Distributed Systems
8L for Part IB

Handout 1

Dr. Steven Hand

Recommended Reading

- “***Distributed Systems: Concepts and Design***”, (5th Ed) Coulouris et al, Addison-Wesley 2012
- “***Distributed Systems: Principles and Paradigms***” (2nd Ed), Tannenbaum et al, Prentice Hall, 2006
- “***Operating Systems, Concurrent and Distributed S/W Design***“, Bacon & Harris, Addison-Wesley 2003
  - or “***Concurrent Systems***”, (2nd Ed), Jean Bacon, Addison-Wesley 1997

What are Distributed Systems?

- A set of discrete computers (“nodes”) which cooperate to perform a computation
  - Operates “as if” it were a single computing system
- Examples include:
  - Compute clusters (e.g. CERN, HPCF)
  - BOINC (aka SETI@Home and friends)
  - Distributed storage systems (e.g. NFS, Dropbox, …)
  - The Web (client/server; CDNs; and back-end too!)
  - Vehicles, factories, buildings (?)

Distributed Systems: Advantages

- **Scale and performance**
  - Cheaper to buy 100 PCs than a supercomputer…
  - … and easier to incrementally scale up too!
- **Sharing and Communication**
  - Allow access to shared resources (e.g. a printer) and information (e.g. distributed FS or DBMS)
  - Enable explicit communication between machines (e.g. EDI, CDNs) or people (e.g. email, twitter)
- **Reliability**
  - Can hopefully continue to operate even if some parts of the system are inaccessible, or simply crash

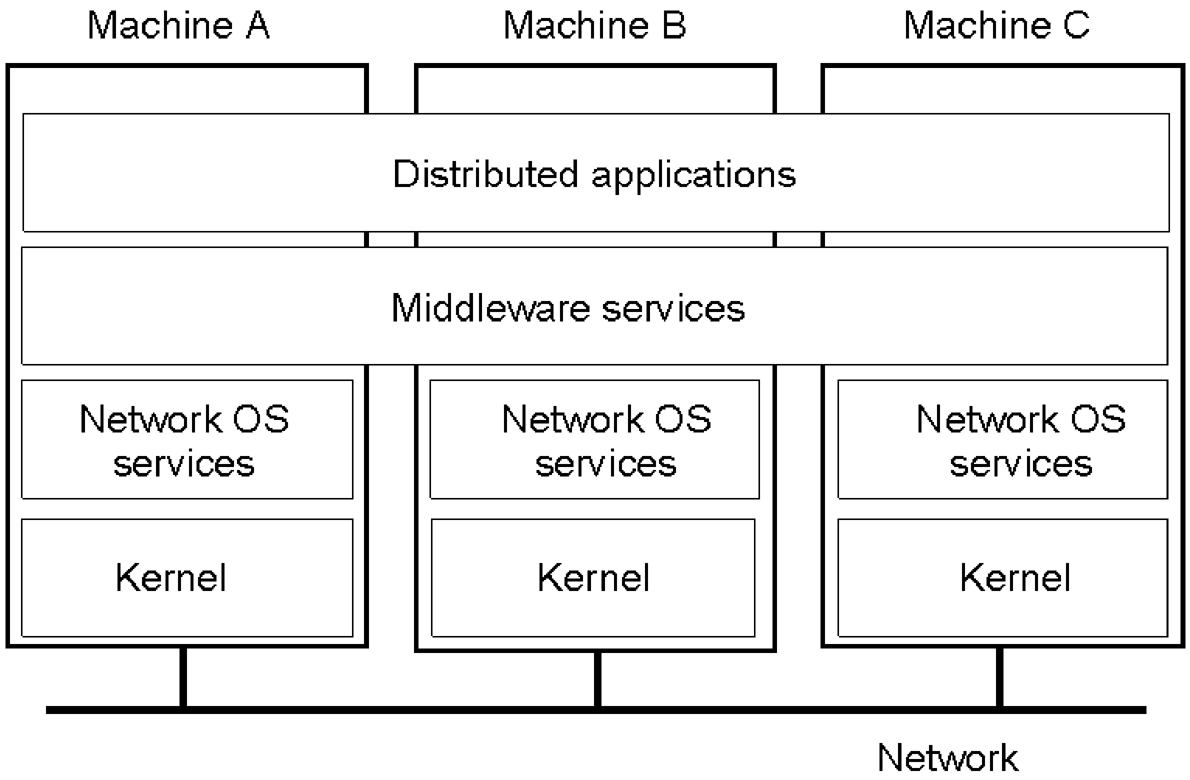
Distributed Systems: Challenges

- **Distributed Systems are *Concurrent Systems***
  - Need to coordinate independent execution at each node (c/f first part of course)
- **Failure of any components (nodes, network)**
  - At any time, for any reason
- **Network delays**
  - Can't distinguish congestion from crash/partition
- **No global time**
  - Tricky to coordinate, or even agree on ordering!

