



## Chapter 10: Concurrent and Distributed Programming

# 10.4 Queue-based Message Passing and Socket-based Networking

## 基于队列的消息传递和Socket网络接口

---

Wang Zhongjie  
[rainy@hit.edu.cn](mailto:rainy@hit.edu.cn)

May 26, 2019

# Outline

- **Two models for concurrency**
- **Message passing with threads**
  - Implementing message passing with queues
  - Thread safety arguments with message passing
- **Sockets & Networking: Message passing between two computers**
  - Client/server design pattern
  - Network sockets and I/O
  - Using network sockets and wire protocols
- **Summary**

本章关注复杂软件系统的构造。这里的“复杂”包括三方面：

- (1) 多线程程序
- (2) 分布式程序
- (3) GUI 程序

本节关注第二和第三方面：网络环境下的分布式程序的基本原理、多线程之间基于消息传递的协作机制

本节大部分内容需要自学

# Reading

- MIT 6.031: 22、23
- CMU 15-214: 12、22
- Java编程思想: 第22章





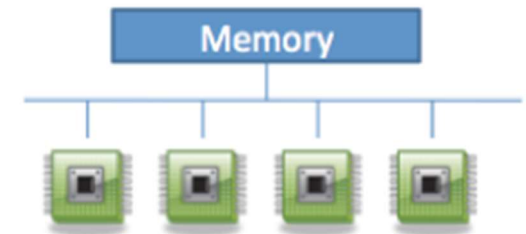
# 1 Two models for concurrency



## Recall: two models for concurrency

- In the **shared memory model**, concurrent modules interact by **reading and writing shared mutable objects in memory**.

- Creating multiple threads inside a single Java process is our primary example of shared-memory concurrency.



- In the **message passing model**, concurrent modules interact by **sending immutable messages to one another over a communication channel**.

- One example of message passing: **the client/server pattern**, in which clients and servers are concurrent processes, often on different machines, and the communication channel is a network socket.



# Advantages of message passing model

- The message passing model has several advantages over the shared memory model, which boil down to **greater safety from bugs**.
  - In message-passing, concurrent modules interact explicitly, by passing messages through the communication channel, rather than implicitly through mutation of shared data. 并发模块显式的交互，通过channel交换信息，而不是通过共享数据；
  - The implicit interaction of shared memory can too easily lead to inadvertent interaction, sharing and manipulating data in parts of the program that don't know they're concurrent and aren't cooperating properly in the thread safety strategy. 隐式交互（在内存中更改数据）导致线程安全问题
  - Message passing also shares only immutable objects (the messages) between modules, whereas shared memory requires sharing mutable objects, which we have already seen can be a source of bugs. 模块之间通过消息传递交换的数据是immutable的，减少bug



## 2 Message passing with threads





# Message passing with threads

- How to implement message passing within a single process?
- We'll use **blocking queues** (an existing threadsafe type) to implement message passing between threads within a process.
- Some of the operations of a blocking queue are *blocking* in the sense that calling the operation blocks the progress of the thread until the operation can return a result.
- Blocking makes writing code easier, but it also means we must continue to contend with bugs that cause deadlock.
- In the next section we'll see how to implement message passing between client/server processes over the network.

