Chapter 17. Data in Space: Networks

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

In Chapter 15, you read about *concurrency*: how to do more than one thing at a time. Now we'll try to do things 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.

Scalability
: Increase your servers to handle load, decrease them to save money.

In this chapter, we work our way up from networking primitives to higher-level concepts. Let's start with TCP/IP and sockets.

TCP/IP

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 whether 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)
```

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 explore general networking patterns.

Sockets

If you like to know how things work, all the way down, this section 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, once with UDP and once with TCP. In the UDP example, 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 string `'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 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.

Example 17-1 presents the first program, *udp_server.py*.

Example 17-1. udp_server.py

```
from datetime import datetime
import socket

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

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 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* (Example 17-2).

Example 17-2. 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 (the bytes value `'Hey'`) to the server, print the reply, and then exit:

```
$ python udp_client.py
Starting the client at 2014-02-05 21:24:56.509682
At 2014-02-05 21:24:56.518670 ('127.0.0.1', 6789) said b'Are you talking to me?
```

Finally, the server will print the message it received, and exit:

```
At 2014-02-05 21:24:56.518473 ('127.0.0.1', 56267) 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`.

NOTE

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. UDP is useful when you need to push packets quickly, and can tolerate a lost packet now and then, such as with VoIP (voice over IP).

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. This makes TCP a bit slower than UDP, but usually a better choice when you need all the packets, in the right order.

NOTE

The first two versions of the web protocol HTTP were based on TCP, but HTTP/3 is based on a protocol called QUIC, which itself uses UDP. So choosing between UDP and TCP can involve many factors.

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 in Example 17-3.

Example 17-3. tcp_client.py

```
import socket
from datetime import datetime

address = ('localhost', 6789)
max_size = 1000

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.

As Example 17-4 demonstrates, *tcp_server.py* also differs from its UDP cousin.

Example 17-4. tcp_server.py

```
from datetime import datetime
import socket

address = ('localhost', 6789)
max_size = 1000

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 sockets really operate at a low level:

- 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 12.

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

Scapy

Sometimes you need to dip into the networking stream and watch the bytes swimming by. You may want to debug a web API, or track down some security issue. The `scapy` library and program provide a domain-specific language to create and inspect packets in Python, which is much easier than writing and debugging the equivalent C programs.

A standard install uses `pip install scapy`. The docs are extremely thorough. If you use tools like `tcpdump` or `wireshark` to investigate TCP issues, you should look at `scapy`. Finally, don't confuse `scapy` with `scrapy`, which is covered in "Crawl and Scrape".

Netcat

Another tool to test networks and ports is Netcat, often abbreviated to `nc`. Here's an example of an HTTP connnection to Google's website, and requesting some basic information about its home page:

```
$ $ nc www.google.com 80
HEAD / HTTP/1.1

HTTP/1.1 200 OK
Date: Sat, 27 Jul 2019 21:04:02 GMT
...
```

In the next chapter, there's an example that uses "Test with telnet" to do the same.

Networking Patterns

You can build networking applications from some basic patterns:

- The most common pattern is *request-reply*, also known as *request-response* or *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 are ignored.

Let's show some request-reply examples, and later some pub-sub ones.

The Request-Reply Pattern

This is the most familiar pattern. You request DNS, web, or email data from the appropriate servers, and they reply, or tell you whether there's a problem.

I just showed you how to make some basic requests with UDP or TCP, but it's hard to build a networking application at the socket level. Let's see if ZeroMQ can help.

ZeroMQ

ZeroMQ is a library, not a server. 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 them 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 choose from multiple languages for each code example. The Python examples are also viewable. In this chapter, I show you some basic request-reply ZeroMQ examples.

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)
- 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*, as shown in Example 17-5.

Example 17-5. 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.

Example 17-6 shows the code for the corresponding request (client), *zmq_client.py*. Its type is `REQ` (for REQuest), and it calls `connect()` rather than `bind()`.

Example 17-6. zmq_client.py

```
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')
    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.

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 17-1). 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.

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".

