Reading 21: Sockets & Networking

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we examine *client/server communication* over the network using the *socket* abstraction.

Network communication is inherently concurrent, so building clients and servers will require us to reason about their concurrent behavior and to implement them with thread safety. We must also design the *wire protocol* that clients and servers use to communicate, just as we design the operations that clients of an ADT use to work with it.

Some of the operations with sockets are *blocking* : they block the progress of a thread until they can return a result. Blocking makes writing some code easier, but it also foreshadows a new class of concurrency bugs we'll soon contend with in depth: deadlocks.

Client/server design pattern

In this reading (and in the problem set) we explore the **client/server design pattern** for communication with message passing.

In this pattern there are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally, the client disconnects. A server might handle connections from many clients concurrently, and clients might also connect to multiple servers.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines, connected only by the network, but it doesn't have to be that way — the server can be a process running on the same machine as the client.

Network sockets

IP addresses

A network interface is identified by an IP address . IPv4 addresses are 32-bit numbers written in four 8-bit parts. For example (as of this writing):

- `18.9.22.69` is the IP address of a MIT web server. Every address whose first octet is `18` is on the MIT network.
- `18.9.25.15` is the address of a MIT incoming email handler.
- `173.194.123.40` is the address of a Google web server.
- `127.0.0.1` is the loopback or localhost address: it always refers to the local machine. Technically, any address whose first octet is `127` is a loopback address, but `127.0.0.1` is standard.

You can ask Google for your current IP address . In general, as you carry around your laptop, every time you connect your machine to the network it can be assigned a new IP address.

Hostnames

Hostnames are names that can be translated into IP addresses. A single hostname can map to different IP addresses at different times; and multiple hostnames can map to the same IP address. For example:

- `web.mit.edu` is the name for MIT's web server. You can translate this name to an IP address yourself using `dig` , `host` , or `nslookup` on the command line, e.g.:

  ```
  $ dig +short web.mit.edu
  18.9.22.69
  ```

- `dmz-mailsec-scanner-4.mit.edu` is the name for one of MIT's spam filter machines responsible for handling incoming email.

- `google.com` is exactly what you think it is. Try using one of the commands above to find `google.com` 's IP address. What do you see?

- `localhost` is a name for `127.0.0.1` . When you want to talk to a server running on your own machine, talk to `localhost` .

Translation from hostnames to IP addresses is the job of the Domain Name System (DNS) . It's super cool, but not part of our discussion today.

Port numbers

A single machine might have multiple server applications that clients wish to connect to, so we need a way to direct traffic on the same network interface to different processes.

Network interfaces have multiple ports identified by a 16-bit number from 0 (which is reserved, so we effectively start at 1) to 65535.

A server process binds to a particular port — it is now **listening** on that port. Clients have to know which port number the server is listening on. There are some well-known ports which are reserved for system-level processes and provide standard ports for certain services. For example:

- Port 22 is the standard SSH port. When you connect to `athena.dialup.mit.edu` using SSH, the software automatically uses port 22.
- Port 25 is the standard email server port.
- Port 80 is the standard web server port. When you connect to the URL `http://web.mit.edu` in your web browser, it connects to `18.9.22.69` on port 80.

When the port is not a standard port, it is specified as part of the address. For example, the URL `http://128.2.39.10:9000` refers to port 9000 on the machine at `128.2.39.10` .

When a client connects to a server, that outgoing connection also uses a port number on the client's network interface, usually chosen at random from the available *non* -well-known ports.

Network sockets

A **socket** represents one end of the connection between client and server.

- A **listening socket** is used by a server process to wait for connections from remote clients.

  In Java, use `ServerSocket` to make a listening socket, and use its `accept` method to listen to it.

- A **connected socket** can send and receive messages to and from the process on the other end of the connection. It is identified by both the local IP address and port number plus the remote address and port, which allows a server to differentiate between concurrent connections from different IPs, or from the same IP on different remote ports.

