NETWORK COMPUTING SOCKET PROGRAMMING

Spring 2012

Yung-Ting Chuang

Client/Server Model

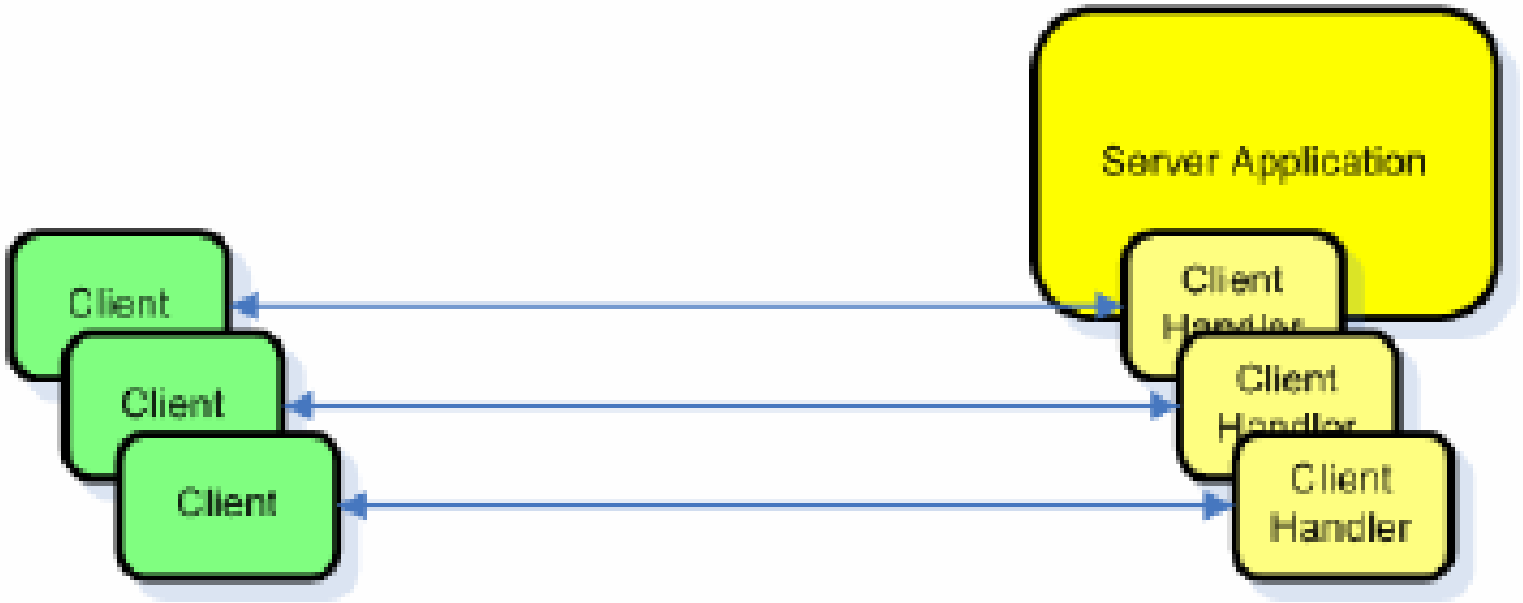
- The Client/Server-Model is a base topic to describe models of **interacting Clients and Servers**
- In the Client/Server-Model, a Server **waits for requests from Clients, and after** receiving such a request, the Server **processes it and send a response**
  - Possible to keep connection open for future requests
- The communication between the two peers is based on a **Protocol,** which is defining the possible interaction patterns and the information being exchanged

Threads

- In order to service multiple clients simultaneously, a server application needs to be **multi-threaded.**
- Each client connection results in a new thread that is dedicated to servicing that particular client.
- A class with thread attribute can be created by extending Thread class or implementing Runnable Interface in Java.
- You will need to have **run()** method and make a call to **start()** method after thread object is created

Sockets

- Main server application keeps each Client Handler in a **Vector.**
- Each of these client Handler is a **Thread created** to serve individual Client connection

