CSE 451: Operating Systems

Messaging and Remote Procedure Call (RPC)
Messages

• Initially, people hand-coded messages to send requests and responses
  – message is a stream of bytes – “op codes” and operands

• Lots of drawbacks
  – need to worry about message format
  – have to pack and unpack data from messages
  – servers have to decode messages and dispatch to handlers
  – messages are often asynchronous
    • after sending one, what do you do until response comes back?
  – messages aren’t a natural programming model
Client/Server communication

• The prevalent model for structuring distributed computation is the client/server paradigm
  – a server is a program (or collection of programs) that provides a service to other programs
    • e.g., file server, name server, web server, mail server …
    • server/service may span multiple nodes (clusters)
      – often, nodes are called servers too
      – e.g., the web server runs on a Dell server computer
  – a client is a program that uses the service
    • the client first binds to the server
      – locates it, establishes a network connection to it
    • the client then sends requests (with data) to perform actions, and the server sends responses (with data)
      – e.g., web browser sends a “GET” request, server responds with a web page
• TCP/IP is the transport, but what is the higher-level programming model?
Procedure calls

• Procedure calls are a natural way to structure multiple modules inside a single program
  – every language supports procedure calls
  – semantics are well-defined and well-understood
  – programmers are used to them

• “Server” (called procedure) exports an API

• “Client” (calling procedure) calls the server procedure’s API

• Linker binds the two together
Procedure call example

- If the server were just a library, then “Add” would just be a local procedure call.

Client Program:

```c
... 
sum = server->Add(3,4);
... 
```

Server API:

```c
int Add(int x, int y;
```

Server Program:

```c
int Add(int x, int y) {
    return x + y;
}
```
Remote Procedure Call (RPC)

• Traditional *procedure call syntax and semantics* across a network
• The most common means used for remote communication in *client/server systems*
• Used both by operating systems and applications
  – NFS is implemented as a set of RPCs
  – HTTP is essentially RPC
  – DCOM, CORBA, Java RMI, etc., are just RPC systems
• Someday you, too, will likely have to write an application that uses remote communication
  – you’ll likely model your remote communication on RPC
RPC

• Use procedure calls as the model for distributed (remote) communication
  – have servers export a set of procedures that can be called by client programs
    • similar to library API, class definitions, etc.
  – clients do a local procedure call, as though they were directly linked with the server
    • under the covers, the procedure call is converted into a message exchange with the server
      • *largely invisible to the programmer!*
RPC issues

• There are a bunch of hard issues:
  – how do we make the “remote” part of RPC invisible to the programmer?
    • and is that a good idea?
  – what are the semantics of parameter passing?
    • what if we try to pass by reference?
  – how do we bind (locate/connect-to) servers?
  – how do we handle heterogeneity?
    • OS, language, architecture, …
  – how do we make it go fast?
RPC example invocation

Client Program:

```c
...  
sum = server->Add(3,4);  
...  
```

Server Program:

```c
int Add(int x, int y) {
  return x + y;
}
```

client-side stub:

```c
int Add(int x, int y) {
  alloc message buffer;
  mark as "add" call;
  store x,y in buffer;
  send message;
  receive response;
  unpack response;
  return response;
}
```

server-side stub:

```c
Message Add_Stub(Message m) {
  remove x,y from m;
  r = Add(x,y);
  allocate response buffer;
  store r in response;
  return response;
}
```

RPC runtime system:

```c
send message to server;
receive response;
```

RPC runtime system:

```c
receive message m;
response = Add_Stub(m);
send response to client;
```

Topics:
- interface description
- stubs
- stub generation
- parameter marshalling
- binding
- runtime system
- error handling
- performance
- thread pools
RPC model

• A server defines the service interface using an interface definition language (IDL)
  – the IDL specifies the names, parameters, and types for all client-callable server procedures
    • example: ASN.1 in the OSI reference model
    • example: Sun’s XDR (external data representation)
• A “stub compiler” reads the IDL declarations and produces two stub procedures for each server procedure
  – the server programmer implements the service’s procedures and links them with the server-side stubs
  – the client programmer implements the client program and links it with the client-side stubs
  – the stubs manage all of the details of remote communication between client and server using the RPC runtime system
RPC stubs

• A client-side stub is a procedure that looks to the client as if it were a callable server procedure
  – it has the same API as the server’s implementation of the procedure
  – a client-side stub is just called a “stub” in Java RMI
• A server-side stub looks like a caller to the server
  – it looks like a hunk of code that invokes the server procedure
  – a server-side stub is called a “skeleton” or “skel” in Java RMI
• The client program thinks it’s invoking the server
  – but it’s calling into the client-side stub
• The server program thinks it’s called by the client
  – but it’s really called by the server-side stub
• The stubs send messages to each other, via the runtime, to make the RPC happen transparently
RPC marshalling

• Marshalling is the packing of procedure parameters into a message packet
  – the RPC stubs call type-specific procedure to marshal or unmarshal the parameters of an RPC
    • the client stub marshals the parameters into a message
    • the server stub unmarshals the parameters and uses them to invoke the service’s procedure
  – on return:
    • the server stub marshals the return value
    • the client stub unmarshals the return value, and returns them to the client program
RPC binding

• Binding is the process of connecting the client to the server
  – the server, when it starts up, exports its interface
    • identifies itself to a network name server
    • tells RPC runtime that it is alive and ready to accept calls
  – the client, before issuing any calls, imports the server
    • RPC runtime uses the name server to find the location of the server and establish a connection
• The import and export operations are explicit in the server and client programs
  – a slight breakdown in transparency
    • more to come…
RPC transparency

• One goal of RPC is to be as transparent as possible
  – make remote procedure calls look like local procedure calls
  – we’ve seen that binding breaks this transparency

• What else breaks transparency?
  – failures: remote nodes/networks can fail in more ways than with local procedure calls
    • network partition, server crash
    • need extra support to handle failures
    • server can fail independently from client
      – “partial failure”: a big issue in distributed systems
      – if an RPC fails, was it invoked on the server?
  – performance: remote communication is inherently slower than local communication
    • if you’re not aware you’re doing a remote procedure call, your program might slow down an awful lot…
RPC and thread pools

• What happens if two client threads (or client programs) simultaneously invoke the same server procedure using RPC?
  – ideally, two separate threads will run on the server
  – so, the RPC run-time system on the server needs to spawn or dispatch threads into server-side stubs when messages arrive
    • is there a limit on the number of threads?
    • if so, does this change semantics?
    • if not, what if 1,000,000 clients simultaneously RPC into the same server?