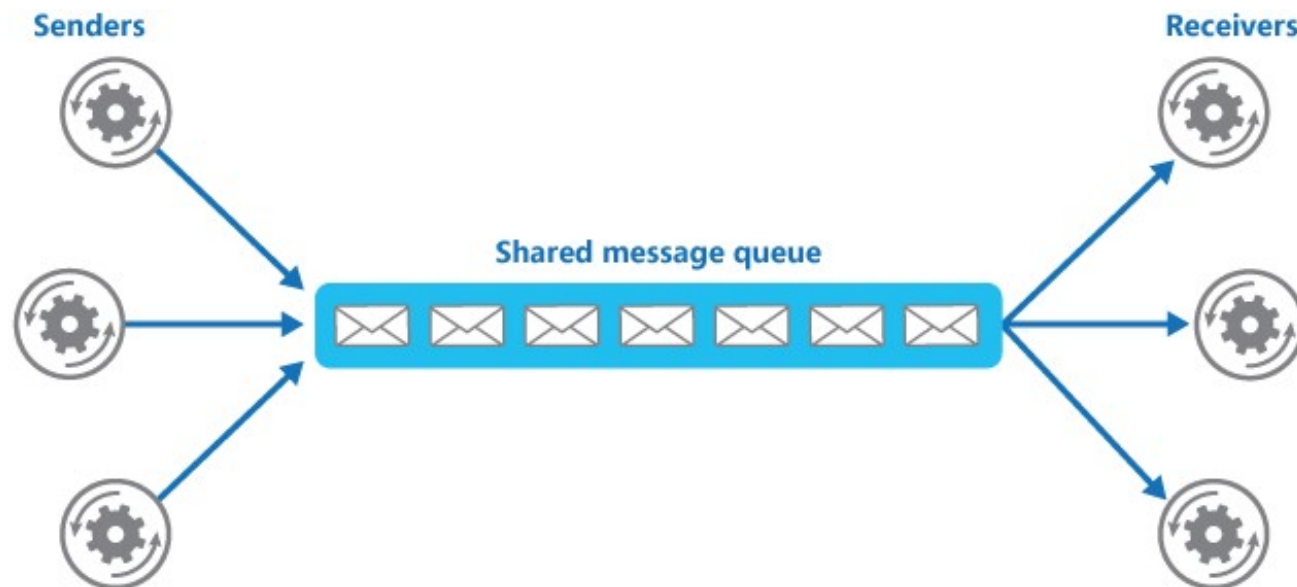


# Blocking Methods

- In locks and synchronization, a thread blocks tries to acquire a lock until the lock has been released by its current owner.
- 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.

# Message passing with threads

- **Message passing between processes: clients and servers communicating over network sockets .**
  - We can also use **message passing between threads within the same process**, and this design is often preferable to a shared memory design with locks.
  - **Use a synchronized queue for message passing between threads.** The queue serves the same function as the buffered network communication channel in client/server message passing.



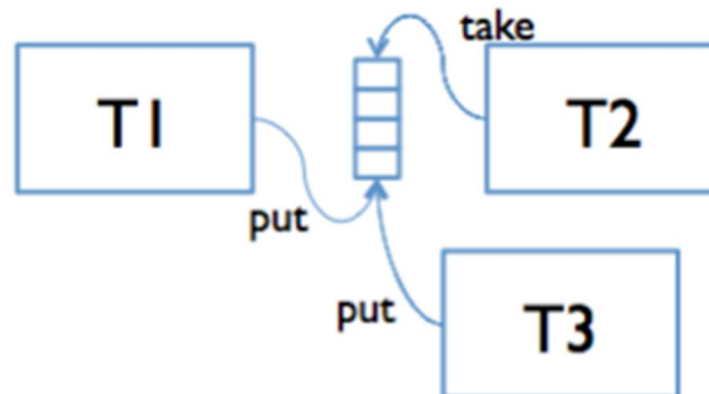
# How Java supports message passing

- **Java provides the BlockingQueue interface for queues with blocking operations:**
- **In an ordinary Queue :**
  - `add(e)` adds element `e` to the end of the queue.
  - `remove()` removes and returns the element at the head of the queue, throws an exception if the queue is empty.
- **A BlockingQueue extends this interface:**
  - Additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
  - `put(e)` blocks until it can add element `e` to the end of the queue (if the queue does not have a size bound, `put` will not block).
  - `take()` blocks until it can remove and return the element at the head of the queue, waiting until the queue is non-empty.

When you are using a `BlockingQueue` for message passing between threads, make sure to use the `put()` and `take()` operations, not `add()` and `remove()` .

# Producer-consumer design pattern

- **Producer-consumer design pattern for message passing between threads.**
  - Producer threads and consumer threads share a synchronized queue.
  - Producers put data or requests onto the queue, and consumers remove and process them.
  - One or more producers and one or more consumers might all be adding and removing items from the same queue.
  - This queue must be safe for concurrency.



# Two implementations of BlockingQueue

- **Java provides two implementations of BlockingQueue :**
  - ArrayBlockingQueue is a fixed-size queue using an array representation. **put** ting a new item on the queue will block if the queue is full.
  - LinkedBlockingQueue is a growable queue using a linked-list representation. If no maximum capacity is specified, the queue will never fill up, so **put** will never block.