  In Java, clients use a `Socket` constructor to establish a socket connection to a server. Servers obtain a connected socket as a `Socket` object returned from `ServerSocket.accept` .

I/O

Buffers

The data that clients and servers exchange over the network is sent in chunks. These are rarely just byte-sized chunks, although they might be. The sending side (the client sending a request or the server sending a response) typically writes a large chunk (maybe a whole string like "HELLO, WORLD!" or maybe 20 megabytes of video data). The network chops that chunk up into packets, and each packet is routed separately over the network. At the other end, the receiver reassembles the packets together into a stream of bytes.

The result is a bursty kind of data transmission — the data may already be there when you want to read them, or you may have to wait for them to arrive and be reassembled.

When data arrive, they go into a **buffer** , an array in memory that holds the data until you read it.

* see ***What if Dr. Seuss Did Technical Writing?*** , although the issue described in the first stanza is no longer relevant with the obsolescence of floppy disk drives

Streams

The data going into or coming out of a socket is a **stream** of bytes.

In Java, `InputStream` objects represent sources of data flowing into your program. For example:

- Reading from a file on disk with a `FileInputStream`
- User input from `System.in`
- Input from a network socket

`OutputStream` objects represent data sinks, places we can write data to. For example:

- `FileOutputStream` for saving to files
- `System.out` is a `PrintStream`, an `OutputStream` that prints readable representations of various types
- Output to a network socket

In the Java Tutorials, read:

- I/O Streams up to and including *I/O from the Command Line* (8 pages)

With sockets, remember that the *output* of one process is the *input* of another process. If Alice and Bob have a socket connection, Alice has an output stream that flows to Bob's input stream, and *vice versa*.

Blocking

Blocking means that a thread waits (without doing further work) until an event occurs. We can use this term to describe methods and method calls: if a method is a **blocking method**, then a call to that method can **block**, waiting until some event occurs before it returns to the caller.

Socket input/output streams exhibit blocking behavior:

- When an incoming socket's buffer is empty, calling `read` blocks until data are available.
- When the destination socket's buffer is full, calling `write` blocks until space is available.

Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the `read` (or `write`) call will always work, no matter what the timing of data arrival. If data (or for `write`, space) is already available in the buffer, the call might return very quickly. But if the read or write can't succeed, the call **blocks**. The operating system takes care of the details of delaying that thread until `read` or `write` *can* succeed.

Blocking happens throughout concurrent programming, not just in I/O (communication into and out of a process, perhaps over a network, or to/from a file, or with the user on the command line or a GUI, …). Concurrent modules don't work in lockstep, like sequential programs do, so they typically have to wait for each other to catch up when coordinated action is required.

We'll see in the next reading that this waiting gives rise to the second major kind of bug (the first was race conditions) in concurrent programming: **deadlock**, where modules are waiting for each other to do something, so none of them can make any progress. But that's for next time.

Using network sockets

Make sure you've read about streams at the Java Tutorial link above, then read about network sockets:

In the Java Tutorials, read:

- All About Sockets (4 pages)

This reading describes everything you need to know about creating server- and client-side sockets and writing to and reading from their I/O streams.

On the second page

The example uses a syntax we haven't seen: the try-with-resources statement. This statement has the form:

```
try (
    // create new objects here that require cleanup after being used,
    // and assign them to variables
) {
    // code here runs with those variables
    // cleanup happens automatically after the code completes
} catch(...) {
    // you can include catch clauses if the code might throw exceptions
}
```

On the last page

Notice how both `ServerSocket.accept()` and `in.readLine()` are *blocking*. This means that the server will need a *new thread* to handle I/O with each new client. While the client-specific thread is working with that client (perhaps blocked in a read or a write), another thread (perhaps the main thread) is blocked waiting to `accept` a new connection.

Unfortunately, their multithreaded Knock Knock Server implementation creates that new thread by *subclassing* `Thread`. That's *not* the recommended strategy. Instead, create a new class that implements `Runnable`, or use an anonymous `Runnable` that calls a method where that client connection will be handled until it's closed. Don't use ~~`extends Thread`~~. And while subclassing was popular when the Java API was designed, we don't discuss or recommend it at all because it has many downsides.

