Announcements

• Reminder:
  – Project 1 due on Monday, Oct. 7th
  – In-class midterm Wednesday, Oct. 9th
Recap

- Transport Level Protocols
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)

Today

- Remote Procedure Call (RPC)
  - Java’s remote method invocation (RMI)
Remote Procedure Call (RPC)

• Mechanism for structuring distributed programs
  – Commonly used in OS kernels, for instance
• Application-level transparency
  – To the programmer, RPC looks just like local procedure call
Remote Procedure Call (RPC)

- **Request/Reply Paradigm**
  - Sender (i.e. client) sends request to server
  - Receiver (i.e. server) sends back response

- **More than UDP**
  - Reliable
  - *At-most-once* message semantics

- **Less than TCP**
  - Not byte-stream oriented
  - Establishing full duplex reliable channels is sometimes overkill
RPC Mechanism

- **Caller (client)**
  - Args
  - Request
  - Client stub
  - RPC protocol
  - Reply
  - Return val.

- **Server stub**
  - Request
  - RPC protocol
  - Reply
  - Return val.

- **Callee (server)**
  - Args
  - Request
  - Server stub
  - RPC protocol
  - Reply
Challenges for RPC

• Dealing with network properties
  – Limited message sizes
  – Lost messages
  – Reordered messages
  – Multiple deliveries

• Dealing with architectural differences
  – big vs. little endian
  – Data structure layout
  – Pointer representation

• Language semantics
  – What about network failure?
  – Garbage collection

Language & Compiler support
Networks & RPC

• Message Size
  – Use fragmentation/reassembly

• Reliability
  – Use ACK’s, timeouts, and retransmission

• In-order & at-most-once delivery
  – Use sequence numbers
Data Exchange Issues

• Different hosts may use different data representations
  – Sizes of integers, floating points, characters
  – Big vs. Little endian
  – Data structure layout in memory
    • Padding of arrays and structs
  – Pointers and structured data
    • Pointer representation might differ
    • Trees, lists, etc. must be \textit{serialized}
  – Objects and functions closures (contain code!)
    • Typically don’t transmit code
Marshalling/Unmarshalling data

- **Marshalling/Unmarshalling**
  - Process of putting data into a form that can be sent over a network

- **Strategy 1: Receiver Makes Right**
  - Send data in sender’s native form
  - Receiver fixes it up if necessary

- **Strategy 2: Canonical Intermediate Representation**
  - Sender marshals data into common format
  - Receiver unmarshals data into its own format
  - (More commonly used)
Tagging Data

• A *tag* is additional information that describes the structure of the marshalled data
  – Type information
  – Array lengths
  – Architecture information

• Untagged representations are also possible
  – Sender and receiver know a priori the types of data being transmitted
Marshalling Using Tags

```
int16 | byte8 | array | 5
ptr16 | struct{ int16, ptr16 }
```
Stubs

• A *stub* is a piece of code that implements marshalling/unmarshalling
  – Can be interpreted or compiled
  – Typically generated from the procedure interface by a *stub compiler*
Stub Compiler

Call Proc. P

Source code for P

Code for P

Client stub

Stub Compiler

RPC protocol

Marshalled args

Client stub

RPC protocol

Marshalled args
Which Procedure to Run?

• Need to identify code to execute
• Host x Port x Procedure Identifier

• Sometimes provided by well-known numbers
• Sometimes given by external mapping
SunRPC

- Implemented on top of UDP
- Designed for LAN use
  - Does not support at-most-once semantics
- A program called Port Mapper
  - Maps ports to program numbers
  - Runs on well-known UDP port 111
- Uses the External Data Representation (XDR)
  - Canonical intermediate form
  - Handles C types (except function pointers)
  - No tags
Distributed Computing Environment

- DEC’s RPC protocol
- Implemented on top of UDP
- Does support at-most-once semantics
- Used with Network Data Representation stub compiler
  - Receiver makes right strategy
  - Architecture tags
  - Description written in Interface Definition Language (IDL)
  - Essentially supports C’s type system

- Underlying mechanism for the Common Object Request Broker Architecture (CORBA)
Java RMI (Remote Method Invocation)

• Provides RPC Mechanism
• See java.rmi.* classes & documentation

• Built on top of TCP
  – Quite heavy weight
  – Reliable

• Incorporates Java’s security model
Java RMI Details

• Classes implement **Remote** interface
• Use the RMI **Naming** class to name and advertise remote objects
  – **Naming.rebind** maps URLs to Objects
  – **Naming.lookup** used by client to find Object references
• Java stub compiler is **rmic**
• Objects passed via RMI must implement the **Serializable** interface
• Non-remote arguments (and results) are passed by **copying**
• Remote arguments (and results) are passed by **reference**
Using Java’s Remote interface

/* File: Hello.java */

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
Server Implementation

/* File: HelloServer.java */
import java.rmi.*;

public class HelloServer
    extends UnicastRemoteObject
    implements Hello
{
    public HelloServer() throws RemoteException
    {
        super();
    }

    public String sayHello() {
        return "Hello World!";
    }

    /* continued on next slide */
public static void main(String args[]) {
    // Create and install a security manager
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(
            new RMISecurityManager());
    }
    try {
        HelloServer obj = new HelloServer();
        // Bind this object instance to the
        // name "HelloServer"
        Naming.rebind("//host/HelloServer", obj);
        System.out.println
            ("HelloServer bound in registry");
    } catch (Exception e) {…}
}
public class HelloClient {
    String message = "blank";
    // "obj" refers to the remote object
    Hello obj = null;

    public static void main(String args[]) {
        try {
            obj = (Hello) Naming.lookup("//host/HelloServer");
            message = obj.sayHello();
            System.out.println(message);
        } catch (Exception e) {
            // Handle exception
        }
    }
}
Compilation & Execution

• Compile both files with **javac**
  – Produces HelloServer.class and HelloClient.class

• Generate stubs by using **rmic** on HelloServer.class
  – Produces HelloServer_stub.class and HelloServer_skel.class

• Run Java’s **rmiregistry** program

• Start the server on host

• Start the client on another