CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2024

LECTURE 13

Tamara Bonaci
t.bonaci@northeastern.edu

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Course logistics
- Review
  - Functional programming in Java
  - Lambdas
  - Streams
  - Functional objects
- Networking in Java

COURSE LOGISTICS

REVIEW – FUNCTIONAL PROGRAMMING IN JAVA

CS 5004, SPRING 2024 – LECTURE 13

REVIEW: JAVA FUNCTIONAL PROGRAMMING

- Key concepts:
  - Functions as first-class objects
  - Pure functions
  - Higher order functions
  - No state
  - No side effect
  - Immutable variables
  - Recursion favored over looping
  - Functional interfaces

REVIEW: JAVA FUNCTIONAL PROGRAMMING

- Functions as first-class objects:
  - We can create an instance of a function
  - We can have a variable referencing to a function
  - Functions can be passed as arguments to other functions
- Note: ordinarily, methods in Java are not first-class objects, but lambda expressions come very close

- Pure functions:
  - The execution of a function has no side effects
  - The return value of a function depends only on input arguments

REVIEW: JAVA FUNCTIONAL PROGRAMMING

- Higher order functions:
  - The function takes one or more functions as input arguments, or
  - The function returns another function as result
- Note: In Java, the closest we can get to a higher order function is a function that takes one a lambda expression as a parameter, and/or returns another lambda expression
- No state external to a function:
  - A method may have local variables containing temporary information, but it cannot reference any member variable of a class or object that it belongs to

REVIEW: JAVA FUNCTIONAL PROGRAMMING

- No side effects:
  - A function cannot change any state outside of that function
- Immutable variables
- Recursions favored over looping
- Functional interfaces
  - An interface that has only one abstract method (i.e., a method that is not implemented on an interface itself)

ACKNOWLEDGEMENT

Notes adapted from Dr. Adrienne Slaughter. Thank you.

REVIEW: STREAMS AND STREAM PIPELINE

Adapted from: https://stackoverflow.com/questions/1216380/what-is-a-stream

- Stream: sequence of elements
- Stream pipeline: sequence of tasks ("processing steps") applied to elements of a stream
- A stream starts with a data source
  - Examples:
    - Terminal I/O
    - Socket I/O
    - File I/O
- A stream can generally be used like a queue– you're reading from it, but you can't go back in the stream
- Once you've pulled an element off the stream, it's no longer in the stream

REVIEW: THE STREAM PIPELINE

```
int total =
IntStream.rangeClosed(1, 10)
                        .sum();
```

The processing step to take, or task to complete using the stream

Internal Iteration:

IntStream handles all the iteration details– we don't write them ourselves

Reduction:

Reduces the stream of values into a single value

REVIEW: THE STREAM PIPELINE

Declarative Programming:

Internal Iteration:
IntStream handles all the iteration details– we don't write them ourselve.

Imperative Programming:

External Iteration:
The programmer specifies the iteration details.

REVIEW: METHOD `MAP()`

- map() - takes a method, and applies it to every element in the stream

```
.map((int x) -> {return x * 2;})
```

Wait, what? A *method*?

REVIEW: LAMBDAS - ANONYMOUS METHODS

- lambda or lambda expression
  - aka anonymous method
  - aka method-without-a-name
  - aka the method that shall not be named

```
(int x) -> {return x * 2;}
```

REVIEW: LAMBDAS - ANONYMOUS METHODS

- Methods that can be treated as data
  - pass lambdas as arguments to other methods (map)
  - assign lambdas to variables for later use
  - return a lambda from a method

```
(int x) -> {return x * 2;}
```

REVIEW: LAMBDAS - SYNTAX

```
(parameter list) -> {statements}
```

```
(int x) -> {return x * 2;}
```

Parameter: one int named x

Statement: return 2*x

REVIEW: LAMBDAS - SYNTAX

```
(parameter list) -> {statements}
```

```
(int x) -> {return x * 2;}
```

Same as:

```
int multiplyBy2(int x){
    return x * 2;
}
```

Difference:

- the lambda doesn't have a name
- compiler infers return type

REVIEW: LAMBDAS - SIMPLIFYING SYNTAX

Eliminate parameter type

```
(int x) -> {return x * 2;}
```

```
(x) -> {return x * 2;}
```

Type is inferred.
If it can't be inferred, compiler throws an error.

REVIEW: LAMBDAS - SIMPLIFYING SYNTAX

Simplify the body

```
(x) -> {return x * 2;}
```

```
(x) -> x * 2
```

- return is inferred
- semicolon and brackets not necessary

REVIEW: LAMBDAS - SIMPLIFYING SYNTAX

Simplify parameter list

```
(x) -> x * 2
```

```
x -> x * 2
```

We can remove parentheses for single parameter

REVIEW: LAMBDAS - SIMPLIFYING SYNTAX

lambda without parameters

```
() -> System.out.println("Hello Lambda!")
```

REVIEW: LAMBDAS - SIMPLIFYING SYNTAX

method references

```
.map(x -> System.out.println(x))
```

```
.map(System.out::println)
```

```
objectName::instanceMethodName
```

Sometimes, you want to just pass the incoming parameter to another method

REVIEW: LAMBDAS - SCOPE

- Lambdas do not have their own scope
  - We cannot shadow a method's local variable with lambda parameters with the same name
  - Lambdas share scope with the enclosing method

INTERMEDIATE AND TERMINAL OPERATIONS

CS 5004, SPRING 2024 – LECTURE 13

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
            .map((int x) -> {return x * 2;})
            .sum();
```

- map() is an intermediate operation
- sum() is a terminal operation

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
            .map((int x) -> {return x * 2;})
            .sum();
```

- map() is an intermediate operations
- sum() is a terminal operation

Intermediate operations use lazy evaluation

The operation produces a new stream object, but no operations are performed on the elements until the terminal operation is called to produce a result

STREAM PIPELINE: INTERMEDIATE AND TERMINAL OPERATIONS

```
int total = IntStream.rangeClosed(1, 10)
                     .map((int x) -> {return x * 2;})
                     .sum();
```

- map() is an intermediate operations
- sum() is a terminal operation

Terminal operations use are eager.
The operation is performed when called.

EXAMPLES

Intermediate Operations

- `filter()`
- `distinct()`
- `limit()`
- `map()`
- `sorted()`

Terminal Operations

- `forEach()`
- `collect()`

Reductions:

- `average()`
- `count()`
- `max()`
- `min()`
- `reduce()`

BACK TO OUR EXAMPLE 2…

```
int total = IntStream.rangeClosed(1, 10)
                     .map((int x) -> {return x * 2;})
                     .sum();
```

For this example, we chose to create a stream of event ints from 2 to 20 by mapping from 1:10, multiplying by 2.

How else can we do this?

BACK TO OUR EXAMPLE 2...

```
int total = IntStream.rangeClosed(1, 20)
            .filter(x -> x%2 == 0)
            .sum();
```

Filter!

The lambda for the filter operation needs to return a boolean indicating whether the given element should be in the output stream.

CLARIFYING ELEMENTS THROUGH A PIPELINE

```
int total = IntStream.rangeClosed(1, 10)
    .filter(
        x -> {
           System.out.printf("%nFilter: %d%n", x);
           return x % 2 == 0;
        })
    .map(
         x -> {
            System.out.printf("map: %d", x);
            return x * 3;
         }
    )
    .sum();
 System.out.println("\n\nTotal: " +total);
```

CLARIFYING ELEMENTS THROUGH A PIPELINE

```
int total = IntStream.rangeClosed(1, 10)
    .filter(
        x -> {
            System.out.printf("%nFilte
            return x % 2 == 0;
        })
    .map(
        x -> {
            System.out.printf("map: %d"
            return x * 3;
        }
    )
    .sum();
 System.out.println("\n\nTotal: " +
```

```
Filter: 1

Filter: 2
map: 2
Filter: 3

Filter: 4
map: 4
Filter: 5

Filter: 6
map: 6
Filter: 7

Filter: 8
map: 8
Filter: 9

Filter: 10
map: 10

Total: 90
```

COLLECTORS

CS 5004, SPRING 2024 – LECTURE 13

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

Returns the number of elements in the stream.

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

Joins the elements of the stream together into a String, with a specified delimiter

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - **Collectors.toList()**
  - Collectors.groupingBy()

Puts the elements of the stream into a List<> and returns it.

COLLECTORS

- The terminal operation collect() combines the elements of a stream into a single object, such as a collection.
- There are many pre-defined collectors:
  - Collectors.counting()
  - Collectors.joining()
  - Collectors.toList()
  - Collectors.groupingBy()

Groups the elements in the stream according to some parameter and returns a HashMap keyed by the "groupingBy" parameter.

ANOTHER TERMINAL: FOREACH

- forEach() applies the given method to each element of the stream
- The method must receive one argument and return void

METHOD REDUCE ( )

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
                     .reduce(1, (x, y) -> x * y);
```

METHOD REDUCE ( )

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
            .reduce(1, (x, y) -> x * y);
```

The starting value.
This is the value for reduce(0)

METHOD REDUCE ( )

- Rather than using predefined reductions (.sum(), .max(), etc), we can write our own reduction.

```
int total = IntStream.rangeClosed(1, 10)
          .reduce(1, (x, y) -> x * y);
```

The operation to perform.
Must take 2 parameters.
(Because it takes 2 params, we need to use the parens in the lambda)

PRODUCING A STREAM FROM AN ARRAY

```
int total = IntStream.of(someInts)
                .sum();
```

PRODUCING A STREAM FROM A COLLECTION

```
List<String> strings = new ArrayList<>();
strings.stream();
```

CREATING A STRING FROM AN ARRAY

```
String out = IntStream.of(someInts)
           .mapToObj(String::valueOf)
           .collect(Collectors.joining(" "));
```

Here, the mapToObj() operator is new.