Wire protocols

Now that we have our client and server connected up with sockets, what do they pass back and forth over those sockets?

A **protocol** is a set of messages that can be exchanged by two communicating parties. A **wire protocol** in particular is a set of messages represented as byte sequences, like `hello world` and `bye` (assuming we've agreed on a way to encode those characters into bytes).

Most Internet applications use simple ASCII-based wire protocols. You can use a program called Telnet to check them out. For example:

HTTP

Hypertext Transfer Protocol (HTTP) is the language of the World Wide Web. We already know that port 80 is the well-known port for speaking HTTP to web servers, so let's talk to one on the command line.

You'll be using Telnet on the problem set, so try these out now. User input is shown in **green**, and for input to the telnet connection, newlines (pressing enter) are shown with ↵:

```
$ telnet www.eecs.mit.edu 80
Trying 18.62.0.96...
Connected to eecsweb.mit.edu.
Escape character is '^]'.
GET /↵
<!DOCTYPE html>
... lots of output ...
<title>Homepage | MIT EECS</title>
... lots more output ...
```

The `GET` command gets a web page. The `/` is the path of the page you want on the site. So this command fetches the page at `http://www.eecs.mit.edu:80/`. Since 80 is the default port for HTTP, this is equivalent to visiting http://www.eecs.mit.edu/ in your web browser. The result is HTML code that your browser renders to display the EECS homepage.

Internet protocols are defined by RFC specifications (RFC stands for "request for comment", and some RFCs are eventually adopted as standards). RFC 1945 defined HTTP version 1.0, and was superseded by HTTP 1.1 in RFC 2616. So for many web sites, you might need to speak HTTP 1.1 if you want to talk to them. For example:

```
$ telnet web.mit.edu 80
Trying 18.9.22.69...
Connected to web.mit.edu.
Escape character is '^]'.
GET /aboutmit/ HTTP/1.1↵
Host: web.mit.edu↵
↵
HTTP/1.1 200 OK
Date: Tue, 31 Mar 2015 15:14:22 GMT
... more headers ...

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
... more HTML ...
<title>MIT – About</title>
... lots more HTML ...
```

This time, your request must end with a blank line. HTTP version 1.1 requires the client to specify some extra information (called headers) with the request, and the blank line signals the end of the headers.

You will also more than likely find that telnet does not exit after making this request — this time, the server keeps the connection open so you can make another request right away. To quit Telnet manually, type the escape character (probably `Ctrl - ]`) to bring up the `telnet>` prompt, and type `quit`:

```
... lots more HTML ...
</html>
Ctrl-]↵
telnet> quit↵
Connection closed.
```

SMTP

Simple Mail Transfer Protocol (SMTP) is the protocol for sending email (different protocols are used for client programs that retrieve email from your inbox). Because the email system was designed in a time before spam, modern email

communication is fraught with traps and heuristics designed to prevent abuse. But we can still try to speak SMTP. Recall that the well-known SMTP port is 25, and `dmz-mailsec-scanner-4.mit.edu` was the name of a MIT email handler.

You'll need to fill in *your-IP-address-here* and *your-username-here* , and the ↵ indicate newlines for clarity. This will only work if you're on MITnet, and even then your mail might be rejected for looking suspicious:

```
$ telnet dmz-mailsec-scanner-4.mit.edu 25
Trying 18.9.25.15...
Connected to dmz-mailsec-scanner-4.mit.edu.
Escape character is '^]'.
220 dmz-mailsec-scanner-4.mit.edu ESMTP Symantec Messaging Gateway
HELO your-IP-address-here↵
250 2.0.0 dmz-mailsec-scanner-4.mit.edu says HELO to your-ip-address:port
MAIL FROM: <your-username-here@mit.edu>↵
250 2.0.0 MAIL FROM accepted
RCPT TO: <your-username-here@mit.edu>↵
250 2.0.0 RCPT TO accepted
DATA↵
354 3.0.0 continue.  finished with "\r\n.\r\n"
From: <your-username-here@mit.edu>↵
To: <your-username-here@mit.edu>↵
Subject: testing↵
This is a hand-crafted artisanal email.↵
.↵
250 2.0.0 OK 99/00-11111-22222222
QUIT↵
221 2.3.0 dmz-mailsec-scanner-4.mit.edu closing connection
Connection closed by foreign host.
```

