CS 138: Communication II
Writing Distributed Programs

• Concerns
  – transparency
  – portability
  – interoperability

• Solutions
  – RPC
  – RMI
Common communication pattern

Client

Hey, do something

Server

working {

Done/Result
Writing it by hand...

- eg, if you had to write a, say, radio station server (as in CS168’s snowcast)

```c
typedef struct {
    uint8_t type;
    union {
        struct {
            uint16_t udpPort;
        } handshake;
        struct {
            uint16_t stationNo;
        } set_station;
    }
} command;

int sendCommand(int socket, command *c){
    char outputBuff[10];
    //have to set the network byte order properly.
    int ret, bytesread=0;
    c->handshake udpPort = htons(c->handshake.udpPort);
    memcpy(outputBuff = htonl(c->handshake.udpPort, 1);
    memcpy(outputBuff+1, &c->handshake.udpPort, 2);
    ret = send_all(socket, outputBuff, 3);
    do { if (ret<= 0) return ret; bytesread += ret; } while(0)
    c->handshake.udpPort = ntohs(c->handshake.udpPort);
    return bytesread;
}

int res;
command c;
//send to the server the init/subscribe packet
  c.type = COMMAND_HELLO;
  c.handshake.udpPort = UDPPort;
res = sendCommand(serverSocket, &c);
if (res != sizeof(c.type) +
    sizeof(c.handshake.udpPort))
    //...
Idea

Make remote functions look like local functions!

Historical note: Seems obvious in retrospect, but RPC was only invented in the ‘80s. See Birrell & Nelson, “Implementing Remote Procedure Call” ... or Bruce Nelson, Ph.D. Thesis, Carnegie Mellon University: Remote Procedure Call., 1981 :)
RPC

• A type of client/server communication

• Attempts to make remote procedure calls look like local ones