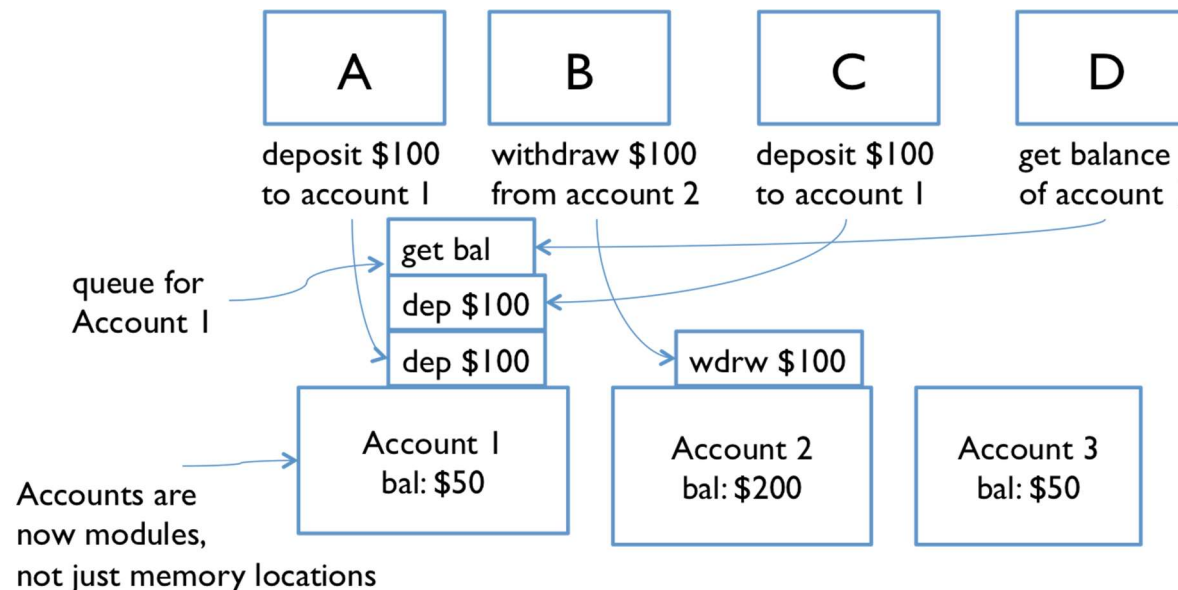
# Designing our own queues as threadsafe ADTs

- Like other collections classes in Java, these synchronized queues can hold objects of an arbitrary type.
- We must choose or design a type for messages in the queue. **It must be an immutable type** to avoid the problems of shared memory.
- Producers and consumers will communicate only by sending and receiving messages, and there will be no opportunity for (mis)communication by mutating an aliased message object.
- And just as we designed the operations on a threadsafe ADT to prevent race conditions and enable clients to perform the atomic operations they need, we will design our message objects with those same requirements.

# Bank account example

- Each cash machine and each account is its own module, and modules interact by sending messages to one another. Incoming messages arrive on a queue.
- We designed messages for **get-balance** and **withdraw**, and said that each cash machine checks the account **balance** before withdrawing to prevent overdrafts:

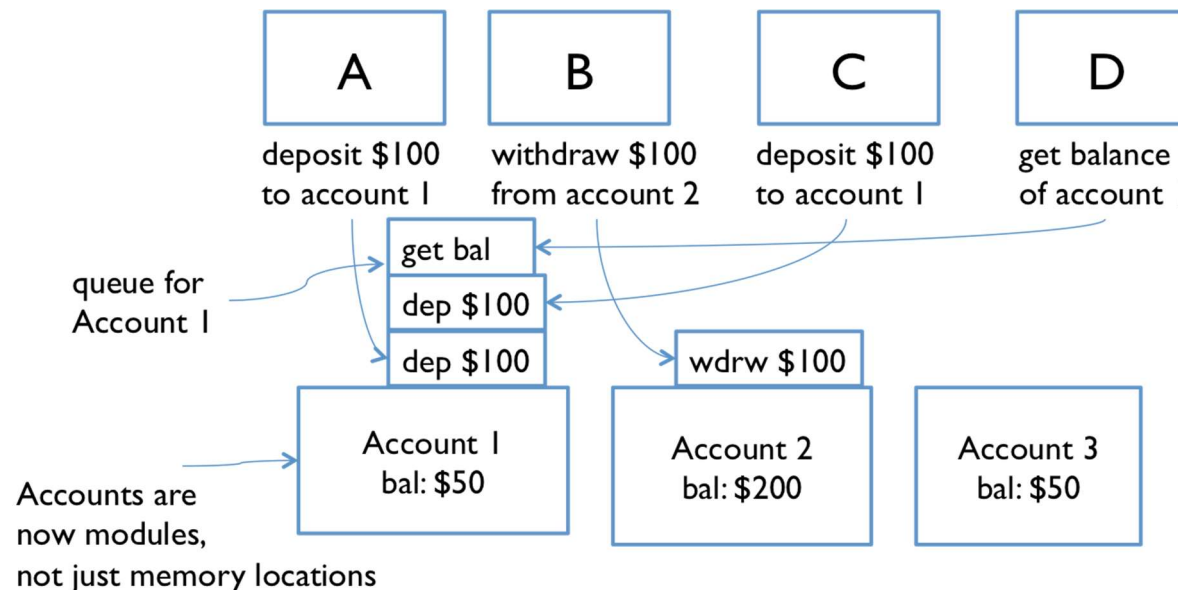
get-balance: if balance >= 1 then withdraw 1





# Bank account example

- But it is still possible to interleave messages from two cash machines so they are both fooled into thinking they can safely withdraw the last dollar from an account with only \$1 in it.
- We need to choose a better atomic operation: **withdraw-if-sufficient-funds** would be a better operation than just withdraw.





