Distributed Applications

More Networking Idioms and Styles

Week 8

FIRST:

- Time to sign up for next round of in-class presentations!
- ~10 minute talks, next class
- Technical issues, solutions, problems, designs
- Examples:
  - Strategies for debugging networked programs
  - Design process: something you tried to build in a particular way, but didn't work. How did you solve the problem?
  - Code modularity: how did you structure the connections between your network code and your GUI code?
- Should discuss code/architecture of project

Grading Criteria

Feature	Points
Overall Program Structure	**20**
• GUI code still works!	5
• Correct "main" handling	5
• Ability to configure server IP address and port number	5
• Good modularity, with networking code in net.py, which loads correctly	5
Basic Protocol Functionality	**30**
• Correct REGISTER handling	10
Sends well-formatted REGISTER message at startup time	
• Correct ONLINE_USERS handling	10
Should receive and process ONLINE_USERS messages from server	
Should update online users list appropriately	
The current user should not be displayed in the list of online users	
• Correct GOODBYE handling	10
Send GOODBYE message at time client shuts down	
Conversation Management	**50**
• Should be able to support multiple chats at one time	10
In other words, everything below should work when there are two or more chats going	
• Correct INVITE handling	5
Send INVITE message when new conversation is attempted	
Optionally: show a "pending" window, or use other mechanism to indicate no one else is in the conversation	
• Correct INVITATION handling	5
Display invitation window if INVITATION is from a different user	
Show no invitation window if INVITATION is from you	
• Correct JOIN handling	5
Generate JOIN message to server when user accepts an invitation	
• Correct LEAVE handling	5
Generate a LEAVE message to server when user exits a chat	
• Correct CONVERSATIONS handling	10
Receive and process CONVERSATIONS messages from server	
Correctly update all current conversations with information about currrent users	
GUI should correctly show current members of each conversation	
• Correct SEND_MESSAGE handling	10
Sending text should cause a SEND_MESSAGE to be sent to the server	
Sent text should appear (once) in the transcript	
Correctly handle received SEND_MESSAGES from the server	
Any received text should appear (once) in the transcript, tagged by whom it is from	
Bonus	**20**
• Server Enhancements (sending icons, for instance)	10
• Exploit STATUS messages to let users change online status, display this in the GUI, propagate to other users	5
• Exception handling above and beyond the call of duty (survive server crashes, malformatted messages from server, etc.)	5

Network Coding Idioms

Modularizing Message Handling

- Your code may be getting a little convoluted with the messages that need to be handled

```
if command == “ONLINE_USERS”:
elif command == “SEND_MESSAGE”:
elif ....
```

- Nothing really wrong with this, but code can be hard to maintain/ update

Common Idiom: Table-Based Dispatch

- One solution is to have each message handled by a single function
- Create a new function to *dispatch* control to your handlers, based on incoming message
- Use a data structure to map from incoming message type to handler function
- (This is how the server is implemented)

Example

```
class Dispatcher:
    def __init__(self):
        self.dispatchMap = {}
        self.dispatchMap["ONLINE_USERS"] = self.handleOnlineUsers
        self.dispatchMap["SEND_MESSAGE"] = self.handleSendMessage
        # ... add other mappings from commands to handlers here

    def dispatchCommands(self, socket):
        data = socket.recv(1024)
        command = data[0:data.find(' ')]
        handler = dispatchMap[command]
        handler()
```

New Styles of Networking

- Our networking so far: client-server
  - One server at well-known address
  - Accessible through (generally) any Internet-connected machine
  - Clear delineation between roles of clients and servers

Downsides of Client-Server

- Requires a server!
- Must be at a well-known address
  - Not always practical
  - Example: mobile services may have changing IP addresses
- Accessible through any Internet-connected machine
  - Clients and services may not have access to the full network
  - Example: when away from a wired or wireless network
- Clear delineation of clients and servers
  - Sometimes you may want a single device to act as each

Common Networking Style: Peer-to-Peer

- No distinction between clients and services: *any* peer can act as either
- Peer: just a machine capable of networking with other peers
- No need for connection to the Greater Internet
  - E.g., multiple peers may be in a park, able to connect with each other but not with the rest of the net
- No need to know a server's address ahead of time
  - You may not know what peers are available to you until you start up

Advantages of P2P

- No need for every machine to be on the Internet (although they do have to be on *a* network)
- No need for hard-coded IP addresses
  - Thus, also no need for *configuring* such addresses by hand
- Infrastructureless: no need for fixed networking (routers, access points, etc.) nor fixed server machines
  - Good for impromptu communications applications

More on P2P

- Some parts of some systems are P2P while others are client-server; not mutually exclusive
- Example:
  - Napster used P2P for transferring files...
  - ...but used a centralized server to allow peers to find out about each other
- Example:
  - iTunes uses P2P for music sharing...
  - ...but a centralized server for music store purchases, authorization, etc.

Implementing P2P Systems

- For the most part, nothing special
- Uses sockets, just like client-server
- Each peer creates a socket to *listen* on
  - Thus, in effect it acts like a server
- Each peer can also create a socket and *connect* it to another peer
  - Thus, in effect it acts like a client
- Protocols work basically the same as client-server
  - Thus, same formatting idioms, same protocol design issues, etc.

One Additional Twist

- How do peers "know about" each other?
- Clients know about servers by having the server's well-known IP address hard coded into them, or configured by a user
- In contrast:
  - In P2P, peers don't know in advance what other peers may be out there
  - In P2P, peers may come and go more rapidly than fixed servers (e.g., mobile services)

Discovery Protocols

- A low-level protocol for letting peers know about each other
  - Effectively: tells peers about the IP addresses of other available peers
- Common in ubicomp, mobile applications, etc.
- Allows "configuration free" use of resources on the network
  - In other words, no need for manual configuration of peers' IP addresses
- Not all P2P systems use a "real" discovery protocol
  - Napster peers register themselves with a centralized server that contains their IP addresses
  - IP address of server must be known ahead of time, but addresses of individual clients need not be
- Examples of discovery protocols:
  - Rendezvous (iTunes, iChat, ...)
  - SSDP (Universal Plug and Play)

How Discovery Protocols Work

- Earlier I said that an IP address uniquely identifies a service on the Internet
  - I lied
- Certain ranges of addresses are known as *multicast addresses*
  - 224.0.0.0 to 239.255.255.255
- Can be used to talk to a whole set of machines at one time
- Programs can listen on multicast addresses, just as they can listen on the "normal" address for their machine
- Programs can send to a multicast address, just as they can send to a normal IP address
- Messages sent to multicast addresses are passed to every host listening on that address
- Multicast is not *broadcast* because messages don't go to *every* host, just those listening on a certain address

Multicast Scoping

- You can set a *scope* for multicast messages that say how “far” they go in the network
  - Defined in terms of hops through network routers
  - Allows you to limit the multicast *radius* to a certain area of the network
- You can’t really use multicast to send a message to every listener across the entire Internet
  - Many routers won’t pass traffic with a scope this large
- Generally used to reach a set of hosts on a *single* network
- Example: rendezvous is scoped so that only hosts on the same network can communicate with each other using multicast addresses

Example Discovery Protocol