Transparency & Middleware

- Recall a distributed system should appear “as if” it were executing on a single computer
- We often call this **transparency**:
  – User is unaware of multiple machines
  – Programmer is unaware of multiple machines
- How “unaware” can vary quite a bit
  – e.g. web user probably aware that there’s network communication ... but not the number or location of the various machines involved
  – e.g. programmer may explicitly code communication, or may have layers of abstraction: **middleware**

The Role of Middleware



- Note that the middleware layer extends over multiple machines

Types of Transparency

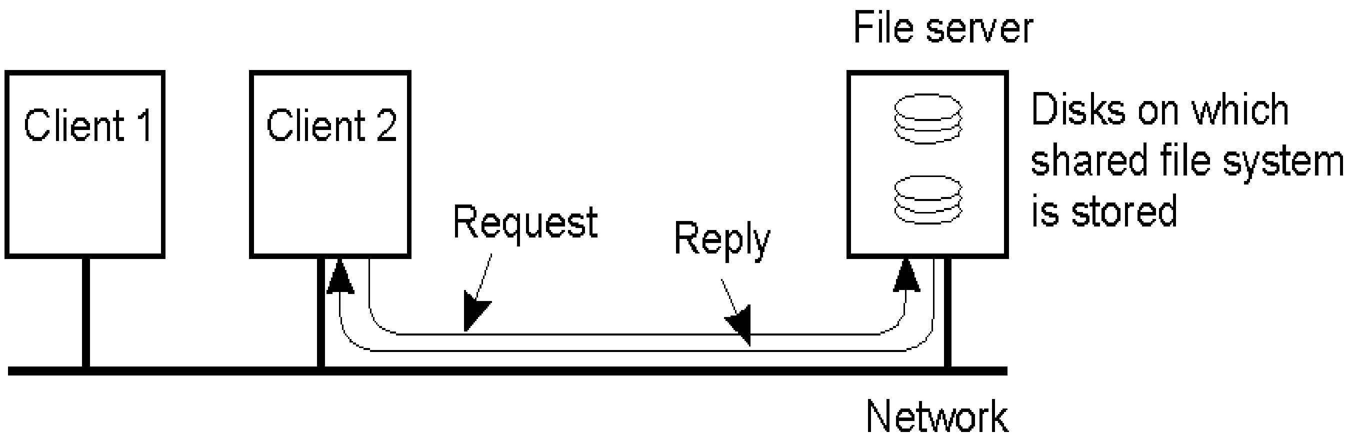
Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be provided by multiple cooperating systems
Concurrency	Hide that a resource may be simultaneously shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

In this Course

- We will look at techniques, protocols & algorithms used in distributed systems
  - in many cases, these will be provided for you by a middleware software suite
  - but knowing how things work will still be useful!
- Assume OS & networking support
  - processes, threads, synchronization
  - basic communication via messages
  - (will see later how assumptions about messages will influence the systems we [can] build)
- Let’s start with a simple client-server systems

Client-Server Model

- 1970s: development of LANs
- 1980s: standard deployment involves small number of **servers**, plus many **workstations**
  - Servers: always-on, powerful machines
  - Workstations: personal computers
- Workstations request 'service' from servers over the network, e.g. access to a shared file-system:



Request-Reply Protocols

- Basic scheme:
  - Client issues a request message
  - Server performs request, and sends reply
- Simplest version is **synchronous**:
  - client blocks awaiting reply
- Example: HTTP 1.0
  - Client (browser) sends “GET /index.html”
  - Web server fetches file and returns it
  - Browser displays HTML web page

Handling Errors & Failures

- **Errors** are **application-level** things => easy ;-)
  - E.g. client requests non-existent web page
  - Need special reply (e.g. “404 Not Found”)
- **Failures** are **system-level** things, e.g.:
  - lost message, client/server crash, network down,…
- To handle failure, client must timeout if it doesn’t receive a reply within a certain time T
  - On timeout, client can **retry** request
  - (Q: what should we set T to?)

