Logistics

- Teams stabilized.
- Programming assignment 1 posted.
  - Due Oct. 21, 2pm.
  - Feel free to e-mail me if you have issues during the exercise.
- Today:
  - Wrap up TCP, discuss RMI.
  - Probably won't get through RMI today entirely. We'll spill into next week a little.
    - I built in some wiggle room in the schedule.

Questions summarizing weeks 1-2

- Any questions about the material over the last 2 weeks?

- You should consider the lectures so far as intended to lay a foundation upon which the really interesting stuff happens in distributed systems.

TCP stream communications

- The abstraction TCP provides is that of a stream of bytes between sender and receiver.
  - Versus the bounded length datagrams of UDP.
- What does this mean?
  - TCP handles bundling data into IP packets, so application-level message sizes are sent. Packetization is hidden.
  - TCP acknowledges receipt of messages. So, if a message is lost (i.e.: no ACK before a timeout), TCP automatically retransmits it.
  - Flow control: Backs of rate sender transmits data if receiver is slower.
  - Prevents reordering and duplication by attaching IDs to each packet.
  - Once a connection is established, it persists so both sides can read and write to it.

Caveats

- What is put into the stream must be read out in the same order on the other end.
  - E.g.: Writing an int and then a double requires reading an int, then a double. If this cooperation doesn't occur, the data is likely to be interpreted incorrectly.
- Blocking: If a sender is throttled due to flow control, a send may block. A receive may block if no data has arrived yet.
- Threads
  - Typically, a server accepts() a connection and spawns a thread to deal with that connection so it can listen for new ones.
  - Polling via select() is an alternative. This can have lower overhead and work on systems without threads, but it is trickier to manage.

TCP failure model

- TCP retries address omission failures, and checksums address corruption and arbitrary failures. The protocol masks these by defining how retries and retransmissions are handled.
- If a connection is truly bad and the data simply cannot be properly transmitted (i.e.: resend limit exceeded), the TCP layer may break the connection.
- TCP will notify both sides when they attempt to use the socket that it is no longer valid.
  - This means a bad communication channel (network failure) is indistinguishable from a process failure on the other side.
  - A process can't tell if recently sent messages were received properly.

Uses of TCP

- Most familiar protocols are built on top of TCP.
  - HTTP
  - SMTP
  - FTP

- Why? These protocols require reliability and TCP allows them to gain it without each application or higher level protocol being responsible for implementing it themselves.
- Typically the cost paid for TCP overhead versus UDP is acceptable for this benefit.

Java socket API : Server side

- Servers create a `ServerSocket` object to bind to a local port and listen for incoming requests.
- The `accept()` method on the `ServerSocket` blocks until a request arrives, and the result is a `Socket` object representing the connection.
- The `Socket` provides access to `InputStream` and `OutputStream` objects for reading and writing.

- If a server wishes to be able to handle more than one connection at a time, one can bundle the handling of the Socket IO in a Java `Thread`.
- Figure 4.6 has an example of this.

Java socket API: Client side

- Clients create `Socket` objects by passing in the hostname and port of the server to connect to.
- Like the server side, the `Socket` object provides `InputStream` and `OutputStream` objects for I/O.
- Java Sockets conveniently encapsulate name resolution when you create them, so you can provide a symbolic name and port without having to explicitly look up the `InetAddress` first.

Java socket API: Error conditions

- In the event of a failure in some part of the process, Java exceptions allow for processes on either side of the connection to gracefully deal with them.

External data representations

- In a previous lecture we pointed out that **heterogeneity** is a challenge in designing and implementing distributed systems.
- One of the reasons is that not all systems choose to represent information the same way internally.
  - Does the most significant byte of an integer come first or last?
  - Does a system use 8-bit ASCII or 16-bit Unicode?
  - Are floating point numbers represented the same way?
  - Are arrays stored contiguously following row or column major ordering?
- All of these prohibit the direct sharing of raw data between systems. You need to put data into a common form that every participant agrees upon in advance.

External data representations

- We call this agreed upon form the *external data representation*. Some packages abbreviate this to XDR.
- The act of putting data into this agreed upon form is called *marshalling*.
- The intermediate form can be either:
  - A fully specified data format. E.g.: All text will be Unicode, all integers will be big-endian, etc…
  - The native format of the sender, with a header that the receiver can read to determine what format the sender assumed.

Common representations

- CORBA common data representation
- Java Object serialization
- XML

- A popular older one is the IETF standard XDR format intended to live at the presentation layer of the stack (between the application and lower level protocols).
  - See RFC 1832 for information.
  - NFS and other tools based on ONC RPC use this XDR.
    - Open Network Computing, Remote Procedure Call : a close relative of SunRPC.

