

61FIT3JSD

Fall 2022

Lecture 11

Networked Application Development

Lecture outline

- TCP/IP review
- Client/server application
- Application protocol
- Design & implementation
 - single-threaded application
 - multi-threaded application



TCP/IP review

- TCP/IP protocol stack
- TCP
- UDP

TCP/IP protocol stack

- Defines layers of protocols that enable data communications between Internet hosts

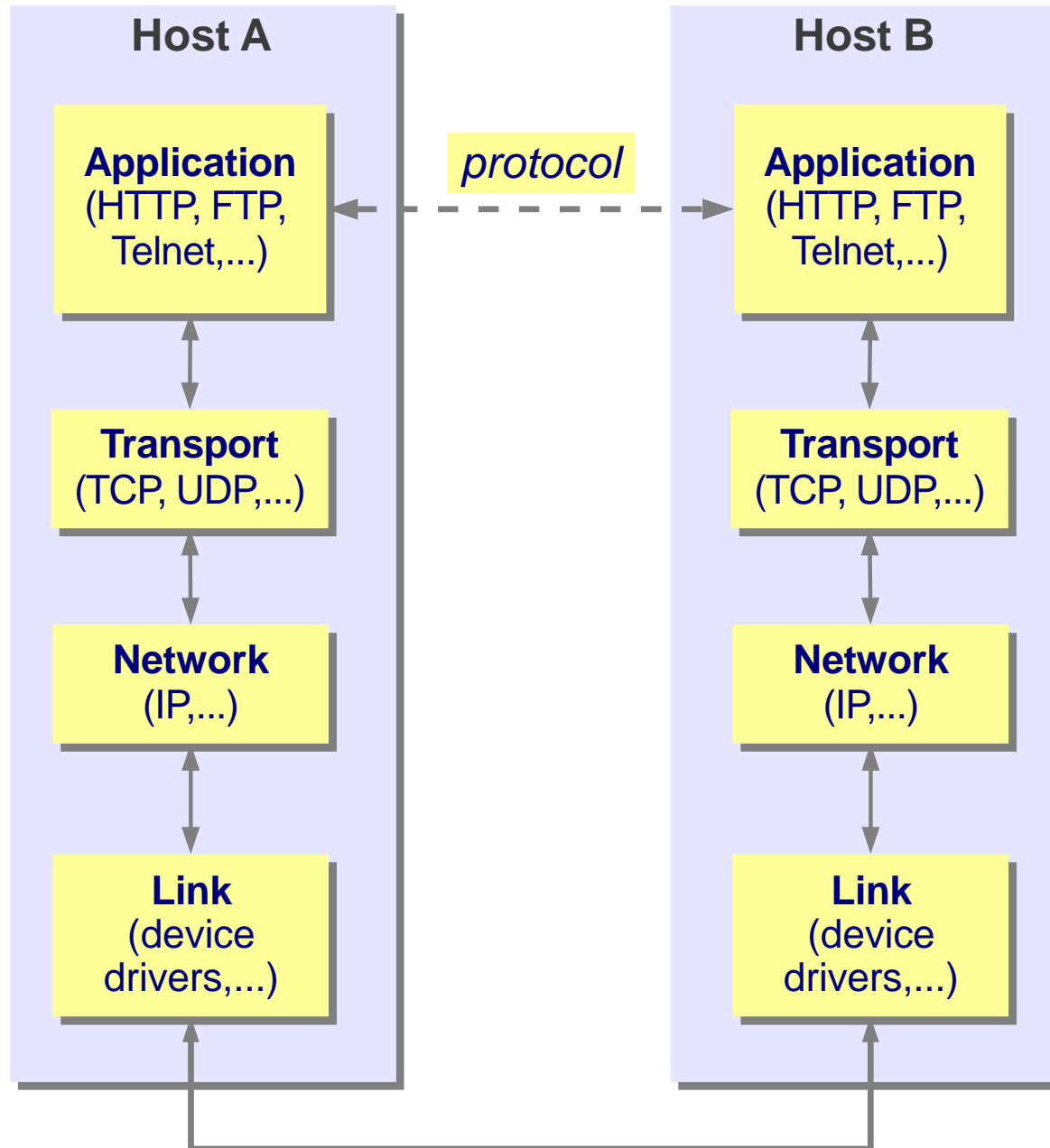
Application
(HTTP, FTP,
Telnet,...)

Transport
(TCP, UDP,...)

Network
(IP,...)

Link
(device
drivers,...)

TCP/IP hosts



TCP

- Transmission Control Protocol
- Transport layer
- Connection-based
- Reliable transmission:
 - guarantee of data arrival
 - guarantee of consistency between sent and received data

