Distributed Systems
File Systems

Wouter Joosen

DistriNet, KULeuven

October 24 & 31, 2023






KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

Introduction

- Objectives of this study
  - ***Study existing subsystems: a combination of modern real world, and pedagogical (slightly older)***
    - Services offered
    - Architecture & design trade-offs
    - Limitations
  - Learn about **non-functional requirements**: initial focus is on performance, availability and fault tolerance, data consistency
    - Techniques used
    - Results
    - Limitations
  - Non function requirements: Security & Privacy – ***new & intro in 2023-2024!***

Where to apply know how?

Storage? Persistence?
Concurrency Control?

Applications, services

Interaction, or more?

Middleware

Operating system

Computer and network hardware

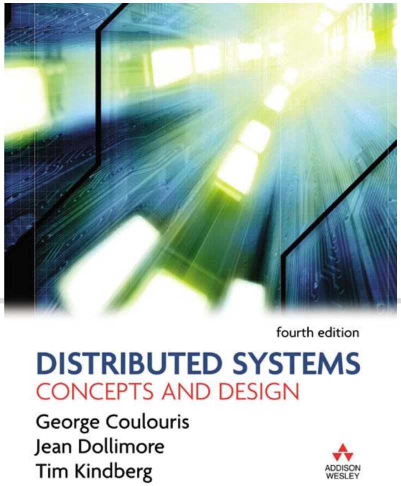
Platform

Build up experience:

- In using middleware (building applications)
  - (system perspective)
- In customizing/adapting middleware
  - (development perspective)
- Eventually in building middleware

Existing services: *from the textbook*

- Distributed file systems
- Name services (see DNS in computer networks!)
- ...



KATHOLIEKE UNIVERSITEIT LEUVEN

Distributed Systems:

File Systems

Main focus of this case study– caching & performance

Distributed file systems

- Overview
  - File service architecture (quick intro)
  - case study: NFS
  - case study AFS
  - comparison NFS <> AFS

Distributed file systems
File service architecture

- Definitions:
  - file
  - directory
  - file system

  cf. Operating systems

Distributed file systems
File service architecture

- Requirements:
    - access transparency
    - location transparency
    - failure transparency
    - performance transparency
    - hardware and operating system heterogeneity
    - Scalability
    - concurrent updates

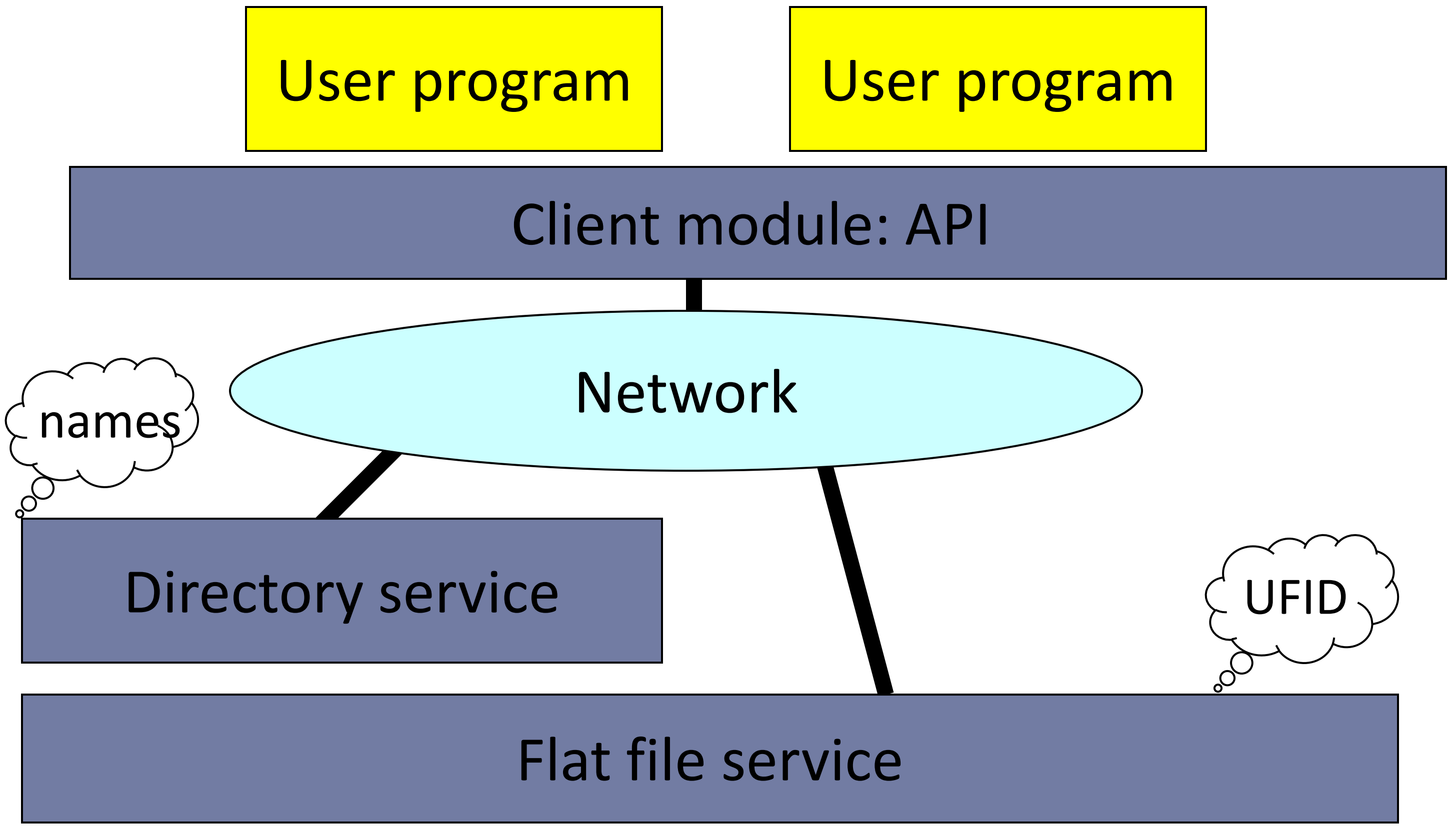
- Historical case studies (pioneering cases 80'- 90's):
    - Low interest in concurrency,
    - String focus on remote access

