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

From Colours, chapter 4
## System calls for using TCP

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>create socket</td>
</tr>
<tr>
<td>bind*</td>
<td>assign address, port</td>
</tr>
<tr>
<td>listen</td>
<td>listen for clients</td>
</tr>
<tr>
<td>socket</td>
<td>create socket</td>
</tr>
<tr>
<td>bind*</td>
<td>assign address (optional)</td>
</tr>
<tr>
<td>connect</td>
<td>connect to listening socket</td>
</tr>
<tr>
<td>accept</td>
<td>accept connection</td>
</tr>
</tbody>
</table>

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:

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

Client

Hey, do something

working {

Server

Done/Result
Writing it by hand...

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

```c
struct foomsg {
    u_int32_t len;
};

send_foo(char *contents) {
    int msglen = sizeof(struct foomsg) + strlen(contents);
    char buf = malloc(msglen);
    struct foomsg *fm = {struct foomsg *buf;
    fm->len = htonl(strlen(contents));
    memcp(buf + sizeof(struct foomsg),
           contents,
           strlen(contents));
    write(outsock, buf, msglen);
}
```

Then wait for response, etc.
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)

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
The basic theory of operation of RPC is pretty straightforward. But, to understand *remote* procedure calls, let’s first make sure that we understand *local* procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a (shallow) copy of the arguments (other languages, such as C++, provide other argument-passing modes, but copying is all that is provided in C). In the usual implementation, the callee’s copy of the arguments have been placed on the runtime stack by the caller—the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be *out* arguments—changes to their value are reflected back to the caller. This is handled in C indirectly—the actual argument, passed by copying, is a pointer to some value. The callee follows the pointer and modifies the value.
Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee returns data through out arguments by following a pointer into the space of the caller. These techniques simply don’t work in the remote case.
The solution is to use *stub procedures*: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the *client-side stub*. This proxy gathers together all of the arguments (actually, just the in and in-out arguments) and packages them into a message that it sends to the server. The server has a corresponding *server-side stub* that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call—they behave no differently for the remote case.
We will see later how these are generated.
“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.
Today’s Lecture

- Sockets
- RPC
  - Overview
  - Challenges
  - Examples
So, what are the differences between local and distributed, and can we create a proper illusion?
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 → srv lost
- Reply from srv → 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

- **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.
Consider this slide first with the assumption that RPC is layered on UDP. Thus, since the response acts as the acknowledgement, there is uncertainty as to whether the request was handled by the server.

Does this uncertainty go away if RPC is layered on TCP? If you consider the possibility that the TCP connection might be lost, perhaps due to a transient network problem, the answer is clearly no. For example, suppose the TCP connection is lost just after the server receives the request. With no connection, the server cannot send a response, so the client is uncertain about what happened.
A procedure is *idempotent* if the effect of executing it twice in a row is the same as executing it just once. With such procedures, the client may repeatedly send a request until it finally gets a response. If an RPC protocol depends on such retries, it is said to have *at-least-once semantics* — clients are assured that, after all the retries, the remote procedure is executed at least once.
Not everything is idempotent! If we have non-idempotent procedures, then RPC requests should not be blindly retried, but instead should be sent just once. RPC protocols that do this are said to have at-most-once semantics. DCE RPC guarantees at-most-once semantics by default, though a remote procedure may be declared (in its IDL description) to be idempotent, in which case calls are done using at-least-once semantics.
The server might keep track of what operations it has already performed and what the responses were. If it gets a repeat of a previous request, it merely repeats the original response.
If the server crashed and no longer has its history information, it can respond by raising an exception at the client, indicating that it has no knowledge as to whether the operation has taken place. But it guarantees that it hasn’t taken place more than once.
This table is Figure 5.9 of the text. Note the distinction made between “maybe” and “at-most-once” semantics.

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</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>
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
- 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:

// 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:

```go
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.Fatal("failed to serve: %v", err)
    }
}
```
This example (extending through the next five slides) is taken from http://download.oracle.com/javase/tutorial/rmi/overview.html (and the code is copyrighted by Oracle). The idea is that we are designing a “compute server” to which we can send an object representing a computation to be performed. The server performs the computation and sends back the result.

We start with a declaration of the server’s interface. By extending Remote, it becomes the interface to a remote object, meaning that clients with references to the object can invoke its methods remotely. Note that all methods of remote objects must be declared as throwing RemoteException, which occurs when there is some sort of problem, such as a communication error.

The interface provides one method, that takes a parameterized type, Task<T> as an argument, and returns something of type T. Thus effectively the method has two parameters: Task<T> and T.
Here is the declaration of Task<T>. Note that it is not a remote object, but will be passed (by copying) to the Compute object.
Now that we’ve seen the declaration of the interface, we look at the server itself, whose code begins on this slide. The class Compute Engine implements the Compute interface. Its constructor simply calls upon the superclass constructor. Its executeTask method simply invokes the execute method of the supplied Task<T> argument.
The main routine, which is static, after setting up a security manager (which restricts what remote invocations of the object can do) creates an instance of a `ComputeEngine` object and makes it available for clients to access. The first step in this is to set things up so that the object can receive invocations of its method from remote clients. The `exportObject` method of `UnicastRemoteObject` does this: the first argument is the remote interface being offered to clients and the second is the port on which to receive requests (0 means to use the default port).

Note that remote interfaces do not have constructors, thus clients must be given some external means for getting references to remote objects. This is done here by putting the name of the object in the local registry along with the reference to the remote object provided by `exportObject`. Thus clients can contact the registry and get the reference associated with the name.
Here we have code that’s run by clients. As with servers, a security manager must be set up to enforce restrictions on what objects can do that are returned by servers. The client then gets the object reference from the server’s registry (the name of the server is passed to the client code as an argument). It then creates an instance of an object that represents a computation to be done (in this case: compute the value of pi to a given number of digits). This object is passed to the remote object as an argument and the result of the computation is returned.
Lastly we have the skeleton of the object that computes pi. Note that the object must implement `Serializable` so that it can be marshalled and unmarshalled.
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