Distributed Applications

Networking Basics

Georgia Tech

What is a "Network?"

- Depends on what level you're at
- One person's "network" is another person's "application"
- OSI Seven Layer Model
  - The physical wire itself
  - Ethernet, 802.11b
  - Routing protocols
  - ...

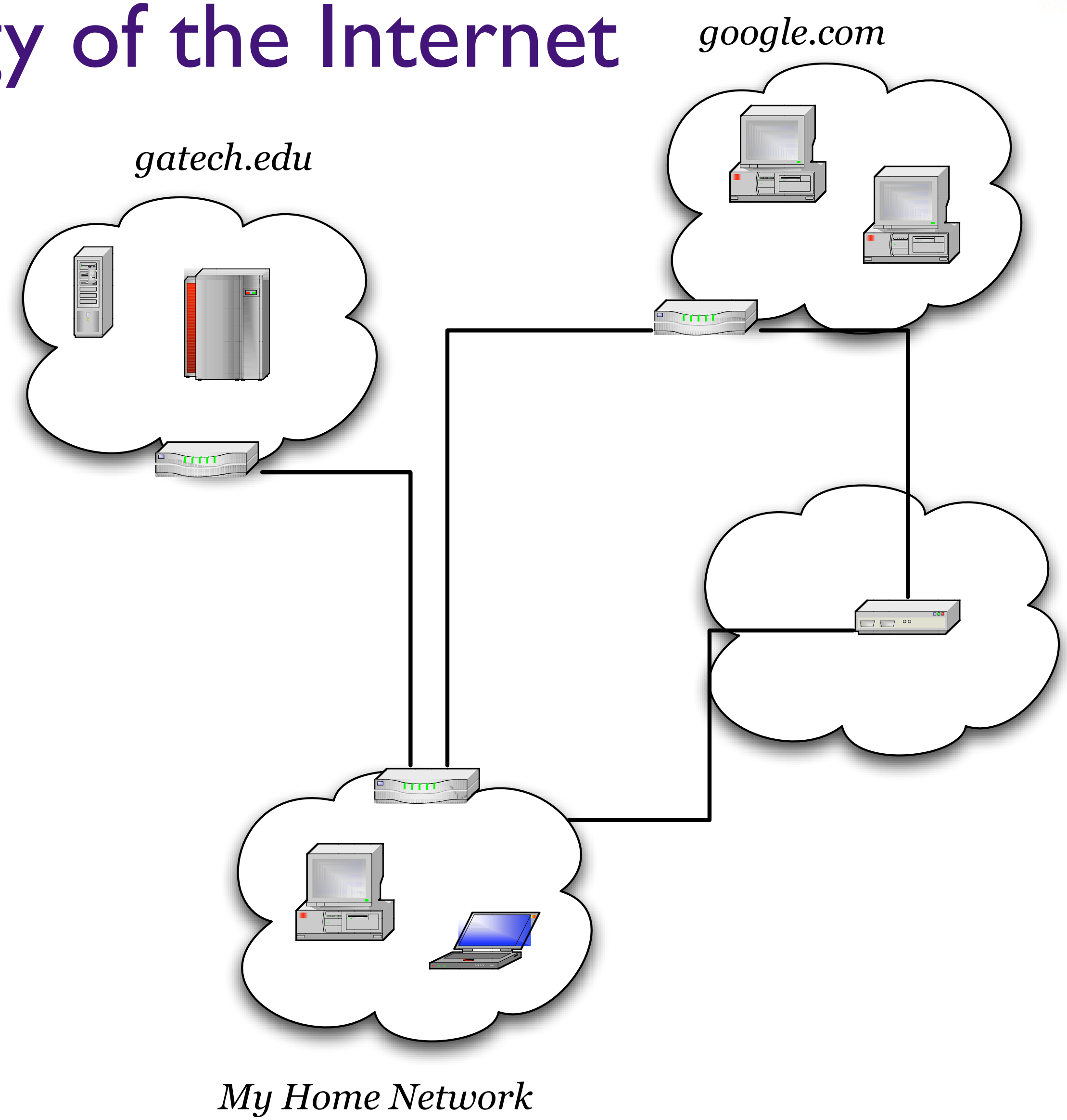
7. Application	FTP, HTTP, SMTP, etc.
6. Presentation	
5. Session	
4. Transport	TCP
3. Network	IP
2. Data Link	ARP, RARP
1. Physical	Ethernet

For Our Purposes: The Internet

- We're *application programmers*
- In terms of OSI, we're defining/using our own *application-layer protocol*
- Sits atop TCP/IP, the lingua franca of the Internet
- For almost every networked application you will ever want to build, this will be the lowest layer in the stack you'll need to care about