# 3 Implementing message passing with queues



# Implementing message passing with queues

```

/** Squares integers. */
public class Squarer {

    private final BlockingQueue<Integer> in;
    private final BlockingQueue<SquareResult> out;
    // Rep invariant: in, out != null

    /** Make a new squarer.
     * @param requests queue to receive requests from
     * @param replies queue to send replies to */
    public Squarer(BlockingQueue<Integer> requests,
                   BlockingQueue<SquareResult> replies) {
        this.in = requests;
        this.out = replies;
    }

    /** Start handling squaring requests. */
    public void start() {
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    // TODO: we may want a way to stop the thread
                    try {
                        // block until a request arrives
                        int x = in.take();
                        // compute the answer and send it back
                        int y = x * x;
                        out.put(new SquareResult(x, y));
                    } catch (InterruptedException ie) {
                        ie.printStackTrace();
                    }
                }
            }
        }).start();
    }
}

```

Message passing  
module

```

/** An immutable squaring result message. */
public class SquareResult {
    private final int input;
    private final int output;

    /** Make a new result message.
     * @param input input number
     * @param output square of input */
    public SquareResult(int input, int output) {
        this.input = input;
        this.output = output;
    }

    @Override public String toString() {
        return input + "^2 = " + output;
    }
}

```

Outgoing message

```

public static void main(String[] args) {

    BlockingQueue<Integer> requests = new LinkedBlockingQueue<>();
    BlockingQueue<SquareResult> replies = new LinkedBlockingQueue<>();

    Squarer squarer = new Squarer(requests, replies);
    squarer.start();

    try {
        // make a request
        requests.put(42);
        // ... maybe do something concurrently ...
        // read the reply
        System.out.println(replies.take());
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
}

```

Main() method that  
uses the squarer

# Stopping

- **What if we want to shut down the Squarer so it is no longer waiting for new inputs?**
  - In the client/server model, if we want the client or server to stop listening for our messages, we close the socket.
  - If we want the client or server to stop altogether, we can quit that process. But here, the squarer is just another thread in the same process, and we can't "close" a queue.
- **One strategy is a poison pill : a special message on the queue that signals the consumer of that message to end its work.**
  - To shut down the squarer, since its input messages are merely integers, we would have to choose a magic poison integer (everyone knows the square of 0 is 0 right? no one will need to ask for the square of 0...) or use null (don't use null).

# Stopping

- Instead, we might change the type of elements on the requests queue to an ADT:

$\text{SquareRequest} = \text{IntegerRequest} + \text{StopRequest}$

with operations:

$\text{input} : \text{SquareRequest} \rightarrow \text{int}$

$\text{shouldStop} : \text{SquareRequest} \rightarrow \text{boolean}$

and when we want to stop the squarer, we enqueue a StopRequest where shouldStop returns true.

# start()

```
public void run() {  
    while (true) {  
        try {  
            // block until a request arrives  
            SquareRequest req = in.take();  
            // see if we should stop  
            if (req.shouldStop()) { break; }  
            // compute the answer and send it back  
            int x = req.input();  
            int y = x * x;  
            out.put(new SquareResult(x, y));  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
    }  
}
```



# interrupt()

- It is also possible to interrupt a thread by calling its `interrupt()` method.
- If the thread is blocked waiting, the method it's blocked in will throw an `InterruptedException` (that's why we have to `try-catch` that exception almost any time we call a blocking method).
- If the thread was not blocked, an interrupted flag will be set.
- The thread must check for this flag to see whether it should stop working.

```
public void run() {  
    // handle requests until we are interrupted  
    while ( ! Thread.interrupted()) {  
        try {  
            // block until a request arrives  
            int x = in.take();  
            // compute the answer and send it back  
            int y = x * x;  
            out.put(new SquareResult(x, y));  
        } catch (InterruptedException ie) {  
            // stop  
            break;  
        }  
    }  
}
```





## 4 Thread safety arguments with message passing



# Thread safety arguments with message passing

- **A thread safety argument with message passing might rely on:**
  - **Existing thread-safe data types for the synchronized queue.** This queue is definitely shared and definitely mutable, so we must ensure it is safe for concurrency.
  - **Immutability** of messages or data that might be accessible to multiple threads at the same time.
  - **Confinement** of data to individual producer/consumer threads. Local variables used by one producer or consumer are not visible to other threads, which only communicate with one another using messages in the queue.
  - **Confinement** of mutable messages or data that are sent over the queue but will only be accessible to one thread at a time. This argument must be carefully articulated and implemented. But if one module drops all references to some mutable data like a hot potato as soon as it puts them onto a queue to be delivered to another thread, only one thread will have access to those data at a time, precluding concurrent access.

