CIS 415: Operating Systems

RPC

Prof. Kevin Butler
Spring 2012
Today’s Lecture

• Remote procedure calls
• Distributed RPC (MapReduce & Hadoop)
• Threads

• Reminders
  ‣ Assignment 1 due April 23
  ‣ Project 1 due April 25
IPC -- Message Passing

• Establish communication link
  ‣ Producer sends on link
  ‣ Consumer receives on link

• IPC Operations
  ‣ Y: Send(X, message)
  ‣ X: Receive(Y, message)

• Issues
  ‣ What if X wants to receive from anyone?
  ‣ What if X and Y aren’t ready at same time?
  ‣ What size message can X receive?
  ‣ Can other processes receive the same message from Y?
IPC -- Synchronous Messaging

• Direct communication from one process to another

• Synchronous send
  ‣ Send(X, message)
  ‣ Producer must wait for the consumer to be ready to receive the message

• Synchronous receive
  ‣ Receive(id, message)
  ‣ Id could be X or anyone
  ‣ Wait for someone to deliver a message
  ‣ Allocate enough space to receive message

• Synchronous means that both have to be ready!
• Indirect communication from one process to another

• Asynchronous send
  ‣ Send(M, message)
  ‣ Producer sends message to a buffer M (like a mailbox)
  ‣ No waiting (modulo busy mailbox)

• Asynchronous receive
  ‣ Receive(M, message)
  ‣ Receive a message from a specific buffer (get your mail)
  ‣ No waiting (modulo busy mailbox)
  ‣ Allocate enough space to receive message

• Asynchronous means that you can send/receive when you’re ready
  ‣ What are some issues with the buffer?
IPC -- Sockets

• Communication end point
  ‣ Connect one socket to another (TCP/IP)
  ‣ Send/receive message to/from another socket (UDP/IP)

• Sockets are named by
  ‣ IP address (roughly, machine)
  ‣ Port number (service: ssh, http, etc.)

• Semantics
  ‣ Bidirectional link between a pair of sockets
  ‣ Messages: unstructured stream of bytes

• Connection between
  ‣ Processes on same machine (UNIX domain sockets)
  ‣ Processes on different machines (TCP or UDP sockets)
  ‣ User process and kernel (netlink sockets)
Files and file descriptors

• Remember open, read, write, and close?
  ‣ POSIX system calls for interacting with files
  ‣ open() returns a file descriptor
    • an integer that represents an open file
    • inside the OS, it’s an index into a table that keeps track of any state associated with your interactions, such as the file position
    • you pass the file descriptor into read, write, and close
Networks and sockets

• UNIX likes to make all I/O look like file I/O
  ‣ the good news is that you can use read( ) and write( ) to interact with remote computers over a network!
  ‣ just like with files....
    • your program can have multiple network channels open at once
    • you need to pass read( ) and write( ) a file descriptor to let the OS know which network channel you want to write to or read from
  ‣ a file descriptor used for network communications is a socket
Examples of sockets

• HTTP / SSL
• email (POP/IMAP)
• ssh
• telnet
IPC: Sockets

Internet

- **Socket**
  - **client X**: 10.12.3.4:5544
  - **client Y**: 44.1.19.32:7113
  - **web server**: 128.95.4.33:80

- **Networks**
  - **10.12.3.4**
  - **44.1.19.32**
  - **128.95.4.33**
128.95.4.33

Web server

fd 5  fd 8  fd 9  fd 3

index.html  pic.png

10.12.3.4 : 5544  44.1.19.32 : 7113

OS's descriptor table

<table>
<thead>
<tr>
<th>file descriptor</th>
<th>type</th>
<th>connected to?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pipe</td>
<td>stdin (console)</td>
</tr>
<tr>
<td>1</td>
<td>pipe</td>
<td>stdout (console)</td>
</tr>
<tr>
<td>2</td>
<td>pipe</td>
<td>stderr (console)</td>
</tr>
</tbody>
</table>
| 3               | TCP socket | local: 128.95.4.33:80  
remote: 44.1.19.32:7113 |
| 5               | file     | index.html                             |
| 8               | file     | pic.png                                |
| 9               | TCP socket | local: 128.95.4.33:80  
remote: 102.12.3.4:5544 |
Types of sockets