7. Application	FTP, HTTP, SMTP, etc.
6. Presentation	
5. Session	
4. Transport	TCP
3. Network	IP
2. Data Link	ARP, RARP
1. Physical	Ethernet

Topology of the Internet



Some Terminology: Protocols

- **Protocols**: rules that facilitate information exchange among programs on a network
  - Example from human world: "roger" and "over" for radio geeks
- Similar to how you design the interfaces between objects in your program
  - A callback expects to get a certain set of parameters in a certain order
  - You need to know this in order to use the callback
- Likewise:
  - A networked program expects you to communicate with it in certain ways (using certain messages, in a known format)
  - You need to know this in order to use the program

Some Terminology: Servers

- **Server**: a (generally) long-lived program that sits around waiting for connections to it
  - Examples: web server, mail server, file server, IM server
- "Server" implies that it does something useful (delivers a *service*)
  - Web server: provides access to HTML documents
  - Mail server: allows retrieval, sending, organization of email messages
  - File server: provides remote access to files and directories
  - IM server: provides info about online users, passes messages between them

Some Terminology: Clients

- **Client**: a program that connects to a server to use whatever service it provides
  - Examples:
    - Web browser connects to web servers to access/view HTML documents
    - Mail client (Outlook, etc.) connects to mail servers for mail storage, transmission
    - IM clients connect to IM servers to access info about who is on, etc.
- Most servers can be connected to by multiple clients at the same time

Some Terminology: Host

- **Host**: Simply a machine that's connected to the network
- Generally running clients and/or servers
  - The machine "hosts" a server

The Next Phase of the Project

- We'll be building the networking part of the IM program
  - Enhancing the GUI code to talk to an an IM server on the network
- For the IM assignment:
  - I'll provide a sample IM server, and documentation on its protocol
- Important concept: understanding a protocol specification
  - Useful for when you want to write a program that talks to an existing server (and thus has its own existing, documented protocol)
  - Side concept: designing your *own* protocols
    - We'll talk about this, but won't do it for the project (unless you want to go nuts and get all fancy...)
- Should give you experience in using basic Internet-style networking, debugging, etc.

What Will You Have to Do?

1. Connect to the other machine(s)
   - Know how to refer to it: which machine do you want to connect to?
   - Know how to perform the connection
   - Know how to deal with errors (server is down, etc.)
2. Send messages to it (e.g., "I'm online now!")
   - Know how to "marshall" arguments
   - Know how to do the transmission
   - Know how to deal with errors (server crashed while sending, etc.)
3. Receive messages from it (e.g., list of online users)
   - Know how to "unmarshall" arguments
   - Know how to read data
   - Know how to deal with errors (e.g., got unexpected data from server, etc.)
4. Disconnect from it
   - This is the easy part!

Why All the Focus on Errors?

- Networking in inherently error-prone
- Different than single application programming
  - Errors generally result from a bug, and just crash entire program
- Networking: errors may be caused by reasons outside of your control
  - Network is down, server has crashed, server slow to respond, etc.
  - During a chat I could shut my laptop and walk away
  - Someone could trip over the power cord for an access point
  - Networks can't even guarantee that messages will get from A to B
- Good goal: *robustness*
  - Your program should survive the crash of another program on the network, receiving malformed data, etc
  - "Defensive programming"

Networking 101

Internet Addressing

- Every machine on the Internet has an *address*
- Internet addresses are sequences of 4 bytes
  - Usually written in "dotted quad" notation
  - Examples: 192.168.13.40, 13.2.117.14
- Addresses identify a particular machine on the Internet
  - Example: 64.223.161.104 is the machine www.google.com
- One special address
  - 127.0.0.1
  - *localhost*
  - Refers to the local machine always

Where do IP Addresses Come From?

- You can't just set your IP address to any random value and have it work
  - The rest of the Internet won't know how to reach you
  - You have to use values that are compatible with whatever network you're on
- In most cases a service called *DHCP* will take care of this for you
  - *Dynamic Host Configuration Protocol*
  - Assigns you a valid IP address when you boot your machine, wake your laptop, etc.
  - E.g., LAWN at Georgia Tech
  - IP address may change from time to time: in other words, don't count on this being your address forever
- If DHCP isn't available, you may have to set your IP address by hand, but only with a value provided by an administrator

Why Do You Need to Know This?

