6.005 Software Construction
Fall 2011
Prof. Rob Miller

# L11: Processes & Sockets

## Today

- Client/server
- Network sockets
- Blocking
- Wire protocols
- Deadlocks

## Required reading (from the Java Tutorial)

- I/O Streams (up to I/O from the Command Line)
- Network Sockets

## Review

Shared memory vs. message passing

Race conditions caused by shared memory access

Today: dig deeper into message passing, and see our first example of deadlock

## Client/Server Design Pattern

in today's lecture (and in PS5) we're going to use a well-established design pattern for message passing called **client/server**.

This pattern has multiple processes communicating by message passing. 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.

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

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

## Network Sockets

a network interface is identified by an IP address (or a hostname, which translates into an IP address; so there may be many synonyms)

examples: 127.0.0.1, localhost; web.mit.edu

an interface has 65536 ports, numbered from 0 to 65535

a server process binds to a port (the listening port). clients have to know which number it's binding to. Some numbers are well-known (port 80 is the standard web server port, port 22 is the SSH port, port 25 is the standard SMTP email server port). When it's not a standard port for the kind of

*These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.*

server, you just treat it as part of the address (you may have seen URLs like http://128.2.39.10:9000? The 9000 is the port number to connect to on the computer at IP address 128.2.39.10.

the listening port is just used to accept incoming client connections. Once the connection is accepted, the server creates a new socket for the actual connection, with a fresh port number (unrelated to the listening port number). Both the client and server sockets have port numbers.

## Buffers

Data is sent over a network in chunks. Rarely just byte-sized chunks (though they may be). The sending side typically writes a big chunk (maybe a whole string like "Hello, world!", or maybe 20 megabytes worth of video data all at once). The network chops that chunk up into packets, which are routed separately over the network. And the receiver reassembles the packets together to a stream of bytes.

The result is a bursty kind of data transmission – the data may be there when you want to read it, or you may have to wait for it.

When data arrives, it is put into a **buffer**, which is simply an array in memory that is holding it until you read it.

## Streams

Stream abstraction: a sequence of bytes

Most modern languages support Unicode, in which characters are 16 bits. But file storage and network transmission uses bytes, which are only 8 bits long.

character set (Unicode) vs. character encoding (Latin-1, UTF-8, UTF-16, Windows horrible)

InputStream/OutputStream vs. Reader/Writers

## Blocking

**Blocking** means a thread waits (doing nothing) until an event occurs. It's usually used to refer to a method call: when a method call **blocks**, it delays returning to its caller until the event occurs.

Socket streams exhibit blocking behavior:

- When an incoming socket's buffer is empty, read() blocks.

- When the destination socket's buffer is full, write() blocks.

Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the read() call will always succeed, no matter what the timing of data arrival. The operating system takes care of the details of delaying your thread until read() *can* succeed.

Blocking happens throughout concurrent programming, not just in I/O. Concurrent modules don't work in lockstep, like sequential programs do, so they typically have to wait for each other to catch up.

We'll see, though, that all this waiting causes the second major kind of bug in concurrent programming: **deadlocks**.

## Wire Protocols

Now that we've got our client and our server and they're 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".

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

HTTP:

```
telnet web.mit.edu 80
GET /
```

The GET command gets a web page; the / is the path of the page you want on the web.mit.edu server. So this command effectively fetches the page at http://web.mit.edu:80/

Internet protocols are defined by RFC specifications (RFC stands for "request for comment").

## Designing a Wire Protocol

similar to defining operations for an abstract data type: small, coherent, adequate

the equivalent of representation independence is platform-independence

ready for change – e.g., version number that client and server can announce to each other. GET / HTTP/1.0

## Data Serialization

Java object serialization

XML

JSON

## Deadlock

When buffers fill up, message passing systems can experience deadlock.

Deadlock: two concurrent modules are both blocked waiting for each other to do something. Since they're blocked, neither will be able to make it happen, and neither will break the deadlock.

In general, in a system of multiple concurrent modules communicating with each other, we can imagine drawing a graph in which the nodes are the modules and there's an edge from A to B if A is blocked waiting for B to do something. The system is deadlocked if at some point in time, there's a cycle in this graph. The simplest case is the two-node deadlock, A -> B and B -> A, but more complex systems can have larger deadlocks.

Deadlocked systems appear to simply hang. They're not done, there's still work to be done, they just can't make any progress.

One solution to deadlock is to design the system so that there is no possibility of a cycle – so that if it's possible for A to wait for B, then it's never possible for B to wait for A.

Another approach to deadlock is *timeouts* – if a module has blocked for too long (maybe 100 milliseconds?  maybe 10 seconds? it depends on the application and how long you need to wait), then you stop blocking and throw an exception.  Then the problem becomes what do you do when that exception gets thrown.

We'll come back to deadlocks again when we talk about locking (which is what gave deadlock its name).