SMTP is quite chatty in comparison to HTTP, providing some human-readable instructions like `continue. finished with "\r\n.\r\n"` to tell us how to terminate our message content.

Designing a wire protocol

When designing a wire protocol, apply the same rules of thumb you use for designing the operations of an abstract data type:

- Keep the number of different messages **small** . It's better to have a few commands and responses that can be combined rather than many complex messages.
- Each message should have a well-defined purpose and **coherent** behavior.
- The set of messages must be **adequate** for clients to make the requests they need to make and for servers to deliver the results.

Just as we demand representation independence from our types, we should aim for **platform-independence** in our protocols. HTTP can be spoken by any web server and any web browser on any operating system. The protocol doesn't say anything about how web pages are stored on disk, how they are prepared or generated by the server, what algorithms the client will use to render them, etc.

We can also apply the three big ideas in this class:

- **Safe from bugs**
  - The protocol should be easy for clients and servers to generate and parse. Simpler code for reading and writing the protocol (whether written with a parser generator like ANTLR, with regular expressions, etc.) will have fewer opportunities for bugs.
  - Consider the ways a broken or malicious client or server could stuff garbage data into the protocol to break the process on the other end.

    Email spam is one example: when we spoke SMTP above, the mail server asked *us* to say who was sending the email, and there's nothing in SMTP to prevent us from lying outright. We've had to build systems on top of SMTP to try to stop spammers who lie about `From:` addresses.

    Security vulnerabilities are a more serious example. For example, protocols that allow a client to send requests with arbitrary amounts of data require careful handling on the server to avoid running out of buffer space, or worse .
- **Easy to understand** : for example, choosing a text-based protocol means that we can debug communication errors by reading the text of the client/server exchange. It even allows us to speak the protocol "by hand" as we saw above.
- **Ready for change** : for example, HTTP includes the ability to specify a version number, so clients and servers can agree with one another which version of the protocol they will use. If we need to make changes to the protocol in the future, older clients or servers can continue to work by announcing the version they will use.

Serialization is the process of transforming data structures in memory into a format that can be easily stored or transmitted (not the same as serializability from *Thread Safety*). Rather than invent a new format for serializing your

data between clients and servers, use an existing one. For example, JSON (JavaScript Object Notation) is a simple, widely-used format for serializing basic values, arrays, and maps with string keys.

Specifying a wire protocol

In order to precisely define for clients & servers what messages are allowed by a protocol, use a grammar.

For example, here is a very small part of the HTTP 1.1 request grammar from RFC 2616 section 5 :

```
request ::= request-line
            ((general-header | request-header | entity-header) CRLF)*
            CRLF
            message-body?
request-line ::= method SPACE request-uri SPACE http-version CRLF
method ::= "OPTIONS" | "GET" | "HEAD" | "POST" | ...
...
```

Using the grammar, we can see that in this example request from earlier:

```
GET /aboutmit/ HTTP/1.1
Host: web.mit.edu
```

- `GET` is the `method` : we're asking the server to get a page for us.
- `/aboutmit/` is the `request-uri` : the description of what we want to get.
- `HTTP/1.1` is the `http-version` .
- `Host: web.mit.edu` is some kind of header — we would have to examine the rules for each of the `...-header` options to discover which one.
- And we can see why we had to end the request with a blank line: since a single `request` can have multiple headers that end in CRLF (newline), we have another CRLF at the end to finish the `request` .
- We don't have any `message-body` — and since the server didn't wait to see if we would send one, presumably that only applies for other kinds of requests.