- First off: **don't change your IP address for this class!**
  - You can only do harm!
- Second: if you get an address from DHCP (which you probably do), you can't count on having this address forever
  - So don't hard-code it into any programs
- Third: if you want to debug clients and servers on the same machine, you can use the localhost address
  - But don't hardcode this either, since it would keep you from working when client and server are on *different* machines

Public Versus Private Addressing

- Not all IP addresses may be *reachable* from any given machine
- Simple case: machines behind a *firewall*
  - Example: my old machine at PARC was 13.1.0.128, but only reachable from within PARC
- More complex case:
  - Some IP addresses are *private* (also called *non-routable*)
  - Three blocks of addresses that cannot be connected to from the larger Internet
    - 10.0.0.0 - 10.255.255.255
    - 172.16.0.0 - 172.31.255.255
    - 192.168.0.1 - 192.168.255.255

Why Private Addresses?

- Two reasons: IP address conservation and security
  - Public addresses *uniquely* define a given machine
    - There's a limited number of these, and they're running out
  - Private addresses can be reused (although not on the same network)
    - Probably hundreds of thousands of machines with 192.168.0.1 on private networks (corporation internal, homes, etc.)
  - Certain network configs let you *share* a single public IP address across multiple private machines
    - *Network Address Translation*
    - Built into most home routers
      - E.g., BellSouth gives me the address 68.211.58.142
      - My router gives my home machines 192.168 addresses
      - Connections *out* are translated so that it looks like they come from 68.211.58.142
      - Internal machines are "invisible" since they have non-routed addresses

Why Do You Need to Know This?

- Servers running on machines with private IP addresses are not reachable from machines not on that network
  - Ok if you're running your client and service on the same network
  - Ok if you're running your client and service on the same machine
  - **Not** ok if, e.g., your server is at home and you client is at Georgia Tech
- *Aside:* this is the reason that many people pay for an extra "static" IP address at home--so that they can run servers that have a fixed IP address that is visible throughout the Internet

Naming

- When you go to a web browser, you don't type in 64.223.161.104, you type in www.google.com
- The *Domain Name Service*
  - A big distributed database of all the machines on the Internet
  - Each organization manages its own little portion of it
  - Maps from host names to IP addresses
- Ultimately, the Internet runs on IP addresses. Names are a convenience for humans
  - When you type www.google.com, the browser *resolves* that name to an IP address by talking to a DNS server
  - If *name resolution* can't be done (DNS is down; you're not connected to the network), then browsing will fail

Naming Configuration

- Much like IP addressing, you may not have much control over the DNS name for your machine
  - In general, you *won't* have a name resolvable by DNS, even if your machine has a "local" name
  - In the CoC, CNS sets up DNS names for the machines they administer, mapping them to fixed IP addresses
    - If you were to take these machines to different networks (where they get different IP addresses), those names would no longer work
    - Resolve to the incorrect address
  - Personally owned machines, even if they get an IP address from DHCP, generally get sucky names, if they get a name at all
    - Example: lawn-199-77-214-212 on my laptop

Why Do You Need to Know This?

- General all-around erudition and cocktail party conversation :-)
- Even though we're used to using names to refer to machines on the Greater Internet, you'll probably be reduced to using IP addresses for this assignment
- We may be able to run a server on a well-known machine, administered by TSO, in which case you'd be able to specify it by name

Ports

- What if you've got multiple servers running on a single host?
  - E.g., a machine might have a web server, mail server, FTP server, ...
- When you tell a client to connect to a given machine, how does it know which server running on that machine to talk to?
- **Ports**: Let you address different servers running on the same machine
  - Think of IP addresses as the street address for an apartment building
  - Ports specify the individual apartments
- Ports are just numbers that range from 0-65,535

More On Ports

- Back to the question: when I type www.google.com into my browser...
  - It knows to go to 64.233.161.104
  - But how does it know which is the port for the google web server?
- *Well-known ports*: certain common Internet services use standard port numbers:
  - Web servers: port 80
  - FTP servers: port 21
- Terminology: we say that the FTP server *runs on* port 21, meaning that this is the port at which it is waiting for clients to connect to it
- *Reserved ports*: ports 0-1024 reserved for privileged programs
- Servers specify which port they run on when they start
- Clients specify *both* the IP address of the desired host, and the port number, when they connect to a server
- Clients outgoing connections *also* have a port, but generally you don't need to know what it is
- Only one client or service can run on a port at any given time

Why Do You Need to Know This?