UDP

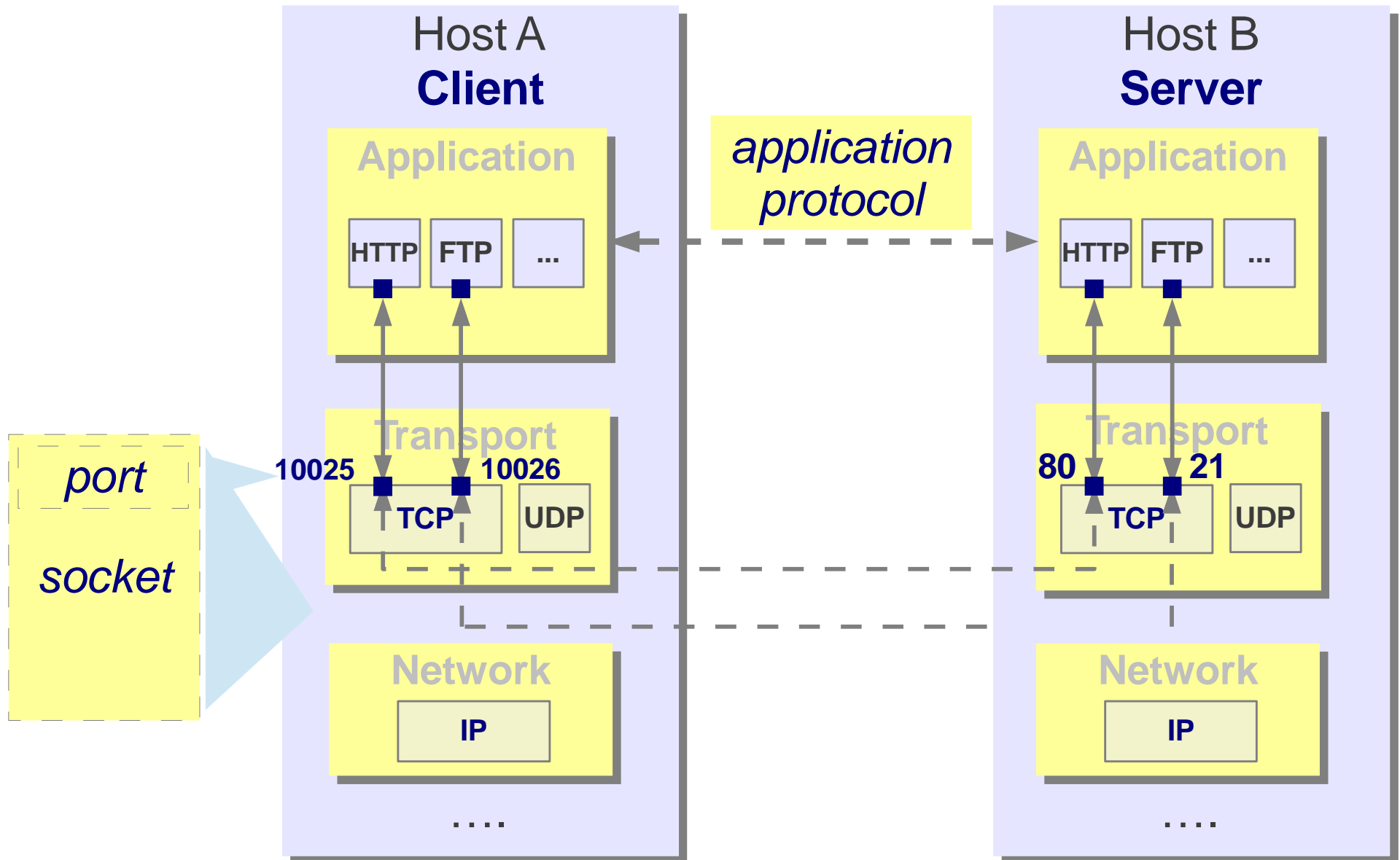
- User Datagram Protocol
- Transport layer
- Connectionless
- Unreliable transmission
 - no guarantee of data arrival



Client-server application

- Applications reside at the top layer of the TCP/IP stack
- Uses a transport layer protocol to send/receive data
- Follows a client/server architecture:
 - client application: sends requests to a server application
 - server application: processes each client's request to send a suitable response

Client/server application concepts



Client

- An application process that makes requests
- Knows the host and port of the server
- A **local port** is assigned to the client to receive responses from the server
- Local ports are automatically assigned from a range different from the well-known ports

Server

- The application process that handles the requests
- Performs domain-specific tasks (e.g. web functions, file management, etc.) that are shared among many clients
- Listens on a well-known port for connection requests from clients
- A new socket is created for each client to send/receive data

Port and socket

- **Port:** a unique number assigned to each application:
 - well-known port numbers: 80 (HTTP), 21 (FTP)
- **Socket:** a combination of IP address and port number that uniquely identifies an application on the network:
 - e.g.: HTTP on host 1.1.1.1 is different from HTTP on host 1.1.1.2



Application protocol

- Manage connections
- Manage data exchange between client(s) and server

Example: Knock-knock game

- Sample data exchange:

Client

Server

Knock! Knock!

Who is there?

Turnip

Turnip who?

Turnip the heat, it's cold in here!

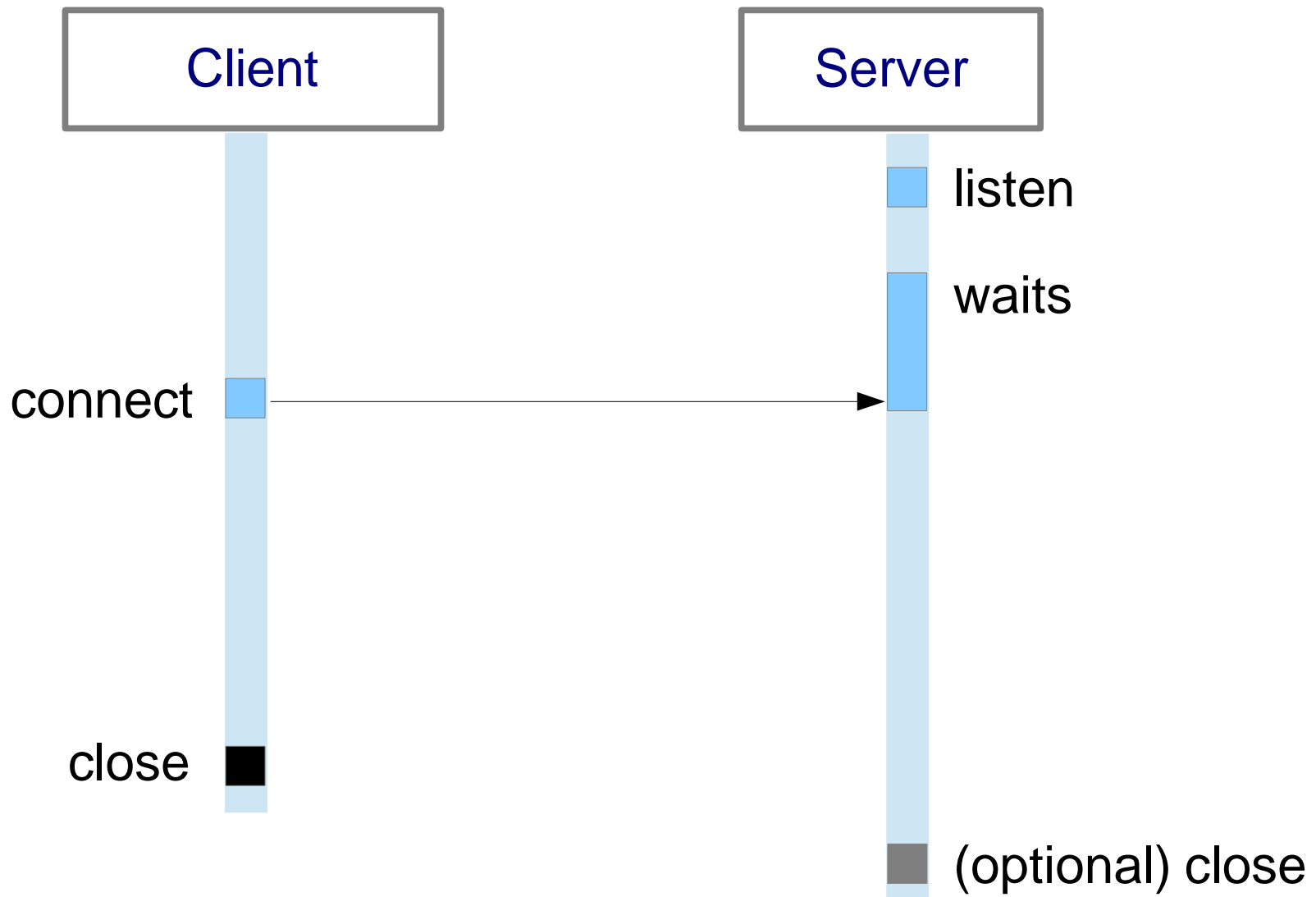
Want another? (y/n)

