Distributed Systems

Summer Term 2020

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: March 12, 2020

Distributed Systems

Summer Term 2020

9 Distributed File Systems

Contents

- General
- Case study: NFS

Literature

- Tanenbaum, van Steen: Ch. 10
- Colouris, Dollimore, Kindberg: Ch. 8

[Coulouris, 8.1-8.3]

9.1 General

- Objective: support the sharing of information (files) in an **intra**net
  - in the Internet: WWW
- Allows applications to access remote files in the same way as local files
  - similar (or even better) performance and reliability
- Allows operation of diskless nodes
- Examples:
  - NFS (standard in the UNIX area)
  - AFS (goal: scalability), CIFS (Windows), CODA, xFS, ...

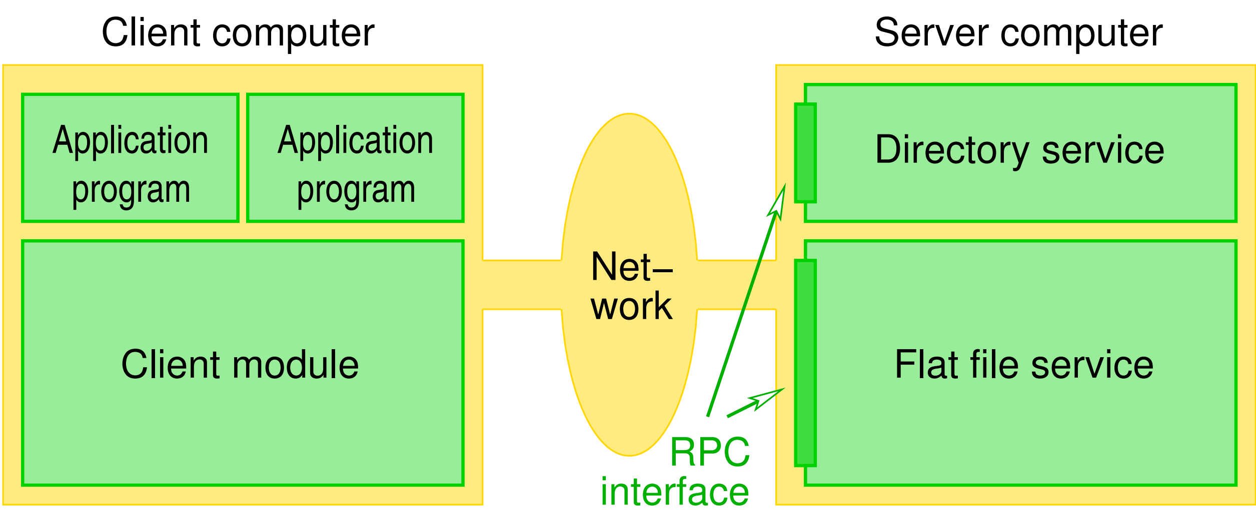
9.1 General ...

Requirements

- Transparency: access, location, mobility, performance and scaling transparency
- Concurrent file updates (e.g., locks)
- File replication (often: local caching)
- Heterogeneity of hardware and operating system
- Fault tolerance (especially in case of server failure)
  - often: at-least-once semantics + idempotent operations
  - advantageous: stateless server (easy reboot)
- Consistency (☞ 8)
- Security (access control, authentication, encryption)
- Efficiency

9.1 General ...

Model Architecture of a Distributed File System



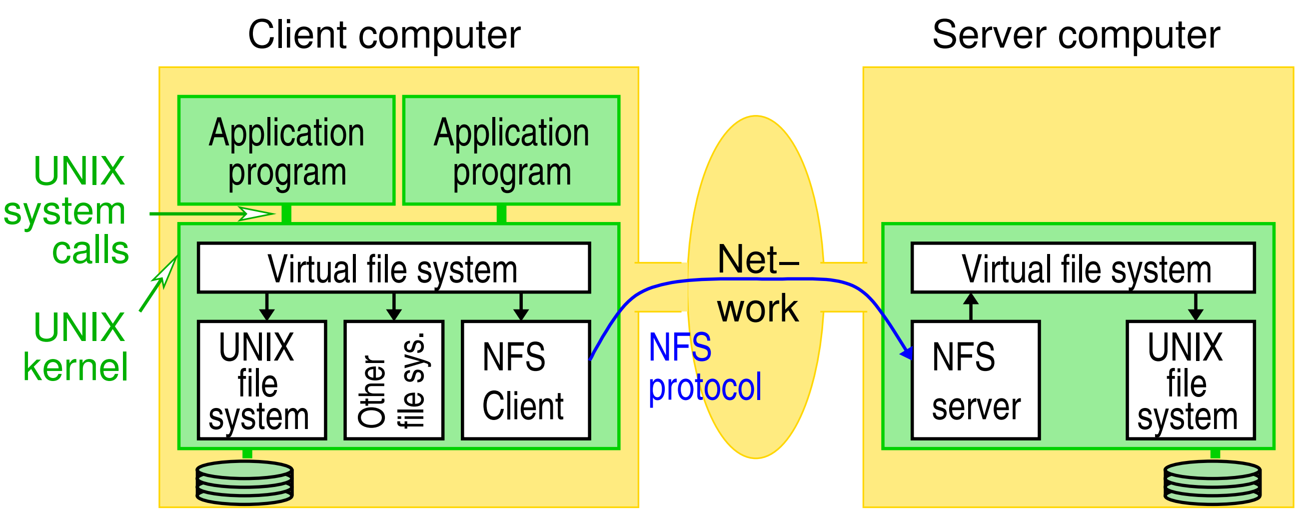
- Tasks of the client module:
  - emulation of the file interface of the local OS
  - if necessary caching of files or file sections

Model Architecture of a Distributed File System ...