The grammar is not enough: it fills a similar role to method signatures when defining an ADT. We still need the specifications:

- **What are the preconditions of a message?** For example, if a particular field in a message is a string of digits, is any number valid? Or must it be the ID number of a record known to the server?

  Under what circumstances can a message be sent? Are certain messages only valid when sent in a certain sequence?

- **What are the postconditions?** What action will the server take based on a message? What server-side data will be mutated? What reply will the server send back to the client?

Testing client/server code

Remember that concurrency is hard to test and debug . We can't reliably reproduce race conditions, and the network adds a source of latency that is entirely beyond our control. You need to design for concurrency and argue carefully for the correctness of your code.

Separate network code from data structures and algorithms

Most of the ADTs in your client/server program don't need to rely on networking. Make sure you specify, test, and implement them as separate components that are safe from bugs, easy to understand, and ready for change — in part because they don't involve any networking code.

If those ADTs will need to be used concurrently from multiple threads (for example, threads handling different client connections), our next reading will discuss your options. Otherwise, use the thread safety strategies of confinement, immutability, and existing threadsafe data types .

Separate socket code from stream code

A function or module that needs to read from and write to a socket may only need access to the input/output streams, not to the socket itself. This design allows you to test the module by connecting it to streams that don't come from a socket.

Two useful Java classes for this are `ByteArrayInputStream` and `ByteArrayOutputStream` . Suppose we want to test this method:

```
void upperCaseLine(BufferedReader input, PrintWriter output) throws IOException
  requires: input and output are open
  effects: attempts to read a line from input
           and attempts to write that line, in upper case, to output
```

The method is normally used with a socket:

```
Socket sock = ...
```

```
// read a stream of characters from the socket input stream
BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
// write characters to the socket output stream
PrintWriter out = new PrintWriter(sock.getOutputStream(), true);

upperCaseLine(in, out);
```

If the case conversion is a function we implement, it should already be specified, tested, and implemented separately. But now we can now also test the read/write behavior of `upperCaseLine` :

```
// fixed input stream of "dog" (line 1) and "cat" (line 2)
String inString = "dog\ncat\n";
ByteArrayInputStream inBytes = new ByteArrayInputStream(inString.getBytes());
ByteArrayOutputStream outBytes = new ByteArrayOutputStream();

// read a stream of characters from the fixed input string
BufferedReader in = new BufferedReader(new InputStreamReader(inBytes));
// write characters to temporary storage
PrintWriter out = new PrintWriter(outBytes, true);

upperCaseLine(in, out);

// check that it read the expected amount of input
assertEquals("expected input line 2 remaining", "cat", in.readLine());
// check that it wrote the expected output
assertEquals("expected upper case of input line 1", "DOG\n", outBytes.toString());
```

In this test, `inBytes` and `outBytes` are **test stubs** . To isolate and test just `upperCaseLine` , we replace the components it normally depends on (input/output streams from a socket) with components that satisfy the same spec but have canned behavior: an input stream with fixed input, and an output stream that stores the output in memory.

Testing strategies for more complex modules might use a **mock object** to simulate the behavior of a real client or server by producing entire canned sequences of interaction and asserting the correctness of each message received from the other component.

Summary

In the *client/server design pattern* , concurrency is inevitable: multiple clients and multiple servers are connected on the network, sending and receiving messages simultaneously, and expecting timely replies. A server that *blocks* waiting for one slow client when there are other clients waiting to connect to it or to receive replies will not make those clients happy. At the same time, a server that performs incorrect computations or returns bogus results because of concurrent modification to shared mutable data by different clients will not make anyone happy.

All the challenges of making our multi-threaded code **safe from bugs** , **easy to understand** , and **ready for change** apply when we design network clients and servers. These processes run concurrently with one another (if on different machines), and any server that wants to talk to multiple clients concurrently (or a client that wants to talk to multiple servers) must manage that multi-threaded communication.