CMSC 433

Nov 30th

Project 3

- Distributed chat room
- First, we'll do a local chat room
  - Using a ConcurrentPublisher
- Then make it distributed
  - Using object serialization and hand written stubs and skeletons

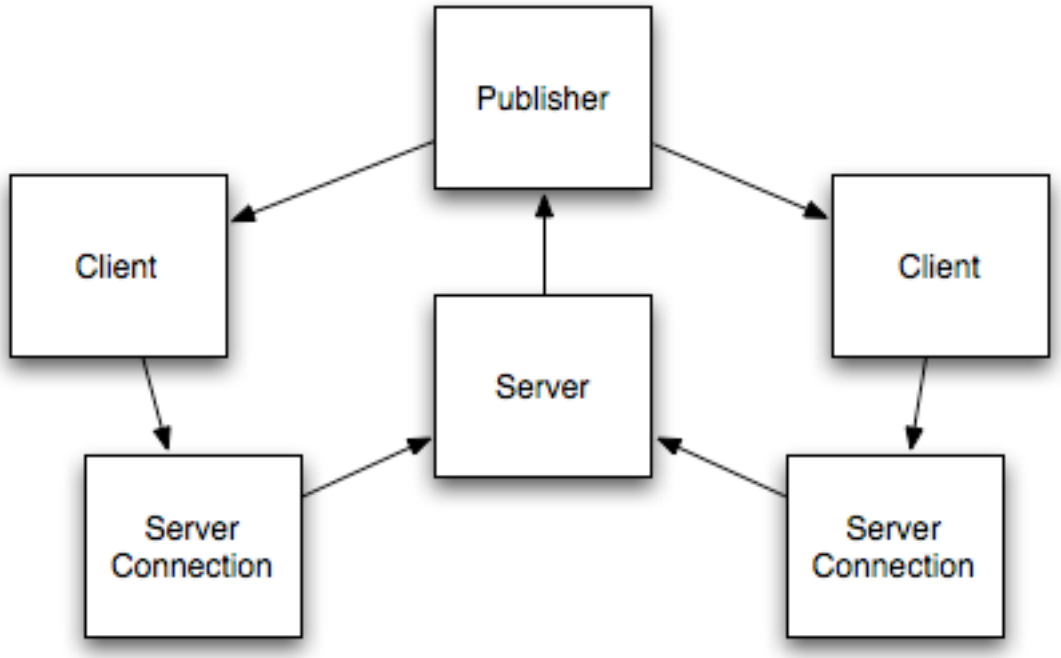
ConcurrentPublisher

- Concurrent Observable/Observer implementation
  - one slow observer doesn't slow down anything else
- We provide you with an implementation of this

Chat room

- There is a Server interface
  - You can register with the server
    - Get back a ServerConnection
- There are msgs that can be sent to a ServerConnection or to a Client
  - Command pattern

Local Chat Room



Why ServerConnections?

- Stores information we will need to know about a client
- Acts as an authorization token
  - Unless you have a ServerConnection object, you can't invoke certain methods on the server

Distributed applications

- Machines are not all the same
  - But all adhere to same communication protocol
- Network is “slow”
  - Sending a message takes a lot of time
- Network is unreliable
  - Machines may join and leave with no warning
  - Part of the network may fail

Making it distributed

- Want to have remote proxies for
  - Servers
  - ServerConnections
  - Clients
- Have special interfaces for each of these
  - Containing the methods that we want to be able to invoke remotely

Remote Procedure Call

- A remote procedure call is a way for one program to invoke a method in another process or machine
  - Old, old technique, before OO
  - But even more common now days

Implementing RPC

- Issues:
  - How do you make a call?
  - How is the information transmitted?
  - How is it handled in the receiving machine/process?

Making a RPC call through a proxy

- Have an class that defines the interface you want to be able to invoke remotely
- Create an object that acts as a remote proxy for a particular instance in a particular process
- Just invoke the method on the proxy

Transmitting data

- Lots of different ways to handle transmitting data
  - See discussion of object serialization
- Also need to consider the fact that I/O faults happen
  - Can't be ignored, particularly in wide area situation

In the receiving machine

- Use TCP sockets
- Could have a dedicated TCP connection for each remote proxy, or create a new TCP connection for each invocation
- Listen for requests, forward to correct object

Receiving machine, continued

- If dedicated TCP connections, probably have dedicated thread listening to the connection
  - Could use NIO select
- Otherwise, dedicated thread listening for new socket connections

Multiplexing

- Could have multiple proxies from the same machine share the same TCP connection
  - Have to transmit id identifying target with each call
- Could also multiplex server sockets