# Thread safety arguments with message passing

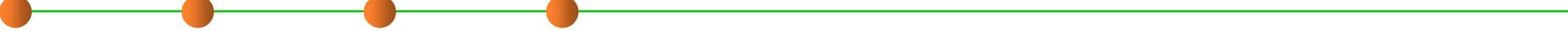
- In comparison to synchronization, message passing can make it easier for each module in a concurrent system to maintain its own thread safety invariants.
- We don't have to reason about multiple threads accessing shared data if the data are instead transferred between modules using a threadsafe communication channel.
- Rather than synchronize with locks, message passing systems synchronize on a shared communication channel, e.g. a stream or a queue.
- Threads communicating with blocking queues is a useful pattern for message passing within a single process.



# 5 Sockets & Networking: Message passing between processes of two computers



# Objectives

- 
- 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.



# (1) Client/server design pattern



# Client/server design pattern

- **Client/server design pattern** is for communication with 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.
  - 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 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.



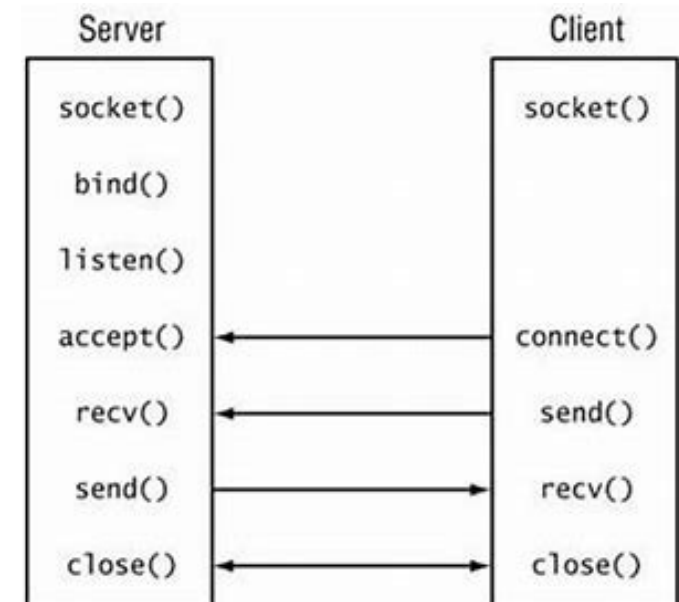


## (2) Sockets and streams



# Sockets & Networking

- **Client/server communication over the network using the `socket` abstraction.** 客户端/服务器模式，通过`socket`进行
  - 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: deadlocks.



# Sockets and streams

- Input/output (I/O) refers to 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 graphical user interface.
  - IP Address
  - Hostnames
  - Port Numbers
  - Network Sockets
  - Buffers
  - Byte Streams
  - Character Streams
  - Blocking

# IP addresses

- A network interface is identified by an IP address . IPv4 addresses are 32-bit numbers written in four 8-bit parts.
  - 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.

Google IP4 Address

216.58.216.164

Google IP6 Address


2607:f8b0:4005:805::200e

# 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.
  - `web.mit.edu` is the name for MIT's web server. You can translate this name to an IP address.
  - `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.
  - `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).

# 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 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.
  - 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.

# Port numbers

- **There are some well-known ports which are reserved for system-level processes and provide standard ports for certain services.**
  - 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 .**



# Network sockets

- **A socket is an endpoint in a network connection used to send and/or receive data**
- **Transport protocol:** TCP or UDP (or Raw IP, but not in Java)
- **Socket address:** local or remote IP address and port number
- **Sockets make network I/O feel like file I/O**
  - Support read, write, open, and close operations
  - Consistent with Unix philosophy “Everything’s a file.”
  - History: first appeared In Berkeley (BSD) Unix in 1983

# 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**.

# TCP networking in Java – `java.net`

- **IP Address – `InetAddress`**
  - `static InetAddress getByName(String host);`
  - `static InetAddress getByAddress(byte[] b);`
- **Ordinary socket – `Socket`**
  - `Socket(InetAddress addr, int port);`
  - `InputStream getInputStream();`
  - `OutputStream getOutputStream();`
  - `void close();`
- **Server socket – `ServerSocket`**
  - `ServerSocket(int port);`
  - `Socket accept();`
  - `void close();`
  - ...