- If you're writing a client for an existing service, you'll have to know what port is is running on in order to connect to it
- If you write a service, you'll need to run it on a port that will be known by its clients
  - Can be a fixed port number that you decide on, and tell your clients
  - Can let the system assign you a random one, but then you'll need some way to communicate this to clients
- You can't choose ports in the reserved range
- Good practice is to use relatively high numbers (e.g., 5,000 - 50,000)

Network Programming 101

Basic Network Programming

- One unified concept for dealing with the network at the Internet layer: **sockets**
- Basically similar across all platforms (Java, C, Python, etc.)
- De facto standard (slight differences across platforms, languages)
- So what's a socket?
  - An endpoint for communication
  - May be connected to another endpoint, in another program on the net
  - Lets you read from it and write to it, much like a file
  - Adds some additional operations specific to networking

Network Programming from the Client's Perspective

1. *Create* a socket
2. *Bind* it to an address on a client machine
   - Both endpoints of a communication have addresses, including ports
3. *Connect* it to the server, by specifying its address and port
   - This call blocks until the connection is successful, or times out
4. *Read* and *write* to and from the socket, to get and send data
5. *Close* the socket when you're done with it

Network Programming from the Server's Perspective

1. *Create* a socket
2. *Bind* it to an address on the server machine
   - This sets the port for the socket
3. *Listen* for incoming connections
4. *Accept* any connection that comes in.
   - This call *blocks* until a new connection comes in
   - This produces a **new** socket, paired with the client, and just for communication with that client
   - This socket can be read, written, and closed independently from the socket used for any other client
   - Meanwhile, original listening socket can go back to listening
   - Allows you to have multiple ongoing client connections at one time
5. *Close* the listening socket when you're done accepting connections

Example: Basic Socket Programming in Jython

Need to import the socket module before doing anything:

import socket

How to create a socket (for both clients and servers):

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

How to connect to a server (if you're a client):

s.connect((”192.168.2.54”, 45235))

Note *double parentheses!*
This is because connect() takes a single tuple as an argument; this tuple contains the address and port. (The inner set of parentheses wraps the arguments in a tuple.)

How to bind to an address, listen for connections and accept them (if you're a server):

s.bind((“”, 45235))

Note *double parentheses!*
Same with bind()

s.listen(5)

newSock, clientAddress = s.accept()

How to send and receive data (both clients and servers):

s.send(”hello world”)

reply = sa.recv(1024)

How to close a socket (both clients and servers):

s.close()

Writing a Simple Server

(All of this code is on the web site, as net-sampler.py)

```
import socket
import sys

class SimpleServer:
    def __init__(self, port):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind(('', port))
        self.sock.listen(5)
        while 1:
            requestSock, peerAddress = self.sock.accept()
            print "Accepted connection from", peerAddress
            while 1:
                input = requestSock.recv(1024)
                if not input:
                    print "Peer closed connection"
                    break
                requestSock.send(input)

            requestSock.close()

if __name__ == "__main__":
    port = 7777
    if len(sys.argv) > 1:
        port = sys.argv[1]
    server=SimpleServer(port)
```

Writing a Simple Client

```
import socket
import sys

class SimpleClient:
    def __init__(self, serverAddr, serverPort):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((serverAddr, serverPort))

    def sendToServer(self, message):
        self.sock.send(message)
        return self.sock.recv(1024)

    def close(self):
        self.sock.close()

if __name__ == "__main__":
    if len(sys.argv) != 3:
        sys.exit(1)
    else:
        client = SimpleClient(sys.argv[1], int(sys.argv[2]))

    while 1:
        string = sys.stdin.readline()
        if string == "close\n":
            client.close()
            sys.exit(0)
        else:
            response = client.sendToServer(string)
            print "Server replied '", response,  "'"
```

Extra Useful Tricks

- Figuring out what you're connected to:
  - s.getpeername() returns a tuple of (address, port) indicating what you're connected to (or what has connected to you)
- Figuring out your local address:
  - s.getsockname() returns a tuple of (address, port) indicating your local address. Useful when you need to know what port your service is on
- Making life easier:
  - s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
  - Tells the OS that it's ok to reuse a port number
  - Example: you find a bug, kill your server, fix it, and restart
  - Without this call, OS may prevent the port from being reused until some timeout expires

Multi-threaded Servers

- Problem with previous simple server:
  - While it's processing requests from one client, every other client must queue up
  - Only when first client dies does the next one in the queue get handled
- Bad, since most servers should support connections by multiple clients at the same time
- Common approach: multi-threaded servers
  - One thread to hang around waiting for clients to appear
  - One thread to handle each client; terminates when client is done

Multi-Threaded Server Example

```
import socket
import sys
import threading

class MTServer:
	def __init__(self, port):
		self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
		self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
		self.sock.bind(('', port))
		self.sock.listen(1)
		while 1:
			requestSock, peerAddress = self.sock.accept()
			handler = Handler(requestSock)

class Handler:
	def __init__(self, requestSock):
		self.requestSock = requestSock
		self.thread = threading.Thread(target=self.handle)
		self.thread.start()

	def handle(self):
		while 1:
			input = self.requestSock.recv(1024)
			if not input:
				break
			self.requestSock.send(input)
		self.requestSock.close()

if __name__ == "__main__":
	port = 7777
	if len(sys.argv) > 1:
		port = sys.argv[1]
	server=MTServer(port)
```

Message Formatting

- Any messages you send to a server must be parseable by it
  - Recipient must be able to decipher what you sent it
  - Must know when it has reached the end of the message
- There are many ways of encoding messages

The Joy of ASCII

- Many protocols use a simple text-based encoding
  - Example: HTTP

    GET /index.html HTTP/1.0
  - Example: SMTP

    HELO rutabaga.cc.gatech.edu

    MAIL From: Keith Edwards <keith@cc>

    DATA

    Hello there!