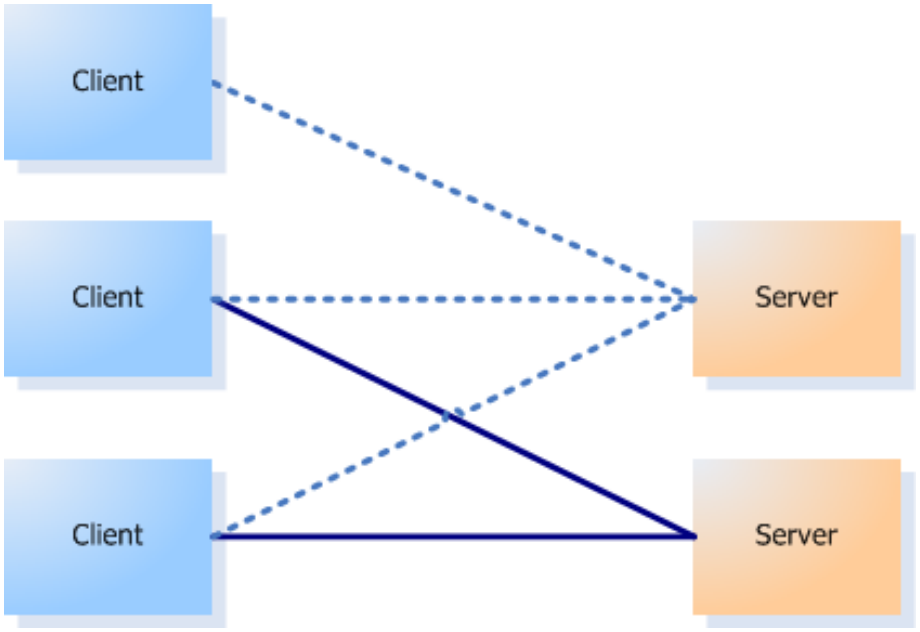


Messages

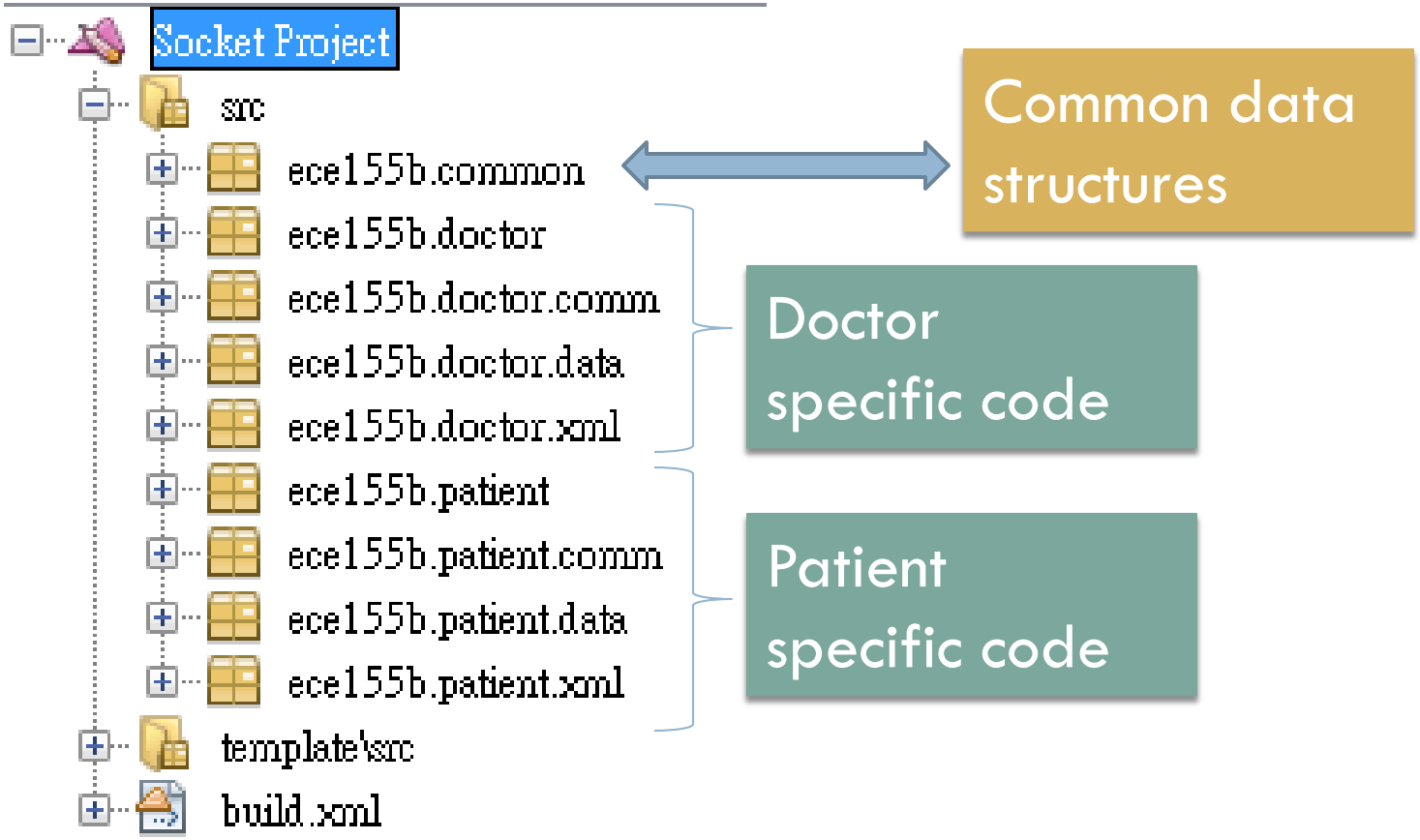
- In the project, the communication will be in well-defined XML formatted messages
- Use XStream to generate/parse the XML content