# java.net

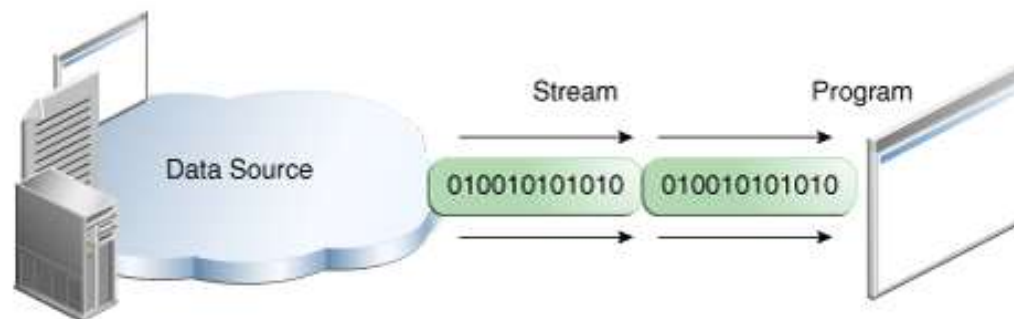
- java.lang.**Object**
  - java.net.**Authenticator**
  - java.net.**CacheRequest**
  - java.net.**CacheResponse**
    - java.net.**SecureCacheResponse**
  - java.lang.**ClassLoader**
    - java.security.**SecureClassLoader**
      - java.net.**URLClassLoader** (implements java.io.Closeable)
  - java.net.**ContentHandler**
  - java.net.**CookieHandler**
    - java.net.**CookieManager**
  - java.net.**DatagramPacket**
  - java.net.**DatagramSocket** (implements java.io.Closeable)
    - java.net.**MulticastSocket**
  - java.net.**DatagramSocketImpl** (implements java.net.SocketOptions)
  - java.net.**HttpCookie** (implements java.lang.Cloneable)
  - java.net.**IDN**
  - java.net.**InetAddress** (implements java.io.Serializable)
    - java.net.**Inet4Address**
    - java.net.**Inet6Address**
  - java.net.**InterfaceAddress**
  - java.net.**NetworkInterface**
  - java.net.**PasswordAuthentication**
  - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - java.security.**BasicPermission** (implements java.io.Serializable)
      - java.net.**NetPermission**
      - java.net.**SocketPermission** (implements java.io.Serializable)
      - java.net.**URLPermission**
  - java.net.**Proxy**
  - java.net.**ProxySelector**
  - java.net.**ResponseCache**
  - java.net.**ServerSocket** (implements java.io.Closeable)
  - java.net.**Socket** (implements java.io.Closeable)
  - java.net.**SocketAddress** (implements java.io.Serializable)
    - java.net.**InetSocketAddress**
  - java.net.**SocketImpl** (implements java.net.SocketOptions)
  - java.net.**StandardSocketOptions**

# 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.**

# Byte streams

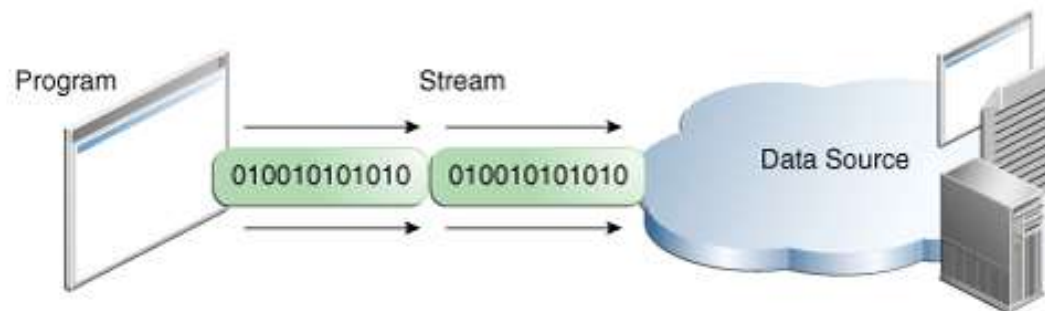
- The data going into or coming out of a socket is a stream of bytes.
- In Java, **InputStream** objects represent sources of data flowing into your program:
  - Reading from a file on disk with a **FileInputStream**
  - User input from **System.in**
  - Input from a network **socket**



- See Chapter 8.1

# Byte streams

- **OutputStream** objects represent data sinks, places we can write data to:
  - **FileOutputStream** for saving to files
  - **System.out** is a **PrintStream**, an **OutputStream** that prints readable representations of various types
  - Output to a network **socket**
- With sockets, remember that the *output* of one process is the *input* of another process.



- See Chapter 8.1



# Character streams

- **The stream of bytes provided by `InputStream` and `OutputStream` is often too low-level to be useful.**
  - We may need to interpret the stream of bytes as a stream of Unicode characters, because Unicode can represent a wide variety of human languages (not to mention emoji).
  - A `String` is a sequence of Unicode characters, not a sequence of bytes, so if we want to use strings to manipulate the data inside our program, then we need to convert incoming bytes into Unicode, and convert Unicode back to bytes when we write it out.
- **In Java, `Reader` and `Writer` represent incoming and outgoing streams of Unicode characters:**
  - `FileReader` and `FileWriter` treat a file as a sequence of characters rather than bytes
  - The wrappers `InputStreamReader` and `OutputStreamWriter` adapt a byte stream into a character stream



# 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.
- **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

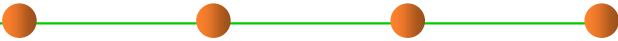
- **Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the `read` (or `write`) call will always work, no matter what the timing of data arrival.**
  - If data (or for write, space) is already available in the buffer, the call might return very quickly.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

# Blocking

- Blocking also introduces the possibility of deadlock, which arises any time two (or more) concurrent modules are both blocked waiting for each other to do something.
  - Since they're blocked, no module will be able to make anything happen, and none of them 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 of the graph are modules, and there's an edge from A to B if module A is blocked waiting for module B to do something.
  - The system is deadlocked if at some point in time, there is a *cycle* in this graph.
  - The simplest case is the two-node deadlock,  $A \rightarrow B$  and  $B \rightarrow A$ , but more complex systems can encounter larger deadlocks.