y

Manage connections

- **Connection** set-up and shut-down
- **Server:**
 - open the server socket
 - wait for client connections
 - (optional) close when finished
- **Client:**
 - connect to the server socket
 - close when finished

Example



Data exchange

- Application:
 - what **data** to send?
- Presentation:
 - what **format** under which the data is represented?
- Session:
 - **when** to send the data?

Message format

- Depend on the application
- Types:
 - unstructured text: format-free
 - structured text: CSV, XML, etc.
 - binary: e.g. Java object, JPEG, MPEG4, etc.

Example: unstructured text

Knock! Knock!

Who is there?

Turnip

Turnip who?

Turnip the heat, it's cold in here! Want another?
(y/n)

y



KnockKnock application

- Package `lect11.knockknocksimple`
 - `KnockKnockClient`
 - `KnockKnockProtocol`
 - `KnockKnockServer`

(Source: *Java Tutorial/Custom networking*)

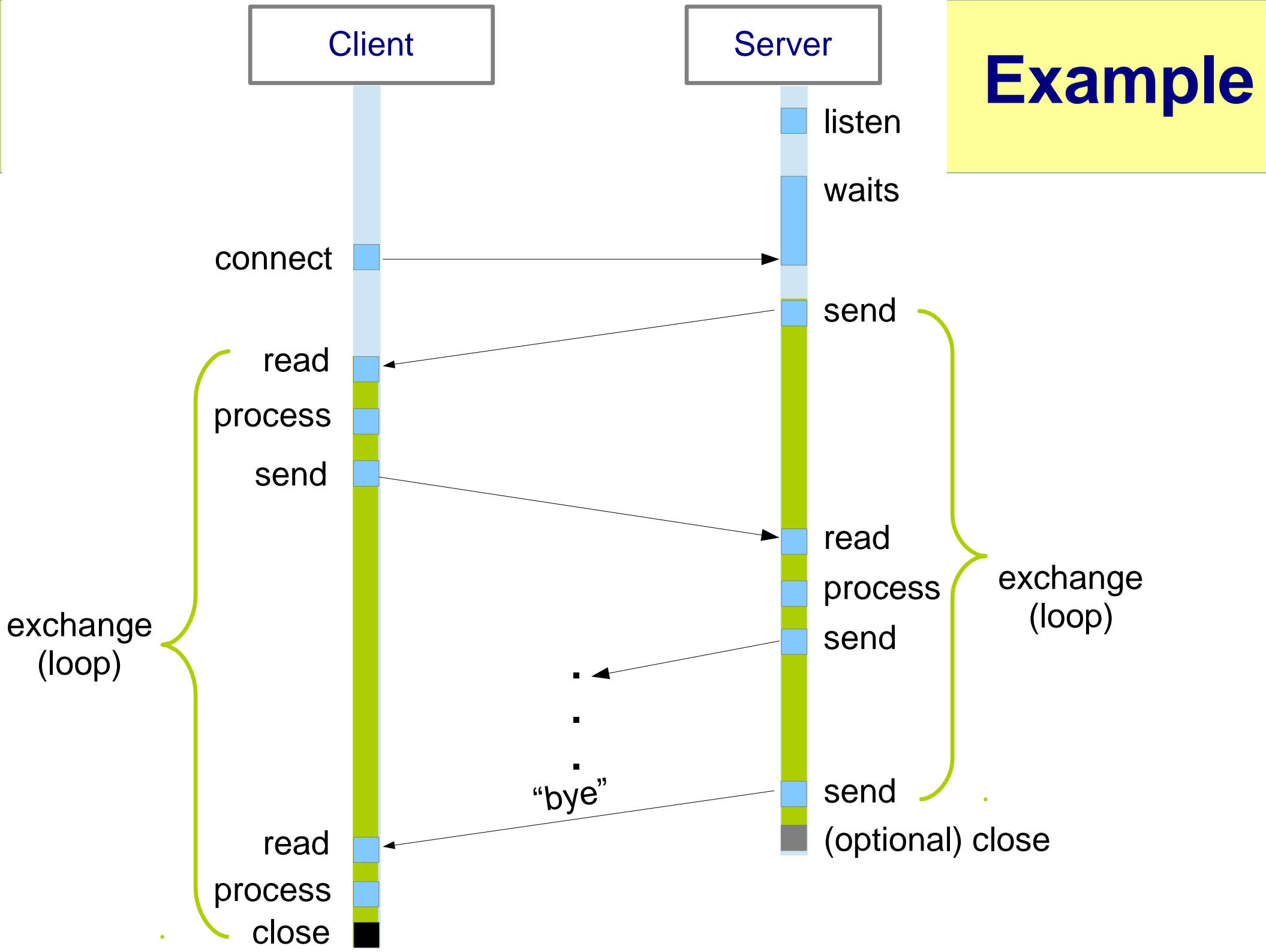
Limitations of knockknocksimple

- Ad-hoc design:
 - specific for the need at hand
- Protocol is only used for server
- Design:
 - does not make clear all the protocol functions
 - single-threaded