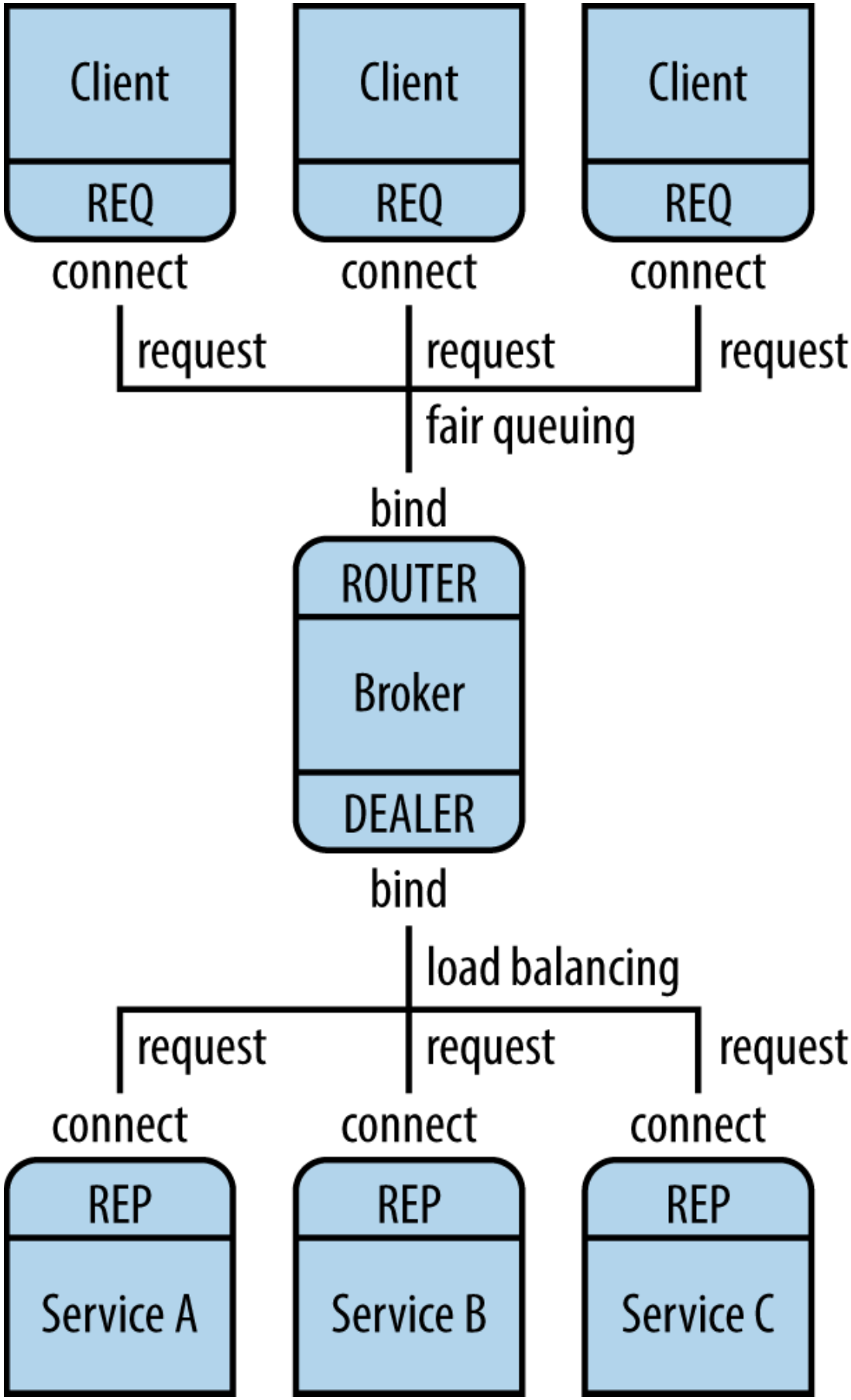

Figure 17-1. Using a broker to connect multiple clients and services

After using ZeroMQ, you may not want to write raw socket code again.

Other Messaging Tools

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", also maintains the ActiveMQ project, including several Python interfaces using the simple-text STOMP protocol.
- RabbitMQ is also popular, and it has useful online Python tutorials.
- NATS is a fast messaging system, written in Go.

The Publish-Subscribe Pattern

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. Like a lonely radio operator, each publisher is just broadcasting and doesn't know who, if anyone, is listening.

Redis

You've seen Redis in Chapter 16, mainly as a data structure server, but it also contains a pub-sub system. The publisher emits messages with a topic and a value, and subscribers say which topics they want to receive.

Example 17-7 contains a publisher, *redis_pub.py*.

Example 17-7. redis_pub.py

```
import redis
import random

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.

Example 17-8 shows a single subscriber, *redis_sub.py*.

Example 17-8. 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))
```

This subscriber 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
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

ZeroMQ has no central server, so each publisher writes to all subscribers. The publisher, *zmq_pub.py*, is provided in Example 17-9.