Retry Semantics

- Client could timeout because:
  1. Request was lost
  2. Request was sent, but server crashed on receipt
  3. Request was sent & received, and server performed operation (or some of it?), but crashed before replying
  4. Request was sent & received, and server performed operation correctly, and sent reply … which was then lost
  5. As #4, but reply has just been delayed for longer than T
- For read-only stateless requests (like HTTP GET), can retry in all cases, but what if request was an order with Amazon?
  – In case #1, we probably want to re-order… and in case #5 we want to wait for a little bit longer, and otherwise we … erm?
- Worse: we don’t know what case it actually was!

Ideal Semantics

- What we want is **exactly-once** semantics:
  - Our request occurs once no matter how many times we retry (or if the network duplicates our messages)
- E.g. add a unique ID to every request
  - Server remembers IDs, and associated responses
  - If sees a duplicate, just returns old response
  - Client ignores duplicate responses
- Pretty tricky to ensure exactly-once in practice
  - e.g. if server explodes ;-)

Practical Semantics

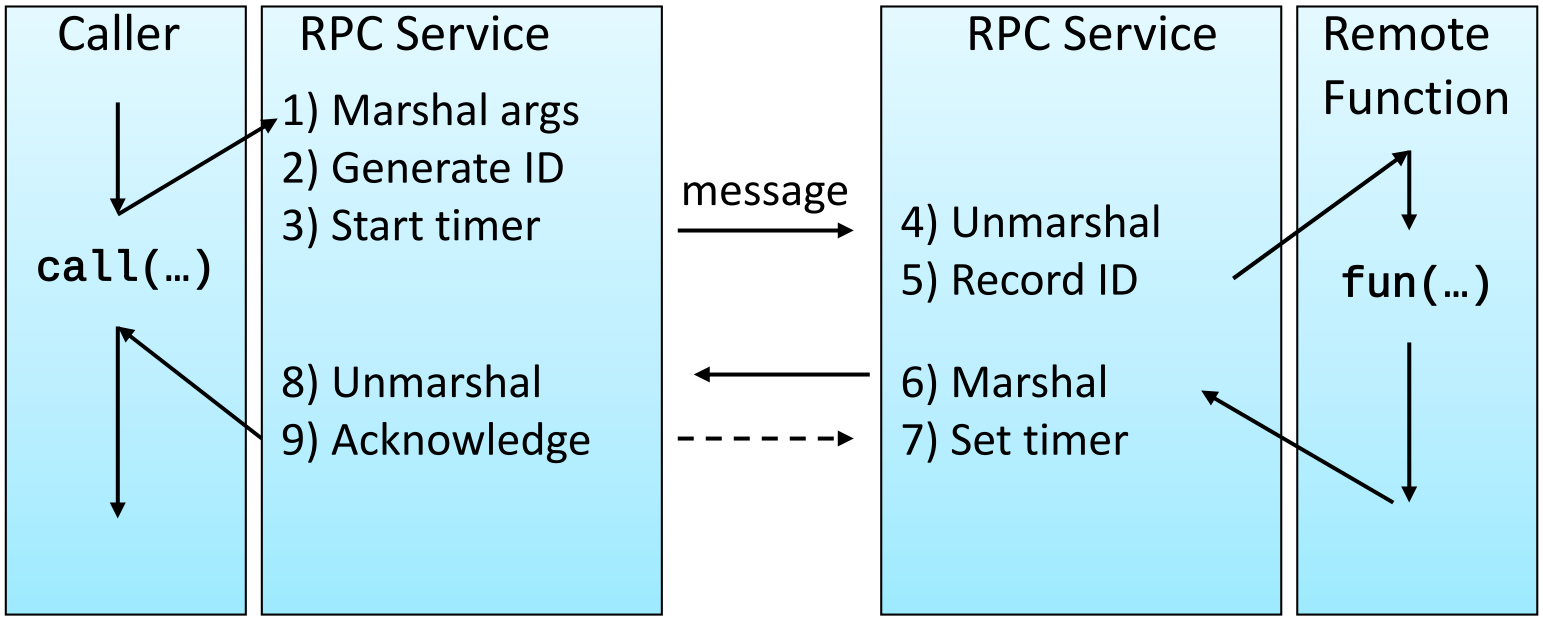
- In practice, protocols guarantee one of the below
- **All-or-nothing** (atomic) semantics
  - Use scheme on previous page, with persistent log
  - (essentially same idea as transaction processing).
- **At-most-once** semantics
  - Request carried out once, or not at all, or don't know
  - e.g. send a single request, and give up if we timeout
- **At-least-once** semantics
  - Retry if we timeout, & risk operation occurring again
  - Ok if the operation is read-only, or **idempotent**