    .
- Parameters and commands encoded using simple, regular format
- *Marshalling*: the process of gathering parameters and encoding them for transmission
- *Unmarshalling*: the process of unpacking the received data for use by your program
- Goal should be *machine* parseability for ease of implementation; *human* parseability for ease of debugging

More Complex Data

- What about very complex data?
- Example: marshalling an arbitrary Jython dictionary
  - {"name":"keith,"location": (2.425, 1.783, 0.892),"info": {"email": "keith@cc","phone": 56783},"buddies":"[ralph","fred","betty"] }
- You *could* create a string representation that is parseable and "rebuildable" on the other end
- Sometimes called *flattening* the dictionary to a string
- Parsing at the recipient can be very difficult
- Need to account for *arbitrary* objects that might be stored in dictionaries (including custom-defined objects)

Is There an Easier Way?

- Most “standard” services just bite the bullet and use ASCII
  - Perhaps with more complex formatting atop it, such as XML
  - ASCII--since it’s universal--lets you program a client in any language that speaks the necessary protocol
- The marshalling/unmarshalling of complicated parameters can be a significant part of the complexity in dealing with a given service
- **But:** If you *know* you’ll only be working with clients in a particular language, you can take some short cuts

Serialization

- *Serialization* is the process of automatically creating a representation of complex data that can be shipped over the wire
- Generally *built in* to the programming language itself
  - So: can work with custom-defined data types without special work by the programmer
  - Present in Java, Python, Jython, ...
- *Opaque*: with most of these systems, you don't care what the on-the-wire representation is
  - Generally complex; generally non-ASCII
  - System takes care of the chores of generating it, and parsing it
- Terminology: a serialization system is one approach to simplifying the marshalling and unmarshalling of arguments

Serialization in Jython/Python

- Serialization provided by the *pickle* library
  - You "pickle" objects for transmission over the wire
- Works for any Jython data type, including custom-defined objects
  - However: some objects may "depickle" with data intact, but not behave as expected
  - Classic example: swing widgets

Sending Dictionaries Using Pickle

- On the sending side:

  import pickle

  dict = {"name":"keith, "location": (2.425, 1.783, 0.892), "info": {"email": "keith@cc", "phone": 56783}, "buddies": "[ralph", "fred", "betty"] }

  data = pickle.dumps(dict)

  s.send(data)

- On the receiving side:

  data = s.recv(1024)

  dict = pickle.loads(data)

Combining Pickling with Other Techniques

- Pickled objects are *opaque*--you can't easily parse the data yourself

  But you generally don't want/need to. Just unpickle it!

- Common approach is to create a data structure that represents the entire message and pickle it
  - Sender:

    ```
    s.send(pickle.dumps((”Hello”, dict)))
    ```

  - Receiver:

    ```
    pickle.loads(s.recv(1024))
    ```

Instant Messaging Assignment

- Turn the GUI front end into a working network-ified program
- Grab the server off the class web page
- Understand the protocol it speaks
- Integrate it into your client
  - Connect to the server
  - Send messages to it in response to starting up, user events (such as new chats), etc.
  - Be prepared to receive messages from it
    - Asynchronous notifications of online users: *necessitates having a thread to listen for messages!*
    - Responses to client-initiated messages (invitations, etc.)

