Distributed systems

Lecture 10: Case study: the Network File System (NFS)

Michaelmas 2018

Dr Richard Mortier and
Dr Anil Madhavapeddy

(With thanks to Dr Robert N. M. Watson
and Dr Steven Hand)

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 model is **Remote Procedure Call (RPC)**
  - Client calls **functions** on the server via network
  - **Middleware** generates stub code which can **marshal** / **unmarshal** arguments/return values – e.g. SunRPC/XDR
  - **Transparency** for the programmer, not just the user

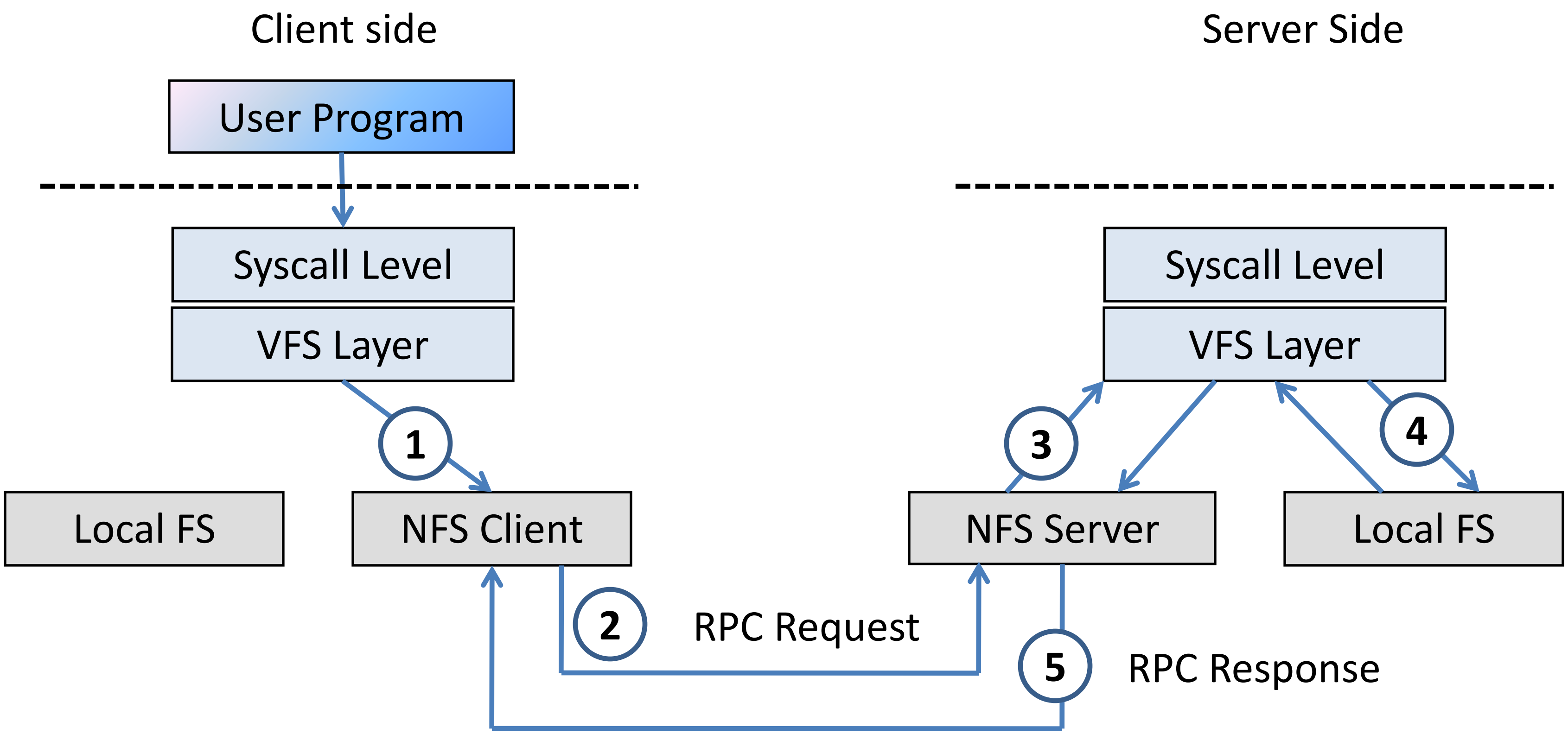
First case study: NFS

- **NFS** = **Networked File System** (developed by Sun)
  - Aimed to provide distributed filing by remote access
- Key design decisions:
  - **Distributed filesystem** vs. **remote disks**
  - Client-server model
  - High degree of transparency
  - Tolerant of node crashes or network failure