Remote Procedure Call (RPC)

- Request/response protocols are useful – and widely used – but rather clunky to use
  - e.g. need to define the set of requests, including how they are represented in network messages
- A nicer abstraction is **remote procedure call**
  - Programmer simply invokes a procedure…
  - …but it executes on a remote machine (the server)
  - RPC subsystem handles message formats, sending & receiving, handling timeouts, etc
- Aim is to make distribution (mostly) transparent
  - certain failure cases wouldn't happen locally

Marshalling Arguments

- RPC is integrated with the programming language
  - Some additional magic to specify things are remote
- RPC layer **marshals** parameters to the call, as well as any return value(s), e.g.



IDLs and Stubs

- To marshal, the RPC layer needs to know:
  - how many arguments the procedure has,
  - how many results are expected, and
  - the types of all of the above
- The programmer must specify this by describing things in an **interface definition language (IDL)**
  - In higher-level languages, this may already be included as standard (e.g. C#, Java)
  - In others (e.g. C), IDL is part of the middleware
- The RPC layer can then automatically generate **stubs**
  - Small pieces of code at client and server (see previous)

Example: SunRPC

- Developed mid 80’s for Sun Unix systems
- Simple request/response protocol:
  - Server registers one or more “programs” (services)
  - Client issues requests to invoke specific procedures within a specific service
- Messages can be sent over any transport protocol (most commonly UDP/IP)
  - requests have a unique transaction id which can be used to detect & handle retransmissions

XDR: External Data Representation

- SunRPC used **XDR** for describing interfaces:

```
// file: test.x
program test {
  version testver {
    int get(getargs) = 1;  // procedure number
    int put(putargs) = 2;  // procedure number
  } = 1;                   // version number
} = 0x12345678;            // program number
```

- **rpcgen** generates [un]marshaling code, stubs
  - Single arguments… but recursively convert values
  - Some support for following pointers too
- Data on the wire always in big-endian format (oops!)