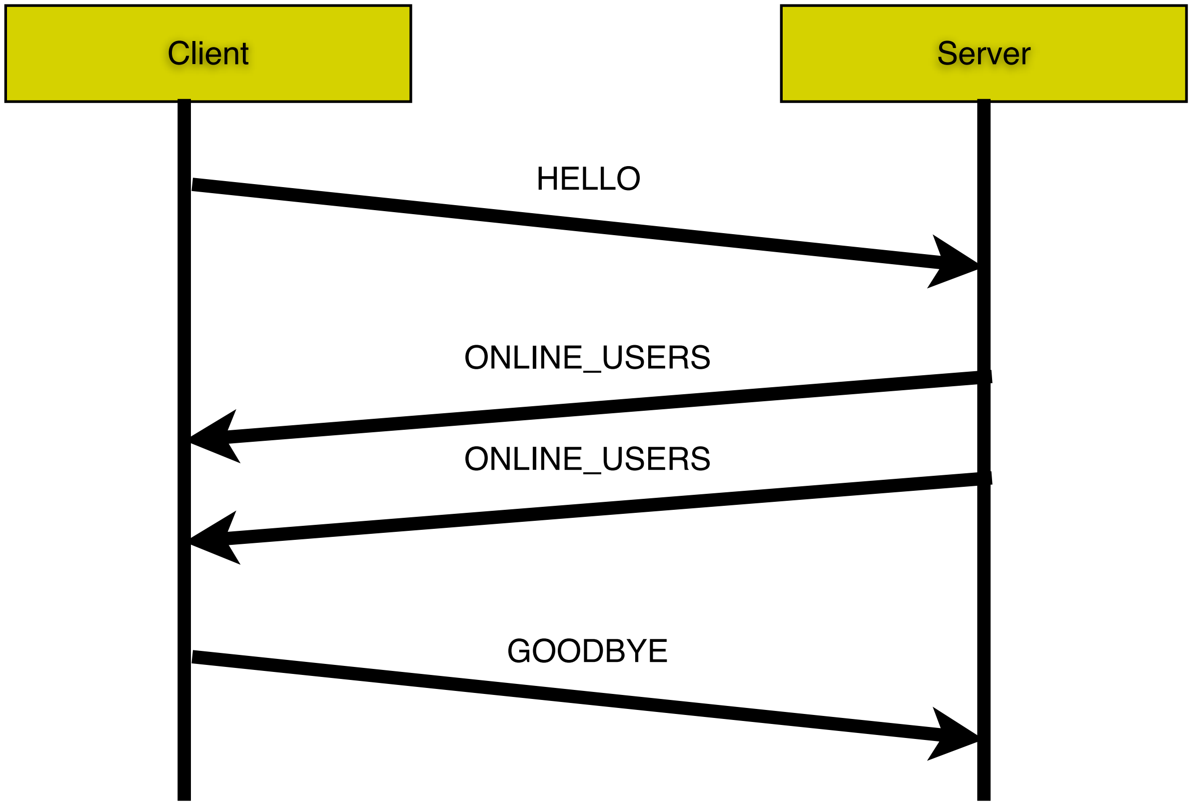
Getting Started

- Get code off the web site: server.py
- Running the server
  - jython server.py
  - Will run on port 6666
  - Generates a lot of debugging messages (don't run under JES though)
  - Look at the *handle* messages in the server if you need to see what it's doing
- Create a client to connect to this port
  - Start small! Create a new file net.py
  - Generate a message to tell the server that you're online
  - Next, make the online user list "real": thread to listen for incoming messages
  - Debug by running multiple instances of the client (as different users)
  - Pay attention to server debugging messages!
  - Iron out the connection, messaging issues *then* integrate it

