Agenda

• Last time (Tues)
  • Interprocess Communication (chpt 4)
  • Sockets demo
• This time and tonight (5-6:15)/tomorrow (3-4:15): chpts 4-5
  • General RPC/RMI example
  • General RPC/RMI terminology
  • Case study: SUNRPC
  • Case study: Java RMI
  • General pros/cons of RPC/RMI
• No class Tues Feb 13 (Marty at conference)
• Note: we are skipping chpt 6
• Note: I’m generally out of contact between 7:30pm – 4am

Before we start: Recap of the demo

• Server: sunfire1.cs.virginia.edu:5451
• Cygwin is a convenient linux-like runtime on the Win platform (it’s NOT required – do it on Linux directly if you wish)
• Use “-Wall” on compilation line
• Google for documentation (e.g., “man exit”) to remove compiler warnings
• Find the appropriate header file
• Differentiate between the “pattern” to establish a sockets-based connection vs. the “application protocol”

Before we start: Assignment #1

• If the server crashes, then you’ll have to re-register
  • Hmm.. Is this not the best server design? Hmmm…
• If you find iteratively poking at the server to determine your bug annoying, you might want to implement the server yourself (your partner?)
  • Having both the client and the server might be easier to debug
  • This will give you a headstart on Assignment #2
  • BUT: the requirement is to get your client working against our server
• Advice: Get the “register” working while I’m still in town!
• Advice: Don’t try to do everything at once – design and test incrementally!

Protocol design in general

• Protocol: the design of the message pattern (and format of the messages) to solve a particular problem
• Your past: designing the UI of an app is a kind of protocol design
  • Distributed systems: machine to machine communication (the machine is NOT as forgiving as the human, so it’s MUCH more difficult)
• What makes a good protocol?
  • Complete (with extensibility?)
  • Unambiguous
  • Secure (?)
  • Minimal?
  • Efficient? Is a protocol efficient? Or is just the implementation of the protocol efficient? And is “efficient” the same as “minimal”? Implementable?
  • Is the protocol in Assignment 1 “good”? Hmm..

Finishing up sockets

• Server tends to be at a “well-known” port; client is not
• Server can be “connectionless” or “connection-oriented”
  • Connection-Oriented
    • Server opens socket for “everybody”
    • As part of client contact, a second socket is opened
    • Server forks a process to handle just that one client on the new port
    • Server and Client “stay connected” during session
  • Connectionless
    • Server opens socket for “everybody”
    • Server multiplexes across all messages arriving to that one socket
    • Server is always listening irrespective of any particular client

Well-known Ports

• ftp: 21
• ssh: 22
• telnet: 23
• finger: 79
• http: 80
• pop3: 110
• sunrpc: 111 (portmapper)
• imap: 143
• https: 443

• rough guide (OS dependent):
  • System ports < 1024;
  • User ports > 1024
• Services/ports defined in /etc/services
• Use “netstat –a –b” to check running services
### Interprocess communication (IPC)

- **Applications, services**
- **RMI and RPC**
- **request-reply protocol**
- **marshalling and external data representation**
- **UDP and TCP**

### Interprocess communication – how?

- **Primitives:** `send` and `receive`
- **Types:**
  - **Synchronous:** The sending and receiving processes synchronize at every message, the send and receive operations are blocking.
  - **Asynchronous:** The send operation is non-blocking, the receive operation may be either blocking or non-blocking.
- **Queue associated with each message destination.**
- **Senders cause messages to be added to remote queues.**
- **Receivers remove messages from local queues.**

### Hiding Message-Passing: RPC (nee. 1976)


  The request/response communication is a basis for the remote procedure call (RPC) model.
  - Think of a server as a module (data + methods).
  - Think of a request message as a call to a server method.
    - Each request carries an identifier for the desired method; the rest of the message contains the arguments.
  - Think of the reply message as a return from a server method.
    - Each reply carries an identifier for the matching call; the rest of the message contains the result.

  *With a little extra glue, the messaging communication can be hidden and made to look “just like a procedure call!” to both the client and the server.*

### Remote Procedure Call – RPC

- **Looks like a nice familiar procedure call**

  \[
  P_0 \xrightarrow{\text{result = foo(param);}} P_1 \xrightarrow{\text{Receive}}
  \]

  - `P_0` is the client (client thread)
  - `P_1` is the server (server thread)
  - `foo` is the method being called
  - `param` is the parameter passed to `foo`
  - `result` is the result returned by `foo`

  *Please do foo for P_0 with param*  
  *Here*  
  *Receive*
Remote Procedure Call – RPC

- Looks like a nice familiar procedure call

```
P_0
    result = foo(param);
    \*blocks\*
    \*here\*
    \*returning \*r to P_0 \*\*\*Reply\*
```

```
P_1
    Receive \*r = foo(param); \*// actual call
    Reply
```

Remote Procedure Call – RPC

```
P_0
    result = foo(param);
```

```
P_1
    Receive \*r = foo(param); \*// actual call
    Reply
```

RPC/RMI Concepts

- Rationale: sockets are nice, but app still knows it’s “shoving bits on the wire”
- RPC/RMI provides transparency
  - Looks like a local procedure invocation, but it’s not
  - “access” + “location” = “network transparency”
- Stubs are necessary
  - client side: connect to server machine, send all the parameters, wait for replies, manipulate the stack, and return
  - server side: wait for messages, read the parameters, present/covert for use by local procedure, send response back to client

General Issues in RPC/RMI

1. How do we specify the interface?
2. How we generate stubs?
3. How are parameters passed?
4. How does a client find/bind to a server?
5. What are the invocation semantics?
6. Ease of use
7. Performance

[1] Specifying the Interface

- Generally through use of some kind of Interface Definition Language (IDL)
- IDL defines primitive language types
Generating Stubs

An explicit “helper program” is invoked to do this.
Uses the IDL, usually generates both client-side and server-side.
Compiler must know which are IN parameters and which are OUT parameters.

How are parameters passed?

Issue: how do I pass information between heterogeneous machines?
Motivation: Information in programs represented as data structures, information in messages consists of sequences of bytes.
You can’t just “put it on the wire”

Issues:
- Implicit typing or explicit typing???
- Certain data types are invalid
- Character sets - ASCII vs. Unicode vs. ???
- Byte order: (given multi-byte numeric rep, what does the first byte signify?)
  - Big endian: the most significant value in the sequence is stored at the lowest storage address (Sparc?)
  - Little endian: Intel microprocessors?
- E.G. decimal 1025 = 00000000 00000000 00000100 00000 001 (Big Endian)
  - Little Endian: 00000001 00000100 00000000 00000000 (Little Endian)
  - Two methods for data exchange:
    - Convert values to agreed external format before transmission.
    - Transmit values in sender’s format (“receiver-make-right”)