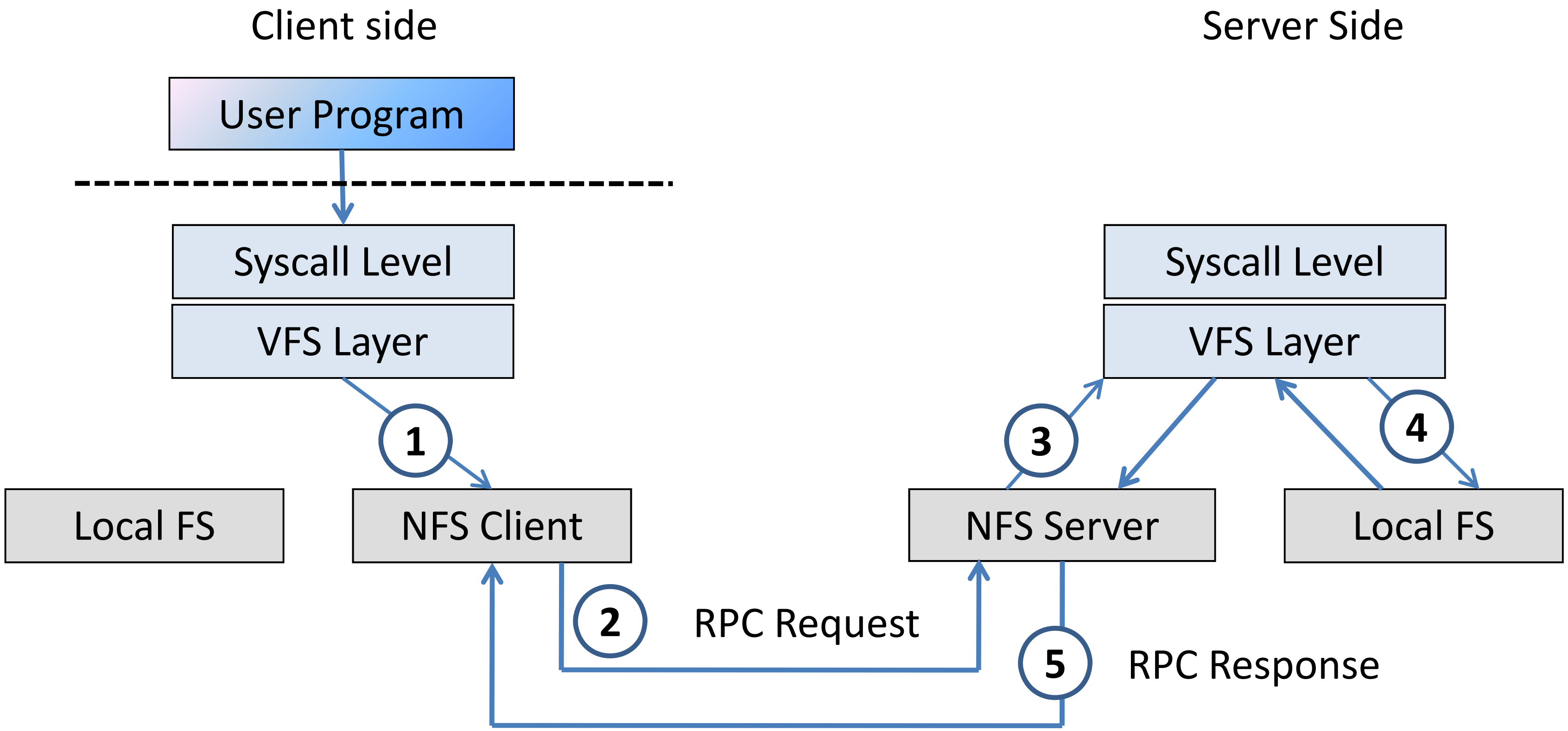
Using SunRPC

1. Write XDR, and use rpcgen to generate skeleton code
2. Fill in blanks (i.e. write actual moving parts for server, and for client(s)), and compile code.
3. Run server program & register with **portmapper**
   – holds mappings from { prog#, ver#, proto } -> port
   – (on linux, try "/usr/sbin/rpcinfo –p")
4. Server process will then listen(), awaiting clients
5. When a client starts, client stub calls clnt_create
   – Sends { prog#, ver#, proto } to portmapper on server, and gets reply with appropriate port number to use
   – Client now invokes remote procedures as needed

Case Study: NFS

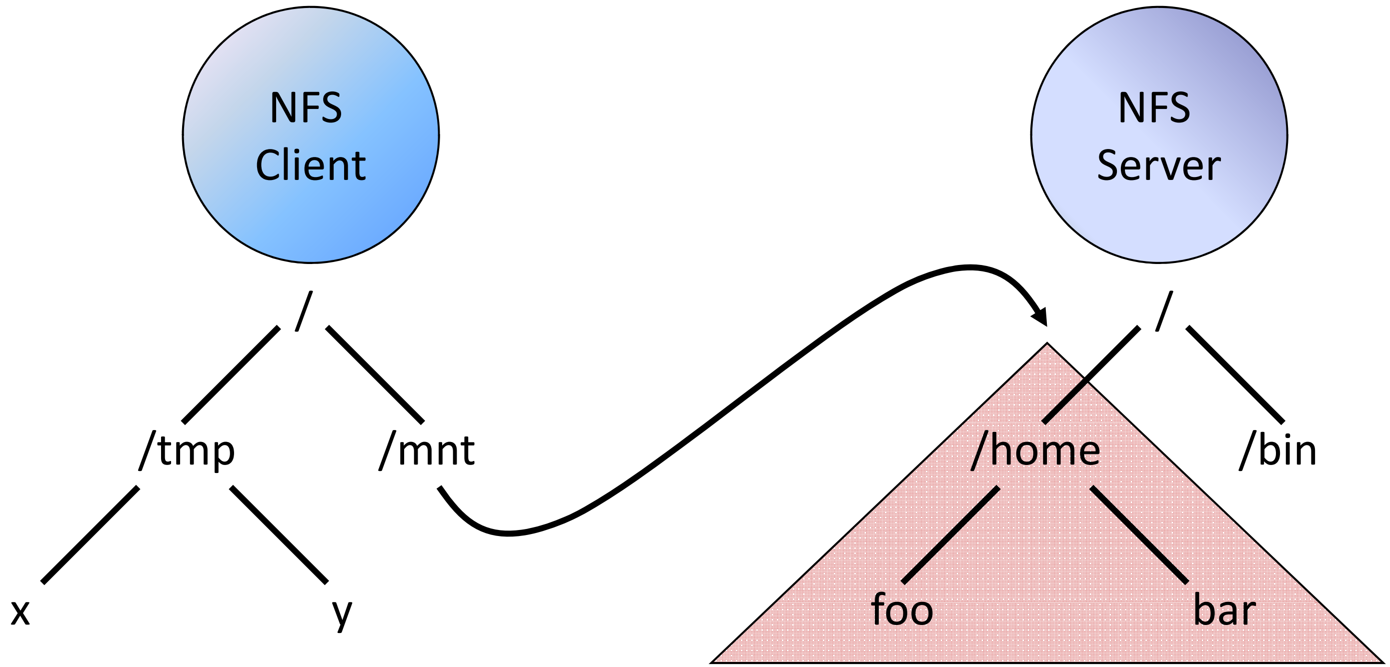
- **NFS = Networked File System** (developed Sun)
  - aimed to provide distributed filing by remote access
- Key design decisions:
  - High degree of transparency
  - Tolerant of node crashes or network failure
- First public version, NFS v2 (1989), did this by:
  - Unix file system semantics (or almost)
  - Integration into kernel (including mount)
  - Simple stateless client/server architecture