It uses the specified method to convert the input element to a new type.

USING LINES IN FILES AS A STREAM

```
Files.lines(Paths.get("src/main/resources/OODAssignment.csv"))
```

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any
other words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                              .map(line ->
splitAtSpaces.splitAsStream(line))
                              .collect(Collectors.toList());
```

What is the type of list after this is run?
How many elements are in the list?
4 elements in the final list.
(one for each entry in someStrings)

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any
other words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                              .map(line ->
splitAtSpaces.splitAsStream(line))
                              .collect(Collectc
```

```
Pattern splitAtSaces = Pattern.compile("\\s+")
String someStrings[] = {"one row", "some more
other words", "and once upon a time"};
Object list =  Stream.of(someStrings)
                              .flatMap(line ->
splitAtSpaces.splitAsStream(line))
                              .collect(Collectors.toList());
```

What is the type of list after this is run?
How many elements are in the list?
4 elements in the final list.
(one for each entry in someStrings)

FLATMAP()?

```
Pattern splitAtSpaces = Pattern.compile("\\s+");
String someStrings[] = {"one row", "some more words", "any other
words", "and once upon a time"};

Object list =  Stream.of(someStrings)
                              .flatMap(line ->
splitAtSpaces.splitAsStream(line))
                              .collect(Collectors.toList());
```

When I really want 13 items in the final list (one for every word in the original input), I use `flatMap()`.
When the output of a map() is a collection, `flatMap()` flattens the result by adding all the items in the output to the stream individually, rather than as a collection.

FUNCTIONAL PROGRAMMING SO FAR - SUMMARY

- Stream that gets mapped, filtered, reduced, and collected… in some order
  - Intermediate operations are not executed until a terminal operation is called
- Lambdas: unnamed methods (functions) that can be applied to a stream
- Declarative vs. imperative

FUNCTIONAL PROGRAMMING SO FAR - SUMMARY

- A tenet of functional programming is immutability
  - An object is not mutable– it can't change
  - Rather than change state (mutate it), create a new copy with the new state
  - Helps with concurrency

FUNCTIONS AS OBJECTS

CS 5004, SPRING 2024 – LECTURE 13

FUNCTIONAL INTERFACES

- Introduced in Java 8
  - An interface that contains only a single abstract (unimplemented) method
- Example 1:

```
public interface MyFunctionalInterface {
    public void run();
    }
```

- Example 2:

```
public interface MyFunctionalInterface2{
    public void execute();
    public default void print(String text) {
        System.out.println(text);
    }
}
```

FUNCTIONAL INTERFACES

- Can be implemented by lambda expressions

- Example 3:

```
MyFunctionalInterface lambda = () ->
 {System.out.println("Executing...");
}
```

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java
  1. Function (**`java.util.function.Function`**) – represents a function that takes a single parameter, and returns a single value

```
public interface Function<T,R> {
   public <R> apply(T parameter); }
```

  2. Predicate (**`java.util.function.Function`**) – represents a simple function that takes a single value parameter, and returns true or false

```
public interface Predicate {
 boolean test(T t); }
```

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java
  3. UnaryOperator – represents an operation which takes a single parameter, and returns a parameter of the same type
     - It can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it again

  4. BinaryOperator– represents an operation that takes two parameters and returns a single value, where both parameters and return value have to be of the same type

FUNCTIONAL INTERFACES

- Built-in Functional Interfaces in Java
  3. Supplier – represents an operation that supplies a value of some sort
     - This interface can be thought of as a factory interface

  4. Consumer – represents a function that consumers a value without returning any value

FUNCTIONS AS OBJECTS

- Let's consider functional interface **`Function`** again

```
public interface Function<T,R> {
    public <R> apply(T parameter); }
```

- The given interface can be implemented with a function object:

```
Function<T, R> functionName = {t → operation returning R}
```

- Calling a function object:

```
T oldObject = new T();
R newObject - functionName.apply(oldObject);
```

USE OF FUNCTIONS AS OBJECTS

- To tidy up stream operations
- If we have a higher-order method, and we need to pass a function as a parameter to it
- If we want a function to be accessible to only one object (e.g., even listeners)
- A design choice

NETWORKING IN JAVA

CS 5004, SPRING 2024 – LECTURE 13

ACKNOWLEDGEMENT

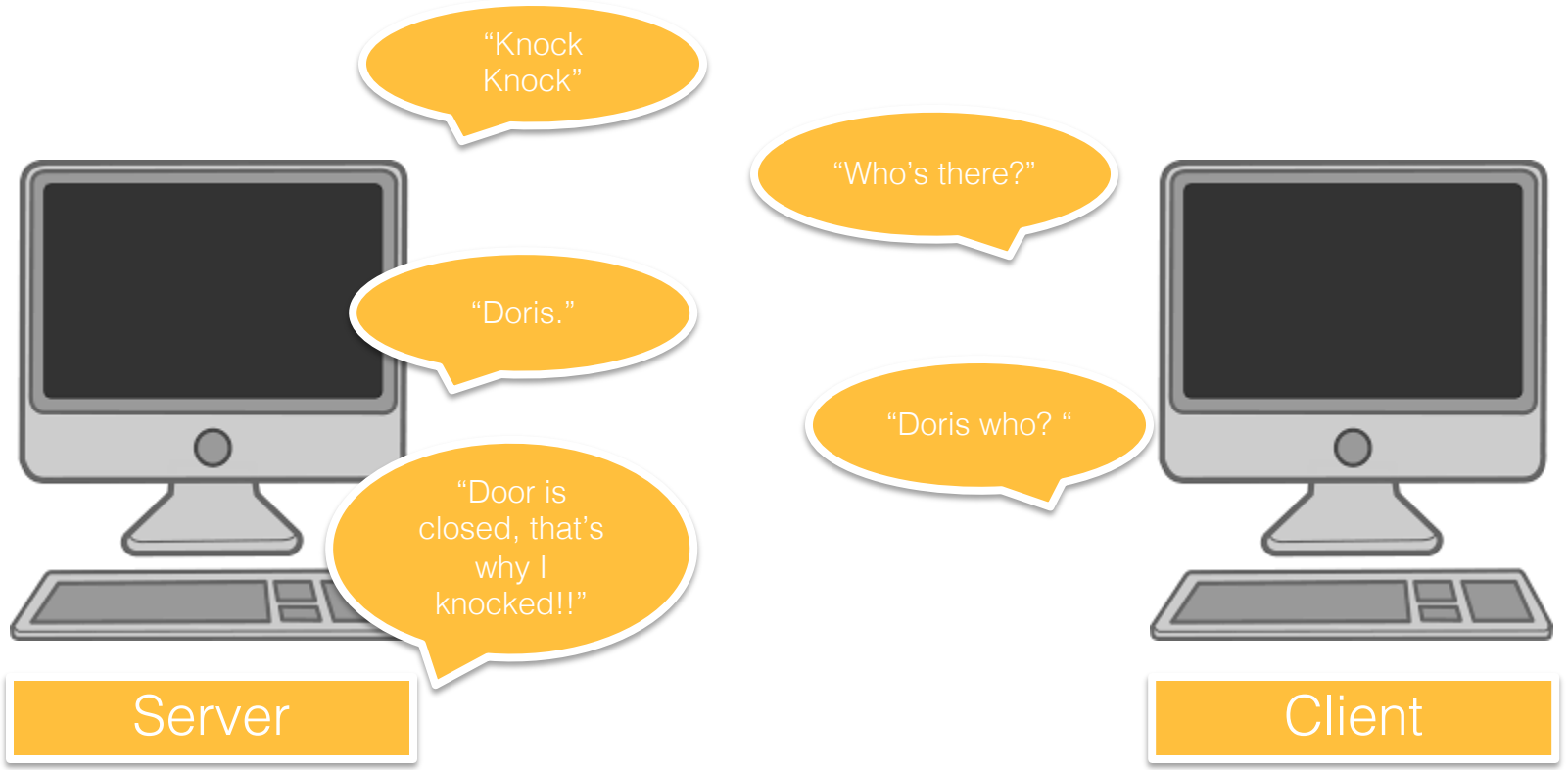
Notes adapted from Dr. Adrienne Slaughter. Thank you.

GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS

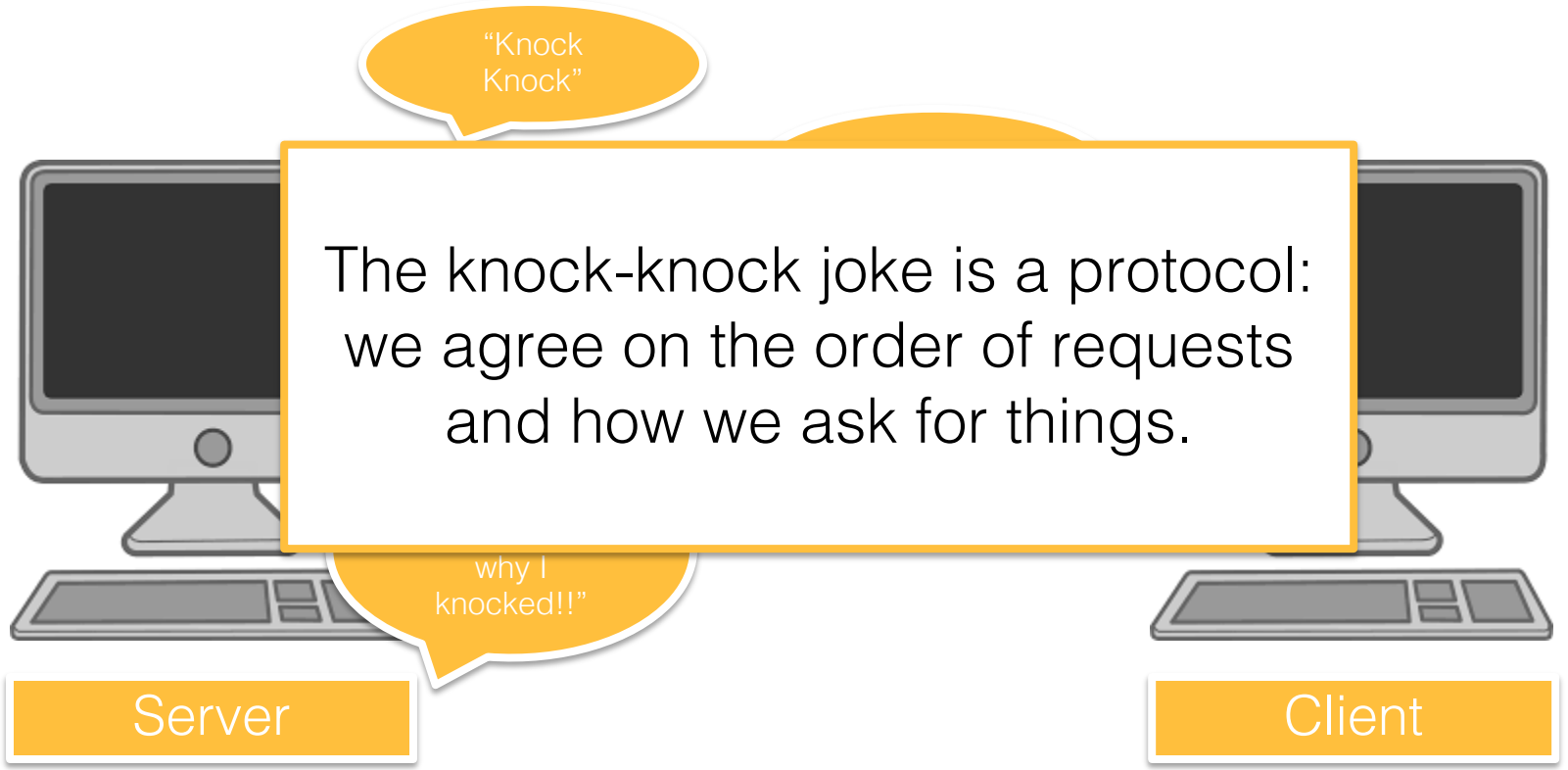
Server

Client

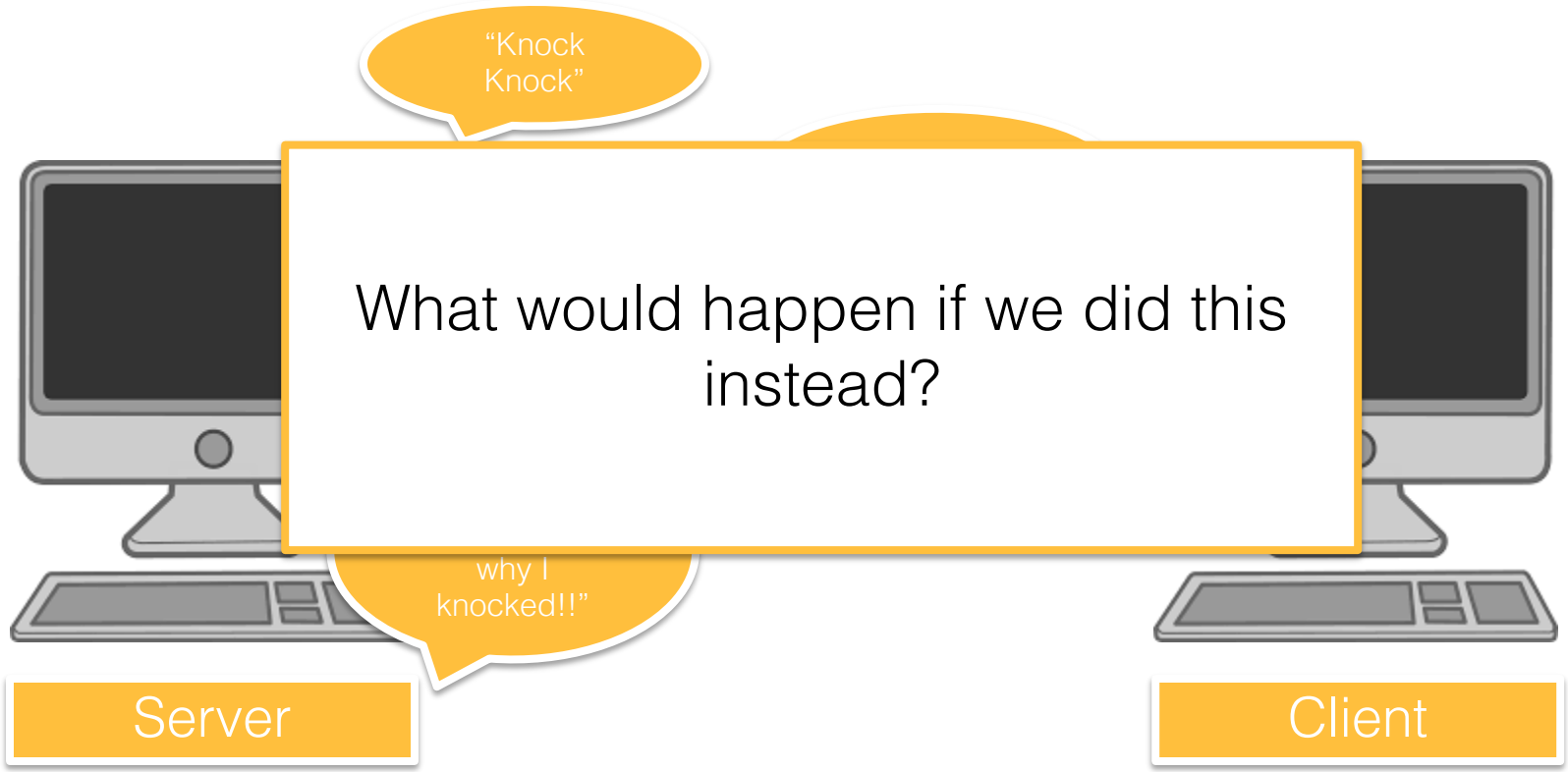
GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS



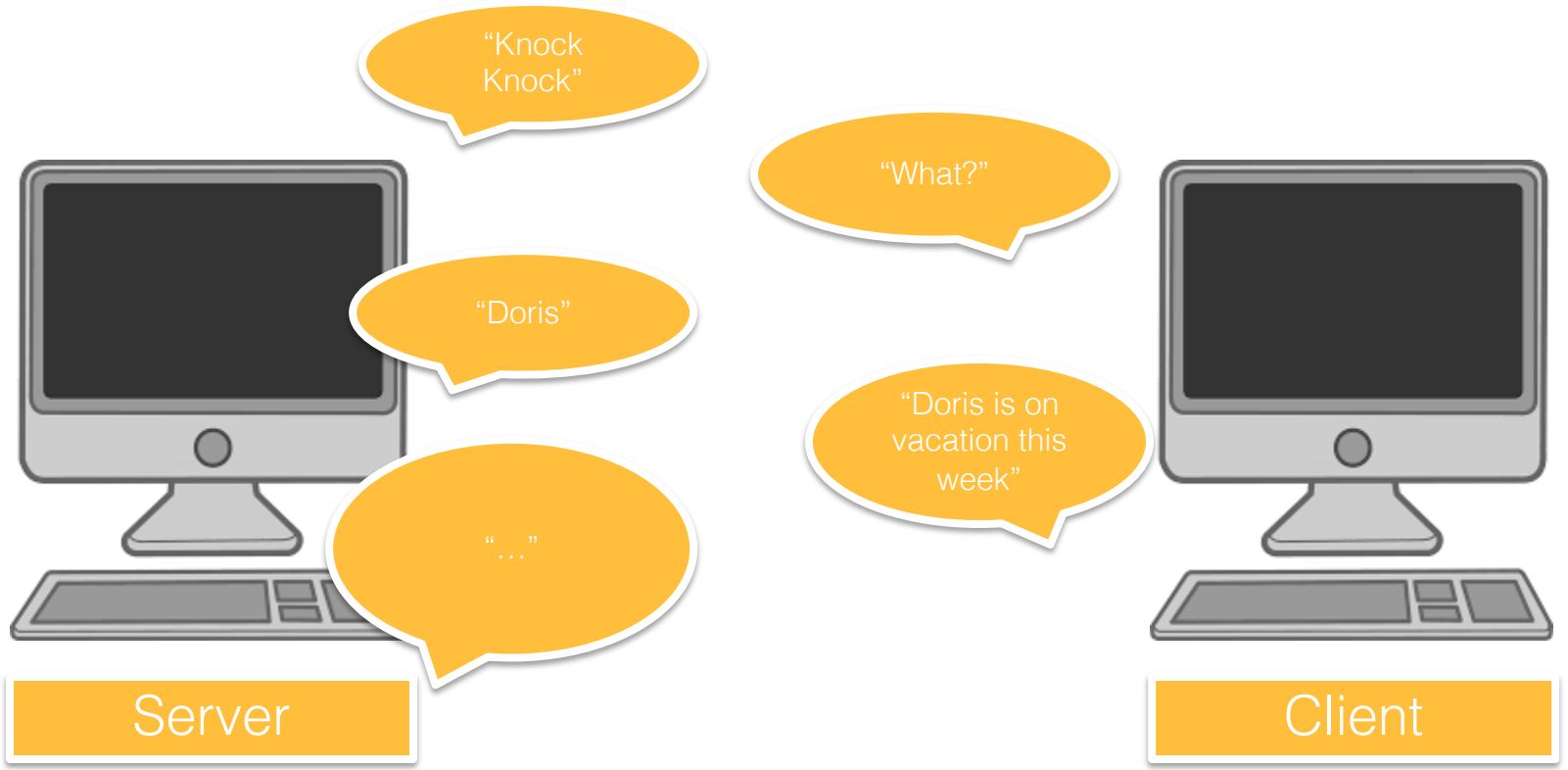
GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS



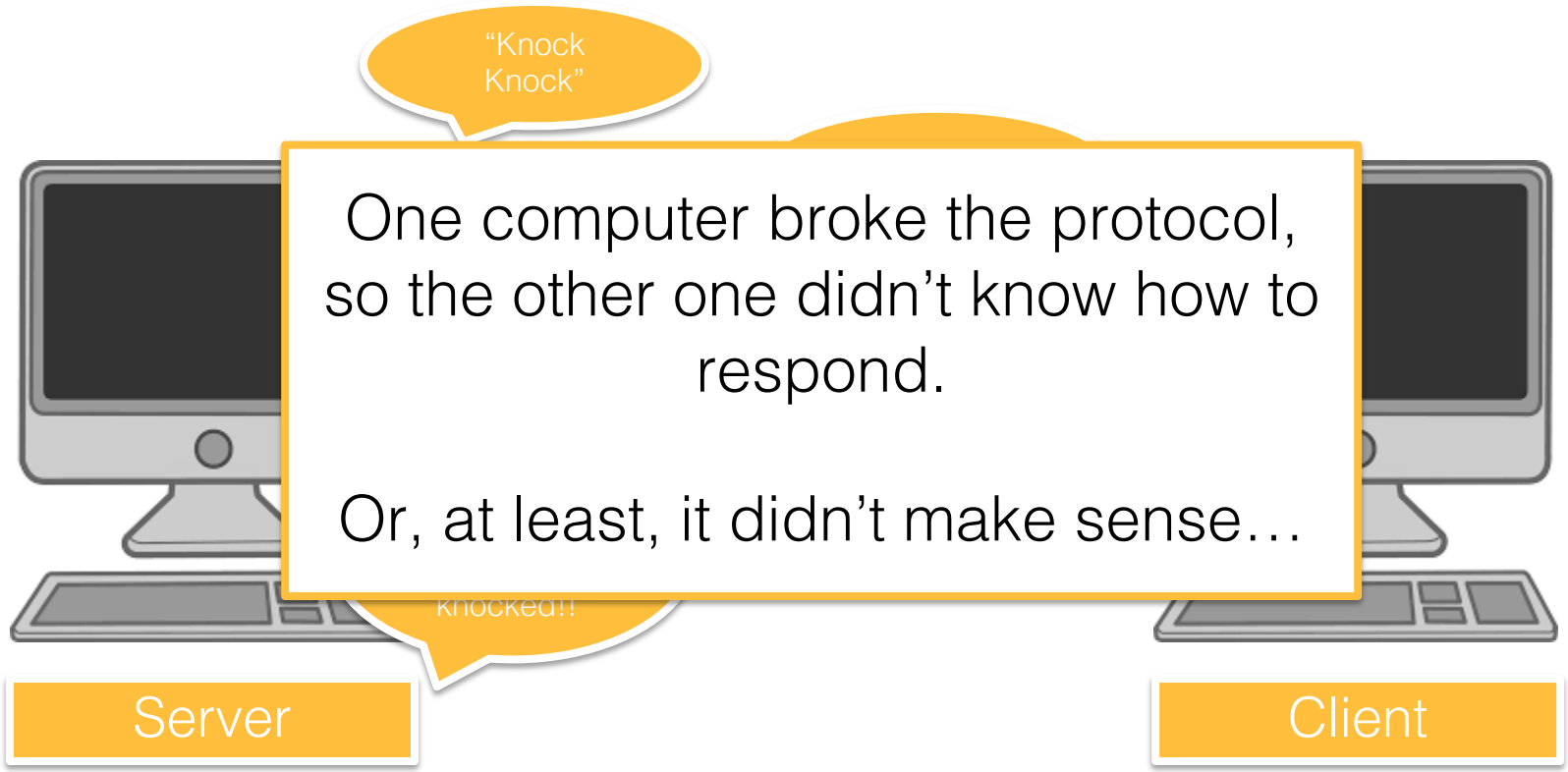
GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS



GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS



GOAL: COMMUNICATING DATA BETWEEN APPLICATIONS



One computer broke the protocol, so the other one didn't know how to respond.

Or, at least, it didn't make sense…

CONSIDER THE WEB

WebServer

Client: Web Browser

CONSIDER THE WEB

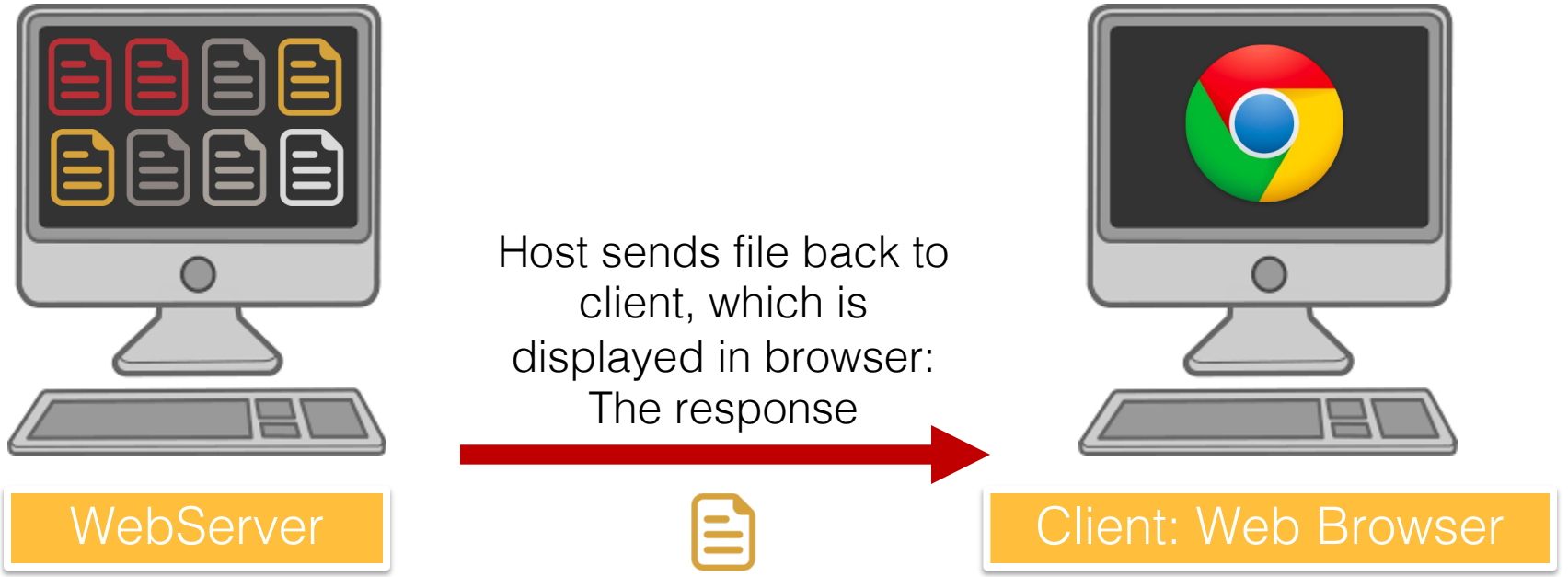
WebServer

Client sends URL to host/server, specifying which document: the request

"Give me the red one"

Client: Web Browser

CONSIDER THE WEB



CONSIDER THE WEB

WebServer

This works because the server and client agree to use the same protocol: HTTP

Client: Web Browser

HTTP

- HyperText Transfer Protocol
- Consists of 2 basic messages:
  - Request
  - Response
- Each of the request/response consists of **headers**

HOW DOES THE DATA GET TRANSFERRED?