Common features of XDRs

- Platform-neutral representation of primitive types (ints, floats, etc…).

- Recursive representation of structured types.
  - C structs.
  - C++ classes, Java classes.
  - Unions, enumerations.

- Metadata beyond the type and contents.
  - Array lengths, dimensions.
  - String lengths.

Java serialization

- Serialization flattens an object and it’s contents (potentially other objects) into a form that can be transmitted to another system.
- Deserialization is the inverse operation of restoring the objects in memory.

- Serialization can also be used to “freeze” objects to store for later, such as in a file. It isn’t restricted to communication uses.

Java serialization

- How does it work?
- Instance variables are written out in a platform-neutral format, along with their datatypes and names.
- This is recursively applied to other objects that are contained within the object being serialized.
- References are serialized by assigning unique handles to each instance of an object, ensuring that multiple references to the same instance will be stored as the same handle.
  - Obviously we can’t store the actual reference address and hope it will be correct when the object is deserialized. Hence the use of handles.

Java serialization

- How do you make an object serializable?
  - Implement the "`serializable`" interface.
- For the most part, you don't need to explicitly write the code to write the raw serialized bytes representing the object or putting an object back together from the stream.
- You generally can assume that if an object came from the Java standard library, it is serializable.
- Java has a nice facility called "reflection" that allows you to interrogate objects to find out about their class definition and structure at runtime.
  - This is how the serialization system can automatically scan through an object and determine the types and names of the fields.

XML representations

- Document markup language.
- You can represent structured data by creating elements and attributes on the elements. The elements can contain other elements.
- E.g.:

```
<person id="12345">
  <name>Bill</name>
  <place>Eugene</name>
  <age>55</age>
</person>
```

XML representations

- Most data is represented as a string equivalent.

- Occasionally binary data (such as hashes or security-related data) must be included. How? It is encoded using a Base64 encoding.
  - Base64 encoding uses the alphanumeric characters , +, / and = to represent binary data.
  - Every 6 bits assigned a character in a-z A-Z 0-9 + /
  - Usually encoded as messages with multiples of 24 bits, so the = character is used to pad the 6, 12, or 18 bits that may remain.

XML representation

- XML provides for schemas that are XML descriptions of the elements, attributes, and nesting relationships of a specific type of XML document.
- Schemas can be used to validate that an XML document is well-formed.
  - Typically this is performed by the XML parser. You, the end-user, are not responsible for implementing this check.

Considerations for XDRs

- Pro:
  - Standardized external representations eliminate a significant hurdle to heterogeneous systems.

- Con:
  - Performance. One must encode and decode data on either endpoint , which takes time.
  - Lack of a single standard.
    - IETF XDR, Java serialization, CORBA CDR, etc…
    - Limits interoperation between distributed systems build using different middleware packages.

Remote object references

- Systems like CORBA and Java allow for distributed programs in which processes can refer to objects that actually are stored in the memory of another process.
- This is achieved through remote object references.
- Remote object references aren’t that hard to represent.
  - Address of host containing the object.
  - Port of the host attached to the process containing the object.
  - A time and object number representing a unique identifier of the object.
- Invocations on the object instance are made over the network.