Multi-Threading

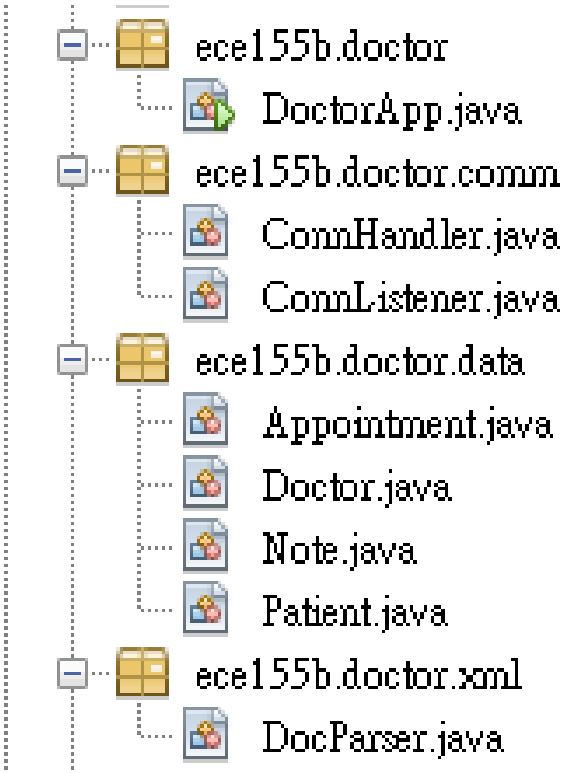
- **Patient**
  - can connect to multiple doctors at the same time
- **Doctor**
  - Can serve multiple patients at the same time