## (3) Using network sockets in Java



# Client code

## Creating server- and client-side sockets and writing to and reading from I/O streams.

```
String hostname = "localhost";  
int port = 4589;  
Socket socket = new Socket(hostname, port);
```

```
OutputStream outToServer = socket.getOutputStream();  
InputStream inFromServer = socket.getInputStream();
```

```
PrintWriter writeToServer =  
    new PrintWriter(new OutputStreamWriter(outToServer, StandardCharsets.UTF_8));  
BufferedReader readFromServer =  
    new BufferedReader(new InputStreamReader(inFromServer, StandardCharsets.UTF_8));  
BufferedReader readFromUser =  
    new BufferedReader(new InputStreamReader(System.in, StandardCharsets.UTF_8));
```

```
while (true) {  
    String message = readFromUser.readLine();  
    writeToServer.println(message);  
    writeToServer.flush();  
    String reply = readFromServer.readLine();  
    if (reply == null) break;  
}
```

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Which lines might be blocked?

```
readFromServer.close();  
writeToServer.close();  
socket.close();
```

# Server code

**ServerSocket** is a java.net class that provides a system-independent implementation of the server side of a client/server socket connection.

```
int port = 4589;
ServerSocket serverSocket = new ServerSocket(port);

Socket socket = serverSocket.accept();
PrintWriter writeToClient =
    new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
                                           StandardCharsets.UTF_8));

BufferedReader readFromClient =
    new BufferedReader(new InputStreamReader(socket.getInputStream(),
                                           StandardCharsets.UTF_8));

while (true) {
    String message = readFromClient.readLine();
    if (message == null) break;
    if (message.equals("quit")) break;


    String reply = "echo: " + message;
    writeToClient.println(reply);
    writeToClient.flush();
}

readFromServer.close();
writeToServer.close();
socket.close();
```

Which lines might  
be blocked?

# Multithreaded server code

```
while (true) {  
    // get the next client connection  
    Socket socket = serverSocket.accept();  
    // handle the client in a new thread, so that the main thread  
    // can resume waiting for another client  
    new Thread(new Runnable() {  
        public void run() {  
            handleClient(socket);  
        }  
    }).start();  
}  
  
private static void handleClient(Socket socket) {  
    // same server loop code as above:  
    // open readFromClient and writeToClient streams  
    // while (true) {  
    //     read message from client  
    //     prepare reply  
    //     write reply to client  
    // }  
    // close streams and socket  
}
```

A red arrow originates from the `handleClient(socket);` line within the `Runnable` thread's `run()` method and points to the `handleClient` method definition below the main loop.

# Closing streams and sockets with try-with-resources

Recall Chapter 7-2 Exception Handling

```
try (  
    // preamble: declare variables initialized to objects that need closing after use  
) {  
    // body: runs with those variables in scope  
} catch(...) {  
    // catch clauses: optional, handles exceptions thrown by the preamble or body  
} finally {  
    // finally clause: optional, runs after the body and any catch clause  
}  
// no matter how the try statement exits, it automatically calls  
// close() on all variables declared in the preamble
```

Useful for any object that should be closed after use:

- byte streams: **InputStream**, **OutputStream**
- character streams: **Reader**, **Writer**
- files: **FileInputStream**, **FileOutputStream**, **FileReader**, **FileWriter**
- sockets: **Socket**, **ServerSocket**



# Questions

- **Which of these are necessary for a client to know in order to connect to and communicate with a server?**
  - Server IP address
  - Server hostname
  - Server port number
  - Server process name
  - Wire protocol
- **Alice has a connected socket with Bob. How does she send a message to Bob?**
  - Write to her socket's input stream
  - Write to her socket's output stream
  - Write to Bob's socket's input stream
  - Write to Bob's socket's output stream

# Questions

- Since `BufferedReader.readLine()` is a blocking method, which of these is true:
  - When a thread calls `readLine`, all other threads block until `readLine` returns
  - When a thread calls `readLine`, that thread blocks until `readLine` returns
  - When a thread calls `readLine`, the call can be blocked and an `exception` is thrown
  - `BufferedReader` has its own thread for `readLine`, which runs a block of code passed in by the client

```
String message = readFromClient.readLine();
```