Data exchange session

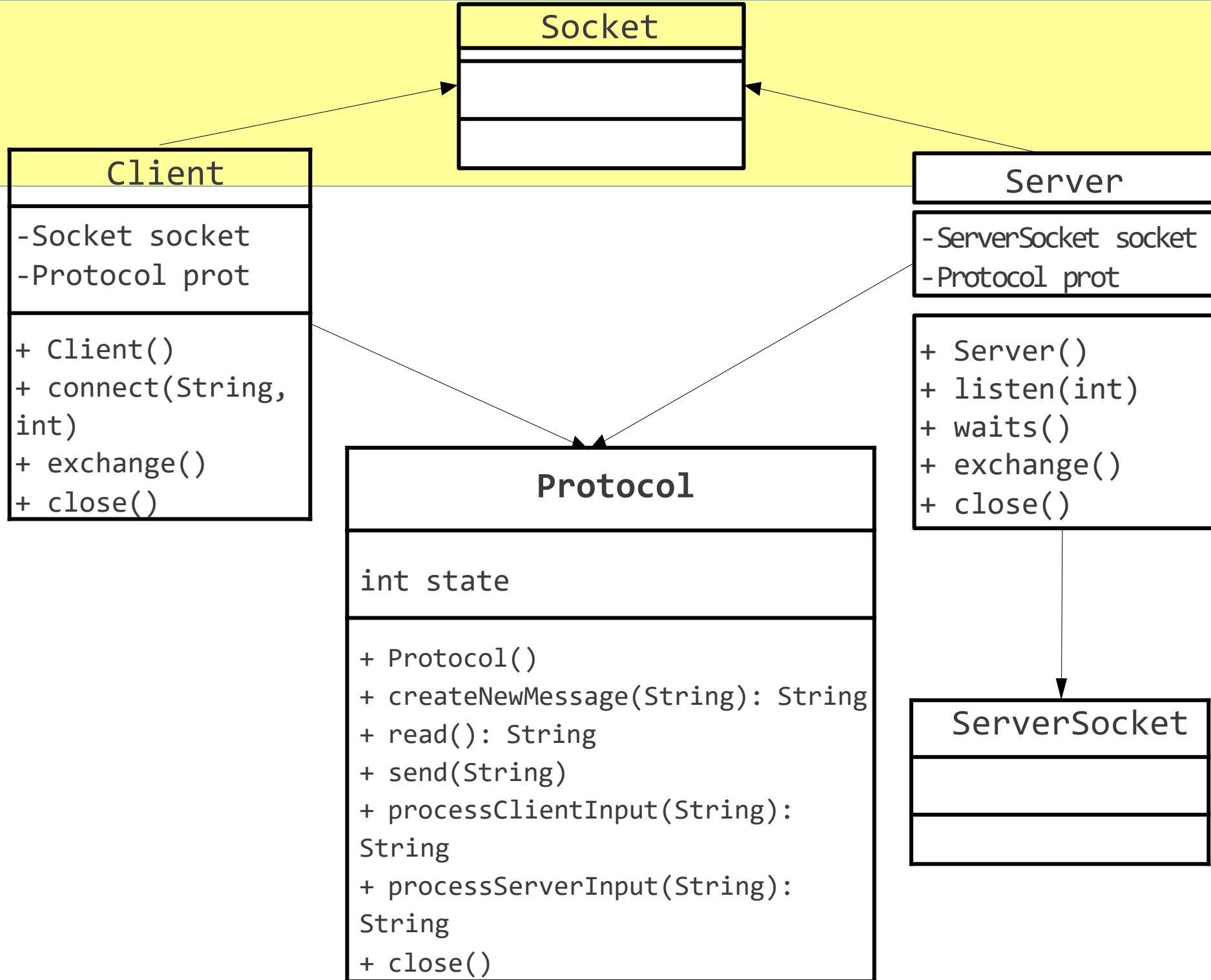
- Consists of a finite sequence of 3-step blocks:
 1. read next message from peer (client or server)
 2. process message and prepare a response
 3. send the response to peer
- Steps 1,3 are common for both client and server
- Step 2 differs between client and server
- Raises an end-of-program exception to end normally

Example



- **Assumption:** single threaded
 - one client connection at a time
- Two design solutions:
 - single-class design
 - multi-class design

Single-class protocol design



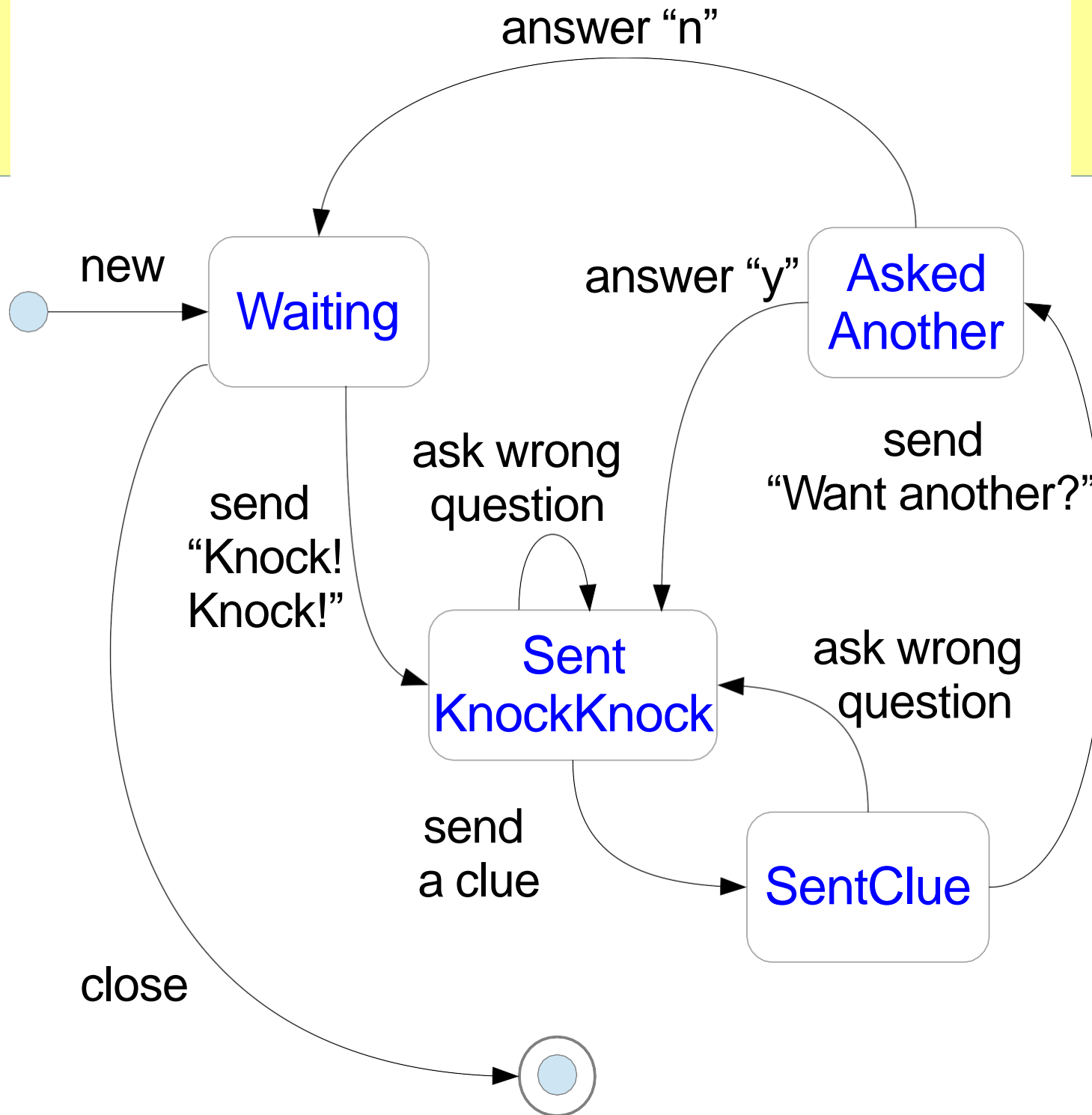
Design note (1)