Example 17-9. zmq_pub.py

```
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* (Example 17-10).

Example 17-10. 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')
    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.

NOTE

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 is a well-known messaging broker, and `pika` is a Python API for it. See the `pika` documentation and a pub-sub tutorial.
- Go to the PyPi search window and type `pubsub` to find Python pack-

ages like `pypubsub`.

- PubSubHubbub enables subscribers to register callbacks with publishers.
- NATS is a fast, open source messaging system that supports pub-sub, request-reply, and queuing.

Internet Services

Python has an extensive networking toolset. In the following sections, we 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'
```

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)

If you want to write your own Python SMTP server, try `smtpd`, or the new asynchronous version `aiosmtpd`.

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"), 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 unforeseen but useful combinations.

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 REST"), but it provides another outlet for those restless bytes.

At the very beginning of this book, you saw a web API query the Internet Archive for an old copy of a website.

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.

Here are some interesting service APIs:

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

You can see examples of APIs for maps in Chapter 21, and others in Chapter 22.

Data Serialization

As you saw in Chapter 16, formats like XML, JSON, and YAML are ways to store structured text data. Networked applications need to exchange data with other programs. The conversion between data in memory and byte sequences "on the wire" is called *serialization* or *marshaling*. JSON is a popular serialization format, especially with web RESTful systems, but it can't express all Python data types directly. Also, as a text format it tends to be more verbose than some binary serialization methods. Let's look at some approaches that you're likely to run into.

Serialize with pickle

Python provides the `pickle` module to save and restore any object in a special binary format.

Remember how JSON lost its mind when encountering a `datetime` object? Not a problem for `pickle`:

```
>>> import pickle
>>> import datetime
>>> now1 = datetime.datetime.utcnow()
>>> pickled = pickle.dumps(now1)
>>> now2 = pickle.loads(pickled)
>>> now1
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
>>> now2
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
```

`pickle` works with your own classes and objects, too. Let's define a little class called `Tiny` that returns the string `'tiny'` when treated as a string:

```
>>> import pickle
>>> class Tiny():
...     def __str__(self):
...         return 'tiny'
...
>>> obj1 = Tiny()
>>> obj1
<__main__.Tiny object at 0x10076ed10>
>>> str(obj1)
'tiny'
>>> pickled = pickle.dumps(obj1)
>>> pickled
b'\x80\x03c__main__\nTiny\nq\x00)\x81q\x01.'
>>> obj2 = pickle.loads(pickled)
>>> obj2
<__main__.Tiny object at 0x10076e550>
>>> str(obj2)
'tiny'
```

`pickled` is the pickled binary string made from the object `obj1`. We converted that back to the object `obj2` to make a copy of `obj1`. Use `dump()` to pickle to a file, and `load()` to unpickle from one.

The `multiprocessing` module uses `pickle` to interchange data among processes.

If `pickle` can't serialize your data format, a newer third-party package called `dill` might.

NOTE

Because `pickle` can create Python objects, the same security warnings that were discussed in earlier sections apply. A public service announcement: don't unpickle something that you don't trust.

Other Serialization Formats

These binary data interchange formats are usually more compact and faster than XML or JSON:

- MsgPack
- Protocol Buffers
- Avro
- Thrift
- Lima
- `Serialize` is a Python frontend to other formats, including JSON, YAML, pickle, and MsgPack.
- A benchmark of various Python serialization packages.

Because they are binary, none can be easily edited by a human with a text editor.

Some third-party packages interconvert objects and basic Python data types (allowing further conversion to/from formats like JSON), and provide *validation* of the following:

- Data types
- Value ranges
- Required versus optional data

These include:

- Marshmallow
- Pydantic—uses type hints, so requires at least Python 3.6
- TypeSystem

These are often used with web servers to ensure that the bytes that came over the wire via HTTP end up in the right data structures for further processing.

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. Your local machine:

- Serializes your function arguments into bytes.
- Sends the encoded bytes to the remote machine.

The remote machine:

- Receives the encoded request bytes.
- Deserializes the bytes back to data structures.
- Finds and calls the service function with the decoded data.
- Encodes the function results.
- Sends the encoded bytes back to the caller.

And finally, the local 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.

XML 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*, as shown in Example 17-11.

Example 17-11. 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*, proudly presented in Example 17-12.