WebServer

Application Data

All that HTTP stuff is just Application Data– data that 2 applications (the web server and web browser) use to communicate.

Client: Web Browser

HOW DOES THE DATA GET TRANSFERRED?

WebServer

Application Data

How do we actually connect to machines and transfer data?

Client: Web Browser

HOW DOES THE DATA GET TRANSFERRED?

WebServer

Application Data

First, we open a socket
on each machine

Client: Web Browser

HOW DOES THE DATA GET TRANSFERRED?

WebServer

Application Data

The apps will use the socket to communicate with the other machine/application.

Client: Web Browser

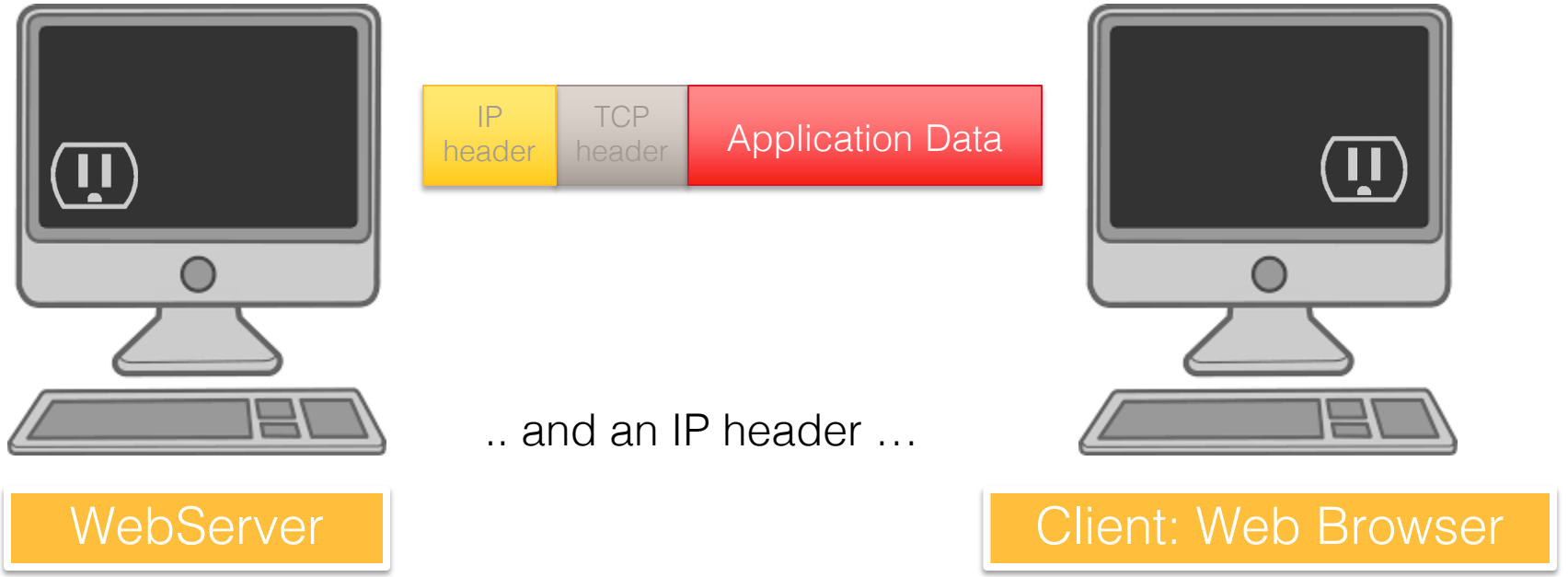
HOW DOES THE DATA GET TRANSFERRED?

The application data gets a TCP header added to it…

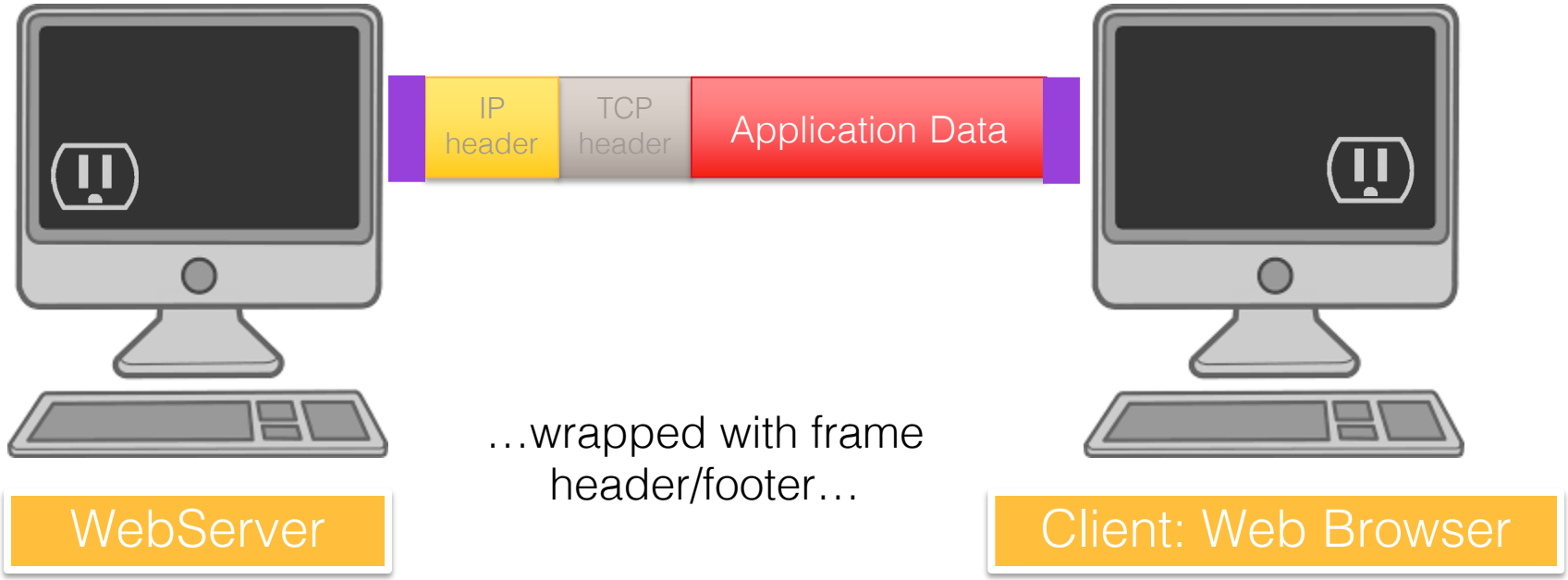
WebServer

Client: Web Browser

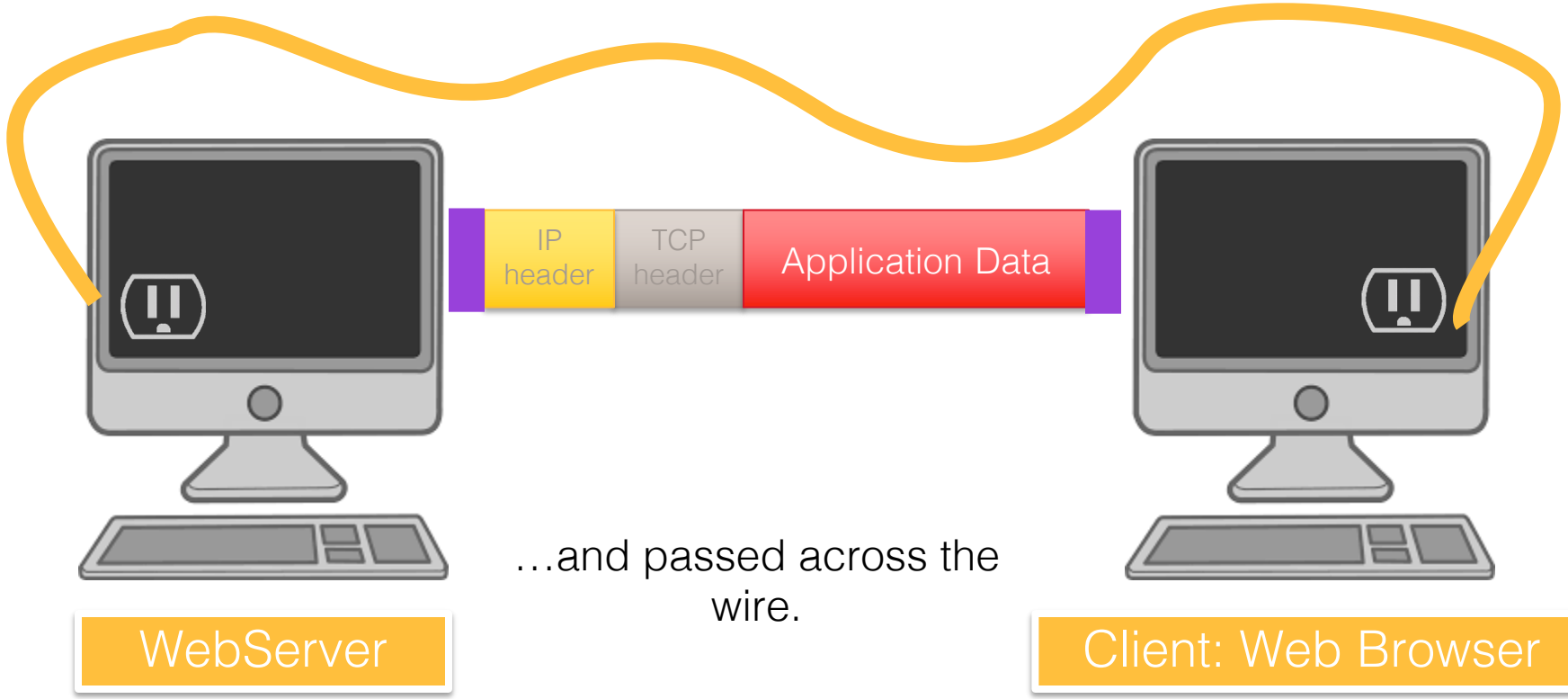
HOW DOES THE DATA GET TRANSFERRED?



