen, developers traditionally avoid slow spots in programs by running them in separate threads or processes. The Apache web server is an example of this design.

One alternative is *event-based* programming. An event-based program runs a central *event loop*, doles out any tasks, and repeats the loop. The nginx web server follows this design, and is generally faster than Apache.

The gevent library is event-based and accomplishes a cool trick: you write normal imperative code, and it magically converts pieces to *coroutines*. These are like generators that can communicate with one another and keep track of where they are. gevent modifies many of Python's standard objects such as socket to use its mechanism instead of blocking. This does not work with Python add-in code that was written in C, as some database drivers are.

Scanned with CamScanner
Scanned with CamScanner

As of this writing, gevent was not completely ported to Python 3, so these code examples use the Python 2 tools pip2 and python2.

You install gevent by using the Python 2 version of pip:

```
$ pip2 install gevent
```

Here's a variation of sample code at the gevent website. You'll see the socket module's gethostbyname() function in the upcoming DNS section. This function is synchronous, so you wait (possibly many seconds) while it chases name servers around the world to look up that address. But you could use the gevent version to look up multiple sites independently. Save this as *gevent_test.py*:

```
import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=[illegible])
for job in jobs:
    print(job.value)
```

There's a one-line for-loop in the preceding example. Each hostname is submitted in turn to a gethostbyname() call, but they can run asynchronously because it's the gevent version of gethostbyname().

Run *gevent_test.py* with Python 2 by typing the following (in bold):

```
$ python2 gevent_test.py
66.6.44.4
74.125.142.[illegible]
78.[illegible]36.12.5[illegible]
```

gevent.spawn() creates a *greenlet* (also known sometimes as a *green thread* or a *micro-thread*) to execute each gevent.socket.gethostbyname(url).

The difference from a normal thread is that it doesn't block. If something occurred that would have blocked a normal thread, gevent switches control to one of the other greenlets.

The gevent.joinall() method waits for all the spawned jobs to finish. Finally, we dump the IP addresses that we got for these hostnames.

Instead of the gevent version of socket, you can use its evocatively named *monkey-patching* functions. These modify standard modules such as socket to use greenlets rather than calling the gevent version of the module. This is useful when you want gevent to be applied all the way down, even into code that you might not be able to access.

At the top of your program, add the following call:

```
from gevent import monkey
monkey.patch_socket()
```

This inserts the gevent socket everywhere the normal socket is called, anywhere in your program, even in the standard library. Again, this works only for Python code, not libraries written in C.

Another function monkey-patches even more standard library modules:

```
from gevent import monkey
monkey.patch_all()
```

Use this at the top of your program to get as many gevent speedups as possible.

Save this program as *gevent_monkey.py*:

```
import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

Again, using Python 2, run the program:

```
$ python2 gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

There are potential dangers when using gevent. As with any event-based system, each chunk of code that you execute should be relatively quick. Although it's nonblocking, code that does a lot of work is still slow.

The very idea of monkey-patching makes some people nervous. Yet, many large sites such as Pinterest use gevent to speed up their sites significantly. Like the fine print on a bottle of pills, use gevent as directed.

> Two other popular event-driven frameworks are tornado and gunicorn. They provide both the low-level event handling and a fast web server. They're worth a look if you'd like to build a fast website without messing with a traditional web server such as Apache.

Scanned with CamScanner
Scanned with CamScanner

twisted

twisted is an asynchronous, event-driven networking framework. You connect functions to events such as data received or connection closed, and those functions are called when those events occur. This is a *callback* design, and if you've written anything in JavaScript, it might seem familiar. If it's new to you, it can seem backwards. For some developers, callback-based code becomes harder to manage as the application grows.

Like gevent, twisted has not yet been ported to Python 3. We'll use the Python 2 installer and interpreter for this section. Type the following to install it:

```
$ pip2 install twisted
```

twisted is a large package, with support for many Internet protocols on top of TCP and UDP. To be short and simple, we'll show a little knock-knock server and client, adapted from twisted examples. First, let's look at the server, *knock_server.py* (notice the Python 2 syntax for print()):

```
from [illegible] import protocol, reactor

class Knock(protocol.Protocol):
    def dataReceived(self, data):
        print 'Client:', data
        if data.startswith("Knock knock"):
            response = "Who's there?"
        else:
            response = data + " who?"
        print 'Server:', response
        self.transport.write(response)

class KnockFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Knock()

reactor.listenTCP([illegible], KnockFactory())
reactor.run()
```

Now, let's take a glance at its trusty companion, *knock_client.py*:

```
from [illegible] import reactor, protocol

class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")

    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
            self.transport.write(response)
        else:
            self.transport.loseConnection()
            reactor.stop()
```


Scanned with CamScanner
Scanned with CamScanner

```
class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient

def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()

if __name__ == '__main__':
    main()
```

Start the server first:

```
$ python2 knock_server.py
```

Then start the client:

```
$ python2 knock_client.py
```

The server and client exchange messages, and the server prints the conversation.

```
Client: Knock knock
Server: Who's there?
Client: Disappearing client
Server: Disappearing client who?
```

Our trickster client then ends, keeping the server waiting for the punch line.

If you'd like to enter the twisted passages, try some of the other examples from its documentation.

asyncio

Recently, Guido van Rossum (remember him?) became involved with the Python concurrency issue. Many packages had their own event loop, and each event loop kind of likes to be the only one. How could he reconcile mechanisms such as callbacks, greenlets, and others? After many discussions and visits, he proposed Asynchronous IO Support Rebooted: the "asyncio" Module, code-named Tulip. This first appeared in Python 3.4 as the asyncio module. For now, it offers a common event loop that could be compatible with twisted, gevent, and other asynchronous methods. The goal is to provide a standard, clean, well-performing asynchronous API. Watch it expand in future releases of Python.

Redis

Our earlier dishwashing code examples, using processes or threads, were run on a single machine. Let's take another approach to queues that can run on a single machine or across a network. Even with multiple singing processes and dancing threads, sometimes

Scanned with CamScanner
Scanned with CamScanner

one machine isn't enough. You can treat this section as a bridge between single-box (one machine) and multiple-box concurrency.