- Client:
 - connect: set up the connection
- Server:
 - listens: start the server socket
 - waits: waits for the next client connection
- Client & Server:
 - exchange: performs the data exchange

Design note (2)

- Protocol: defines state, message format, and common behaviours:
 - state: the current state of the exchange
 - Message format: String
 - processClientInput: process messages from client
 - processServerInput: process message from server

Example

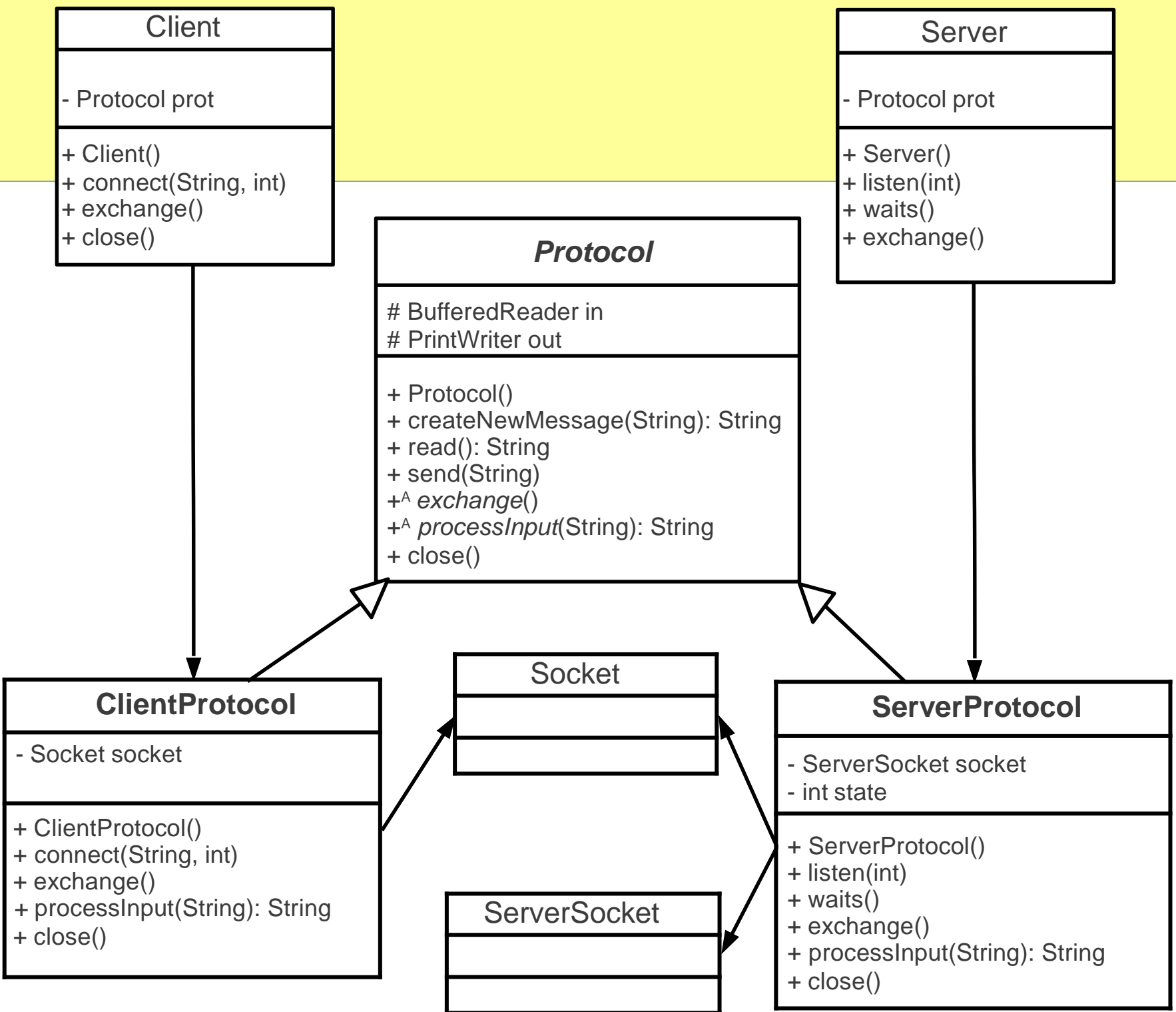


knock-knock states

Features

- One protocol class for both client and server
- Benefits:
 - simple to code and understand
 - easy to update protocol logic (e.g. exchange rules)
- Drawbacks:
 - no listening and connection management in protocol class
 - security concern: server-side logic is visible to client

Multi-class protocol design



Features

- Protocol becomes an abstract super-class
 - defines common data and behaviours
 - abstract methods are common behaviours that have different implementations
- Two protocol classes: one for client and one for server
 - different client and server behaviours are defined in the sub-classes
- Client & server methods are simply the interface for the protocol methods
- Overcome the drawbacks of solution (1)