Project Structure

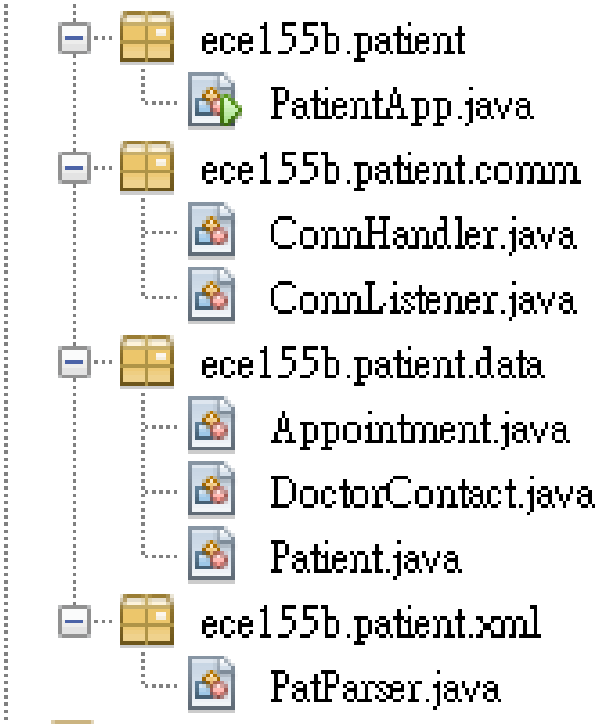


Doctor



- **DoctorApp**
  - Doctor Application (main class for your application)
- **ConnHandler**
  - Handles socket communication
- **ConnListener**
  - Serves a client (separate ConnListener for each Client)
- **DoctorData**
  - Has appointment, doctor, note, and patient objects associates with the doctor's main application. At the end all the object data will be saved into XML file.
- **DoctorXML(Parser)**
  - Parse doctor information from XML file

Patient



- PatientApp
  - Main class for patient
- ConnHandler
  - Handles connection to Doctors
- ConnListener
  - Handler for communication with Doctor
- PatientData
  - Has appointment, doctorContact, and patient objects associates with PatientApp application.
- PatientXML(Parser)
  - Load Patient information from file

Common

- Common files are used by both Patient and Doctor classes
  - If there is any constant that will be used (for example, constant values message types), include the definition into this folder.
- MessageParser
  - Parse the XML message sent or receive from patients/doctors
- Message
  - Message structure that is used during communication in XML format

Server Side

- ServerSocket servers = new ServerSocket(PortNo);

```
while(true) {
        System.out.println("Waiting connections...");
        Socket socket = servers.accept();
        System.out.println("Got one connection :) ");
        addUser(socket);
        System.out.println("Processed..");
}
```

Accept connection, and create Thread for that connection

Client side

- PatientGUI is a JFrame object with Thread enabled to listen for messages from Doctors.
- Client connects to server as below:

```
socket = new Socket("Server URL", SERVERPORT);
bwrite = new BufferedWriter (new
    OutputStreamWriter(socket.getOutputStream()));
tread = new BufferedReader (new
    InputStreamReader(socket.getInputStream()));
new Thread(){
    public void run() {
        // In a while loop wait to read from socket
    }
}.start();
```

Patient & Doctor

- SendMessage (String msg)

```
try {
        bwrite.write(msg);
        bwrite.newLine();
        bwrite.flush();
} catch (Exception ex)
{ System.out.println ("Error, sendMessage: "+ex);}
```

- Sender converts message into XML String and send
- Receiver uses XML Parser to parse & understand the message content

Replace newline with other character before you send your message…

- **BufferedReader** class reads lines separates by newline
- If you don't have newline, it *will not work*, since receiving end looks for a newline feed
- If you send String with multiple newlines in it, this will look separate messages (but you want to send 1 message)
- Solutions:
  - Replace newline with special character before sent:

xml = xml.replaceAll("\n", token);

  - Replace that character with newline again once you receive

xml = xml.replaceAll(token, "\n");

Flush

- If flush is not used, in case there is two message to be sent through TCP/IP, they will be sent in same message.
- Example:
  - The server sending 3 messages, however client receiving them in a single message
- When you call flush, the TCP message is sent, and later messages will be sent in another TCP message

Patient & Doctor (continued…)