To try the examples in this section, you'll need a Redis server and its Python module. You can see where to get them in "Redis" on page 206. In that chapter, Redis's role is that of a database. Here, we're featuring its concurrency personality.

A quick way to make a queue is with a Redis list. A Redis server runs on one machine; this can be the same one as its clients, or another that the clients can access through a network. In either case, clients talk to the server via TCP, so they're networking. One or more provider clients pushes messages onto one end of the list. One or more client workers watches this list with a *blocking pop* operation. If the list is empty, they all just sit around playing cards. As soon as a message arrives, the first eager worker gets it.

Like our earlier process- and thread-based examples, *redis_washer.py* generates a sequence of dishes:

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', num)
conn.rpush('dishes', 'quit')
print('Washer is done')
```

The loop generates four messages containing a dish name, followed by a final message that says "quit." It appends each message to a list called `dishes` in the Redis server, similar to appending to a Python list.

And as soon as the first dish is ready, *redis_dryer.py* does its work:

```
import redis
conn = redis.Redis()
print('Dryer is starting')
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
    print('Dried', val)
print('Dishes are dried')
```

This code waits for messages whose first token is "dishes" and prints that each one is dried. It obeys the *quit* message by ending the loop.

Scanned with CamScanner
Scanned with CamScanner

Start the dryer, and then the washer. Using the & at the end puts the first program in the *background*; it keeps running, but doesn't listen to the keyboard anymore. This works on Linux, OS X, and Windows, although you might see different output on the next line. In this case (OS X), it's some information about the background dryer process. Then, we start the washer process normally (in the *foreground*). You'll see the mingled output of the two processes:

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
Dried bread
Washed entree
Dried entree
Washed dessert
Washer is done
Dried dessert
Dishes are dried
[2]+  Done                    python redis_dryer.py
```

As soon as dish IDs started arriving at Redis from the washer process, our hard-working dryer process started pulling them back out. Each dish ID was a number, except the final *sentinel* value, the string `'quit'`. When the dryer process read that `quit` dish ID, it quit, and some more background process information printed to the terminal (also system-dependent). You can use a sentinel (an otherwise invalid value) to indicate something special from the data stream itself—in this case, that we're done. Otherwise, we'd need to add a lot more program logic, such as the following:

- Agreeing ahead of time on some maximum dish number, which would kind of be a sentinel anyway.
- Doing some special *out of band* (not in the data stream) interprocess communication.
- Timing out after some interval with no new data.

Let's make a few last changes:

- Create multiple dryer processes.
- Add a timeout to each dryer rather than looking for a sentinel.

The new *redis_dryer2.py*:

```
def dryer():
    import redis
```


Scanned with CamScanner
Scanned with CamScanner

```
    import os
    import redis
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes',  timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8' )
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)

import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

Start the dryer processes in the background, and then the washer process in the foreground:

```
$ python redis_dryer2.py &
Dryer process 44447 is starting
Dryer process 44448 is starting
Dryer process 44446 is starting
$ python redis_washer.py
Washer is starting
Washed salad
44447: dried salad
Washed bread
44448: dried bread
Washed entree
44446: dried entree
Washed dessert
Washer is done
44447: dried dessert
```

One dryer process reads the quit ID and quits:

```
Dryer process 44448 is done
```

After 20 seconds, the other dryer processes get a return value of None from their blpop calls, indicating that they've timed out. They say their last words and exit:

```
Dryer process 44447 is done
Dryer process 44446 is done
```

After the last dryer subprocess quits, the main dryer program ends:

Scanned with CamScanner
Scanned with CamScanner

Beyond Queues

With more moving parts, there are more possibilities for our lovely assembly lines to be disrupted. If we need to wash the dishes from a banquet, do we have enough workers? What if the dryers get drunk? What if the sink clogs? Worries, worries!

How will you cope with it all? Fortunately, there are some techniques available that you can apply. They include the following:

Fire and forget
: Just pass things on and don't worry about the consequences, even if no one is there. That's the dishes-on-the-floor approach.

Request-reply
: The washer receives an acknowledgement from the dryer, and the dryer from the put-away-er, for each dish in the pipeline.

Back pressure or throttling
: This technique directs a fast worker to take it easy if someone downstream can't keep up.

In real systems, you need to be careful that workers are keeping up with the demand; otherwise, you hear the dishes hitting the floor. You might add new tasks to a *pending* list, while some worker process pops the latest message and adds it to a *working* list. When the message is done, it's removed from the working list and added to a *completed* list. This lets you know what tasks have failed or are taking too long. You can do this with Redis yourself, or use a system that someone else has already written and tested. Some Python-based queue packages that add this extra level of management—some of which use Redis—include:

`celery`
: This particular package is well worth a look. It can execute distributed tasks synchronously or asynchronously, using the methods we've discussed: `multiprocessing`, `gevent`, and others.

`thoonk`
: This package builds on Redis to provide job queues and *pub-sub* (coming in the next section).

rq
: This is a Python library for job queues, also based on Redis.

Queues
: This site offers a discussion of queuing software, Python-based and otherwise.

Scanned with CamScanner
Scanned with CamScanner

Networks

In our discussion of concurrency, we talked mostly about time: single-machine solutions (processes, threads, green threads). We also briefly touched upon some solutions that can span networks (Redis, ZeroMQ). Now, we'll look at networking in its own right, distributing computing across space.

Patterns

You can build networking applications from some basic patterns.

The most common pattern is *request-reply*, also known as *client-server*. This pattern is synchronous: the client waits until the server responds. You've seen many examples of request-reply in this book. Your web browser is also a client, making an HTTP request to a web server, which returns a reply.

Another common pattern is *push*, or *fanout*: you send data to any available worker in a pool of processes. An example is a web server behind a load balancer.

The opposite of push is *pull*, or *fanin*: you accept data from one or more sources. An example would be a logger that takes text messages from multiple processes and writes them to a single log file.

One pattern is similar to radio or television broadcasting: *publish-subscribe*, or *pub-sub*. With this pattern, a publisher sends out data. In a simple pub-sub system, all subscribers would receive a copy. More often, subscribers can indicate that they're interested only in certain types of data (often called a *topic*), and the publisher will send just those. So, unlike the push pattern, more than one subscriber might receive a given piece of data. If there's no subscriber for a topic, the data is ignored.

The Publish-Subscribe Model

Publish-subscribe is not a queue but a broadcast. One or more processes publish messages. Each subscriber process indicates what type of messages it would like to receive. A copy of each message is sent to each subscriber that matched its type. Thus, a given message might be processed once, more than once, or not at all. Each publisher is just broadcasting and doesn't know who—if anyone—is listening.

Redis

You can build a quick pub-sub system by using Redis. The publisher emits messages with a topic and a value, and subscribers say which topics they want to receive.

Here's the publisher, *redis_pub.py*:

```
import [illegible]
import [illegible]
```


Scanned with CamScanner
Scanned with CamScanner

```
conn = redis.Redis()
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
for msg in range(10):
    cat = random.choice(cats)
    hat = random.choice(hats)
    print('Publish: %s wears a %s' % (cat, hat))
    conn.publish(cat, hat)
```

Each topic is a breed of cat, and the accompanying message is a type of hat.

Here's a single subscriber, *redis_sub.py*:

```
import redis
conn = redis.Redis()

topics = ['maine coon', 'persian']
sub = conn.pubsub()
sub.subscribe(topics)
for msg in sub.listen():
    if msg['type'] == 'message' :
        cat = msg['channel']
        hat = msg['data']
        print('Subscribe: %s wears a %s' % (cat, hat))
```

The subscriber just shown wants all messages for cat types `'maine coon'` and `'persian'`, and no others. The `listen()` method returns a dictionary. If its type is `'message'`, it was sent by the publisher and matches our criteria. The `'channel'` key is the topic (cat), and the `'data'` key contains the message (hat).

If you start the publisher first and no one is listening, it's like a mime falling in the forest (does he make a sound?), so start the subscriber first:

```
$ python redis_sub.py
```

Next, start the publisher. It will send 10 messages, and then quit:

```
$ python redis_pub.py
Publish: maine coon wears a stovepipe
Publish: norwegian forest wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: siamese wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: persian wears a bowler
Publish: norwegian forest wears a bowler
Publish: maine coon wears a stovepipe
```

The subscriber cares about only two types of cat:

```
$ python redis_sub.py
Subscribe: maine coon wears a stovepipe
```


Scanned with CamScanner
Scanned with CamScanner

```
Subscribe: maine coon wears a bowler
Subscribe: maine coon wears a bowler
Subscribe: persian wears a bowler
Subscribe: maine coon wears a stovepipe
```

We didn't tell the subscriber to quit, so it's still waiting for messages. If you restart the publisher, the subscriber will grab a few more messages and print them.

You can have as many subscribers (and publishers) as you want. If there's no subscriber for a message, it disappears from the Redis server. However, if there are subscribers, the messages stay in the server until all subscribers have retrieved them.

ZeroMQ

Remember those ZeroMQ PUB and SUB sockets from a few pages ago? This is what they're for. ZeroMQ has no central server, so each publisher writes to all subscribers. Let's rewrite the cat-hat pub-sub for ZeroMQ. The publisher, *zmq_pub.py*, looks like this:

```
import zmq
import random
import time
host = '*'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
time.sleep(1)
for msg in range(10):
    cat = random.choice(cats)
    cat_bytes = cat.encode('utf-8')
    hat = random.choice(hats)
    hat_bytes = hat.encode('utf-8')
    print('Publish: %s wears a %s' % (cat, hat))
    pub.send_multipart([cat_bytes, hat_bytes])
```

Notice how this code uses UTF-8 encoding for the topic and value strings.

The file for the subscriber is *zmq_sub.py*:

```
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
topics = ['maine coon', 'persian']
for topic in topics:
    sub.setsockopt(zmq.SUBSCRIBE, topic.encode('utf-8'))
while True:
    cat_bytes, hat_bytes = sub.recv_multipart()
    cat = cat_bytes.decode('utf-8')
```


Scanned with CamScanner
Scanned with CamScanner

```
    hat = hat_bytes.decode('utf-8')
    print('Subscribe: %s wears a %s' % (cat, hat))
```

In this code, we subscribe to two different byte values: the two strings in `topics`, encoded as UTF-8.

> It seems a little backward, but if you want *all* topics, you need to subscribe to the empty bytestring `b''`; if you don't, you'll get nothing.

Notice that we call `send_multipart()` in the publisher and `recv_multipart()` in the subscriber. This makes it possible for us to send multipart messages, and use the first part as the topic. We could also send the topic and message as a single string or bytestring, but it seems cleaner to keep cats and hats separate.

Start the subscriber:

```
$ python zmq_sub.py
```

Start the publisher. It immediately sends 10 messages, and then quits:

```
$ python zmq_pub.py
Publish: norwegian forest wears a stovepipe
Publish: siamese wears a bowler
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: maine coon wears a tam-o-shanter
Publish: maine coon wears a stovepipe
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: norwegian forest wears a bowler
Publish: maine coon wears a bowler
```

The subscriber prints what it requested and received:

```
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a tam-o-shanter
Subscribe: maine coon wears a stovepipe
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a bowler
```

Other Pub-sub Tools

You might like to explore some of these other Python pub-sub links:

RabbitMQ
: This is a well-known messaging broker, and `pika` is a Python API for it. See the pika documentation and a pub-sub tutorial.

Scanned with CamScanner
Scanned with CamScanner

pypi.python.org
: Go to the upper-right corner of the search window and type pubsub to find Python packages like pypubsub.

pubsubhubbub
: This mellifluous protocol enables subscribers to register callbacks with publishers.

TCP/IP

We've been walking through the networking house, taking for granted that whatever's in the basement works correctly. Now, let's actually visit the basement and look at the wires and pipes that keep everything running above ground.

The Internet is based on rules about how to make connections, exchange data, terminate connections, handle timeouts, and so on. These are called *protocols*, and they are arranged in *layers*. The purpose of layers is to allow innovation and alternative ways of doing things; you can do anything you want on one layer as long as you follow the conventions in dealing with the layers above and below you.

The very lowest layer governs aspects such as electrical signals; each higher layer builds on those below. In the middle, more or less, is the IP (Internet Protocol) layer, which specifies how network locations are addressed and how *packets* (chunks) of data flow. In the layer above that, two protocols describe how to move bytes between locations:

UDP (User Datagram Protocol)
: This is used for short exchanges. A *datagram* is a tiny message sent in a single burst, like a note on a postcard.

TCP (Transmission Control Protocol)
: This protocol is used for longer-lived connections. It sends *streams* of bytes and ensures that they arrive in order without duplication.

UDP messages are not acknowledged, so you're never sure if they arrive at their destination. If you wanted to tell a joke over UDP:

```
Here's a UDP joke. Get it?
```

TCP sets up a secret handshake between sender and receiver to ensure a good connection. A TCP joke would start like this:

```
Do you want to hear a TCP joke?
Yes, I want to hear a TCP joke.
Okay, I'll tell you a TCP joke.
Okay, I'll hear a TCP joke.
Okay, I'll send you a TCP joke now.
Okay, I'll receive the TCP joke now.
... (and so on)
```


Scanned with CamScanner
Scanned with CamScanner

Your local machine always has the IP address `127.0.0.1` and the name `localhost`. You might see this called the *loopback interface*. If it's connected to the Internet, your machine will also have a *public* IP. If you're just using a home computer, it's behind equipment such as a cable modem or router. You can run Internet protocols even between processes on the same machine.

Most of the Internet with which we interact—the Web, database servers, and so on—is based on the TCP protocol running atop the IP protocol; for brevity, TCP/IP. Let's first look at some basic Internet services. After that, we'll explore general networking patterns.

Sockets

We've saved this topic until now because you don't need to know all the low-level details to use the higher levels of the Internet. But if you like to know how things work, this is for you.

The lowest level of network programming uses a *socket*, borrowed from the C language and the Unix operating system. Socket-level coding is tedious. You'll have more fun using something like ZeroMQ, but it's useful to see what lies beneath. For instance, messages about sockets often turn up when networking errors take place.

Let's write a very simple client-server exchange. The client sends a string in a UDP datagram to a server, and the server returns a packet of data containing a string. The server needs to listen at a particular address and port—like a post office and a post office box. The client needs to know these two values to deliver its message, and receive any reply.

In the following client and server code, `address` is a tuple of (*address*, *port*). The `address` is a string, which can be a name or an *IP address*. When your programs are just talking to one another on the same machine, you can use the name `'localhost'` or the equivalent address `'127.0.0.1'`.

First, let's send a little data from one process to another and return a little data back to the originator. The first program is the client and the second is the server. In each program, we'll print the time and open a socket. The server will listen for connections to its socket, and the client will write to its socket, which transmits a message to the server.

Here's the first program, *udp_server.py*:

```
from [illegible] import datetime
import [illegible]

server_address = ('localhost', [illegible])
max_size = [illegible]
```


Scanned with CamScanner
Scanned with CamScanner

```
print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(server_address)

data, client = server.recvfrom(max_size)

print('At', datetime.now(), client, 'said', data)
server.sendto(b'Are you talking to me?', client)
server.close()
```

The server has to set up networking through two methods imported from the socket package. The first method, `socket.socket`, creates a socket, and the second, `bind`, *binds* to it (listens to any data arriving at that IP address and port). `AF_INET` means we'll create an Internet (IP) socket. (There's another type for *Unix domain sockets*, but those work only on the local machine.) `SOCK_DGRAM` means we'll send and receive datagrams—in other words, we'll use UDP.

At this point, the server sits and waits for a datagram to come in (`recvfrom`). When one arrives, the server wakes up and gets both the data and information about the client. The `client` variable contains the address and port combination needed to reach the client. The server ends by sending a reply and closing its connection.

Let's take a look at *udp_client.py*:

```
import socket
from datetime import datetime

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hey!', server_address)
data, server = client.recvfrom(max_size)
print('At', datetime.now(), server, 'said', data)
client.close()
```

The client has most of the same methods as the server (with the exception of `bind()`). The client sends and then receives, whereas the server receives first.

Start the server first, in its own window. It will print its greeting and then wait with an eerie calm until a client sends it some data:

```
$ python udp_server.py
Starting the server at 2014-02-05 21:17:41.945649
Waiting for a client to call.
```

Next, start the client in another window. It will print its greeting, send data to the server, print the reply, and then exit:

Scanned with CamScanner
Scanned with CamScanner

```
$ python udp_client.py
Starting the client at [illegible]
At [illegible] ('127.0.0.1', [illegible]) said b'Are you talking to me?'
```

Finally, the server will print something like this, and then exit:

```
At [illegible] ('127.0.0.1', [illegible]) said b'Hey!'
```

The client needed to know the server's address and port number but didn't need to specify a port number for itself. That was automatically assigned by the system—in this case, it was 56267.

UDP sends data in single chunks. It does not guarantee delivery. If you send multiple messages via UDP, they can arrive out of order, or not at all. It's fast, light, connectionless, and unreliable.

Which brings us to TCP (Transmission Control Protocol). TCP is used for longer-lived connections, such as the Web. TCP delivers data in the order in which you send it. If there were any problems, it tries to send it again. Let's shoot a few packets from client to server and back with TCP.

tcp_client.py acts like the previous UDP client, sending only one string to the server, but there are small differences in the socket calls, illustrated here:

```
import [illegible]
from [illegible] import datetime

address = ('localhost', [illegible])
max_size = [illegible]

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(address)
client.sendall(b'Hey!')
data = client.recv(max_size)
print('At', datetime.now(), 'someone replied', data)
client.close()
```

We've replaced `SOCK_DGRAM` with `SOCK_STREAM` to get the streaming protocol, TCP. We also added a `connect()` call to set up the stream. We didn't need that for UDP because each datagram was on its own in the wild, wooly Internet.

tcp_server.py also differs from its UDP cousin:

```
from [illegible] import datetime
import [illegible]

address = ('localhost', [illegible])
max_size = [illegible]
```

```
    print('Starting the server at', datetime.now())
    print('Waiting for a client to call.')
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(address)
    server.listen(5)

    client, addr = server.accept()
    data = client.recv(max_size)

    print('At', datetime.now(), client, 'said', data)
    client.sendall(b'Are you talking to me?')
    client.close()
    server.close()
```

server.listen(5) is configured to queue up to five client connections before refusing new ones. server.accept() gets the first available message as it arrives. The client.recv(1000) sets a maximum acceptable message length of 1,000 bytes.

As you did earlier, start the server and then the client, and watch the fun. First, the server:

```
$ python tcp_server.py
Starting the server at 2014-02-06 22:45:13.306971
Waiting for a client to call.
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

Now, start the client. It will send its message to the server, receive a response, and then exit:

```
$ python tcp_client.py
Starting the client at 2014-02-06 22:45:16.038642
At 2014-02-06 22:45:16.049078 someone replied b'Are you talking to me?'
```

The server collects the message, prints it, responds, and then quits:

```
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

Notice that the TCP server called client.sendall() to respond, and the earlier UDP server called client.sendto(). TCP maintains the client-server connection across multiple socket calls and remembers the client's IP address.

This didn't look so bad, but if you try to write anything more complex, you'll see how low-level sockets really are. Here are some of the complications with which you need to cope:

- UDP sends messages, but their size is limited, and they're not guaranteed to reach their destination.

- TCP sends streams of bytes, not messages. You don't know how many bytes the system will send or receive with each call.
- To exchange entire messages with TCP, you need some extra information to reassemble the full message from its segments: a fixed message size (bytes), or the size of the full message, or some delimiting character.
- Because messages are bytes, not Unicode text strings, you need to use the Python bytes type. For more information on that, see Chapter 7.

After all of this, if you find yourself fascinated by socket programming, check out the Python socket programming HOWTO for more details.

ZeroMQ

We've already seen ZeroMQ sockets used for pub–sub. ZeroMQ is a library. Sometimes described as *sockets on steroids*, ZeroMQ sockets do the things that you sort of expected plain sockets to do:

- Exchange entire messages
- Retry connections
- Buffer data to preserve it when the timing between senders and receivers doesn't line up

The online guide is well written and witty, and it presents the best description of networking patterns that I've seen. The printed version (*ZeroMQ: Messaging for Many Applications*, by Pieter Hintjens, from that animal house, O'Reilly) has that good code smell and a big fish on the cover, rather than the other way around. All the examples in the printed guide are in the C language, but the online version lets you pick from multiple languages for each code example. The Python examples are also viewable. In this chapter, I'll show you some basic uses for ZeroMQ in Python.

ZeroMQ is like a Lego set, and we all know that you can build an amazing variety of things from a few Lego shapes. In this case, you construct networks from a few socket types and patterns. The basic "Lego pieces" presented in the following list are the ZeroMQ socket types, which by some twist of fate look like the network patterns we've already discussed:

- REQ (synchronous request)
- REP (synchronous reply)
- DEALER (asynchronous request)
- ROUTER (asynchronous reply)
- PUB (publish)

Scanned with CamScanner
Scanned with CamScanner

- SUB (subscribe)
- PUSH (fanout)
- PULL (fanin)

To try these yourself, you'll need to install the Python ZeroMQ library by typing this command:

```
$ pip install pyzmq
```

The simplest pattern is a single request-reply pair. This is synchronous: one socket makes a request and then the other replies. First, the code for the reply (server), *zmq_server.py*:

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
while True:
    # Wait for next request from client
    request_bytes = server.recv()
    request_str = request_bytes.decode('utf-8')
    print("That voice in my head says: %s" % request_str)
    reply_str = "Stop saying: %s" % request_str
    reply_bytes = bytes(reply_str, 'utf-8')
    server.send(reply_bytes)
```

We create a `Context` object: this is a ZeroMQ object that maintains state. Then, we make a ZeroMQ `socket` of type `REP` (for REPly). We call `bind()` to make it listen on a particular IP address and port. Notice that they're specified in a string such as `'tcp://localhost:6789'` rather than a tuple, as in the plain socket examples.

This example keeps receiving requests from a sender and sending a response. The messages can be very long—ZeroMQ takes care of the details.

Following is the code for the corresponding request (client), *zmq_client.py*. Its type is `REQ` (for REQuest), and it calls `connect()` rather than `bind()`.

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
for num in range(1, 6):
    request_str = "message #%s" % num
    request_bytes = request_str.encode('utf-8')
```


Scanned with CamScanner
Scanned with CamScanner

```
    client.send(request_bytes)
    reply_bytes = client.recv()
    reply_str = reply_bytes.decode('utf 8')
    print("Sent %s, received %s" % (request_str, reply_str))
```

Now it's time to start them. One interesting difference from the plain socket examples is that you can start the server and client in either order. Go ahead and start the server in one window in the background:

```
$ python zmq_server.py &
```

Start the client in the same window:

```
$ python zmq_client.py
```

You'll see these alternating output lines from the client and server:

```
That voice in my head says 'message #1'
Sent 'message #1', received 'Stop saying message #1'
That voice in my head says 'message #2'
Sent 'message #2', received 'Stop saying message #2'
That voice in my head says 'message #3'
Sent 'message #3', received 'Stop saying message #3'
That voice in my head says 'message #4'
Sent 'message #4', received 'Stop saying message #4'
That voice in my head says 'message #5'
Sent 'message #5', received 'Stop saying message #5'
```

Our client ends after sending its fifth message, but we didn't tell the server to quit, so it sits by the phone, waiting for another message. If you run the client again, it will print the same five lines, and the server will print its five also. If you don't kill the *zmq_server.py* process and try to run another one, Python will complain that the address is already is use:

```
$ python zmq_server.py &

[2] 356
Traceback (most recent call last):
  File "zmq_server.py", line 7, in <module>
    server.bind("tcp://%s:%s" % (host, port))
  File "socket.pyx", line 444, in zmq.backend.cython.socket.Socket.bind
      (zmq/backend/cython/socket.c:4076)
  File "checkrc.pxd", line 21, in zmq.backend.cython.checkrc._check_rc
      (zmq/backend/cython/socket.c:6032)
zmq.error.ZMQError: Address already in use
```

The messages need to be sent as byte strings, so we encoded our example's text strings in UTF-8 format. You can send any kind of message you like, as long as you convert it to bytes. We used simple text strings as the source of our messages, so encode() and decode() were enough to convert to and from byte strings. If your messages have other data types, you can use a library such as MessagePack.

Scanned with CamScanner
Scanned with CamScanner

Even this basic REQ-REP pattern allows for some fancy communication patterns, because any number of REQ clients can `connect()` to a single REP server. The server handles requests one at a time, synchronously, but doesn't drop other requests that are arriving in the meantime. ZeroMQ buffers messages, up to some specified limit, until they can get through; that's where it earns the Q in its name. The Q stands for Queue, the M stands for Message, and the Zero means there doesn't need to be any broker.

Although ZeroMQ doesn't impose any central brokers (intermediaries), you can build them where needed. For example, use DEALER and ROUTER sockets to connect multiple sources and/or destinations asynchronously.

Multiple REQ sockets connect to a single ROUTER, which passes each request to a DEALER, which then contacts any REP sockets that have connected to it (Figure 11-1). This is similar to a bunch of browsers contacting a proxy server in front of a web server farm. It lets you add multiple clients and servers as needed.

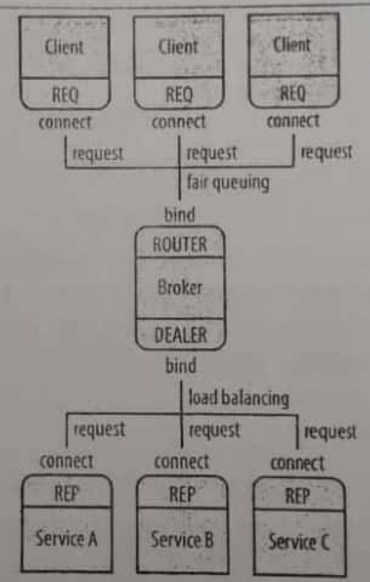

Figure 11-1. Using a broker to connect multiple clients and services

The REQ sockets connect only to the ROUTER socket; the DEALER connects to the multiple REP sockets behind it. ZeroMQ takes care of the nasty details, ensuring that the requests are load balanced and that the replies go back to the right place.

Another networking pattern called the *ventilator* uses PUSH sockets to farm out asynchronous tasks, and PULL sockets to gather the results.

The last notable feature of ZeroMQ is that it scales up *and* down, just by changing the connection type of the socket when it's created:

- `tcp` between processes, on one or more machines
- `ipc` between processes on one machine
- `inproc` between threads in a single process

That last one, `inproc`, is a way to pass data between threads without locks, and an alternative to the `threading` example in "Threads" on page 265.

After using ZeroMQ, you might never want to write raw socket code again.

> ZeroMQ is certainly not the only message-passing library that Python supports. Message passing is one of the most popular ideas in networking, and Python keeps up with other languages. The Apache project, whose web server we saw in "Apache" on page 232, also maintains the ActiveMQ project, including several Python interfaces using the simple-text STOMP protocol. RabbitMQ is also popular, and has useful online Python tutorials.

Scapy

Sometimes you need to dip into the networking stream and see the bytes swimming by. You might want to debug a web API, or track down some security issue. The `scapy` library is an excellent Python tool for packet investigation, and much easier than writing and debugging C programs. It's actually a little language for constructing and analyzing packets.

I planned to include some example code here but changed my mind for two reasons:

- `scapy` hasn't been ported to Python 3 yet. That hasn't stopped us before, when we've used `pip2` and `python2`, but ...
- The installation instructions for scapy are, I think, too intimidating for an introductory book.

If you're so inclined, take a look at the examples in the main documentation site. They might encourage you to brave an installation on your machine.

Finally, don't confuse `scapy` with `scrapy`, which is covered in "Crawl and Scrape" on page 237.

Scanned with CamScanner
Scanned with CamScanner

Internet Services

Python has an extensive networking toolset. In the following sections, we'll look at ways to automate some of the most popular Internet services. The official, comprehensive documentation is available online.

Domain Name System

Computers have numeric IP addresses such as `85.2.101.94`, but we remember names better than numbers. The Domain Name System (DNS) is a critical Internet service that converts IP addresses to and from names via a distributed database. Whenever you're using a web browser and suddenly see a message like "looking up host," you've probably lost your Internet connection, and your first clue is a DNS failure.

Some DNS functions are found in the low-level `socket` module. `gethostbyname()` returns the IP address for a domain name, and the extended edition `gethostbyname_ex()` returns the name, a list of alternative names, and a list of addresses:

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

The `getaddrinfo()` method looks up the IP address, but it also returns enough information to create a socket to connect to it:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
[(2, 2, 17, '', ('66.6.44.4', 80)), (2, 1, 6, '', ('66.6.44.4', 80))]
```

The preceding call returned two tuples, the first for UDP, and the second for TCP (the 6 in the `2, 1, 6` is the value for TCP).

You can ask for TCP or UDP information only:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,
socket.SOCK_STREAM)
[(2, 1, 6, '', ('66.6.44.4', 80))]
```

Some TCP and UDP port numbers are reserved for certain services by IANA, and are associated with service names. For example, HTTP is named `http` and is assigned TCP port 80.

These functions convert between service names and port numbers:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```


Scanned with CamScanner
Scanned with CamScanner

Python Email Modules

The standard library contains these email modules:

- `smtplib` for sending email messages via Simple Mail Transfer Protocol (SMTP)
- `email` for creating and parsing email messages
- `poplib` for reading email via Post Office Protocol 3 (POP3)
- `imaplib` for reading email via Internet Message Access Protocol (IMAP)

The official documentation contains sample code for all of these libraries.

If you want to write your own Python SMTP server, try `smtpd`.

A pure-python SMTP server called Lamson allows you to store messages in databases, and you can even block spam.

Other protocols

Using the standard `ftplib` module, you can push bytes around by using the File Transfer Protocol (FTP). Although it's an old protocol, FTP still performs very well.

You've seen many of these modules in various places in this book, but also try the documentation for standard library support of Internet protocols.

Web Services and APIs

Information providers always have a website, but those are targeted for human eyes, not automation. If data is published only on a website, anyone who wants to access and structure the data needs to write scrapers (as shown in "Crawl and Scrape" on page 237), and rewrite them each time a page format changes. This is usually tedious. In contrast, if a website offers an API to its data, the data becomes directly available to client programs. APIs change less often than web page layouts, so client rewrites are less common. A fast, clean data pipeline also makes it easier to build *mashups*—combinations that might not have been foreseen but can be useful and even profitable.

In many ways, the easiest API is a web interface, but one that provides data in a structured format such as JSON or XML rather than plain text or HTML. The API might be minimal or a full-fledged RESTful API (defined in "Web APIs and Representational State Transfer" on page 236), but it provides another outlet for those restless bytes.

At the very beginning of this book, you can see a web API: it picks up the most popular videos from YouTube. This next example might make more sense now that you've read about web requests, JSON, dictionaries, lists, and slices:

```
import
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = requests.get(url)
```

```
    data = response.json()
    for video in data['feed']['entry'][0:6]:
        print(video['title']['$t'])
```

APIs are especially useful for mining well-known social media sites such as Twitter, Facebook, and LinkedIn. All these sites provide APIs that are free to use, but they require you to register and get a key (a long-generated text string, sometimes also known as a *token*) to use when connecting. The key lets a site determine who's accessing its data. It can also serve as a way to limit request traffic to servers. The YouTube example you just looked at did not require an API key for searching, but it would if you made calls that updated data at YouTube.

Here are some interesting service APIs:

- New York Times
- YouTube
- Twitter
- Facebook
- Weather Underground
- Marvel Comics

You can see examples of APIs for maps in Appendix B, and others in Appendix C.

Remote Processing

Most of the examples in this book have demonstrated how to call Python code on the same machine, and usually in the same process. Thanks to Python's expressiveness, you can also call code on other machines as though they were local. In advanced settings, if you run out of space on your single machine, you can expand beyond it. A network of machines gives you access to more processes and/or threads.

Remote Procedure Calls

Remote Procedure Calls (RPCs) look like normal functions but execute on remote machines across a network. Instead of calling a RESTful API with arguments encoded in the URL or request body, you call an RPC function on your own machine. Here's what happens under the hood of the RPC client:

1. It converts your function arguments into bytes (sometimes this is called *marshalling*, or *serializing*, or just *encoding*).
2. It sends the encoded bytes to the remote machine.

And here's what happens on the remote machine:

1. It receives the encoded request bytes.
2. After receiving the bytes, the RPC client decodes the bytes back to the original data structures (or equivalent ones, if the hardware and software differ between the two machines).
3. The client then finds and calls the local function with the decoded data.
4. Next, it encodes the function results.
5. Last, the client sends the encoded bytes back to the caller.

And finally, the machine that started it all decodes the bytes to return values.

RPC is a popular technique, and people have implemented it in many ways. On the server side, you start a server program, connect it with some byte transport and encoding/decoding method, define some service functions, and light up your *RPC is open for business* sign. The client connects to the server and calls one of its functions via RPC.

The standard library includes one RPC implementation that uses XML as the exchange format: `xmlrpc`. You define and register functions on the server, and the client calls them as though they were imported. First, let's explore the file *xmlrpc_server.py*:

```
from xmlrpc.server import SimpleXMLRPCServer

def double(num):
    return num * 2

server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(double, "double")
server.serve_forever()
```

The function we're providing on the server is called `double()`. It expects a number as an argument and returns the value of that number times two. The server starts up on an address and port. We need to *register* the function to make it available to clients via RPC. Finally, start serving and carry on.

Now, you guessed it, *xmlrpc_client.py*:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
num = 7
result = proxy.double(num)
print("Double %s is %s" % (num, result))
```

The client connects to the server by using `ServerProxy()`. Then, it calls the function `proxy.double()`. Where did that come from? It was created dynamically by the server. The RPC machinery magically hooks this function name into a call to the remote server.

Give it a try—start the server and then run the client:

Scanned with CamScanner
Scanned with CamScanner

```
$ python xmlrpc_server.py
```

Next, run the client:

```
$ python xmlrpc_client.py
Double 7 is 14
```

The server then prints the following:

```
127.0.0.1 - - [13/Feb/2014 20:16:23] "POST / HTTP/1.1" 200 -
```

Popular transport methods are HTTP and ZeroMQ. Common encodings besides XML include JSON, Protocol Buffers, and MessagePack. There are many Python packages for JSON-based RPC, but many of them either don't support Python 3 or seem a bit tangled. Let's look at something different: MessagePack's own Python RPC implementation. Here's how to install it:

```
$ pip install msgpack-rpc-python
```

This will also install `tornado`, a Python event-based web server that this library uses as a transport. As usual, the server comes first (*msgpack_server.py*):

```
from msgpackrpc import Server, Address

class Services():
    def double(self, num):
        return num * 2

server = Server(Services())
server.listen(Address("localhost", 6789))
server.start()
```

The `Services` class exposes its methods as RPC services. Go ahead and start the client, *msgpack_client.py*:

```
from msgpackrpc import Client, Address

client = Client(Address("localhost", 6789))
num = 8
result =  client.call('double', num)
print("Double %s is %s" % (num, result))
```

To run these, follow the usual drill: start the server, start the client, see the results:

```
$ python msgpack_server.py
```

```
$ python msgpack_client.py
Double 8 is 16
```

fabric

The `fabric` package lets you run remote or local commands, upload or download files, and run as a privileged user with sudo. The package uses Secure Shell (SSH; the encrypted text protocol that has largely replaced `telnet`) to run programs on remote

Scanned with CamScanner
Scanned with CamScanner

machines. You write functions (in Python) in a so-called *fabric file* and indicate if they should be run locally or remotely. When you run these with the fabric program (called fab, but not as a tribute to the Beatles or detergent) you indicate which remote machines to use and which functions to call. It's simpler than the RPC examples we've seen.

> As this was written, the author of fabric was merging some fixes to work with Python 3. If those go through, the examples below will work. Until then, you'll need to run them using Python 2.