KnockKnock

- `knockknockmulti.Server`
- `knockknockmulti.Client`



Implementation

- Client & server sockets
- Protocol
- ClientProtocol
- ServerProtocol

Socket

- Class: `java.io.Socket`
- Java's implementation of socket
- Provides methods to:
 - connect to a server
 - obtain input and output streams of the connection
- To use with data streams to send and receive data
 - e.g `BufferedReader` and `PrintWriter`

Server socket

- Class: `java.io.ServerSocket`
- Java's implementation of the server-side socket
- Provides methods for:
 - listening on a (server) port
 - accepting clients connections
- An accepted client connection is a `Socket` object

Protocol

```
public abstract class Protocol {  
    protected PrintWriter out;  
    protected BufferedReader in;  
  
    public Protocol() {  
        //  
    }  
}
```

createNewMessage()

```
public String createNewMessage(String msg) {  
    return msg;  
}
```

read()

```
public String read() {  
    try {  
        return in.readLine();  
    } catch (IOException e) {  
        // prints e  
    }  
    return null;  
}
```

send()

```
public void send(String mesg) {  
    out.println(mesg);  
}
```

(A) exchange()

```
public abstract void exchange();
```


(A) processInput

```
public abstract String processInput(String  
    theInput) throws E0PException;
```

close()

```
public void close() {  
    try {  
        out.close();  
        in.close();  
    } catch (Exception e) {  
        // ignore  
    }  
}  
} // end Protocol
```

ClientProtocol

```
public class ClientProtocol
    extends Protocol {
    protected Socket socket;
    private BufferedReader stdIn;
    public ClientProtocol() {
        super();
        stdIn = new BufferedReader(
            new InputStreamReader(
                System.in));
    }
}
```

connect()

```
public void connect(final String host,  
    final int port) throws  
    UnknownHostException, IOException {  
    socket = new Socket(host, port);  
    out = new PrintWriter(socket.getOutputStream(),  
                           true);  
    in = new BufferedReader(new InputStreamReader(  
        socket.getInputStream()));  
}
```

exchange()

```
@Override
public void exchange() {
    String inputLine;
    String outputLine;

    // start the exchange
    try {
        while (true) {
            // read next input
            inputLine = read();

            System.out.println("Server: " + inputLine);

            if (inputLine != null) {
                // process input
                outputLine = processInput(inputLine);
            }
        }
    }
}
```

...

```
    if (outputLine != null) {
        // next response
        outputLine = createNewMessage(outputLine);

        System.out.println("Client: " + outputLine);
        // send response
        send(outputLine);
    }
} // end if
} // end while
} catch (EOFException e) {
    // end-of-game print Bye.
    System.out.println(e.getMessage());
}
}
```

processInput()

```
public String processInput(String fromServer)
                           throws EOFException {
    String fromUser = null;

    // application specific processing
    if (fromServer.equals("Bye."))
        throw new EOFException("Bye.");

    try {
        fromUser = stdIn.readLine();
    } catch (IOException e) {
        // print error
    }
    return fromUser;
}
```

close()

```
public void close() {  
    super.close();  
    try {  
        socket.close();  
        stdIn.close();  
    } catch (Exception e) {  
        // ignore  
    }  
}  
} // end ClientProtocol
```


ServerProtocol

```
public class ServerProtocol
    extends Protocol {

    private ServerSocket socket;

    // application-specific attributes
    // ...
}
```

listen()

```
public void listen(int port)
    throws UnknownHostException,
           IOException {
    socket = new ServerSocket(port);
}
```

waits()

```
public void waits()  
    throws IOException {  
    Socket clientSocket = socket.accept();  
  
    out = new PrintWriter(clientSocket.  
        getOutputStream(), true);  
    in = new BufferedReader(new  
        InputStreamReader(clientSocket.  
            getInputStream()));  
}
```

exchange()

```
@Override
public void exchange() {
    String inputLine, outputLine;
    try {
        // sends the initial message
        outputLine =
            processInput(null);
        outputLine =
            createNewMessage(outputLine);
        send(outputLine);

        // proceed until end-of-game
        while (true) {
            // read next input
            inputLine = read();
        }
    }
}
```

...

```
// process input
if (inputLine != null) {
    outputLine = processInput(inputLine);
    if (outputLine != null) {
        // new response
        outputLine = createNewMessage(outputLine);

        // send response
        send(outputLine);
    }
}
}
} catch (EOPException e) {
    // end-of-game, send Bye
    send(createNewMessage(e.getMessage()));
}
```

processInput()

```
public void processInput(String mesg)
    throws IOException {
    // application-specific handling
    // ...
}
```

close()

```
public void close() {  
    super.close();  
    try {  
        socket.close();  
    } catch (Exception e) {  
        //  
    }  
}
```

Server

```
// creates a protocol object
Server server = new Server();

// start the socket
try {
    server.listen(port);
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```


...

```
// waits for client connections  
server.waits();
```

```
// data exchange sequence  
server.exchange();
```

```
// close  
server.close();
```

Client

```
// creates a client
Client client = new Client();

// connect
try {
    client.connect(host, port);
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

...

```
// data exchange sequence  
client.exchange();
```

```
// disconnect and close  
client.close();
```



