Distributed systems
Lecture 2: Network File System and Object-Oriented Middleware

Dr. Robert N. M. Watson

Last time

- Distributed systems are everywhere
  - Challenges including concurrency, delays & failures
  - The importance of transparency
- Simplest distributed systems are client/server
  - Client sends request as message
  - Server gets message, performs operation, and replies
  - Some care required handling retry semantics, timeouts
- One popular client/server model is RPC
  - invoking methods on server over the network
  - Middleware generates stub code which can marshal/unmarshal arguments and replies – e.g. SunRPC/XDR
  - Transparency for the programmer, not just the user
Case Study: NFS

- **NFS = Networked File System** (developed Sun)
  - aimed to provide distributed filing by remote access
- Key design decisions:
  - Distributed file system vs. remote disks
  - Client-server model
  - High degree of transparency
  - Tolerant of node crashes or network failure
- First public version, NFS v2 (1989), did this by:
  - Unix file system semantics (or almost)
  - Integration into kernel (including mount)
  - Simple stateless client/server architecture
- A set of RPC “programs”: mountd, nfsd, lockd, statd, ...

NFS: Client/Server Architecture

- Client uses opaque **file handles** to refer to files
- Server translates these to local inode numbers
- SunRPC with XDR running over UDP (originally)
NFS: Mounting

- NFS RPCs are methods on files; file handle is an RPC argument
- Dedicated mount RPC protocol which:
  - Performs authentication (if any);
  - Negotiates any optional session parameters; and
  - Returns root filehandle

Scoping

- Something interesting is going on with names
  - Each layer is aware only certain scopes
  - Layers translate namespaces when transitioning
  - Contents of names between layers are often opaque
- Pure vs impure names (Needham)
NFS is *Stateless*

- Key NFS design decision to ease fault recovery
  - Obviously, file systems aren’t stateless, so...
- Stateless means:
  - Doesn’t keep any record of current clients
  - Doesn’t keep any record of current open files
- Hence server can crash + reboot, and clients shouldn’t have to do anything (except wait ;-))
- Clients can crash, and server doesn’t need to do anything (no cleanup etc)

Implications of Stateless-ness

- No “open” or “close” operations
  - use `lookup(<pathname>)`
- No implicit arguments
  - e.g. cannot support `read(fd, buf, 2048)`
  - Instead use `read(fh, buf, offset, 2048)`
- Note this also makes operations idempotent
  - This use of SunRPC gives at-least-once semantics
  - Tolerate message duplication in network, RPC retries
- Challenges in providing Unix FS semantics...
Semantic Tricks (and Messes)

• rename() is fundamentally non-idempotent
  – Servers-side “cache” recent RPC replies for replay
• unlink() tricky – what if you discard a file that a client has “open”?
  – Local semantics require files to persist even after last unlink()
  – NFS client translates unlink() to rename(): *silly rename*
  – Only works on same client (not server delete, or another client)
  – NFS file handles contain an *inode generation number* - ESTALE
• Stateless file locking seems impossible
  – Add two other daemons: *rpc.lockd* and *rpc.statd*
  – Server reboot => *rpc.lockd* contacts clients
  – Client reboot => server’s *rpc.statd* tries contact

Performance Problems

• Neither side knows if other is alive or dead
  – All writes must be synchronously committed on server before it returns success
• Very limited client caching...
  – Risk of inconsistent updates if multiple clients have file open for writing at the same time
• These two facts alone meant that NFS v2 had truly *dreadful* performance
NFS Evolution

- NFS v3 (1995): mostly minor enhancements
  - Scalability
    - Remove limits on path- and file-name lengths
    - Allow 64-bit offsets for large files
    - Allow large (>8KB) transfer size negotiation
  - Explicit asynchrony
    - Server can do asynchronous writes (write-back)
    - Client sends explicit commit after some writes
    - Timestamps piggybacked on most server replies allowing clients to manage read cache validity: close-to-open consistency
  - Optimized operations (readdirplus, symlink)
- But had major impact on performance

NFSv3 readdirplus

- NFSv2 behaviour for "ls -l"
  - readdir() triggers NFS_READDIR to request names and handles
  - stat() on each file triggers one NFS_GETATTR RPC
- NFS3_READDIRPLUS returns a names, handles, and attributes
  - Eliminates a vast number of round-trip times
  - Principle: mask network latency by batching synchronous operations
NFS Evolution (2)

• NFS v4 (2003): major rethink  
  – **Single stateful protocol** (including mount, lock)  
  – TCP (or at least reliable transport) only  
  – Explicit **open** and **close** operations  
  – Share reservations  
  – Delegation  
  – Arbitrary compound operations  
  – Many lessons learned from AFS (later in term)  
• Now starting to see deployment…