• **Stream sockets**
  ‣ for connection-oriented, point-to-point, reliable bytestreams
    • uses TCP, SCTP, or other stream transports

• **Datagram sockets**
  ‣ for connection-less, one-to-many, unreliable packets
    • uses UDP or other packet transports

• **Raw sockets**
  ‣ for layer-3 communication (raw IP packet manipulation)
Stream sockets

• Typically used for client / server communications
  ‣ but also for other architectures, like peer-to-peer

• Client
  ‣ an application that establishes a connection to a server

• Server
  ‣ an application that receives connections from clients
Datagram sockets

- Used less frequently than stream sockets
  - they provide no flow control, ordering, or reliability
- Often used as a building block
  - streaming media applications
  - sometimes, DNS lookups
IPC -- Sockets

- Issues
- Communication semantics
  - Reliable or not
- Naming
  - How do we know a machine’s IP address? DNS
  - How do we know a service’s port number?
- Protection
  - Which ports can a process use?
  - Who should you receive a message from?
    - Services are often open -- listen for any connection
- Performance
  - How many copies are necessary?
  - Data must be converted between various data types
Remote Procedure Calls

• IPC via a procedure call
  ‣ Looks like a “normal” procedure call
  ‣ However, the called procedure is run by another process
    • Maybe even on another machine

• RPC mechanism
  ‣ Client stub
  ‣ “Marshall” arguments
  ‣ Find destination for RPC
  ‣ Send call and marshalled arguments to destination (e.g., via socket)
  ‣ Server stub
  ‣ Unmarshalls arguments
  ‣ Calls actual procedure on server side
  ‣ Return results (marshall for return)
Remote Procedure Calls

**Diagram:**

**Client Side:**
- User calls kernel to send RPC message to procedure \( X \)
- Kernel sends message to matchmaker to find port number
- Kernel places port \( P \) in user RPC message
- Kernel sends RPC
- Kernel receives reply, passes it to user

**Message Exchange:**
- From: client
  - To: server
  - Port: matchmaker
  - Re: address for RPC \( X \)
  - Matchmaker receives message, looks up answer

- From: server
  - To: client
  - Port: kernel
  - Re: RPC \( X \)
  - Port: \( P \)
  - Matchmaker replies to client with port \( P \)

- From: client
  - To: server
  - Port: port \( P \)
  - <contents>
  - Daemon listening to port \( P \) receives message

- From: RPC
  - Port: \( P \)
  - To: client
  - Port: kernel
  - <output>
  - Daemon processes request and processes send output
Remote Procedure Calls

- Supported by systems
  - Java RMI
  - CORBA

- Issues
  - Support to build client/server stubs and marshalling code
  - Layer on existing mechanism (e.g., sockets)
  - Remote party crashes… then what?

- Performance versus abstractions
  - What if the two processes are on the same machine?
Remote Procedure Calls

• Marshalling

```
// client
val = server.someMethod(A,B)

// stub

// remote object
boolean someMethod (Object x, Object y)
{
    implementation of someMethod
    ...
}

// skeleton

// A, B, someMethod

// boolean return value
```
Example (RMI Server)

```java
public class RmiServer extends UnicastRemoteObject
    implements RmiServerIntf {
    public static final String MESSAGE = "Hello world";

    public RmiServer() throws RemoteException {
    }

    public String getMessage() {
        return MESSAGE;
    }

    public static void main(String args[]) {
        System.out.println("RMI server started");

        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager installed.");
        } else {
            System.out.println("Security manager already exists.");
        }

        ... try {
            // Instantiate RmiServer
            RmiServer obj = new RmiServer();

            // Bind this object instance to the name "RmiServer"
            Naming.rebind("//localhost/RmiServer", obj);

            System.out.println("PeerServer bound in registry");
        } catch (Exception e) {
            System.err.println("RMI server exception:" + e);
            e.printStackTrace();
        }
    }
```