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



- Dedicated mount RPC protocol which:
  - Performs authentication (if any);
  - Negotiates any optional session parameters; and
  - Returns root filehandle

NFS is *Stateless*

- Key NFS design decision to make fault recovery easier
- Stateless means:
  - Doesn’t keep any record of current clients
  - Doesn’t keep any record of current file accesses
- 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**
  - Can tolerate message duplication in network / RPC
- Challenges in providing Unix FS semantics…

Semantic Tricks

- File deletion tricky – what if you discard pages of a file that a client has "open"?
  - NFS changes an unlink() to a rename()
  - Only works for same client (not local delete, or concurrent clients – "stale filehandle")
- 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
  - Optimized operations (**readdirplus**, **symlink**)
- But had *major* impact on performance

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
- Actual success yet to be seen…

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)
  - 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

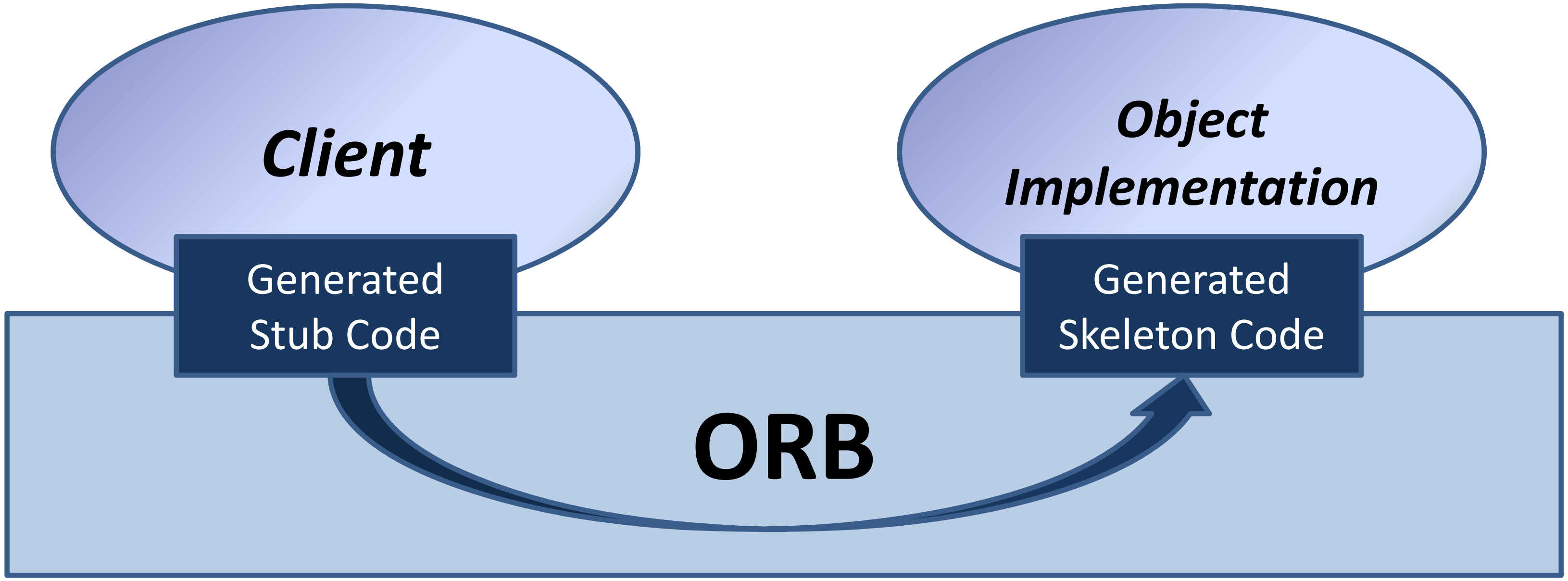
- Neither SunRPC / DCE RPC good at handling types, exceptions, or polymorphism
- Object-Oriented Middleware (OOM) arose in the early 90s to address this
  - Assume programmer is writing in OO-style
  - Provide illusion of ‘remote object’ which can be manipulated just like a regular (local) object
  - 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 (IPIDs) 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