Improving over SunRPC

• SunRPC (now “ONC RPC”) very successful but  
  – Clunky (manual program, procedure numbers, etc)  
  – Limited type information (even with XDR)  
  – Hard to scale beyond simple client/server  
• One improvement was OSF DCE (early 90’s)  
  – Another project that learned from AFS  
  – DCE = “Distributed Computing Environment”  
  – Larger middleware system including a distributed file system, a directory service, and DCE RPC  
  – Deals with a collection of machines – a **cell** – rather than just with individual clients and servers
DCE RPC versus SunRPC

• Quite similar in many ways
  – Interfaces written in Interface Definition Notation (IDN), and compiled to skeletons and stubs
  – NDR wire format: little-endian by default (woot!)
  – Can operate over various transport protocols

• Better security, and location transparency
  – Services identified by 128-bit “Universally” Unique identifiers (UUIDs), generated by uuidgen
  – Server registers UUID with cell-wide directory service
  – Client contacts directory service to locate server...
    which supports service move, or replication

Object-Oriented Middleware

• Neither SunRPC / DCE RPC good at handling types, exceptions, or polymorphism

• Object-Oriented Middleware (OOM) arose in the early 90s to address this
  – Assume programmer is writing in OO-style
  – Provide illusion of ‘remote object’ which can be manipulated just like a regular (local) object
  – Makes it easier to program (e.g. can pass a dictionary object as a parameter)
CORBA (1989)

• First OOM system was CORBA
  – Common Object Request Broker Architecture
  – specified by the OMG: Object Management Group

• OMA (Object Management Architecture) is the general model of how objects interoperate
  – Objects provide services.
  – Clients makes a request to an object for a service.
  – Client doesn’t need to know where the object is, or anything about how the object is implemented!
  – Object interface must be known (public)

Object Request Broker (ORB)

• The ORB is the core of the architecture
  – Connects clients to object implementations
  – Conceptually spans multiple machines (in practice, ORB software runs on each machine)
Invoking Objects

• Clients obtain an **object reference**
  – Typically via the **naming service or trading service**
  – (Object references can also be saved for use later)
• Interfaces defined by CORBA IDL
• Clients can call remote methods in 2 ways:
  1. **Static Invocation**: using stubs built at compile time (just like with RPC)
  2. **Dynamic Invocation**: actual method call is created on the fly. It is possible for a client to discover new objects at run time and access the object methods

CORBA IDL

• Definition of language-independent remote interfaces
  – **Language mappings** to C++, Java, Smalltalk, ...
  – Translation by IDL compiler
• Type system
  – **basic types**: long (32 bit), long long (64 bit), short, float, char, boolean, octet, any, ...
  – **constructed types**: struct, union, sequence, array, enum
  – **objects** (common super type **Object**)
• Parameter passing
  – **in, out, inout** (= send remote, modify, update)
  – basic & constructed types passed by value
  – objects passed by reference
CORBA Pros and Cons

• CORBA has some unique advantages
  – Industry standard (OMG)
  – Language & OS agnostic: mix and match
  – Richer than simple RPC (e.g. interface repository, implementation repository, DII support, ...)
  – Many additional services (trading & naming, events & notifications, security, transactions, ...)
• However:
  – Really really complicated / ugly / buzzwordy
  – Poor interoperability, at least at first
  – Generally to be avoided unless you need it!

Microsoft DCOM (1996)

• An alternative to CORBA:
  – MS had invested in COM (object-oriented local IPC scheme) so didn’t fancy moving to OMA
• Service Control Manager (SCM) on each machine responsible for object creation, invocation, ...
  – essentially a lightweight ‘ORB’
• Added remote operation using MSRPC:
  – based on DCE RPC, but extended to support objects
  – augmented IDL called MIDL: DCE IDL + objects
  – requests include interface pointer IDs (IPIIDs) to identify object & interface to be invoked
DCOM vs. CORBA

• Both are language neutral, and object-oriented
• DCOM supports **objects with multiple interfaces**
  – but not, like CORBA, multiple inheritance of interfaces
• DCOM handles **distributed garbage collection**:  
  – remote objects are reference counted (via explicit calls)  
  – ping protocol handles abnormal client termination
• DCOM is widely used (e.g. SMB/CIFS, RDP, ...)
• But DCOM is MS proprietary (not standard)...
  – and no support for exceptions (return code based)..
  – and lacks many of CORBAs services (e.g. trading)
• Deprecated today in favor of .NET

Next time

• Java remote method invocation (RMI)
• XML-RPC, SOAAP, etc, etc, etc.
• Clocks and clock skew