HOW DOES THE DATA GET TRANSFERRED?



HOW DOES THE DATA GET TRANSFERRED?



WHAT PIECES DO WE NEED TO WORRY ABOUT?

i.e., lecture objectives

- Naming of network resources
  - How to specify which computer you want to connect to
- Sockets
  - How to allow your computer to talk directly to another computer
- Communication protocols
  - Agreeing on the communication
- HTTP connections
  - Because the web
- JSON
  - Also, the web

NETWORKING CONCEPTS, ISSUES AND GOALS

- Naming: How to find the computer/host you want to connect to
- Transfer: The actual connection
- Communicating: Sending data back and forth in a way that both the client and host/server understand

THE GENERAL PROCESS

- Open a socket
- Open an input stream and output stream to the socket
- Read from and write to the stream according to the server's protocol
- Close the streams
- Close the socket

THE GENERAL PROCESS

Naming

Transfer

Communicating

- Open a socket
- Open an input stream and output stream to the socket
- Read from and write to the stream according to the server's protocol
- Close the streams
- Close the socket

NETWORKING IN JAVA - NAMING

CS 5004, SPRING 2024 – LECTURE 13

URL AND URI

- URI: Uniform Resource Identifier
- URL: Uniform Resource Locator
- Often used interchangeably, but there is a difference:
  - URL is very specific: includes item (e.g., a specific file name) and protocol (how to get the item)
    - Example: http://www.northeastern.edu/index.html
  - URI can be less specific:
    - Example: northeastern.edu
    - Doesn't specify access (e.g., ftp? http?) or specific page (index.html)

ANATOMY OF AN URL

ANATOMY OF AN URL

Without protocol & resource name, we can't have a URL. Path and parameters can be null.

ANATOMY OF AN URL

`http://www.theimdbapi.org/api/movie?movie_id=tt0089218`

Protocol **Resource Name:** **Path** **Parameters**

- **Hostname**
- **Filename**
- **Port Number**
- **Reference (optional)**

ANATOMY OF AN URL

`http://www.theimdbapi.org/api/movie?movie_id=tt0089218`

Protocol **Resource Name:**
- **Hostname**
- **Filename**
- **Port Number**
- **Reference (optional)**

Path **Parameters**

All of this information allows a **socket** to be opened up.

But connecting only via URLs is pretty high level– a lot of abstraction is happening.

What if we want to define our own protocol? We need to open a socket directly.

JAVA CLASSES

- `java.net.URL`
- `java.net.URI`
- `java.net.Socket`

JAVA CLASSES - EXAMPLE

```
private static void tryUrl(){
    try {
        // Create URL
        URL myURL = new URL("northeastern.edu");
        System.out.println("The URL is " + myURL);
    }
    catch (MalformedURLException e) {
        // new URL() failed
        e.printStackTrace();
    }
}

private static void tryUri(){
    try {
        // Create URI
        URI myURI = new URI("northeastern.edu");
        System.out.println("The URI is " + myURI);
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
}
```

JAVA CLASSES - EXAMPLE

```
private static void tryUrl(){
    try {
        // Create URL
        URL myURL = new URL("northeastern.edu");
        System.out.println("The URL is " + myURL);
    }
    catch (MalformedURLException e) {
        // new URL() failed
        e.printStackTrace();
    }
}

private static void tryUri(){
    try {
        // Create URI
        URI myURI = new URI("northeastern.edu");
        System.out.println("The URI is " + myURI);
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
}
```

Which one throws an exception?

JAVA CLASSES - EXAMPLE

```
private static void tryUrl(){
    try {
        // Create URL
        URL myURL = new URL("northeastern.edu");
        System.out.println("The URL is " + myURL);
    }
    catch (MalformedURLException e) {
        // new URL() failed
        e.printStackTrace();
    }
}

private static void tryUri(){
    try {
        // Create URI
        URI myURI = new URI("northeastern.edu");
        System.out.println("The URI is " + myURI);
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }
}
```

tryURL() fails, because the string “northeastern.edu” doesn’t tell us enough about the protocol or file that we’re interested in.

Replacing the string with “http://northeastern.edu” will make it work.

SOME POPULAR PROTOCOLS

- HTTP: Hypertext Transfer Protocol
- FTP: File Transfer Protocol
- SMTP: Simple Mail Transfer Protocol

SOME POPULAR PROTOCOLS - EXAMPLE

```
try (
        Socket socket = new Socket(hostName, portNumber); ) {
       // App code goes here:
       // Read from socket, write to socket. (more details soon)
       socket.close();

} catch (UnknownHostException e) {
    System.err.println("Don't know about host " + hostName);
    System.exit(1);
}
```

To go lower-level, open a Socket with a hostname and a portNumber.

SUMMARY OF NAMING

- We have to have a way of specifying which computer we want to connect to
- In Java, we do this with URIs, URLs, and for lower-level client/server programming, sockets
- A socket requires a hostname and a port
- A URL requires a protocol and a resource name

NETWORKING IN JAVA - TRANSFER

CS 5004, SPRING 2024 – LECTURE 13

RELEVANT JAVA CLASSES

- For naming:
  - `java.net.URL`
  - `java.net.URI`
- For connecting:
  - `java.net.URLConnection, java.net.HttpUrlConnection`
  - `java.net.Socket`
- For actual transfer:
  - `java.io.InputStreamReader`
  - `java.io.BufferedReader`
  - `java.io.PrintWriter`

THREE EXAMPLES

1. Reading data from a URL directly
2. Connect to a URL, and initiate a session for input/output
3. Create a socket and connect to it directly

Example 1:
Read directly from URL

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

Open a stream from the defined URL

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

Pass it into an InputStreamReader to handle the input.

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

Pass that into a BufferedReader to make it easy for you to handle the input.

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

While there is still text coming in from the stream connection, get it, and print to console.

```
private static void readUrl(){
    try {
        // Create URL
        URL myURL = new URL("http://www.northeastern.edu");

        BufferedReader in = new BufferedReader(
                new InputStreamReader(myURL.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
    catch (MalformedURLException e) {
        // new URL() failed
        // ...
    }
    catch (IOException e) {
        // openConnection() failed
        // ...
        e.printStackTrace();
    }

}
```

Don’t forget to close your connection!!

EXAMPLE 1 - SUMMARY

- Simple, easy way to get data from a URL
- This example was a web page, but could just as easily be a REST endpoint that contains data
- Transfer was only one way: could only read
- Limited: Some web servers require specific HTTP headers/values, and you can't modify the parameters here

Example 2: Connect to URL for input/output

```
private static void openHttpConnection(){
    try {
        // Create URL
        String theURL = "http://www.theimdbapi.org/api/movie?movie_id=tt0089218";
        URL myURL = new URL(theURL);

        // Connect to URL
        HttpURLConnection connection = (HttpURLConnection) myURL.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("User-Agent", "App/java app demo");
        connection.setRequestProperty("Content-Type", "application/json");

        connection.connect();

        // Read from/Write to the connection
        BufferedReader in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
    // Handle exceptions (omitted for clarity)
}
```

Rather than just calling "openStream()" on the URL, call openConnection() to create a connection object that we can set parameters on before calling.

```
private static void openHttpConnection(){
    try {
        // Create URL
        String theURL = "http://www.theimdbapi.org/api/movie?movie_id=tt0089218";
        URL myURL = new URL(theURL);

        // Connect to URL
        HttpURLConnection connection = (HttpURLConnection) myURL.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("User-Agent", "App/java app demo");
        connection.setRequestProperty("Content-Type", "application/json")

        connection.connect();

        // Read from/Write to the connection
        BufferedReader in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
    // Handle exceptions (omitted for clarity)
}
```

Now, set some parameters:

- requestMethod specifies a GET rather than a POST.
- This particular server requires a User-Agent.
- Content-type just says I expect json in return.
- These are all details that are not always relevant, and change from application to application.

```
private static void openHttpConnection(){
    try {
        // Create URL
        String theURL = "http://www.theimdbapi.org/api/movie?movie_id=tt0089218";
        URL myURL = new URL(theURL);

        // Connect to URL
        HttpURLConnection connection = (HttpURLConnection) myURL.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("User-Agent", "App/java app demo");
        connection.setRequestProperty("Content-Type", "application/json");

        connection.connect();

        // Read from/Write to the connection
        BufferedReader in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
    // Handle exceptions (omitted for clarity)
}
```

Connect!