- processMessage (String)
  - This method will serve to process the received message.
  - Possibly you will first parse the xml formatted string and then behave according to your Application Layer protocol
  - You will use the following code segment to pass String to XML parser:

    ```
    parser.parse(new InputSource(new StringReader(xml)),handler);
    ```

  - Remember last project we provided XML filename as input source, but now we provide String

Doctor (Multi-threaded Doctor)

- This is a multi-threaded server.
- It uses ConnHandler class which deals with client connections
- When there is a new client connection, main application will pass the information to ConnHandler

Connection Handler

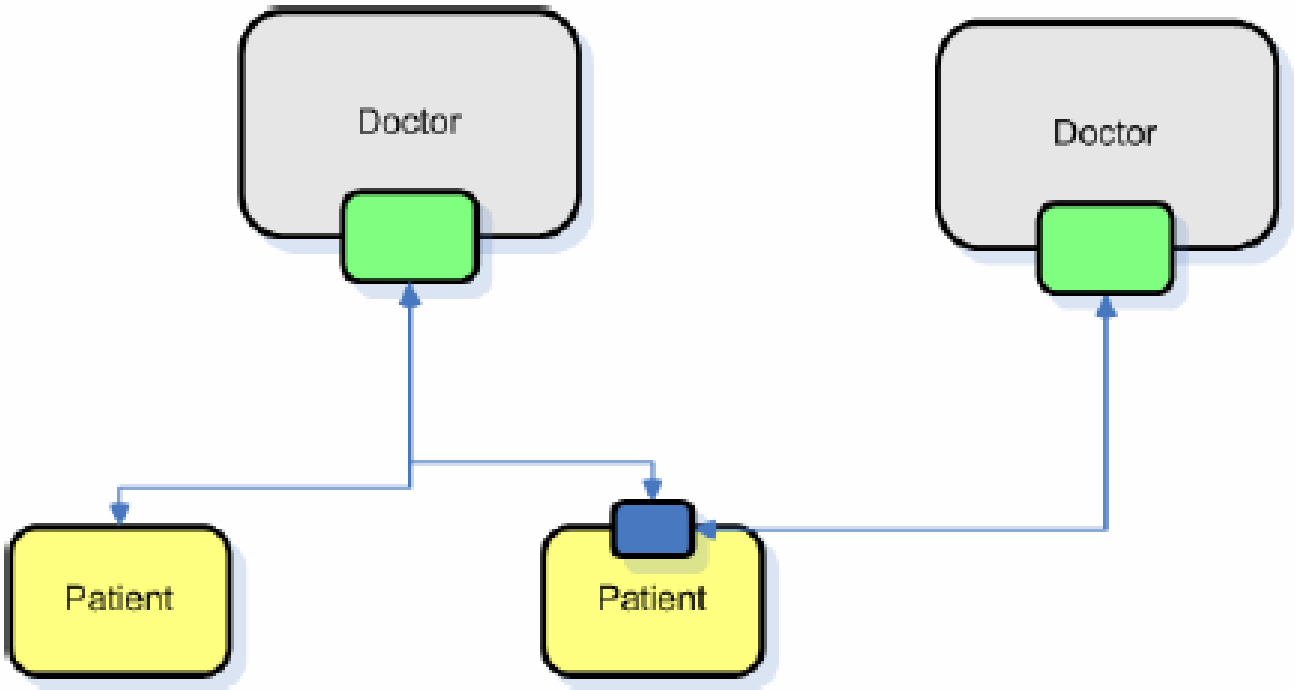
```
public void run(){
    try {
        while(true){
            System.out.println ("Waiting connections");
            Socket socket = servers.accept();
            System.out.println ("Got one connection");
            addUser(socket);
            System.out.println ("Processed..");
        }
    } catch (Exception ex) {}
}
public void addUser(Socket socket) {
    handles.addElement (new ConnListener(this, socket));
}
```

ConnListener

- **ConnListener is thread enabled object serving for** individual client

- Passing the **ConnHandler reference to this object,** message can be passed to upper level, which is **ConnHandler. Then you will be able to send this** information to other clients