Transparency for users and applications, but also **NFS programmers**: hence SunRPC

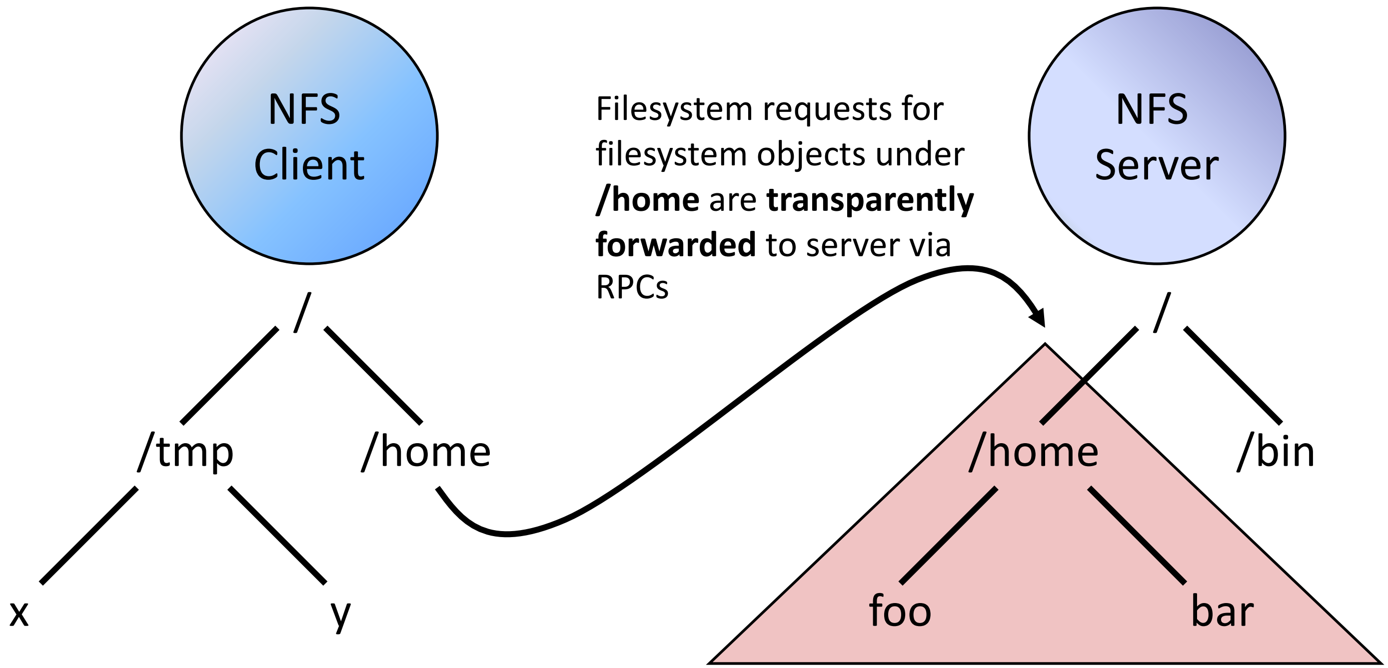
- First public version, NFSv2 (1989), did this via:
  - Unix filesystem semantics (or almost)
  - Integration into kernel (including mount)
  - Simple stateless client/server architecture
- A set of RPC "programs": mountd, nfsd, lockd, statd, ...

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)