Example 17-12. 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:

```
$ python xmlrpc_server.py
```

Run the client again:

```
$ 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.

JSON RPC

JSON-RPC (versions 1.0 and 2.0) is similar to XML-RPC, but with JSON. There are many Python JSON-RPC libraries, but the simplest one I've found comes in two parts: client and server.

Installation of both is familiar: `pip install jsonrpcserver` and `pip install jsonrpclient`.

These libraries provide many alternative ways to write a client and server. In Example 17-13 and Example 17-14, I use this library's built-in server, which uses port 5000 and is the simplest.

First, the server.

Example 17-13. jsonrpc_server.py

```
from jsonrpcserver import method, serve

@method
def double(num):
    return num * 2

if __name__ == "__main__":
    serve()
```

Second, the client.

Example 17-14. jsonrpc_client.py

```
from jsonrpcclient import request

num = 7
response = request("http://localhost:5000", "double", num=num)
print("Double", num, "is", response.data.result)
```

As with most of the client-server examples in this chapter, start the server first (in its own terminal window, or with a following `&` to put it in the background) and then run the client:

```
$ python jsonrpc_server.py &
[1] 10621
$ python jsonrpc_client.py
127.0.0.1 - - [23/Jun/2019 15:39:24] "POST / HTTP/1.1" 200 -
Double 7 is 14
```

If you put the server in the background, kill it when you're done.

MessagePack RPC

The encoding library MessagePack has its 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 (*msgpack_server.py*) comes first (Example 17-15).

Example 17-15. 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* (Example 17-16).

Example 17-16. 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 and client in separate terminal windows,[1] and observe the results:

```
$ python msgpack_server.py
```

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

Zerorpc

Written by the developers of Docker (when they were called dotCloud), zerorpc uses ZeroMQ and MsgPack to connect clients and servers. It magically exposes functions as RPC endpoints.

Type `pip install zerorpc` to install it. The sample code in Example 17-17 and Example 17-18 shows a request-reply client and server.

Example 17-17. zerorpc_server.py

```
import zerorpc

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

server = zerorpc.Server(RPC())
server.bind("tcp://0.0.0.0:4242")
server.run()
```

Example 17-18. zerorpc_client.py

```
import zerorpc

client = zerorpc.Client()
```

```
client.connect("tcp://127.0.0.1:4242")
num = 7
result = client.double(num)
print("Double", num, "is", result)
```

Notice that the client calls `client.double()`, even though there's no definition of it in there:

```
$ python zerorpc_server &
[1] 55172
$ python zerorpc_client.py
Double 7 is 14
```

The site has many more examples.

gRPC

Google created gRPC as a portable and fast way to define and connect services. It encodes data as protocol buffers.

Install the Python parts:

```
$ pip install grpcio
$ pip install grpcio-tools
```

The Python client docs are very detailed, so I'm giving only a brief overview here. You may also like this separate tutorial.

To use gRPC, you write a *.proto* file to define a `service` and its `rpc` methods.

An `rpc` method is like a function definition (describing its arguments and return types) and may specify one of these networking patterns:

- Request-response (sync or async)
- Request-streaming response
- Streaming request-response (sync or async)
- Streaming request-streaming response

Single responses can be blocking or asynchronous. Streaming responses are iterated.

Next, you would run the `grpc_tools.protoc` program to create Python code for the client and server. gRPC handles the serialization and network communication; you add your application-specific code to the client and server stubs.

gRPC is a top-level alternative to web REST APIs. It seems to be a better fit than REST for inter-service communication, and REST may be preferred for public APIs.

Twirp

Twirp is similar to gRPC, but claims to be simpler. You define a `.proto` file as you would with gRPC, and twirp can generate Python code to handle the client and server ends.

Remote Management Tools

- `Salt` is written in Python. It started as a way to implement remote execution, but grew to a full-fledged systems management platform. Based on ZeroMQ rather than SSH, it can scale to thousands of servers.
- `Puppet` and `Chef` are popular and 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

As Google and other internet companies grew, they found that traditional computing solutions didn't scale. Software that worked for single ma-

chines, 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.

Hadoop

After Google published its MapReduce 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* (scatter) and *reduce* (gather) 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". Spotify, known for streaming music, open sourced its Python component for Hadoop streaming, Luigi.

Spark

A rival named Spark was designed to run 10 to 100 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.

Disco

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.

Dask

Dask is similar to Spark, although it's written in Python and is largely used with scientific Python packages like NumPy, Pandas, and scikit-learn. It can spread tasks across thousand-machine clusters.

To get Dask and all of its extra helpers:

```
$ pip install dask[complete]
```

See Chapter 22 for related examples of *parallel programming*, in which a large structured calculation is distributed among many machines.

Clouds

> *I really don't know clouds at all.*
>
> —Joni Mitchell

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, filesystems, 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 wor-

ried 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.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

You can try to build these complex distributed systems, but it's a lot of work, and a different toolset is needed. To borrow an analogy, when you have a handful of servers, you treat them like pets—you give them names, know their personalities, and nurse them back to health when needed. But at scale, you treat servers more like livestock: they look alike, have numbers, and are just replaced if they have any problems.

Instead of building, you can rent servers in the *cloud*. By adopting this model, maintenance is someone else's problem, and you can concentrate on your service, or blog, or whatever you want to show the world. Using web dashboards and APIs, you can spin up servers with whatever configuration you need, quickly and easily—they're *elastic*. You can monitor their status, and be alerted if some metric exceeds a given threshold. Clouds are currently a pretty hot topic, and corporate spending on cloud components has spiked.

The big cloud vendors are:

- Amazon (AWS)
- Google
- Microsoft Azure

Amazon Web Services

As Amazon was growing from hundreds to thousands to millions of servers, developers ran into all the nasty problems of distributed systems. One day in 2002 or thereabouts, CEO Jeff Bezos declared to Amazon employees that, henceforth, all data and functionality needed to be exposed only via network service interfaces—not files, or databases, or local function calls. They had to design these interfaces as though they were being offered to the public. The memo ended with a motivational nugget: *"Anyone who doesn't do this will be fired."*

Not surprisingly, developers got to work, and over time built a huge service-oriented architecture. They borrowed or innovated many solutions, evolving into Amazon Web Services (AWS), which now dominates the market. The official Python AWS library is `boto3`:

- documentation
- SDK pages

Install it with:

```
$ pip install boto3
```

You can use `boto3` as an alternative to AWS's web-based management pages.

Google Cloud

Google uses Python a lot internally, and it employs some prominent Python developers (even Guido van Rossum himself, for some time). From its main and Python pages, you can find details on its many services.

Microsoft Azure

Microsoft caught up with Amazon and Google with its cloud offering, Azure. See Python on Azure to learn how to develop and deploy Python applications.

OpenStack

OpenStack is an open source framework of Python services and REST APIs. Many of the services are similar to those in the commercial clouds.

Docker

The humble standardized shipping container revolutionized international trade. Only a few years ago, Docker applied the *container* name and analogy to a *virtualization* method using some little-known Linux features. Containers are much lighter than virtual machines, and a bit heavier than Python virtualenvs. They allow you to package an application separately from other applications on the same machine, sharing only the operating system kernel.

To install Docker's Python client library:

```
$ pip install docker
```

Kubernetes

Containers caught on and spread through the computing world. Eventually, people needed ways to manage multiple containers and wanted to automate some of the manual steps that have been usually required in large distributed systems:

- Failover
- Load balancing
- Scaling up and down

It looks like Kubernetes is leading the pack in this new area of *container orchestration*.

To install the Python client library:

```
$ pip install kubernetes
```

Coming Up

As they say on television, our next guest needs no

introduction. Learn why Python is one of the best languages to tame the web.

Things to Do

17.1 Use

a plain `socket` to implement a current-time-service. When a client sends the string *time* to the server, return the current date and time as an

ISO string.

17.2 Use ZeroMQ REQ and REP sockets to do the same thing.

17.3 Try the same with XMLRPC.

17.4 You may have seen the classic *I Love Lucy* television episode

in which Lucy and Ethel worked in a choco-late fac-tory. The duo fell be-hind as the con-veyor belt that sup-plied the con-fec-tions for them to process be-gan op-er-at-

ing at an ever-faster rate. Write a simulation that pushes different types of chocolates to a Redis list, and Lucy is a client doing blocking pops of this list.

She needs 0.5 seconds to handle a piece of chocolate. Print the time and type of each chocolate as Lucy gets it, and how many remain to be handled.

17.5 Use

ZeroMQ to publish the poem from exercise 12.4 (from Example 12-1), one word at a time. Write a ZeroMQ consumer that prints every word that starts with a vowel, and another that prints

every word that contains five letters. Ignore punctuation characters.

Or put the server in the background with a final &.