First, install fabric by typing the following:

```
$ pip2 install fabric
```

You can run Python code locally from a fabric file directly without SSH. Save this first file as *fab1.py*:

```
def iso():
    from          import date
    print(date.today().isoformat())
```

Now, type the following to run it:

```
$ fab -f fab1.py -H localhost iso

[localhost] Executing task 'iso'


Done.
```

The -f fab1.py option specifies to use fabric file *fab1.py* instead of the default *fabfile.py*. The -H localhost option indicates to run the command on your local machine. Finally, iso is the name of the function in the fab file to run. It works like the RPCs that you saw earlier. You can find options on the site's documentation.

To run external programs on your local or remote machines, they need to have an SSH server running. On Unix-like systems, this server is sshd; service sshd status will report if it's up, and service sshd start will start it, if needed. On a Mac, open System Preferences, click the Sharing tab, and then click the Remote Login checkbox. Windows doesn't have built-in SSH support; your best bet is to install putty.

We'll reuse the function name iso, but this time have it run a command by using local(). Here's the command and its output:

```
from          import local

def iso():
    local('date -u')

$ fab -f fab2.py -H localhost iso
```

```
[localhost] Executing task 'iso'
[localhost] local: date -u
Sun Feb [illegible] UTC 2014

Done.
Disconnecting from [illegible]
```

The remote counterpart of `local()` is `run()`. Here's *fab3.py*.

```
from [illegible] import run

def iso():
    run('date -u')
```

Using `run()` instructs fabric to use SSH to connect to whatever hosts were specified on the command line with `-H`. If you have a local network and can connect via SSH to a host, use that hostname in the command after the `-H` (shown in the example that follows). If not, use `localhost`, and it will act as though it were talking to another machine; this can be handy for testing. For this example, let's use `localhost` again:

```
$ fab -f fab3.py -H localhost iso

[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] Login password for 'yourname':
[localhost] out: Sun Feb [illegible] UTC
[localhost] out:


Done.
Disconnecting from [illegible]
```

Notice that it prompted for my login password. To avoid this, you can embed your password in the fabric file as follows:

```
from [illegible] import run
from [illegible] import env

env.password = "your password goes here"

def iso():
    run('date -u')
```

Go ahead and run it:

```
$ fab -f fab4.py -H localhost iso

[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] out: Sun Feb [illegible] UTC
[localhost] out:
```


Scanned with CamScanner
Scanned with CamScanner

```
Done.
Disconnecting from localhost... done.
```

Putting your password in your code is both brittle and insecure. A better way to specify the necessary password is to configure SSH with public and private keys, by using ssh-keygen.

Salt

Salt started as a way to implement remote execution, but it grew to a full-fledged systems management platform. Based on ZeroMQ rather than SSH, it can scale to thousands of servers.

Salt has not yet been ported to Python 3. In this case, I won't show Python 2 examples. If you're interested in this area, read the documents, and watch for announcements when they do complete the port.

Alternative products include puppet and chef, which are closely tied to Ruby. The ansible package, which like Salt is written in Python, is also comparable. It's free to download and use, but support and some add-on packages require a commercial license. It uses SSH by default and does not require any special software to be installed on the machines that it will manage.

salt and ansible are both functional supersets of fabric, handling initial configuration, deployment, and remote execution.

Big Fat Data and MapReduce

As Google and other Internet companies grew, they found that traditional computing solutions didn't scale. Software that worked for single machines, or even a few dozen, could not keep up with thousands.

Disk storage for databases and files involved too much *seeking*, which requires mechanical movement of disk heads. (Think of a vinyl record, and the time it takes to move the needle from one track to another manually. And think of the screeching sound it makes when you drop it too hard, not to mention the sounds made by the record's owner.) But you could *stream* consecutive segments of the disk more quickly.

Developers found that it was faster to distribute and analyze data on many networked machines than on individual ones. They could use algorithms that sounded simplistic, but actually worked better overall with massively distributed data. One of these is MapReduce, which spreads a calculation across many machines and then gathers the results. It's similar to working with queues.

Scanned with CamScanner
Scanned with CamScanner

After Google published its results in a paper, Yahoo followed with an open source Java-based package named *Hadoop* (named after the toy stuffed elephant of the lead programmer's son).

The phrase *big data* applies here. Often it just means "data too big to fit on my machine": data that exceeds the disk, memory, CPU time, or all of the above. To some organizations, if *big data* is mentioned somewhere in a question, the answer is always Hadoop. Hadoop copies data among machines, running them through map and reduce programs, and saving the results on disk at each step.

This batch process can be slow. A quicker method called *Hadoop streaming* works like Unix pipes, streaming the data through programs without requiring disk writes at each step. You can write Hadoop streaming programs in any language, including Python.

Many Python modules have been written for Hadoop, and some are discussed in the blog post "A Guide to Python Frameworks for Hadoop". The Spotify company, known for streaming music, open sourced its Python component for Hadoop streaming, Luigi. The Python 3 port is still incomplete.

A rival named Spark was designed to run ten to a hundred times faster than Hadoop. It can read and process any Hadoop data source and format. Spark includes APIs for Python and other languages. You can find the installation documents online.

Another alternative to Hadoop is Disco, which uses Python for MapReduce processing and Erlang for communication. Alas, you can't install it with pip; see the documentation.

See Appendix C for related examples of *parallel programming*, in which a large structured calculation is distributed among many machines.

Working in the Clouds

Not so long ago, you would buy your own servers, bolt them into racks in data centers, and install layers of software on them: operating systems, device drivers, file systems, databases, web servers, email servers, name servers, load balancers, monitors, and more. Any initial novelty wore off as you tried to keep multiple systems alive and responsive. And you worried constantly about security.

Many hosting services offered to take care of your servers for a fee, but you still leased the physical devices and had to pay for your peak load configuration at all times.

With more individual machines, failures are no longer infrequent: they're very common. You need to scale services horizontally and store data redundantly. You can't assume that the network operates like a single machine. The eight fallacies of distributed computing, according to Peter Deutsch, are as follows:

- The network is reliable.
- Latency is zero.

Scanned with CamScanner

Scanned with CamScanner

Scanned with CamScanner