CSE 120
Principles of Operating Systems

Fall 2001

Lecture 14: Remote Procedure Call
Geoffrey M. Voelker

Why is RPC Interesting?

- Remote Procedure Call (RPC) is the most common means for remote communication
- It is used both by operating systems and applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are all basically just RPC
- Someday (soon?) you will most likely have to write an application that uses remote communication (or you already have)
  - You will most likely use some form of RPC for that remote communication
  - So it’s good to know how all this RPC stuff works
    » "Debunking the magic"
Clients and Servers

- The prevalent model for structuring distributed computation is the client/server paradigm.
- A server is a program (or collection of programs) that provide a service (file server, name service, etc.):
  - The server may exist on one or more nodes.
  - Often the node is called the server, too, which is confusing.
- A client is a program that uses the service:
  - A client first binds to the server (locates it and establishes a connection to it).
  - A client then sends requests, with data, to perform actions, and the servers sends responses, also with data.

Messages

- Initially with network programming, people hand-coded messages to send requests and responses.
- Hand-coding messages gets tiresome:
  - Need to worry about message formats.
  - Have to pack and unpack data from messages.
  - Servers have to decode and dispatch messages to handlers.
  - Messages are often asynchronous.
- Messages are not a very natural programming model:
  - Could encapsulate messaging into a library.
  - Just invoke library routines to send a message.
  - Which leads us to RPC…
Procedure Calls

- Procedure calls are a more natural way to communicate
  - Every language supports them
  - Semantics are well-defined and understood
  - Natural for programmers to use
- Idea: Have servers export a set of procedures that can be called by client programs
  - Similar to module interfaces, class definitions, etc.
- Clients just do a procedure call as if they were directly linked with the server
  - Under the covers, the procedure call is converted into a message exchange with the server

Remote Procedure Calls

- So, we would like to use procedure call as a model for distributed (remote) communication
- Lots of issues
  - How do we make this invisible to the programmer?
  - What are the semantics of parameter passing?
  - How do we bind (locate, connect to) servers?
  - How do we support heterogeneity (OS, arch, language)?
  - How do we make it perform well?
**RPC Model**

- A server defines the server’s interface using an interface definition language (IDL)
  - The IDL specifies the names, parameters, and types for all client-callable server procedures
- A stub compiler reads the IDL and produces two stub procedures for each server procedure (client and server)
  - The server programmer implements the server 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 are responsible for managing all details of the remote communication between client and server

**RPC Stubs**

- A client-side stub is a procedure that looks to the client as if it were a callable server procedure
- A server-side stub looks to the server as if a client called it
- The client program thinks it is calling the server
  - In fact, it’s calling the client stub
- The server program thinks it is called by the client
  - In fact, it’s called by the server stub
- The stubs send messages to each other to make the RPC happen “transparently”
**RPC Example**

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

Server Interface:
```c
int Add(int x, int y);
```

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

- If the server were just a library, then Add would just be a procedure call

**RPC Example: Call**

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

Server Program:
```c
int Add(int x, int y) {}
```

Client Stub:
```c
int Add(int x, int y) {
    Alloc message buffer;
    Mark as "Add" call;
    Store x, y into buffer;
    Send message;
}
```

Server Stub:
```c
Add_Stub(Message) {
    Remove x, y from buffer
    r = Add(x, y);
}
```

RPC Runtime:
```c
Send message to server;
```

RPC Runtime:
```c
Receive message;
Dispatch, call Add_Stub;
```
RPC Example: Return

```
Client Program:
sum = server->Add(3,4);

Server Program:
int Add(int x, int, y) {} 

Client Stub:
Int Add(int x, int y) {
Create, send message;
Remove r from reply;
return r;
}

Server Stub:
Add_Stub(Message) {
Remove x, y from buffer
r = Add(x, y);
Store r in buffer;
}

RPC Runtime:
Return reply to stub;
RPC Runtime:
Send reply to client;
```

RPC Marshalling

- Marshalling is the packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshal (or unmarshal) the parameters to a call:
  - The client stub marshals the parameters into a message.
  - The server stub unmarshals parameters from the message and uses them to call the server procedure.
- On return:
  - The server stub marshals the return parameters.
  - The client stub unmarshals return parameters 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 its 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 a server and establish a connection
- The import and export operations are explicit in the server and client programs
  - Breakdown of transparency

RPC Transparency

- One goal of RPC is to be as transparent as possible
  - Make remote procedure calls look like local procedure calls
- We have seen that binding breaks transparency
- What else?
  - Failures – remote nodes/networks can fail in more ways than with local procedure calls
    » Need extra support to handle failures well
  - Performance – remote communication is inherently slower than local communication
    » If program is performance-sensitive, could be a problem
RPC Summary

- RPC is the most common model for communication in distributed applications
  - “Cloaked” as DCOM, CORBA, Java RMI, etc.
  - Also used on same node between applications
- RPC is essentially language support for distributed programming
  - What else have we seen use language support?
- RPC relies upon a stub compiler to automatically generate client/server stubs from the IDL server descriptions
  - These stubs do the marshalling/unmarshalling, message sending/receiving/replying