Java RMI

- 1995: Sun extended Java to allow RMI
  - RMI = **Remote Method Invocation**
- Essentially an OOM scheme for Java with clients, servers and an **object registry**
  - object registry maps from names to objects
  - supports **bind()/rebind()**, **lookup()**, **unbind()**, **list()**
- RMI was designed for Java only
  - no goal of OS or language interoperability
  - hence cleaner design and tighter language integration

RMI: New Classes

- **remote class**:
  - one whose instances can be used remotely
  - within home address space, a regular object
  - within foreign address spaces, referenced indirectly via an **object handle**
- **serializable class**: [nothing to do with transactions!]
  - object that can be marshalled/unmarshalled
  - if a serializable object is passed as a parameter or return value of a remote method invocation, the value will be copied from one address space to another
  - (for remote objects, only the object handle is copied)

RMI: New Classes

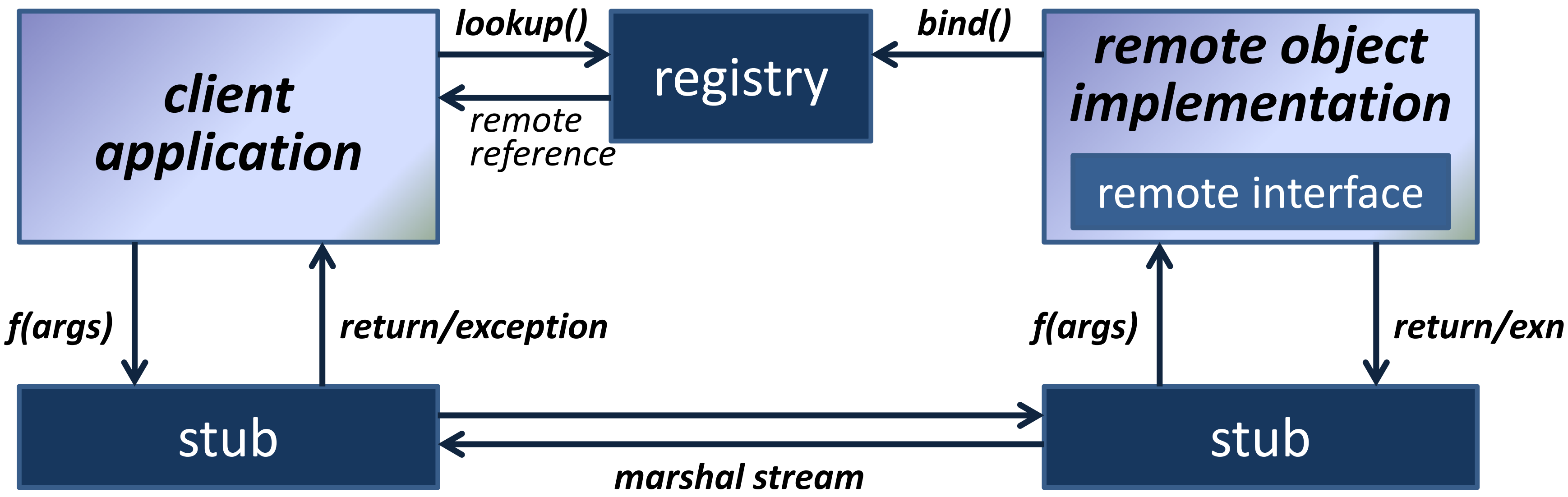
- **remote class**:
  - one whose instances can be used remotely
  - within home address space, a regular object
  - within foreign address spaces, referenced indirectly via an **object handle**

needed for remote objects

- **serializable class**:
  - object that can be marshalled/unmarshalled
  - if a serializable object is passed as a parameter or return value of a remote method invocation, the value will be copied from one address space to another
  - (for remote objects, only the object handle is copied)

needed for parameters

RMI: The Big Picture



- Registry can be on server... or one per distributed system
  - client and server can find it via the **LocateRegistry** class
- Objects being serialized are annotated with a URL for the class
  - unless they implement **Remote** => replaced with a remote reference