This actually opens the connection with the given parameters.

```
private static void openHttpConnection(){
    try {
        // Create URL
        String theURL = "http://www.theimdbapi.org/api/movie?movie_id=tt0089218";
        URL myURL = new URL(theURL);

        // Connect to URL
        HttpURLConnection connection = (HttpURLConnection) myURL.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("User-Agent", "App/java app demo");
        connection.setRequestProperty("Content-Type", "application/json");

        connection.connect();

        // Read from/Write to the connection
        BufferedReader in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
    // Handle exceptions (omitted for clarity)
}
```

But now, just do the same thing we did last time: Create an inputStreamReader, wrap it in a BufferedReader, and dump the response to the console.

```
private static void openHttpConnection(){
    try {
        // Create URL
        String theURL = "http://www.theimdbapi.org/api/movie?movie_id=tt0089218";
        URL myURL = new URL(theURL);

        // Connect to URL
        HttpURLConnection connection = (HttpURLConnection) myURL.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("User-Agent", "App/java app demo");
        connection.setRequestProperty("Content-Type", "application/json");

        connection.connect();

        // Read from/Write to the connection
        BufferedReader in = new BufferedReader(new InputStreamReader(
                connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
    // Handle exceptions (omitted for clarity)
}
```

Don’t forget to close!!

EXAMPLE 2 SUMMARY

- Fairly easy way to connect to a URL
- Gives more control over the connection:
  - Can set parameters, header info
- We didn't use this, but we can use the connection to do output as well
- Still constrained to using a pre-specified protocol (HTTP, FTP, …)

Example 3:
Connect to Socket

In this example, we're looking at an implementation of the Knock-Knock client-server we saw earlier

KNOCK-KNOCK DEMO COMPONENTS

- KnockKnockServer:
  - Listens for clients
  - Parses client input
  - Sends a response
- KnockKnockClient:
  - Takes in user input
  - Sends it to the server
  - Displays server response to the user
- KnockKnockProtocol: (We'll talk about this in the next section)
  - Determines appropriate output for given input

First the client…

(It's pretty similar to what we've seen before)

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

This time, start by opening a socket, giving a hostname and a portnumber.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

In addition to reading from the server, we need to write to the server.
Do this by creating a PrintWriter.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

But since we also need to read from the server, also create the BufferedReader from an InputStreamReader.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

This client takes input from the user and sends it to the server.
Use another BufferedReader with another InputStreamReader to get input from System.in.

Note this pattern: System.in is a source of input to your program, just as the data we get from the server either via a socket or URLConnection.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

While the server is still sending us data, keep getting input from the user and sending it.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

The server sent us a message saying “Bye”, which is defined by the protocol as being time to finish.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

Read a line from the terminal.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

Write that line to the terminal, then send the text to the server.

KnockKnockClient.java

```
try (
        Socket kkSocket = new Socket(hostName, portNumber);
        PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
) {
    BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    while ((fromServer = in.readLine()) != null) {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye."))
            break;

        fromUser = stdIn.readLine();
        if (fromUser != null) {
            System.out.println("Client: " + fromUser);
            out.println(fromUser);
        }
    }
    kkSocket.close();
} catch (Exceptions)//Handle exceptions properly here. Omitted for clarity.
```

Don't forget to close the connection when you're done!!

Now the server…

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Set up the socket to be a server listening on a specified port number (keep it >1000).

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

When a client comes along and connects to the socket, go ahead and accept the connection. Now you have a way to communicate directly with the client!

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Use the PrintWriter to send data out through the clientSocket.

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Once again, get the input stream from the socket, wrap it in a input stream, then wrap it in a BufferedReader.

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

We'll discuss this later, but it keeps track of the joke state and determines what should be said.

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Read the input from the client.

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Send the input from the client to the protocol to determine how to respond.

KnockKnockServer.java

```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

If the protocol says to say "Bye", the session is over and we can quit.

KnockKnockServer.java

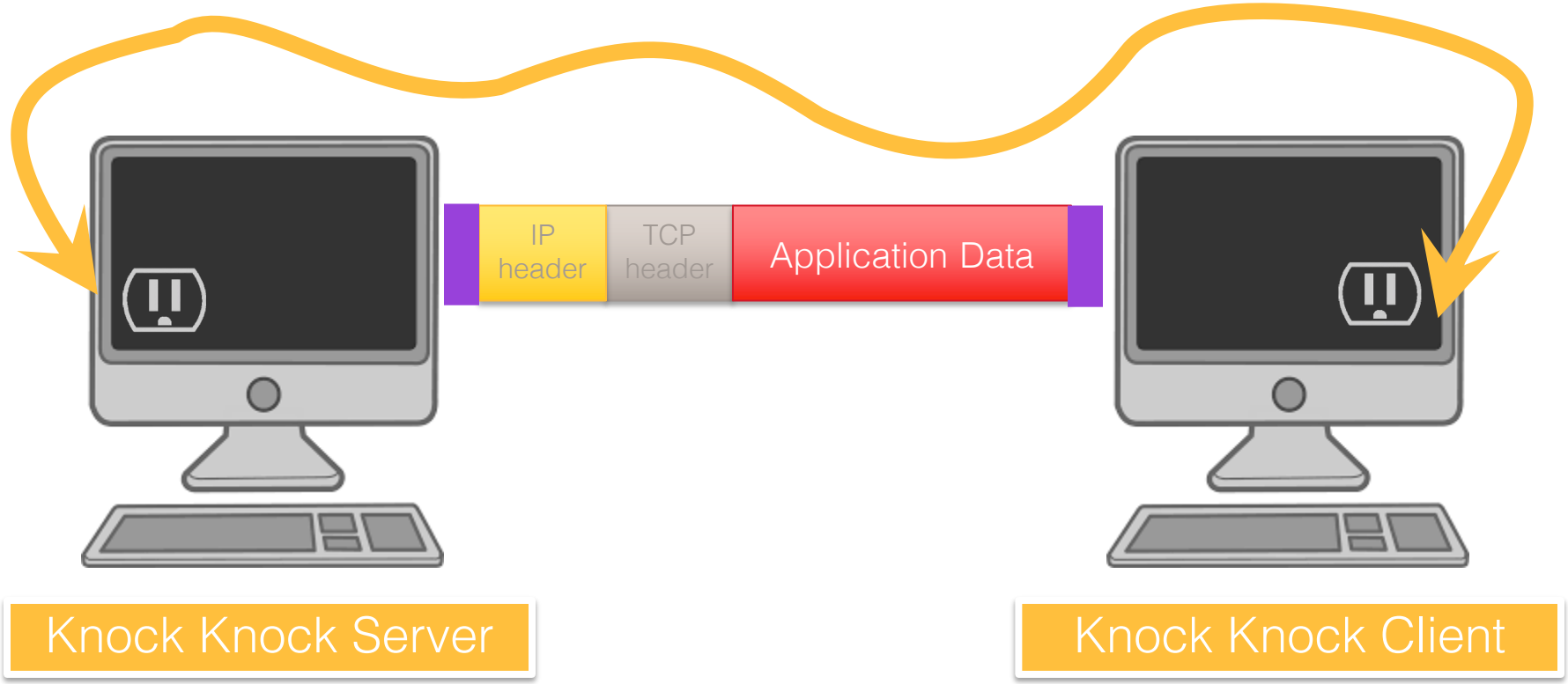
```
try (
        ServerSocket serverSocket = new ServerSocket(portNumber);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
) {
    String inputLine, outputLine;
    out.println("The knock knock server is here! Just come on along. ");
    // Initiate conversation with client
    KnockKnockProtocol kkp = new KnockKnockProtocol();
    outputLine = kkp.processInput(null);
    out.println(outputLine);