Binding to registry
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiServerIntf extends Remote {
    public String getMessage() throws RemoteException;
}
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

public class RmiClient {
    // "obj" is the reference of the remote object
    RmiServerIntf obj = null;

    public String getMessage() {
        try {
            obj = (RmiServerIntf) Naming.lookup("//localhost/RmiServer");
            return obj.getMessage();
        } catch (Exception e) {
            System.err.println("RmiClient exception: " + e);
            e.printStackTrace();
            return e.getMessage();
        }
    }

    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        RmiClient cli = new RmiClient();
        System.out.println(cli.getMessage());
    }
}
Modern RPC

• Google Protocol Buffers
  ‣ Data serialization methods that are language and platform neutral: define how data structured, use generated source to interface with objects in C++, Java, Python
  ‣ Handles marshalling and versioning, RPC interface but no implementation

• Apache (Facebook) Thrift
  ‣ Serialization, versioning and RPC implementation
  ‣ Bindings for Java, C++, Python, C#, Cocoa, Erlang, Haskell, OCaml, Perl, PHP, Ruby, Smalltalk

• Others: XML RPC, JSON RPC
MapReduce

• Distributed computing framework for working on large data sets on compute clusters

• Divide data into subset that are “mapped” to each node involved in computation

• Collect all subproblem answer and “reduce” to form the final output

• Uses:
  ‣ distributed sort and grep
  ‣ graph reversal and search
  ‣ statistical analysis and web analytics, bioinformatics
MapReduce: Word Count

Input: Deer Bear River, Car Car River, Deer Car Bear

Split: Deer Bear River, Car Car River, Deer Car Bear

Map: Deer, 1, Bear, 1, River, 1

Shuffle: Deer, 1, Car, 2, River, 1

Reduce: Bear, 1, Car, 2, Car, 1, Deer, 1, River, 1

Final: Deer, 2, Car, 3, Bear, 2, Car, 3, Deer, 2, River, 2
void map(String name, String document):

// name: document name
// document: document contents
for each word w in document:
    EmitIntermediate(w, "1");

void reduce(String word, Iterator partialCounts):
// word: a word
// partialCounts: a list of aggregated partial counts
int sum = 0;
for each pc in partialCounts:
    sum += ParseInt(pc);
    Emit(word, AsString(sum));

Concepts come from functional programming
(pay attention in CIS 425!)
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.*;

public class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken()); /* splits lines into words */
                output.collect(word, one);
            }
        }
    }
    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get(); /* sums all the collected words */
            }
            output.collect(key, new IntWritable(sum));
        }
    }
}
```java
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class); /* collects all values together */
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}
```

Scalable framework: works on single-node machine, “pseudo-distributed” (single machine, multiple processes), or fully distributed cluster (depending on how Hadoop installation is set up)
IPC Summary

• Lots of mechanisms
  ‣ Pipes
  ‣ Shared memory
  ‣ Sockets
  ‣ RPC

• Trade-offs
  ‣ Ease of use, functionality, flexibility, performance

• Implementation must maximize these
  ‣ Minimize copies (performance)
  ‣ Synchronous vs Asynchronous (ease of use, flexibility)
  ‣ Local vs Remote (functionality)
Summary

• Process
  ‣ Execution state of a program

• Process Creation
  ‣ fork and exec
  ‣ From binary representation

• Process Description
  ‣ Necessary to manage resources and context switch

• Process Scheduling
  ‣ Process states and transitions among them

• Interprocess Communication
  ‣ Ways for processes to interact (other than normal files)
• Next time: Threads