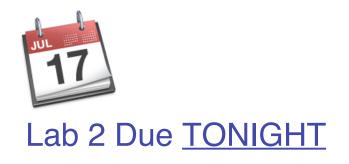
# Lecture 15: Network File System / RPC

CSE 120: Principles of Operating Systems
Alex C. Snoeren





### Network File System



- Simple idea: access disks attached to other computers
  - Share the disk with many other machines as well
- NFS is a protocol for remote access to a file system
  - Does not implement a file system per se
  - Remote access is transparent to applications
- File system, OS, and architecture independent
  - Originally developed by Sun
  - Although Unix-y in flavor, explicit goal to work beyond Unix
- Client/server architecture
  - Local file system requests are forwarded to a remote server
  - Requests are implemented as remote procedure calls (RPCs)

#### Remote Procedure Call



- RPC is 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
- Let's focus on RPC before getting back to NFS
  - Once we have RPC, NFS isn't too hard to build...

#### 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 it 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"





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

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

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

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





```
Client Program:
                                               Server Program:
                                              int Add(int x, int, y) {}
sum = server->Add(3,4);
Client Stub:
                                               Server Stub:
                                              Add_Stub(Message) {
Int Add(int x, int y) {
                                                Remove x, y from buffer
 Alloc message buffer;
 Mark as "Add" call;
                                                r = Add(x, y);
 Store x, y into buffer;
 Send message;
                                              RPC Runtime:
RPC Runtime:
                                               Receive message;
                                              Dispatch, call Add_Stub;
Send message to server;
```

CSE 120 – Lecture 15: Network File System / RPC





```
Client Program:
                                                Server Program:
                                               int Add(int x, int, y) {}
sum = server->Add(3,4);
Client Stub:
                                               Server Stub:
Int Add(int x, int y) {
                                               Add_Stub(Message) {
 Create, send message;
                                                 Remove x, y from buffer
 Remove r from reply;
                                                 r = Add(x, y);
 return r;
                                                 Store r in buffer;
RPC Runtime:
                                                RPC Runtime:
Return reply to stub;
                                               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 it's 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
- Now we're ready to continue talking about NFS

## NFS Binding: Mount



- Before a client can access files on a server, the client must mount the file system on the server
  - The file system is mounted on an empty local directory
  - Same way that local file systems are attached
  - Can depend on OS (e.g., Unix dirs vs Windows drive letters)
- Servers maintain ACLs of clients that can mount their directories
  - When mount succeeds, server returns a file handle
  - Clients use this file handle as a capability to do file operations
- Mounts can be cascaded
  - Can mount a remote file system on a remote file system

#### NFS Protocol



- The NFS protocol defines a set of operations that a server must support
  - Reading and writing files
  - Accessing file attributes
  - Searching for a file within a directory
  - Reading a set of directory links
  - Manipulating links and directories
- These operations are implemented as RPCs
  - Usually by daemon processes (e.g., nfsd)
  - A local operation is transformed into an RPC to a server
  - Server performs operation on its own file system and returns

#### Statelessness



- Note that NFS has no open or close operations
- NFS is stateless
  - An NFS server does not keep track of which clients have mounted its file systems or are accessing its files
  - Each RPC has to specify all information in a request
    - » Operation, FS handle, file id, offset in file, sequence #

#### Robust

- No reconciliation needs to be done on a server crash/reboot
- Clients detect server reboot, continue to issue requests
- Writes must be synchronous to disk, though
  - Clients assume that a write is persistent on return
  - Servers cannot cache writes

### Consistency



- Since NFS is stateless, consistency is tough
  - NFS can be (mostly) consistent, but limits performance
  - NFS assumes that if you want consistency, applications will use higher-level mechanisms to guarantee it
- Writes are supposed to be atomic
  - But performed in multiple RPCs (larger than a network packet)
  - Simultaneous writes from clients can interleave RPCs (bad)
- Server caching
  - Can cache reads, but we saw that it cannot cache writes

## Consistency (2)



- Client caching can lead to consistency problems
  - Caching a write on client A will not be seen by other clients
  - Cached writes by clients A and B are unordered at server
  - Since sharing is rare, though, NFS clients usually do cache
- NFS statelessness is both its key to success and its Achilles' heel
  - NFS is straightforward to implement and reason about
  - But limitations on caching can severely limit performance
    - » Dozens of network file system designs and implementations that perform much better than NFS
  - But note that it is still the most widely used remote file system protocol and implementation

### Summary



- NFS allows access to file systems on other machines
  - Implementation uses RPC to access file server
- RPC enables transparent distribution of applications
  - Also used on same node between applications
- RPC islanguage support for distributed programming
  - Examples include DCOM, CORBA, Java RMI, etc.
  - What else have we seen use language support?
- Stub compiler automatically generates client/server code from the IDL server descriptions
  - These stubs do the marshalling/unmarshalling, message sending/receiving/replying
  - Programmer just implements functions as before

### For Next Time...



- Project 3 on the Web, due Friday, 11/31
- Read Chapter 15
- We'll do TA evaluations in class on Tuesday