    while ((inputLine = in.readLine()) != null) {
        outputLine = kkp.processInput(inputLine);
        out.println(outputLine);
        if (outputLine.equals("Bye."))
            break;
    }
} catch (IOException e)// Do the right thing here. You should know by now.
```

Don't forget to close your connection!!

SOME NOTES, NOW THAT WE'VE SEEN THE CODE

- The server runs and opens up a socket on a specific port (e.g. 1200)
- The client runs, and we provide it with the name of the server (hostname) and the port (e.g. 1200)
- When the server and client are running on the same machine (e.g., testing), the hostname is "localhost"

REMEMBER THIS PICTURE?



EXAMPLE 3 SUMMARY

- The client reads input from the server, and sends data to the server
- The server reads input from the client, and sends the data to the client
- The protocol decides how to interpret the messages sent between the client and the server

NETWORKING IN JAVA - COMMUNICATING

CS 5004, SPRING 2024 – LECTURE 13

Imagine two people talking to each other.
One is speaking in French, the other is speaking in English.
How much communication is happening?

True communication can’t happen if we don’t agree on what do words mean

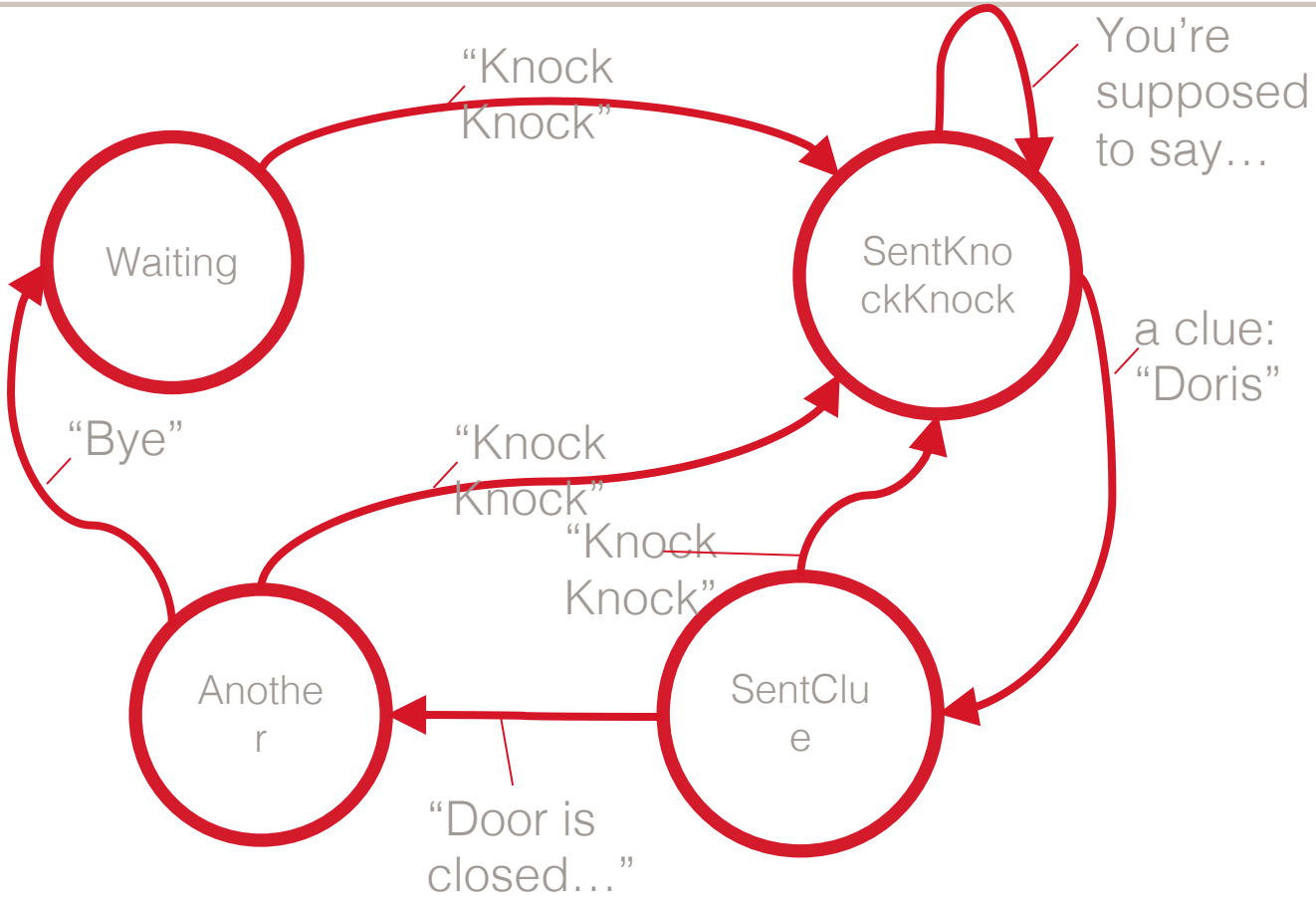
This is where the protocol comes in

ALL ABOUT PROTOCOLS

- Usually defined in a document
- Sometimes implemented as a library that can be included in your code
- Whether your code uses an external library or not, it needs to conform to the protocol

KNOCK KNOCK PROTOCOL

- Can be represented by a state diagram (next slide)
- The output is a combination of the current state and the input (from the client)


"Knock Knock"
You're supposed to say…
Waiting
SentKnockKnock
a clue: "Doris"
"Bye"
"Knock Knock"
"Knock Knock"
Another
SentClue
"Door is closed…"

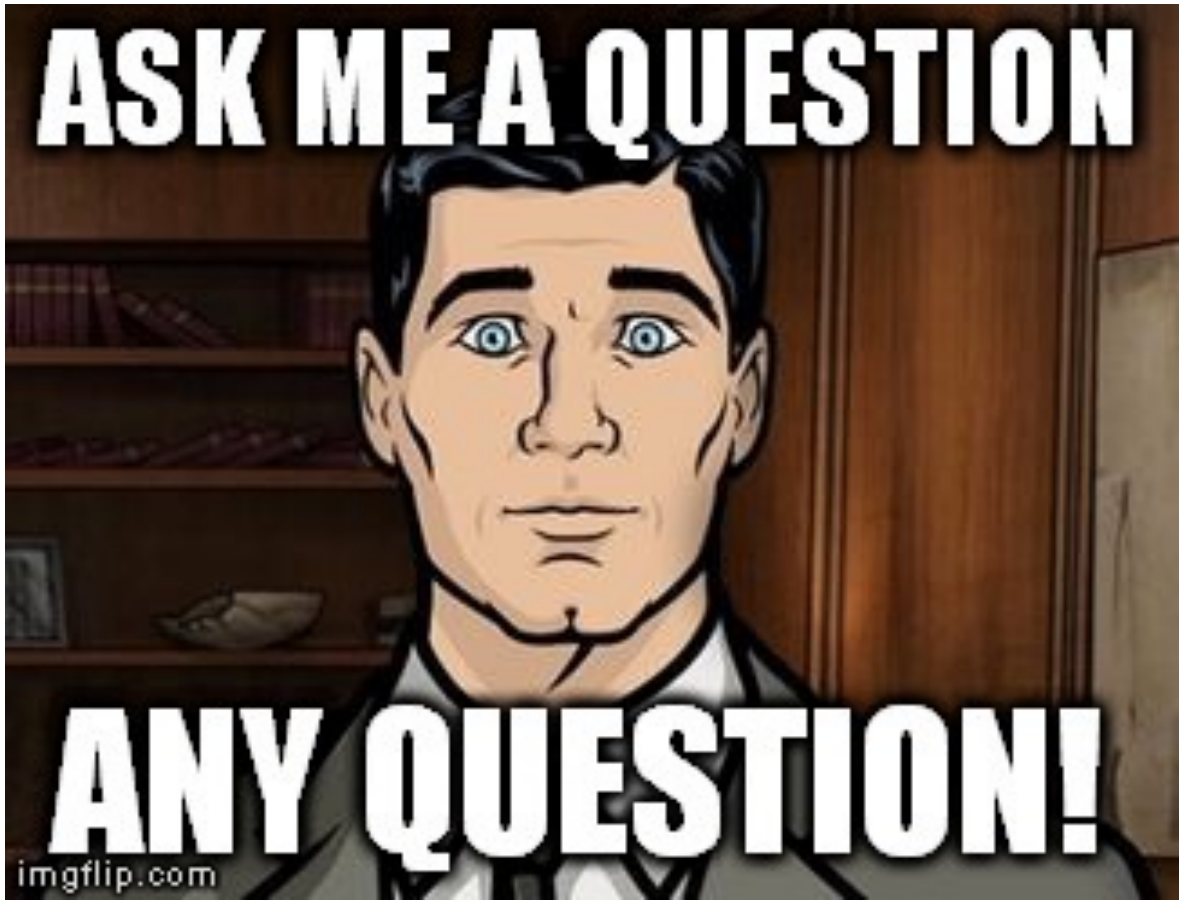
```
switch(state){
    case WAITING:
        theOutput = "Knock Knock";
        state = SENTKNOCKKNOCK;
        break;

    case SENTKNOCKKNOCK:

        if (theInput.equalsIgnoreCase("")){
            theOutput = clues[currentJoke];
            state = SENTCLUE;
        }
        else{
            theOutput = "You're supposed to say Who's there?";
        }
        break;

    case SENTCLUE:
        if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")){
            theOutput = answers[currentJoke] + " Want another? (y/n)";
            state = ANOTHER;
        }
        else{//...
```

```
        case SENTCLUE:
            if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")){
                theOutput = answers[currentJoke] + " Want another? (y/n)";
                state = ANOTHER;
            }
            else{


                theOutput = "You're supposed to say... ";
                state = WAITING;
            }
            break;

        case ANOTHER:
            if (theInput.equalsIgnoreCase("y")) {
                theOutput = "Knock! Knock!";
                if (currentJoke == (NUMJOKES - 1))
                    currentJoke = 0;
                else
                    currentJoke++;
                state = SENTKNOCKKNOCK;
            } else {
                theOutput = "Bye.";
                state = WAITING;
            }
            break;
        default:
            theOutput = "Whaaaat?";
            state = WAITING;
            break;
    }
```

SUMMARY: WAYS OF NETWORKING IN JAVA

- Via URL Connection
  - Create a URL
  - Establish a connection
  - Make requests:
    - PUT
    - GET
  - Process response
  - Can either read directly, or establish session and communicate
- Via Sockets
  - Direct connection to a server via a socket listening on a port
  - Must follow agreed-upon protocol

YOUR QUESTIONS



[Meme credit: imgflip.com]