```c
{ ...
    foo()
}
void foo() {
    invoke_remote_foo()
}
```
RPC Goals

• Ease of programming
• Hide complexity
• Automates task of implementing distributed computation
• Familiar model for programmers (just make a function call)
Hiding Complexity

• Makes a call to a remote service look like a local call
  – RPC makes transparent whether server is local or remote
  – RPC allows applications to become distributed transparently
  – RPC makes architecture of remote machine transparent
But it’s not always simple

• Calling and called procedures run on different machines, with different address spaces
  – And perhaps different environments .. or operating systems, or architectures ..
• Must convert to local representation of data
• Machines and network can fail
Local Procedure Calls

// Client code
... result = procedure(arg1, arg2);
...

// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
... return(result);
}
Remote Procedure Calls (1)

// Client code
... result = procedure(arg1, arg2);
...

// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
    ... return(result);
}
Remote Procedure Calls (2)

// Client code
...
result = procedure(arg1, arg2);
...

// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
    ...
    return(result);
}
Stubs: obtaining transparency

• Compiler generates from API stubs for a procedure on the client and server

• Client stub
  – **Marshals** arguments into machine-independent format
  – Sends request to server
  – Waits for response
  – **Unmarshals** result and returns to caller

• Server stub
  – **Unmarshals** arguments and builds stack frame
  – Calls procedure
  – Server stub **marshals** results and sends reply
“stubs” and IDLs

• RPC stubs do the work of marshaling and unmarshaling data
• *But how do they know how to do it?*
• Two basic alternatives
  – Write a description of the function signature using an IDL -- interface definition language.
    - Lots of these. Some look like C, some look like XML, ... details don’t matter much.
  – Use reflection information to do this
    - Java RMI
Remote Procedure Calls (1)

• A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client’s OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

Continued ...
Remote Procedure Calls (2)

...  
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server’s OS sends the message to the client’s OS.
9. The client’s OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.
Block Diagram

Client

Application

Stub

RPC support code

Transport protocol

Server

Remote procedure

Stub

RPC support code

Transport protocol
Local vs. Distributed

- Latency
- Memory access
- Partial failure
- Concurrency
Latency

• Remote invocation of objects takes much longer than local invocation
  – can this be ignored at first and dealt with later?
Synchronous RPC

- The interaction between client and server in a traditional RPC.
Asynchronous RPC (1)

- The interaction using asynchronous RPC.
Asynchronous RPC (2)

- A client and server interacting through two asynchronous RPCs.
Concurrency

• Distributed programs have the same concurrency issues as multithreaded programs
  – Do they? In a single address space…
    - all threads are under control of a common OS
    - synchronization is easy
    - fate sharing
RPC failures

• Request from cli $\rightarrow$ srv lost
• Server crashes after receiving request
  — Before executing
  — After executing
• Reply from srv $\rightarrow$ cli lost
• Client crashes after sending request
Partial failures

- In local computing:
  - if machine fails, application fails
- In distributed computing:
  - if a machine fails, part of application fails
  - one cannot tell the difference between a machine failure and network failure
  - one cannot (in principle) tell the difference between a failure and a really long execution!
- How to make partial failures transparent to client?
Strawman solution

• Make remote behavior identical to local behavior:
  – Every partial failure results in complete failure
    - You abort and reboot the whole system
  – You wait patiently until system is repaired

• Problems with this solution:
  – Many catastrophic failures
  – Clients block for long periods
    - System might not be able to recover
Let’s try to do better

• Can we maintain transparency?
Uncertainty

Client \rightarrow \text{Request} \rightarrow \text{Server}

Client \rightarrow \text{Server}

Client \rightarrow \text{Request} \rightarrow \text{Server}

A

Client \rightarrow \text{Request} \rightarrow \text{Server}

Client \rightarrow \text{Server}

B

Client \rightarrow \text{Request} \rightarrow \text{Server}

Client \rightarrow \text{Response} \rightarrow \text{Server}
Idempotent Procedures

Client
Write File A Block 0
Server

Client
Done
Server

Client
(Ahem …) Write File A Block 0
Server

Client
Done
Server

At-least-once semantics
Non-Idempotent Procedures

Client → Server
Transfer $1000 from Kyle’s Account to Andy’s

Client → Server
Done

Client → Server
Do it again and again and again!

Client → Server
Done

At-most-once semantics
Maintaining History

Client

Transfer $1000 from Kyle’s Account to Andy’s

Server

Done

Client

Do it again!

Server

Client

Done (replay)

Server

At-most-once semantics
No History

Client → Transfer $1000 from Kyle’s Account to Andy’s → Server

Client → Server → Done

Client → Server → CRASH!!

Client → Server → Do it again!

Client → Server → Sorry ...

At-most-once semantics
In practice

• It is also possible to have “undo” procedures
  – E.g., reconciliation in the case of bank accounts
  – Apologies, reimbursements, etc.
## Coping

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td></td>
</tr>
<tr>
<td>Duplicate filtering</td>
<td>Re-execute procedure or retransmit reply</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Some content from David Andersen.
Exactly-Once?

• Sorry - no can do *in general*.
• Imagine that message triggers an external physical thing (say, a robot fires a nerf dart at the professor)
• The robot could crash immediately before or after firing and lose its state. Don’t know which one happened. Can, however, make this window very small.
Real solution: break transparency

- **At-least-once**: Just keep retrying on client side until you get a response.
  - Server just processes requests as normal, doesn’t remember anything. Simple!

- **At-most-once**: Server might get same request twice...
  - Must re-send *previous* reply and not process request (implies: keep cache of handled requests/responses)
  - Must be able to identify requests
  - Strawman: remember *all* RPC IDs handled. -> Ugh! Requires infinite memory.
  - Real: Keep sliding window of valid RPC IDs, have client number them sequentially.
Real solution: break transparency

• Possible semantics for RPC:
  – At least once:
    - Only safe for idempotent operations
  – At most once
    - Zero, don’t know, or once
  – Exactly-once
    - Impossible in practice
Memory Access

• Pointers work locally
• Can they be made to work remotely?
  – yes ... (but it’s complicated)
  – but don’t use a remote pointer thinking it’s just like a local pointer
Implementation Concerns

- As a general library, performance is often a big concern for RPC systems
- Major source of overhead: copies and marshaling/unmarshaling overhead
- Zero-copy tricks:
  - Representation: Send on the wire in native format and indicate that format with a bit/byte beforehand. What does this do? Think about sending uint32 between two little-endian machines (DEC RPC)
  - Scatter-gather writes (writev() and friends)
Summary:
expose remoteness to client

• Expose RPC properties to client, since you cannot hide them

• Application writers have to decide how to deal with partial failures
  – Consider: E-commerce application vs. game
Important Lessons

• Procedure calls
  – Simple way to pass control and data
  – Elegant transparent way to distribute application
  – Not only way…

• Hard to provide true transparency
  – Failures
  – Performance
  – Memory access
  – Etc.

• How to deal with hard problem → give up and let programmer deal with it
  – “Worse is better”
Today’s Lecture

• Sockets
• RPC
  – Overview
  – Challenges
  – Examples
Two styles of RPC implementation

• Shallow integration. Must use lots of library calls to set things up:
  – How to format data
  – Registering which functions are available and how they are invoked.

• Deep integration.
  – Data formatting done based on type declarations
  – (Almost) all public methods of object are registered.

• Sun RPC, XMLRPC, GRPC, Thrift
• Java uses the latter.
Example: gRPC

- Google’s open source RPC package
- Uses the IDL approach
  - Services defined using protocol buffers
  - Active support for C++, Java, Python, Go, Ruby, Node.js, C#, Objective-C, PHP

- Usage:
  - Create service definition (.proto)
  - Compile with protoc
  - Use generated
Helloworld example

helloworld.proto:

```protobuf
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {} 
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```
Helloworld example

Client:

import pb "google.golang.org/grpc/examples/helloworld/helloworld"
...
func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial(address, grpc.WithInsecure())
    ...
    defer conn.Close()
    c := pb.NewGreeterClient(conn)

    // Contact the server and print out its response.
    name := defaultName
    ...
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{Name: name})
    ...
    log.Printf("Greeting: %s", r.Message)
}
Helloworld example

Server:

```go
import pb "google.golang.org/grpc/examples/helloworld/helloworld"

// server is used to implement helloworld.GreeterServer.
type server struct{}

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, error) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

func main() {
    lis, err := net.Listen("tcp", port)
    ...
    s := grpc.NewServer()
    pb.RegisterGreeterServer(s, &server{})
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```
RMI: a Remote Interface

package compute;

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

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
RMI: the Argument

package compute;

public interface Task<T> {
    T execute();
}

Some content from David Andersen.
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() { super(); }
    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
}
RMI: the Server (2)

```java
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
```
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {
    private final int digits;
    public Pi(int digits) {this.digits = digits;} // constructor
    // lots of stuff deleted …
    public BigDecimal execute() {
        return computePi(digits);
    }
    // lots more stuff deleted …
}
Other examples

• SunRPC (RFC4506)
  – IDL-based, used by NFS
  – C/C++ focus, was very popular

• DCE
  – Developed in the 90’s
  – More complex (pointers, server state, arrays)
  – Basis for Microsoft RPC

• Thrift.
  – Similar to GRPC, originally dev. by Facebook