Easiest thing to do

- For each remote proxy
  - Dedicate a TCP connection
  - Have a dedicated remote skeleton on receiving machine
    - Has a socket and a local object
    - In a thread, listen for remote requests and translates them into calls on the local object

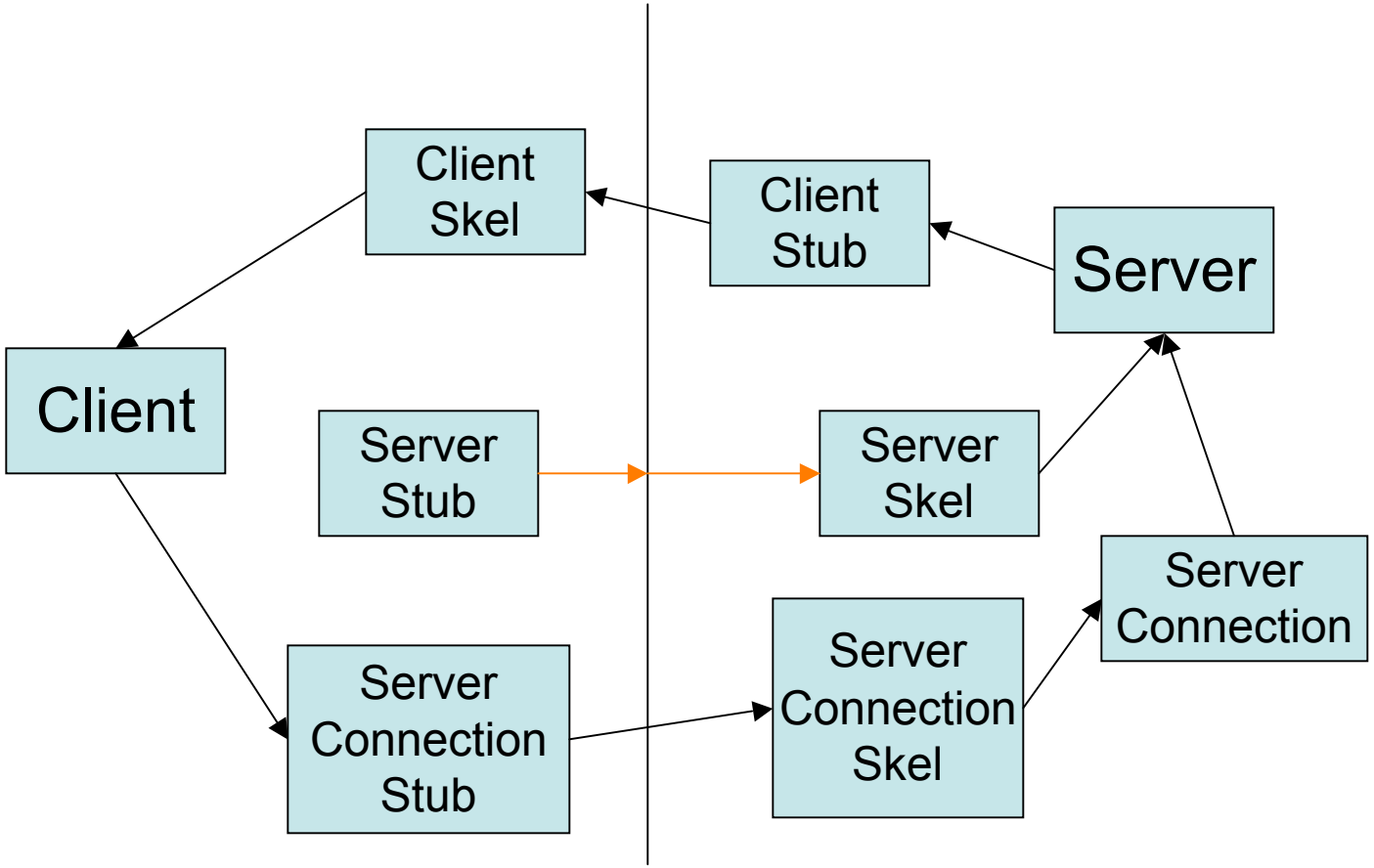
Return values

- Do remote methods have return values?
- Do we know when a remote invocation has completed successfully?
- What happens if, when the method is invoked on the remote machine, an exception is thrown

For P3, we'll use async msgs

- When an async method is invoked remotely, we don't need to wait for method to complete on remote server
  - And we don't see a return value or thrown exception
- In P3, call to Server.register is synchronous, others are asynchronous.

P3 design


Client Skel
Client Stub
Server
Client
Server Stub
Server Skel
Server Connection
Server Connection Stub
Server Connection Skel