CS 138: Communication II

Today's Lecture

- Sockets
- RPC
 - Overview
 - Challenges
 - Examples

Sockets



Sockets

- TCP and UDP allow sending and receiving of bytes over the network
 - TCP: reliable infinite stream of bytes between two processes
 - UDP: unreliable messages (up to 64KB)
- How do applications access these protocols?

Using TCP/IP

Sockets API.

- Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
- Important do know and do once
- Higher-level APIs build on them
- After basic setup, much like files
 - Sockets are file descriptors

Types of Sockets

- Datagram sockets: unreliable message delivery
 - With IP, gives you UDP
 - Send atomic messages, which may be reordered or lost
- Stream sockets: bi-directional pipes
 - With IP, gives you TCP
 - Bytes written on one end read on another
- There are other types
 - Eg. Unix domain sockets
 - Endpoints are filenames

Sockets



System calls for using TCP

<u>Client</u>

<u>Server</u>

socket – create socket bind – assign address, port listen – listen for clients

socket – create socket

- bind* assign address (optional)
- connect connect to listening socket

accept - accept connection

Both can read and write from the connection. Both can call close to end the connection.

Go's Interface is not too different

Server:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

Go's Interface is not too different

Client:

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err :=
bufio.NewReader(conn).ReadString('\n')
// ...
```

Limitations

- Strictly an interface to the transport layer
 - (or lower)
- Reliability
 - if the receiving machine is temporarily not available, will sent data eventually reach it?
 - how is the sender notified if sent data does not arrive at destination machine?
 - how is the sender notified if sent data does not arrive at destination application?

Writing Distributed Programs

- Concerns
 - transparency
 - portability
 - interoperability
- Solutions
 - RPC
 - RMI

Common communication pattern



Writing it by hand...

 eg, if you had to write a, say, password cracker

Then wait for response, etc.

RPC

- A type of client/server communication
- Attempts to make remote procedure calls look like local ones

Go Example

Need some setup in advance of this but...

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args,
&reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B,
reply)
```

RPC Goals

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

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 :)

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 ..

- Must convert to local representation of data
- Machines and network can fail

Local Procedure Calls







Remote Procedure Calls (1)



Remote Procedure Calls (2)



Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
 - <u>Marshals</u> arguments into machine-independent format
 - Sends request to server
 - Waits for response
 - <u>Unmarshals</u> result and returns to caller
- Server stub
 - <u>Unmarshals</u> 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 description language.
 - Lots of these. Some look like C, some look like XML, ... details don't matter much.
 - Use reflection information to do this

- Go-rpc, 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.



Today's Lecture

- Sockets
- RPC
 - Overview
 - Challenges
 - Examples

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
- Reply from srv \rightarrow cli lost
- Server crashes after receiving request
- 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

Real solution: break transparency

- Possible semantics for RPC:
 - Exactly-once
 - Impossible in practice
 - At least once:
 - Only for idempotent operations
 - At most once
 - Zero, don't know, or once
 - Zero or once
 - Transactional semantics

Real solution: break transparency

- <u>At-least-once</u>: Just keep retrying on client side until you get a response.
 - Server just processes requests as normal, doesn't remember anything. Simple!
- <u>At-most-once</u>: 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.

Uncertainty



Idempotent Procedures



Non-Idempotent Procedures



Maintaining History



VI-43

No History



Coping

Fau	Invocation Semantics		
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

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.

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
- Go and Java use the latter.

Example: Sun XDR (RFC 4506)

- External Data Representation for SunRPC
- Types: most of C types
- No tags (except for array lengths)
 - Code needs to know structure of message
- Usage:
 - Create a program description file (.x)
 - Run rpcgen program
 - Include generated .h files, use stub functions
- Very C/C++ oriented
 - Although encoders/decoders exist for other languages

Example

```
typedef struct {
    int comp1;
    double comp2;
    long long comp3[6];
    char *annotation;
} value_t;
```

```
typedef struct {
    value_t element;
    value_t *next;
} list_t;
```

```
char add(int key, value_t value);
char remove(int key, value_t value);
list_t query(int key);
```

A Specification

typedef struct value { int comp1; double comp2; hyper comp3[6]; annotation<255>; string } value t; **typedef struct** list { value t element: **struct list** *next; } list t; program DB { version DBVERS { **bool** add(**int** key, **value** t value) = 1; **bool** remove(**int** key, **value_t** value) = 2; **list t** query(**int** key) = 3; $\} = 1;$ = 0x200000A;