Distributed Garbage Collection

- With RMI, can have local & remote object references scattered around a set of machines
- Build distributed GC by leveraging local GC:
  - When a server exports object O, it creates a skeleton S[O]
  - When a client obtains a remote reference to O, it creates a proxy object P[O], and remotely invokes **dirty(O)**
  - Local GC will track the liveness of P[O]; when it is locally unreachable, client remotely invokes **clean(O)**
  - If server notices no remote references, can free S[O]
  - If S[O] was last reference to O, then it too can be freed
- Like DCOM, server removes a reference if it doesn't hear from that client for a while (default 10 mins)

OOM: Summary

- OOM enhances RPC with objects
  - types, interfaces, exceptions, …
- Seen CORBA, DCOM and Java RMI
  - All plausible, and all still used today
  - CORBA most general (language and OS agnostic), but also the most complex: design by committee
  - DCOM is MS only, & being phased out for .NET
  - Java RMI decent starting point for simple distributed systems… but lacks many features
  - (EJB is a modern CORBA/RMI/<stuff> megalith)

XML-RPC

- Systems seen so far all developed by large industry, and work fine in the local area…
  - But don't (or didn't) do well through firewalls ;-)
- In 1998, Dave Winer developed XML-RPC
  - Use XML to encode method invocations (method names, parameters, etc)
  - Use HTTP POST to invoke; response contains the result, also encoded in XML
  - Looks like a regular web session, and so works fine with firewalls, NAT boxes, transparent proxies, …

XML-RPC Example

XML-RPC Request

```
<?xml version="1.0"?>
<methodCall>
<methodName>util.InttoString</methodName>
  <params>
    <param>
        <value><i4>55</i4></value>
    </param>
  </params>
</methodCall>
```

XML-RPC Response

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
       <value><string>Fifty Five</string></value>
    </param>
  </params>
</methodResponse>
```

- Client side names method (as a string), and lists parameters, tagged with simple types
- Server receives message (via HTTP), decodes, performs operation, and replies with similar XML
- Inefficient & weakly typed… but simple, language agnostic, extensible, and eminently practical!

SOAP & Web Services

- XML-RPC was a victim of its own success
- WWW consortium decided to embrace it, extend it, and generally complify it up
  - SOAP (**Simple Object Access Protocol**) is basically XML-RPC, but with more XML bits
  - Support for namespaces, user-defined types, multi-hop messaging, recipient specification, …
  - Also allows transport over SMTP (!), TCP & UDP
- SOAP is part of the **Web Services** world
  - As complex as CORBA, but with more XML ;-)

Moving away from RPC

- SOAP 1.2 defined in 2003
  - Less focus on RPC, and more on moving XML messages from A to B (perhaps via C & D)
- One major problem with all RPC schemes is that they were synchronous:
  - Client is blocked until server replies
  - Poor responsiveness, particularly in wide area
- 2006 saw introduction of AJAX
  - **A**synchronous **Ja**vascript with **X**ML
  - Chief benefit: can update web page without reloading
- Examples: Google Maps, Gmail, Google Docs, …

REST

- AJAX still does RPC (just asynchronously)
- Is a procedure call / method invocation really the best way to build distributed systems?
- **Representational State Transfer** (REST) is an alternative 'paradigm' (or a throwback?)
  - Resources have a name: URL or URI
  - Manipulate them via PUT (insert), GET (select), POST (updated) and DELETE (delete)
  - Send state along with operations
- Very widely used today (Amazon, Flickr, Twitter)

Client-Server Interaction: Summary

- Server handles requests from client
  - Simple request/response protocols (like HTTP) useful, but lack language integration
  - RPC schemes (SunRPC, DCE RPC) address this
  - OOM schemes (CORBA, DCOM, RMI) extend RPC to understand objects, types, interfaces, exns, …
- Recent WWW developments move away from traditional RPC/RMI:
  - Avoid explicit IDLs since can slow evolution
  - Enable asynchrony, or return to request/response