Distributed file systems
File service architecture

- Requirements *(cont.)* :
  - scalable to a very large number of nodes
    - replication transparency
    - migration transparency
  - future:
    - tolerance to network partitioning

Distributed file systems
File service architecture

- File service components



Distributed file systems
File service architecture

- Flat file service
  - file = data + attributes
  - data = sequence of items
  - operations: simple read & write
  - attributes: in flat file & directory service
  - UFID: unique file identifier

Distributed file systems
File service architecture

- Flat file service: operations
  - *Read (FileID, Position, n) ➔ Data*
  - *Write (FileID, Position, Data)*
  - *Create () ➔ FileID*
  - *Delete (FileID)*
  - *GetAttributes (FileId) ➔ Attr*
  - *SetAttributes (FileID, Attr)*

Distributed file systems
File service architecture

- Flat file service: in light of *fault tolerance (easy recovery)*
  - straightforward for simple servers
  - idempotent operations
  - stateless servers

Distributed file systems
File service architecture

- Directory service
  - translate file name in UFID
  - substitute for open
  - responsible for access control

Distributed file systems
File service architecture

- Directory service: operations
  - *Lookup (Dir, Name, AccessMode, UserID)*
    ➔ UFID
  - *AddName (Dir, Name, FileID, UserID)*
  - *UnName (Dir, Name)*
  - *ReName (Dir, OldName, NewName)*
  - *GetNames (Dir, Pattern) ➔ list-of-names*

Distributed file systems
File service architecture

- Implementation techniques:
  - known techniques from OS experience
  - remain important
  - distributed file service: comparable in
    - performance
    - reliability

Distributed file systems
File service architecture

- Implementation techniques: overview (in line with text book)
  - file groups
  - space leaks
  - capabilities and access control (*no priority at this point*)
  - access modes
  - file representation
  - file location
  - group location
  - file addressing
  - Caching: *main focus of this case study*

Distributed file systems
File service architecture

- Implementation techniques: file groups
  - (similar: file system, partition)
  - = collection of files mounted on a server computer
  - unit of distribution over servers
  - transparent migration of file groups
  - once created file is locked in file group
  - UFID = file group identifier + file identifier

- Note: operational management and administrative aspects often ignored, underestimated in importance….