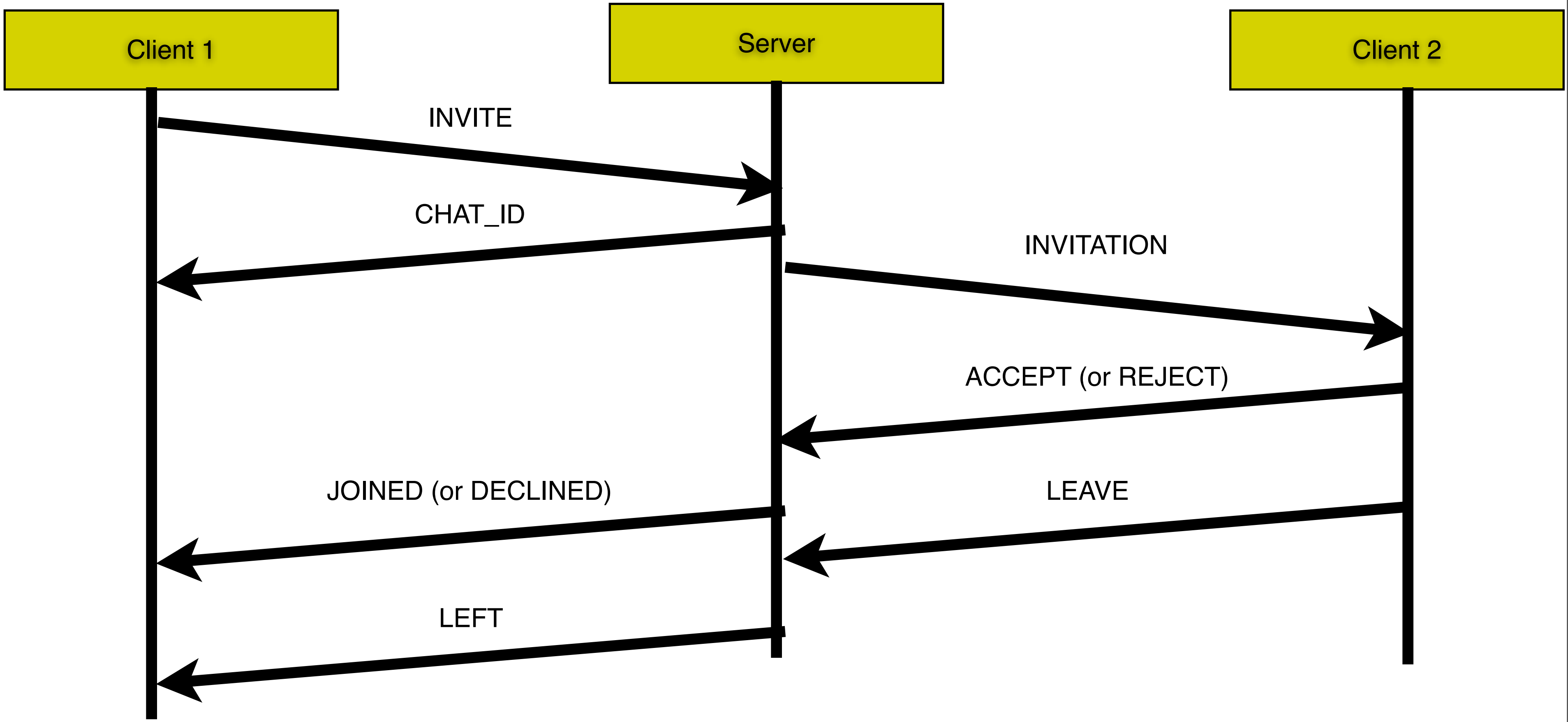
The IM Server Protocol

- Uses the "pickled argument list" approach
  - Every message is a pickled list; first item is the "command"
- Clients announce themselves when they first start; server periodically sends updated online user status
- Each chat has a unique ID *that the server generates*
- Clients only communicate with server, not each other
  - Tell the server to *invite*, specifying desired users
  - Server creates a chat, giving it a unique *chat ID*, and sending this ID back to the initiator
  - Server sends *invitations* to all clients, indicating the chat ID
  - Clients *accept* or *reject* the invitation, providing the specified chat ID
  - Clients tell server to send message to parties in a chat, by specifying both the message and the chat ID
  - Server propagates message to all members of the chat
  - Clients can leave chats by specifying their ID

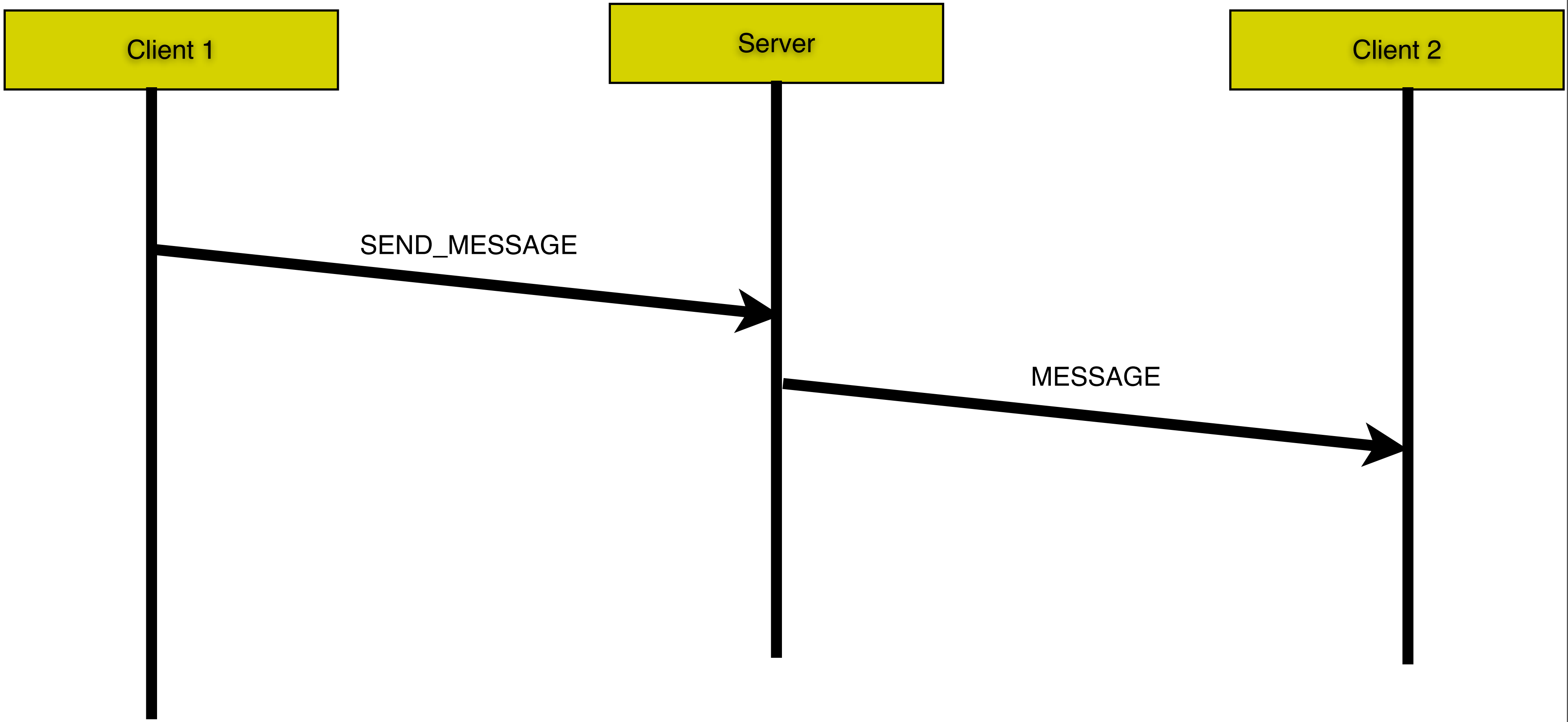
The IM Server protocol: Basic status info