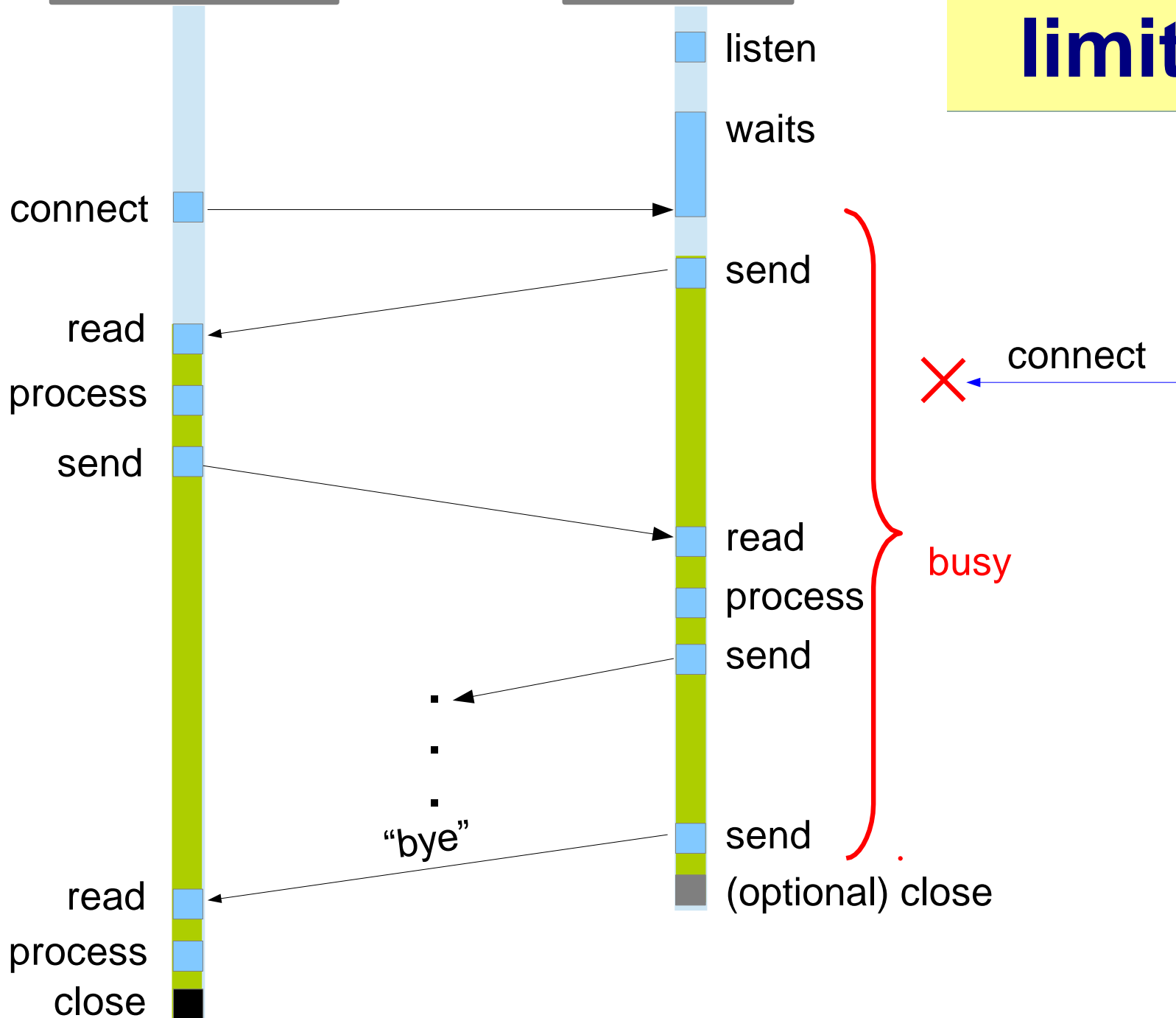
Multi-threaded design

- How to handle concurrent client connections?
- Limitations of the current design:
 - server is busy during the entire exchange session
 - a new client has to wait for the current exchange to finish
- Solution:
 - minimise the busy window of the server,
 - place the exchange session in a thread