Project 2



Project 2(continued…)

- Patient has:
  - Appointments other than notes
- Appointments:
  - Appointment information
  - Notes
  - Prescriptions
- Prescriptions:
  - Written by doctor
  - Prescription Date
  - Expiration Date
  - List of medicines

Message class

- Message
  - int type
    - AUTHENTICATE_PATIENT(0) → AUTHENTICATE_PATIENT_REPLY(1)
    - REQUEST_TIME (2) → REQUEST_TIME_REPLY (3)
    - REQUEST_APPOINTMENT(4) →REQUEST_APPOINTMENT_REPLY(5)
    - Other optional type as you wish
  - Other optional type can be confirmation message, error messages, or terminate message
  - Object attachment
    - Can be any type of object
    - We will use Doctor, Patient, Note, Appointment objects in the project

Authenticate

Patient sends

```
<Message>
   <MessageType>Authenticate</MessageType>
   <MessageFrom>1</MessageFrom>
   <MessageContent>Authenticate Patient</MessageContent>
</Message>
```

Please note that "MessageFrom" is the ID number which server will recognize during the execution(so the server knows which patient is making request). It can be in string, not necessary to be in int type

Authenticate Response

Doctor replies

```
<Message>
  <MessageType>AuthenticateR</MessageType>
  <MessageFrom>Doctor's Name</MessageFrom>
  <MessageContent>
        Authenticate Patient Reply
   </MessageContent>
</Message>
```

Request Time, Request Time Reply

- Patient send requests time availability to the doctor for a specific date

- Doctor Reply with:
  - Times that's available on the requested date patient made.
  - If there is no time available on the date patient requested, doctor replies with "No Times Available" to patient.
  - Patient then reissue his/her request for another date.

Multiple requests at the same time...

- If there is another patient concurrently communicating with the doctor to request time availability, we need to inform this patient to send his/her request later.
- We can implement this task by having a "lock" in ConnHandler.
  - Patient A requests for time availability. Since no one is using lock, we permits Patient A to request time availability.
  - Patient B now requests for time availability. ConnHandler sees the lock value is not equal to original, so it replies "Busy" back to Patient B and ask Patient B to try again in next few seconds.
  - After Patient A finishes requesting appointment, ConnHandler release lock.
  - Now Patient B requests for time availability again. ConnHandler grants permission for Patient B to make time availability

Request Time, Request Time Reply

- Note that the lock can only be released after the current patient successfully made his/her request.
- So that means…
  - *Request Time Availability of Doctor* ***(Grant lock)***
  - *Request Time Availability Reply* ***(Hold lock)***
  - *Request Appointment* ***(Hold lock)***
  - *Request Appointment Reply* ***(Release Lock)***

Request Appt and Request Appt Reply

- Request Appointment
  - The Patient requests an appointment for a specific date/time with the Doctor.

- Request Appointment Reply

  The Doctor sends an acknowledgement to the Patient, indicating that the appointment at the particular date/time has been made for him/her.

Request Appt and Request Appt Reply

- Patients sends **Appointment** object which includes
  - Int time
  - Int date
  - Patient's name
  - Doctor's name of this appointment
- Doctor replies with
  - Appointment set
  - Not set if the some error occurred…

You need to create an Appointment object

How you start your Project2…

1. Convert JApplet into Jframe
2. Change your codes so now you can load two different XMLs for two Doctors
3. Create Patient GUI
4. Change template codes so we have dynamic port # specified from user, rather than static port #
5. Assign clientID from Doctor's side, so each time a new client would have new ID (Hint: modify addUser() from ConnListenner.java from Doctor's side)
6. Write codes so now when a new patient try to connect to the doctor, the doctor would first assign clientID for patient, and send message back to this patient with newly assigneed clientID
7. Write codes so after doctor sends message back to patient, patient can extract its current clientID.
8. Now you will know what to do rest…