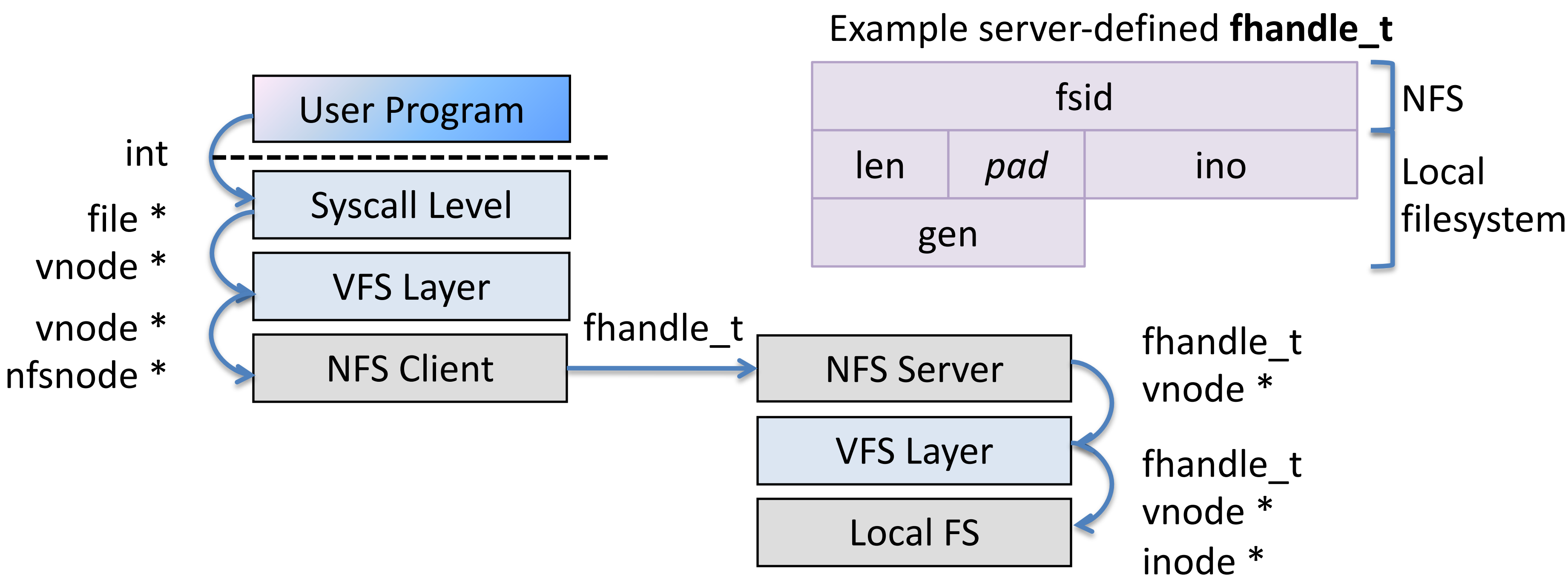
NFS: mounting remote filesystems



- NFS RPCs are methods on files identified by **file handle(s)**
- Bootstrap via dedicated **mount** RPC 'program' that:
  - Performs authentication (if any);
  - Negotiates any optional session parameters; and
  - Returns **root file handle**

NFS file handles and scoping

- Arguments at each layer are with specific **scopes**
  - Layers translate between namespaces for **encapsulation**
  - Contents of names between layers often **opaque**



- **Pure names** expose no visible semantics (e.g., NFS handle)
- **Impure names** have exposed semantics (e.g., file paths)

NFS is stateless

- Key NFS design decision to ease fault recovery
  - Obviously, filesystems aren't stateless, so…
- **Stateless** means the protocol doesn't require:
  - Keeping any record of **current clients**
  - Keeping any record of **current open files**
- **Server can crash + reboot**, and clients do not have to do anything (except wait!)
- **Clients can crash**, and servers do not need to do anything (no cleanup etc)

Implications of stateless-ness

- No "open" or "close" operations
  - **`fh = lookup(<directory fh>, <filename>)`**
  - All file operations are via per-file handles
- No implied state linking multiple RPCs; e.g.,
  - UNIX file descriptor has "current offset" for I/O:

    `read(fd, buf, 2048)`
  - NFS file handle has no offset; operations are explicit:

    `read(fh, buf, offset, 2048)`
- This makes many 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(<old filename>, <new filename>)`**
  - Fundamentally non-idempotent
  - Strong expectation of atomicity
  - Server-side, "cache" recent RPC replies for replay
- **`unlink(<old filename>)`**
  - UNIX requires open files to persist after **`unlink()`**
  - What if the server removes a file that is open on a client?
  - **Silly rename**: clients translate **`unlink()`** to **`rename()`**
  - Only within client (not server delete, nor for other clients)
  - Other clients will have a **stale** file handle: **`ESTALE`**
- Stateless file **locking** seems impossible
  - Problem avoided (?): separate RPC protocols