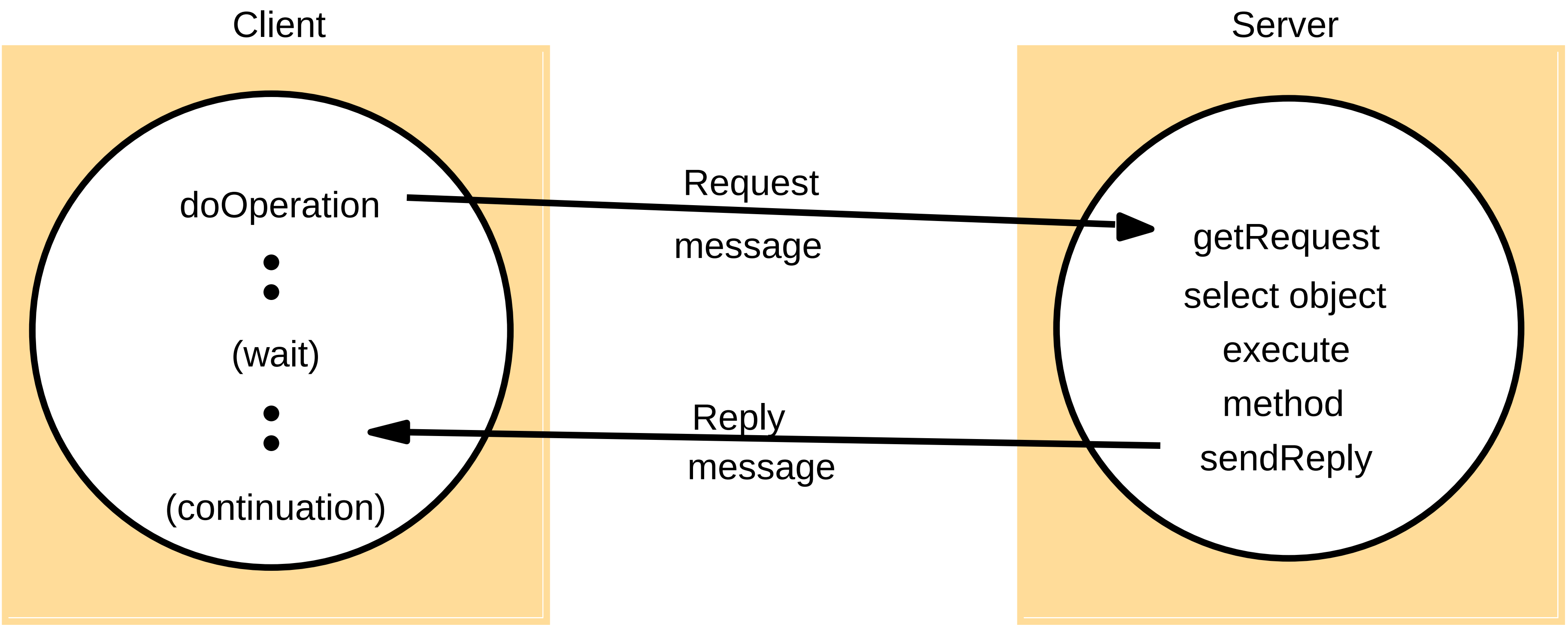
Client-server communication

- Given an object instance, what can we do with it?
  - Look at it's data.
  - Invoke methods on it.
- So, naturally we are interested in invoking methods on remote object references.
  - Remote method invocation.
- Note that we usually don't have access to instance variables remotely without going through a method interface (e.g.: setter/getter).
- How do we make this happen? RPC exchange protocols.

Request-reply protocol

- The protocol defines the set of messages passed back and forth from the client (caller) to the server (callee).
  - doOperation : Used by the client to invoke remote operations given a remote object reference.
  - getRequest : Used by the server to retrieve requests submitted by clients and execute them.
  - sendReply : Used by the server to respond to the request with the reply, possibly containing return values. Client unblocks when reply received.

Request-reply protocol



Message structure

- Messages have a simple structure:

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

- Seems redundant with TCP, right? This is intended to go over UDP too.

Considerations: Over TCP or UDP?

- Requests are followed by replies. So, a reply is essentially an acknowledgement.
  - TCP ACKs are redundant.
- Establishing a connection requires message exchange in addition to the request/reply pair.
  - Wasteful communication overhead.
- Majority of RPC calls pass few and small arguments and return values.
  - Flow control largely unnecessary.

- So, Request-Reply for RMI is perfectly fine over UDP.

Failure model

- Omission failures, obviously when over UDP.
- Reordering possible.

- The requestID is incremented for each message, so it is both unique and monotonically increasing.
  - Can be used to put messages back in order on other side and identify duplicates.

- Timeouts on doOperation on the client side lead to interesting questions.

Timeouts

- Timeouts in doOperation can result from:
  - Request never getting to the server. Resending is harmless.
  - Replies never getting back to the client.

- The first case isn't hard to deal with. The server can keep track of the most recent message ID it has received from each client host and throw out duplicates.
- The second case is harder. The reply getting lost means the computation occurred already. What to do?

Operations

- A server can either maintain a history or not.
  - If it maintains a history, this is easy – just resend the reply when the client asks for it again without recomputing.
  - If there is no history, the server has to recompute.

- Recomputation poses a problem if the computation is not idempotent. Idempotent means that the operation can be performed repeatedly with the same result each time.
  - Special measures need to be implemented if an operation provided over RPC is not idempotent.

Exchange protocol variants

- **Request** (R): Client sends a request once, and never looks for a reply.
- **Request/Reply** (RR): Client sends a request, and the server responds with a reply that the client consumes.
- **Request/Reply/Acknowledge** (RRA): RR with a client to server acknowledgement sent after the reply. The client doesn't block on the acknowledge, and the server considers an acknowledgement for requestID "X" to imply acknowledgement for "X-1" and below in the event that their acknowledgements were lost.

Example of a request/reply protocol

- HTTP