The IM Server protocol: Basic chat setup



The IM Server protocol: Sending text



The IM Server Protocol

Messages from Server to Client	Messages from Client to Server
["ONLINE_USERS", {user1: status1, user2: status2, ...}] Sent by the server every 5 seconds to all clients to indicate what users are online and their status. The parameter is a dict.	["HELLO", username, status] Sent by client when it first starts; username and status are both strings
["CHAT_ID", chatID, [invitee1, invitee2, ...]] Sent in response to an invite, providing the initiating client with the chatID. The invitees list allows the client to tell which chat this ID is for.	["GOODBYE"] Sent by client when it shuts down
["INVITATION", invitor, [invitee1, invitee2, ...], chatID] Sent by the server to invitees to a new chat. Invitor is a string that indicates who started the chat; the invitees list indicates the other invitees; chatID is an integer that identifies the new chat.	["INVITE", [user1, user2, ...]] Sent by client to ask the server to invite a list of users to a new chat. Each user name should be a string.
["JOINED", chatID, userName] Sent to all members of a chat when an invitee (indicated by the userName string) joins. chatID indicates the chat.	["ACCEPT", chatID] Used to accept an invitation to chat. The client should provide the chatID it is accepting; this is an integer.
["DECLINED", chatID, userName] Sent to all members of a chat when an invitee (indicated by userName) declines an invitation. chatID indicates the chat.	["REJECT", chatID] Used to reject an invitation to chat. The client should provide the chatID it is rejecting; this is an integer.
["MESSAGE", chatID, userName, message] Sent to all members of a chat when a user sends a message. chatID indicates the chat; userName indicates the sender; and the message string is the message itself.	["SEND_MESSAGE", chatID, message] Used to send a text message to the members in a chat. chatID is an integer and message is the text to be sent.
["LEFT", chatID, userName] Sent to all members of a chat when a member (userName) leaves.	["LEAVE", chatID] Used by a client when it wishes to leave the chat indicated by the chatID.
	["CHANGE_STATUS", newStatus] (*optional!)* Change the status message associated with the client. newStatus is a string.

A Few Tips

- Write some utility methods for common operations
  - Example: receiving variable-length replies:

```
def receive(self, sock):
    reply = “”
    while 1:
        data = sock.recv(1024)        # read up to 1024 bytes
        if not data:                  # no data means connection closed
            break
        elif len(data) == 1024:       # read the data, there may be more; keep going
            reply = reply + data
        else:                         # read less than 1024, so that’s the end of the msg
            reply = reply + data
            break
    return reply
```

More Tips

- Remember: messages from the server may come *at any time* (asynchronously!)
    - Invitations to chat, updated user list, a new message, etc.
- Create your socket and connect it to the server
- Whenever you need to *write* to the server, you can do it pretty much anywhere
- But for *reading*, start a new thread that'll always be hanging out, waiting for messages to show up (like the Handler class we saw before):

```
class Handler:
    def __init__(self, sock):
        self.thread = threading.Thread(target=self.handle)
        self.thread.start()

    def handle(self):
        input = self.sock.recv(1024)
        # deal with the message you just read from the server here
        ....
```