- Flat file service:
  - provides idempotent access operations to files
    - e.g., *read*, *write*, *create*, *remove*, *getAttributes*, *setAttributes*
    - no *open* / *close*, no implicit file pointer
  - files are identified by UFIDs (Unique File IDs)
    - (long) integer IDs, can serve as capabilities
- Directory service:
  - maps file or path names to UFIDs
    - if necessary first authenticates the client and verifies its access rights
  - services for creating, deleting and modifying directories

9.2 Case Study: NFS

- Introduced in 1984 by Sun
- Open, OS independent protocol
- Architecture:



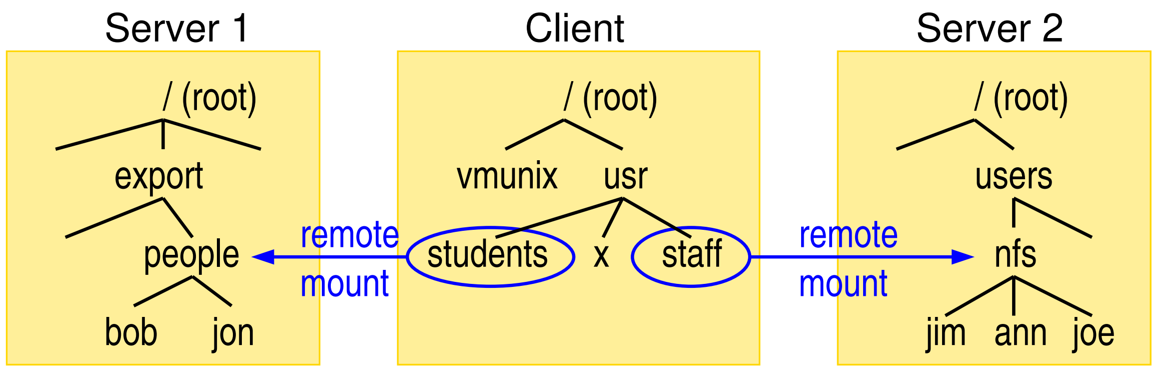
Access Control and Authentication

- NFS server is stateless (up to and including NFS3)
- UFID (file handle): essentially just the file system ID and i-node
  - not a capability
- Thus, access rights are checked with each request
  - by the RPC protocol
- Authentication usually only via user and group ID
  - extremely insecure!
- More possibilities in NFS3:
  - Diffie-Hellman key exchange (insecure)
  - Kerberos
- NFS4: secure RPC (RPCSEC_GSS)

9.2 Case Study: NFS ...

Mount Service

- An NFS file system can be mounted in the local directory tree



- Collaboration of `mount` command in the client with the mount service of the NFS server
  - on request, the mount service provides file handles of the exported directories (for name resolution)

Notes for slide 293:

A directory exported from an NFS server A may contain a subdirectory that this server imports from another NFS server B. However, A is not be allowed to export this subdirectory to its clients. A client importing the directory from A must therefore also import the subdirectory from B.

Translation of Pathnames

- Iteratively (NFS3): for each directory one request to NFS server
  - necessary because path can cross mount points
  - inefficiency is mitigated by client caching

Automounter

- Goal: set up an NFS mount only when it is accessed
  - better fault tolerance, load balancing is possible
- Automounter is local NFS server
  - thereby it sees the *lookup()*-requests of the client
- On request: set up the NFS mount and create a symbolic link to the mount point
- After prolonged inactivity: release the mount

Server Caching

- Traditional file caching in UNIX:
  - buffer in main memory for most recently used disk blocks
  - *read ahead*: sequential blocks are loaded into cache beforehand
  - *delayed write*: modified blocks only written back when space is needed; additionally every 30s by `sync`
- Server caching in NFS: two modes
  - *write through*: write requests are executed in the server cache and immediately also on disk
    - advantage: no data loss in case of server crash
  - *delayed write*: modified data will remain in the cache until a *commit* operation is executed (i.e. file is closed)
    - advantage: better performance if many write operations

Client Caching

- NFS client buffers the results of (among other things) *read* / *write* and *lookup* operations in a local cache
  - leads to consistency issues, since now multiple copies
- Client is responsible for maintaining consistency
- Timeliness of the cache entry is checked with each access
  - for that: compare whether the modification timestamp in the cache matches the modification timestamp on the server
  - in case of negative validation: cache entry is deleted
  - if validation is successful: cache entry is considered current for a certain time (3 - 30 s) without further checks
    - i.e. changes only become visible after a few seconds
    - compromise between consistency and efficiency

Client Caching ...

- Treatment of write operations:
  - file block is marked as *dirty* in the cache
  - marked blocks are sent asynchronously to the server:
    - when closing the file
    - at a `sync` operation on client machine
    - possibly more often by *block-input/output* (bio)-demons
- Bio-demons realize asynchronous operations for *read ahead* and *delayed write*
  - for performance optimization
- NFS does not guarantee real consistency of client caches