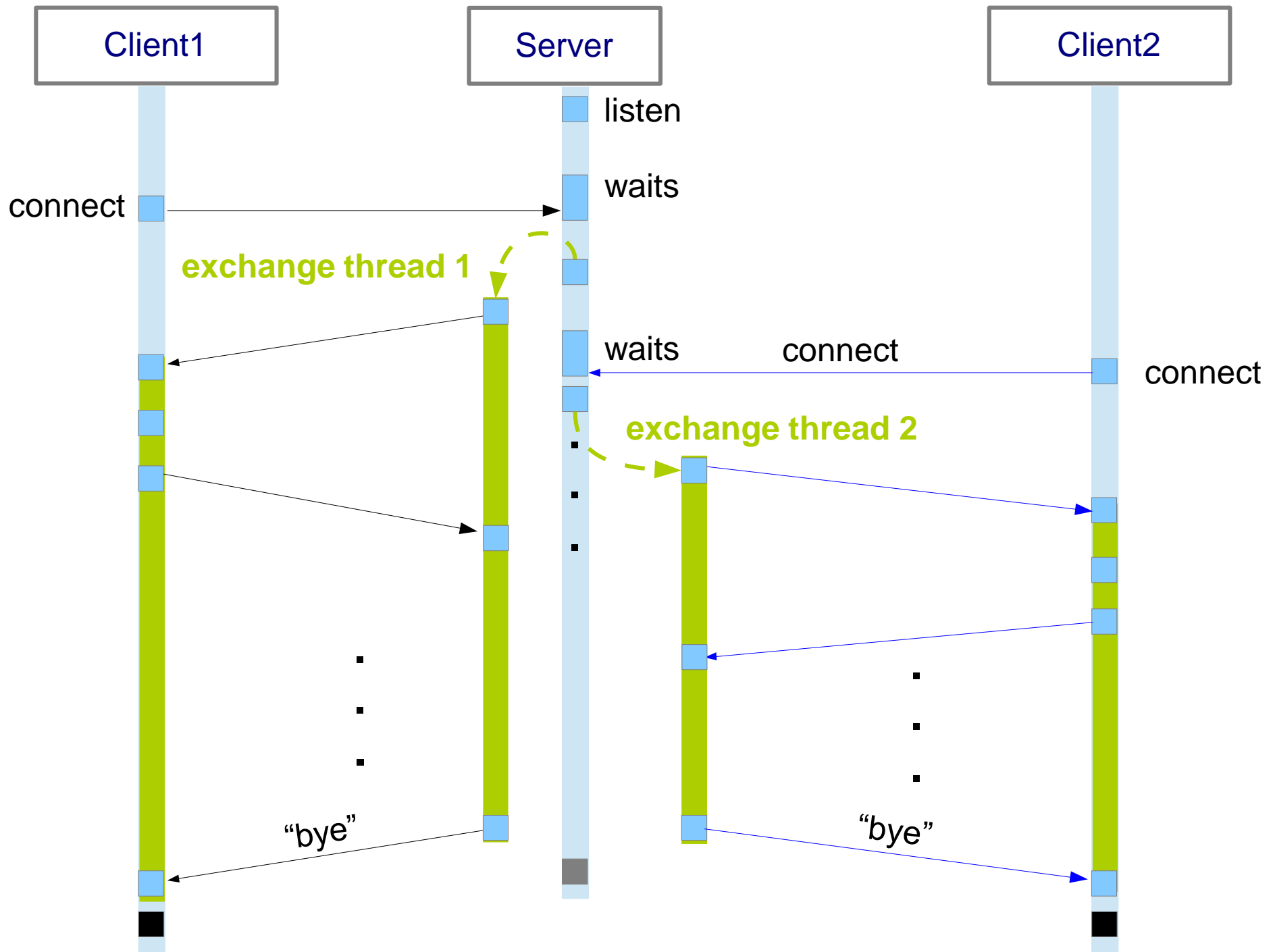
Client

Server

Current limitations



A multi-threaded protocol



DEMO

Multi-threaded knockknock

- `knockknock.Server`
- `knockknock.Client`

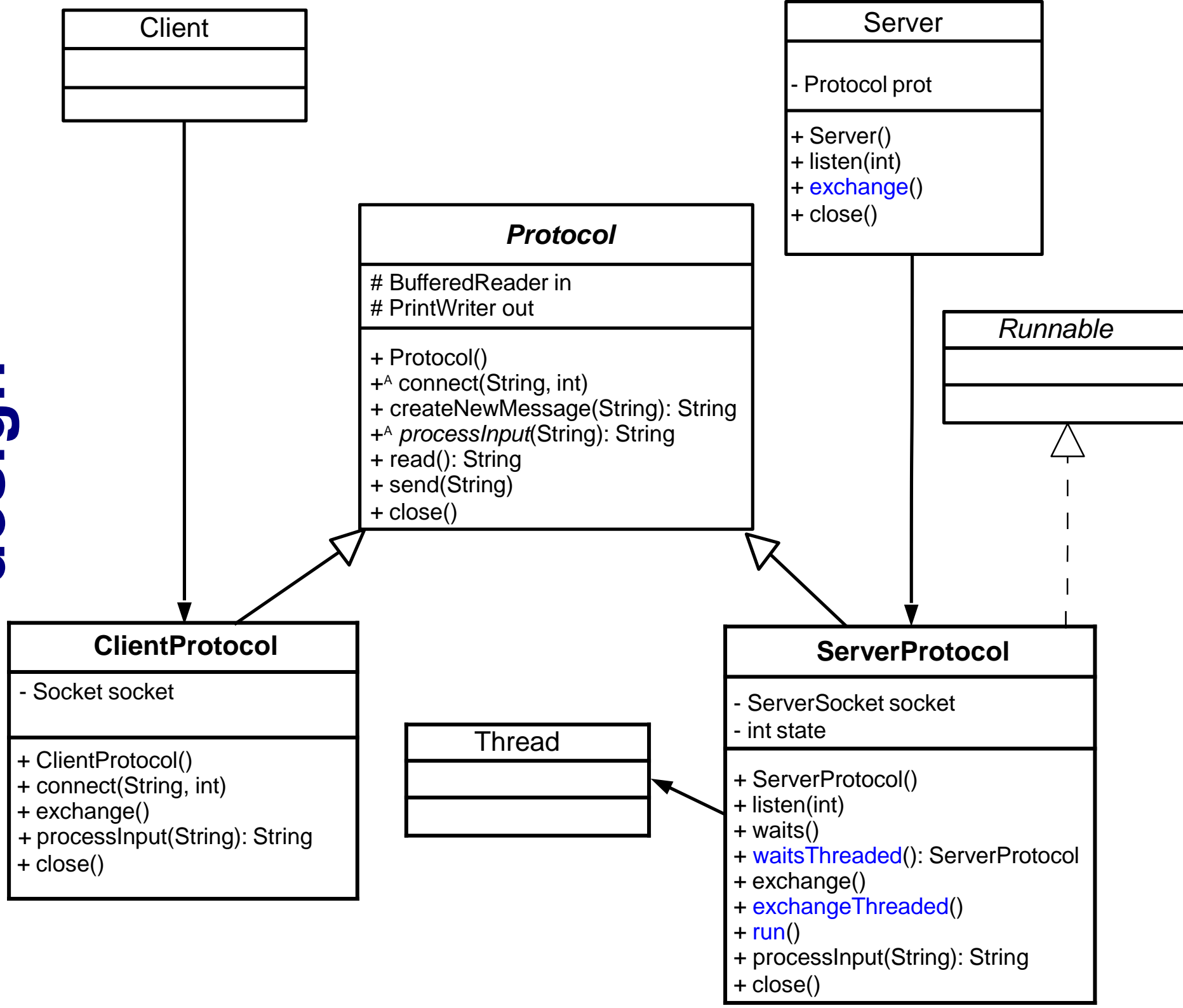
Design changes (1)

- `ServerProtocol`:
 - implements `Runnable`
 - three new methods:
 - `exchangeThreaded`, `waitThreaded`, `run`
- `waitThreaded`:
 - return a new `ServerProtocol` instance
- `exchangeThreaded`:
 - invoke `waitThreaded` to obtain a new protocol object
 - create a new `Thread` to run the object

Design changes (2)

- `Server.exchange()`:
 - invokes `prot.exchangeThreaded()`

Multi-threaded protocol design



Implementation

- ServerProtocol:
 - waitsThreaded
 - exchangeThreaded
 - run

exchangeThreaded()

```
public void exchangeThreaded() {  
    while (true) {  
        try {  
            // waits  
            ServerProtocol protocol = waitsThreaded();  
  
            // creates a new exchange thread  
            Thread t = new Thread(protocol);  
            t.start();  
            System.out.println("Started thread " +  
                t.getName());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

waitsThreaded()

```
// creates a rotocol instance to handle each new
// client connection
public ServerProtocol waitsThreaded() throws
                                   IOException {
    Socket clientSocket = socket.accept();
    PrintWriter out = new PrintWriter(
        clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()
                           ));
    ServerProtocol kks = new ServerProtocol();
    kks.in = in;
    kks.out = out;
    return kks;
}
```

run()

```
// implements Runnable  
public void run() {  
    exchange();  
}
```

Server.exchange()

```
// exchange data with clients
public void exchange() {
    // use threads
    prot.exchangeThreaded();
}
```

Summary

- Networked applications use the TCP/IP protocol stack for data exchange
- Servers listen on well-known ports and handle clients requests
- Client/server applications are implemented using the `Socket` and `ServerSocket` class in `java.io` package
- Two design solutions: single- and multi-class
- Multi-class design can easily support multiple client connections using threads

References

Oracle, The Java Tutorial, Oracle,
<http://docs.oracle.com/javase/tutorial>

- Trail: Custom networking