 Rpcgen generates marshalling/unmarshalling code, stub functions, you fill out the actual code

XDR Primitive Types

- Integer
- Unsigned integer
- Boolean
- Hyper integer
- Unsigned hyper integer
- Fixed-length opaque data
- Variable-length opaque data
- String

XDR Structured Types

- Fixed-length array
- Variable-length array
- Discriminated union
- Linked lists

Generated Header File

```
struct value {
    int comp1;
    double comp2;
    int64_t comp3[6];
    char *annotation;
};
```

typedef struct value value;

```
typedef value value_t;
```

```
struct list {
    value_t element;
    struct list *next;
};
```

```
typedef struct list list;
```

typedef list list_t;

```
struct add_1_argument {
    int key;
    value_t value;
};
typedef struct add_1_argument
```

```
add_1_argument;
```

```
struct remove_1_argument {
    int key;
    value_t value;
};
typedef struct remove_1_argument
```

```
remove_1_argument;
```

Placing Calls

```
result_1 = add_1(add_1_key, add_1_value, clnt);
if (result_1 == (bool_t *) NULL) {
      clnt_perror (clnt, "call failed");
result_2 = remove_1(remove_1_key, remove_1_value, clnt);
if (result_2 == (bool_t *) NULL) {
      clnt perror (clnt, "call failed");
}
result_3 = query_1(query_1_key, clnt);
if (result_3 == (list_t *) NULL) {
      clnt_perror (clnt, "call failed");
clnt_destroy (clnt);
```

DCE RPC

- Designed by Apollo and Digital in the 1980s
 both companies later absorbed by HP
- Does everything ONC RPC can do, and more
- Basis for Microsoft RPC

Same Example ...

typedef struct {

double comp1; int comp2; long long comp3; char *annotation; } value_t;

char add(int key, value_t value);
char remove(int key, value_t value);
int query(int key, int number, value_t values[]);

An Interface Specification

```
interface db {
 typedef struct {
   double comp1;
   long comp2;
   hyper comp3;
   [string, ptr]
     ISO LATIN 1
       *annotation;
 } value_t;
 boolean add(
   [in] long key,
   [in] value_t value
 );
```

```
boolean remove(
   [in] long key,
   [in] value_t value
);
```

```
long query(
  [in] long key,
  [in] long number,
  [out, size_is(number)]
    value_t values[ ]
);
```

An Interface Specification (notes continued)

```
interface db {
 typedef struct {
   double comp1;
   long comp2;
   hyper comp3;
   [string, ptr]
     ISO LATIN 1
       *annotation;
 } value_t;
 boolean add(
   [in] long key,
   [in] value_t value
 );
```

```
boolean remove(
   [in] long key,
   [in] value_t value
);
```

```
long query(
   [in] long key,
   [in] long number,
   [out, size_is(number)]
    value_t values[ ]
);
```

Representing an Array

Length	Item 1	Item 2	•••	ltem n

Representing Pointers



Complications



Marshalling Unrestricted Pointers



Referring to Server State



Server

Maintaining Client State on Servers

```
interface trees {
 typedef [context handle] void *tree t;
 void create (
      [in] long value,
      [out] tree t pine
  );
 void insert (
      [in] long value,
      [in, out] tree t pine
  );
```

Go RPC

package server

```
type Args struct {
   A, B int
}
type Quotient struct {
   Quo, Rem int
}
type Arith int
func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}
func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

Server Startup

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

Client Startup

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```
Client Call

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args,
&reply)
if err != nil {
      log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B,
reply)
```

Client Call

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args,
quotient, nil)
replyCall := <-divCall.Done // will be equal to
divCall
// check errors, print, etc.</pre>
```

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();
}
```

RMI: the Server (1)

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)

```
public static void main(String[] args) {
           (System.getSecurityManager() == null) {
        if
            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();
        }
```

}

RMI: the Client

```
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();
```

RMI: the Client's Compute Object

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

- Grpc. Developed at Google
 - Protocol Buffers as an IDL
 - HTTP2 as a transport
 - Many languages (C, C++, Java, Go, Node.js, Python, Ruby, Objective-C, PHP and C#)
- Thrift.
 - Developed at Facebook.
 - Now Apache Open Source. Supports multiple data encodings & transport mechanisms. Even more languages.
- Avro. Also Apache standard. Created as part of Hadoop project. Uses JSON. Not as elaborate as Thrift.