## (4) Wire protocols



# 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.
  - HTTP
  - FTP
  - SMTP
  - ...

# Hypertext Transfer Protocol (HTTP)



- It is a web browser's responsibility to parse the output and show in browser.

```
$ 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, 18 Apr 2017 15:25:23 GMT
... more headers ...
```

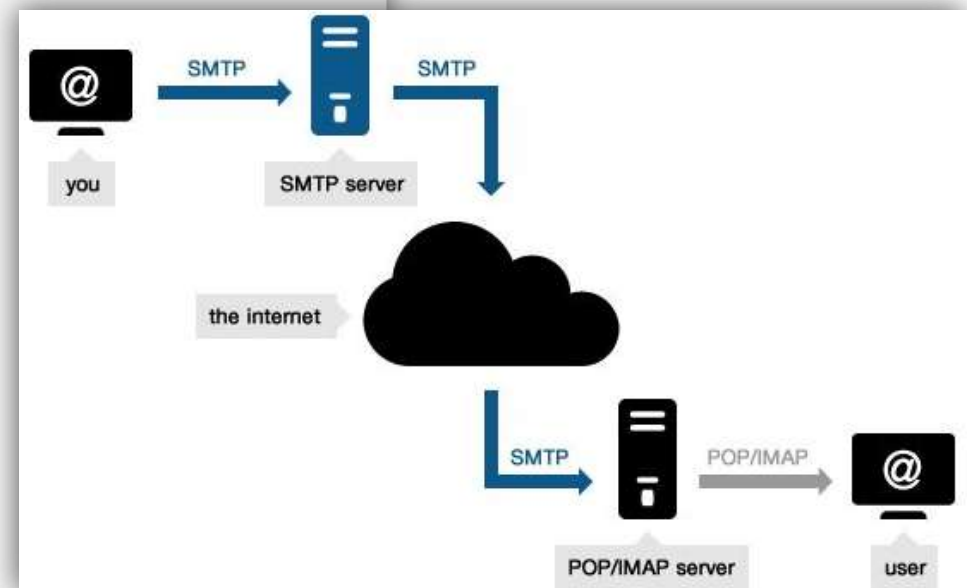
```
9b7
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
... more HTML ...
<title>MIT - About</title>
... Lots more HTML ...
</html>
```

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

# Simple Mail Transfer Protocol (SMTP)

- Simple Mail Transfer Protocol is the protocol for sending email.

```
$ 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.
```





# Designing a wire protocol

- When designing a wire protocol, apply the same rules of thumb you use for designing the operations of an ADT:
  - 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.
- As representation independence from ADTs, we should aim for **platform-independence** in wire 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.

# 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" | ...
...
```

```
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 don't have any message-body



# Specifying a wire protocol

- 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?

# An Example

## ■ Messages from the client to the server

- The client can turn lights, identified by numerical IDs, on and off. The client can also request help.

```
MESSAGE ::= ( ON | OFF | HELP_REQ ) NEWLINE
ON ::= "on " ID
OFF ::= "off " ID
HELP_REQ ::= "help"
NEWLINE ::= "\r"? "\n"
ID ::= [1-9][0-9]*
```

## ■ We'll use ↵ to represent a newline.

Which of these is a valid message from the client?

- on ↵
- on 0↵
- help
- off 1000000000000000↵
- OFF 1↵

# An Example

## ■ Messages from the server to the client

- The server can report the status of the lights and provides arbitrary help messages.

```
MESSAGE ::= ( STATUS | HELP ) NEWLINE
STATUS ::= ONE_STATUS ( NEWLINE "and " ONE_STATUS )*
ONE_STATUS ::= ID " is " ( "on" | "off" )
HELP ::= [^\r\n]+
NEWLINE ::= "\r"? "\n"
ID ::= [1-9][0-9]*
```

Which of these is a valid message from the server?

- 1 is off
- 1 is off↵
- 1 is off↵ and 2 is on↵
- Isn't this awesome? You can turn lights on and off!↵
- Turn lights on and off using the commands:↵on↵off↵



## (5) Testing client/server code



# Testing client/server code

- **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.**
- **(1) 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 .

# Testing client/server code

- **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.



# Summary



# Summary

## ■ Message passing with threads

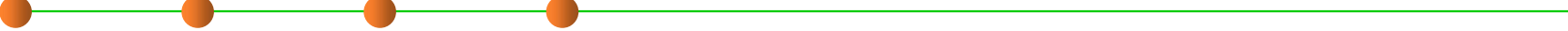
- Rather than synchronize with locks, message passing systems synchronize on a shared communication channel, e.g. a stream or a queue.
- Threads communicating with blocking queues is a useful pattern for message passing within a single process.

## ■ Sockets & Networking: Message passing between two computers

- 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.



# Summary

- 
- 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 (often 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.



The end

May 26, 2019