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

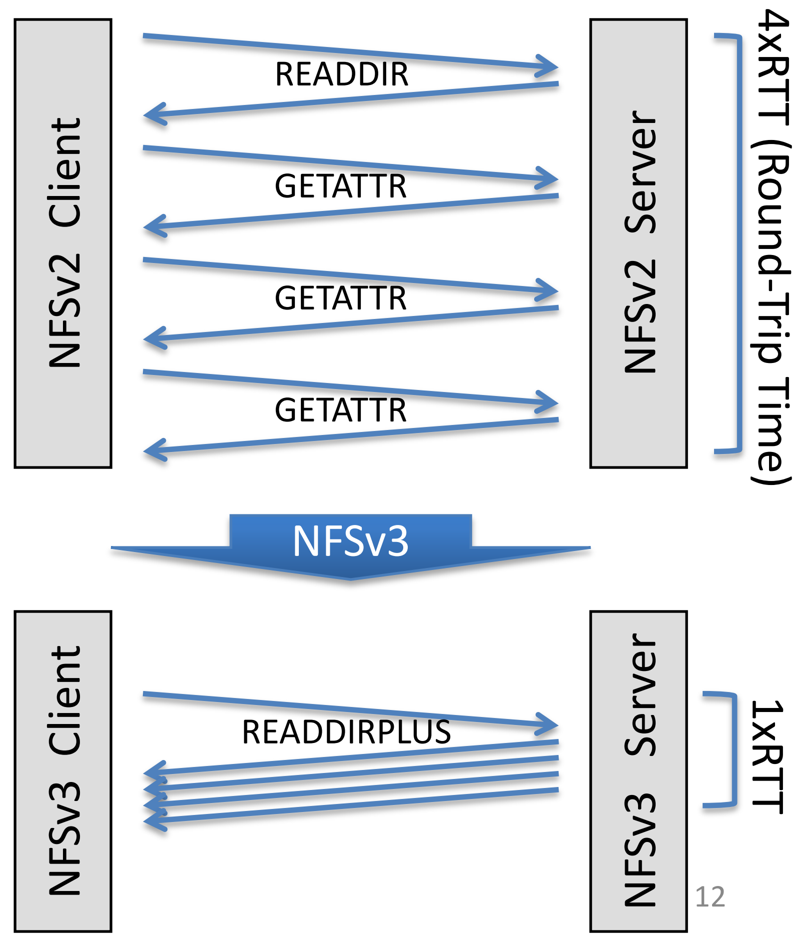
NFSv3 (1995)

- Mostly minor protocol 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
    - File timestamps piggybacked on server replies allow clients to manage cache: **close-to-open consistency**
  - Optimized RPCs (**readdirplus, symlink**)
- But had ***major*** impact on performance

NFSv3 readdirplus

```
drwxr-xr-x  55 al565    al565    12288 Feb  8 15:47 al565/
drwxr-xr-x 115 am21     am21     49152 Feb 10 18:19 am21/
drwxr-xr-x 214 atm26    atm26    36864 Feb  1 17:09 atm26/
```

- 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**



Distributed filesystem consistency

- Can a **distributed application** expect data **written on client A** to be **visible to client B**?
  - After **`write()`** on **A**, will a **`read()`** on **B** see it?
  - What if a process on **A** writes to a file, and then sends a message to a process on **B** to read the file?
- In NFSv3, **no!**
  - **A** may have freshly written data in its cache that it has not yet sent to the server via a write RPC
  - The server will return stale data to **B**'s read RPC

  Or:
  - **B** may return stale data in its cache from a prior read RPC
- This problem is known as **inconsistency**:
  - Clients may see **different versions** of the **same object**

NFS close-to-open consistency (1)

- Guaranteeing **global visibility** for every **`write()`** required synchronous RPCs and prevented caching
- NFSv3 implements **close-to-open consistency**, which reduces synchronous RPCs and permits caching
  1. For each file it stores, the server maintains a **timestamp** of the last write performed
  2. When a file is **opened**, the client receives the timestamp; if the timestamp has changed since data was cached, the client **invalidates** its read cache, forcing fresh read RPCs
  3. While the file is **open**, data reads/writes for the file can be cached on the client, and write RPCs can be deferred
  4. When the file is **closed**, pending writes must be sent to the server (and ack'd) before **`close()`** can return

NFS close-to-open consistency (2)

- We now have a **consistency model** that programmers can use to reason about when writes will be visible in NFS:
  - If a program on host **A** needs writes to a file to be visible to a program on host **B**, it must **`close()`** the file
  - If a program on host **B** needs reads from a file to include those writes, it must **`open()`** it **after** the corresponding **`close()`**
- This works quite well for some applications
  - E.g., distributed builds: inputs/outputs are whole files
  - E.g., UNIX maildir format (each email in its own file)
- It works very badly for others
  - E.g., long-running databases that modify records within a file
  - E.g., UNIX mbox format (all emails in one large file)
- Applications using NFS to share data **must be designed for these semantics**, or they will behave very badly!

NFSv4 (2003)

- Time for a 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 seeing widespread 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 90s)
  - 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!
  - 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

Summary + next time

- NFS as an RPC, distributed-filesystem case study
  - Retry semantics vs. RPC semantics
  - Scoping, pure vs. impure names
  - Close-to-open consistency
  - Batching to mask network latency
- DCE RPC

- Object-Oriented Middleware (OOM)
- Java remote method invocation (RMI)
- XML-RPC, SOAP, etc, etc, etc.
- Starting to talk about distributed time