Distributed file systems
File service architecture

- Implementation techniques: space leaks
  - 2 steps for creating a file
    - create (empty) file and get new UFID
    - enter name + UFID in directory
  - failure after step 1:
    - file exists in file server
    - unreachable: UFID not in any directory

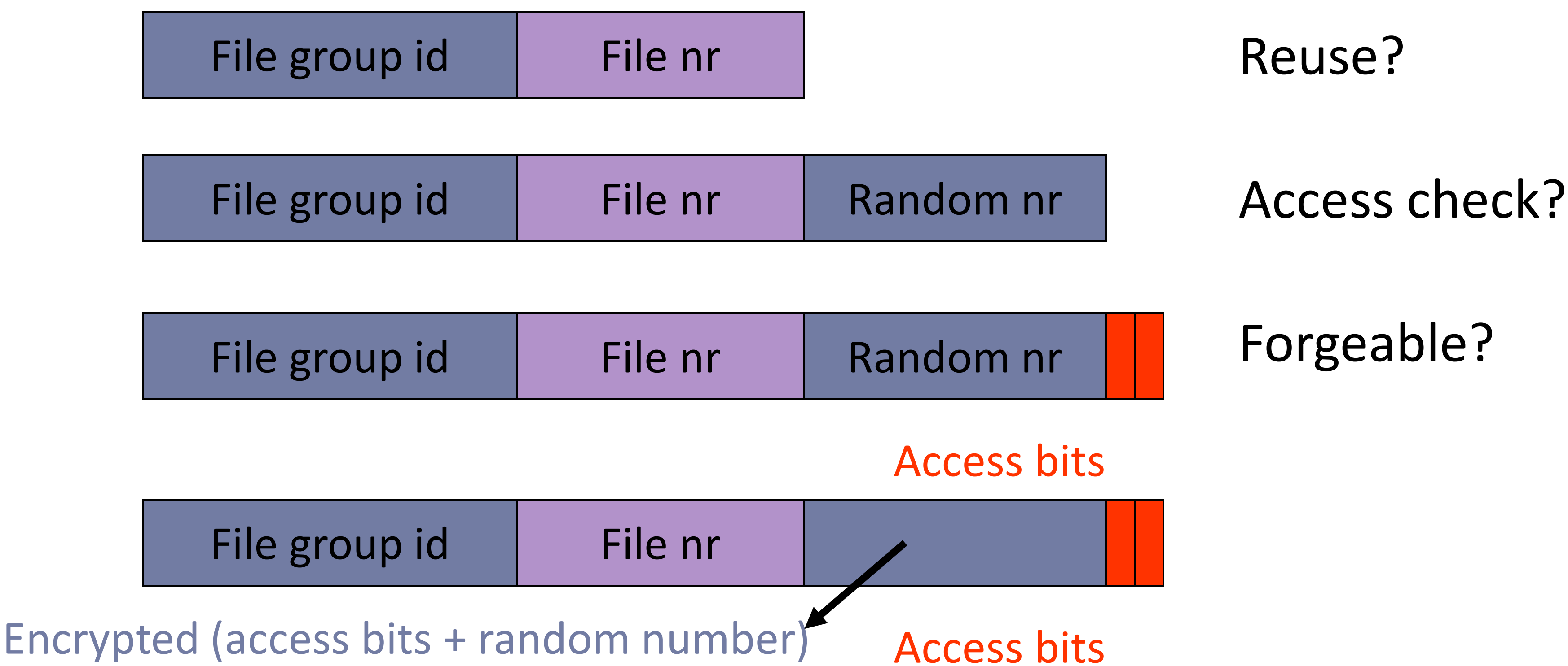
  ➔ lost space on disk
  - detection requires co-operation between
    - file server
    - directory server

Distributed file systems
File service architecture

- Implementation techniques: capabilities
  - = digital key: access to resource granted on presentation of the capability
  - request to directory server: file name + user id + mode of access
  - ➔ UFID including permitted access modes
  - construction of UFID
  - unique
  - encode access
  - unforgeable

Distributed file systems
File service architecture

- Implementation techniques: capabilities



Distributed file systems
File service architecture

- Implementation techniques: file location
  - from UFID ➔ location of file server
  - use of replicated group location database

    file group id, PortId

  - why replication?
  - why location not encoded in UFID?

Distributed file systems
File service architecture

- Implementation techniques: caching
  - server cache: reduce delay for disk I/O
    - selecting blocks for release
    - coherence:
      - dirty flags
      - write-through caching
  - client cache: reduce network delay
    - always use write-through
    - synchronisation problems with multiple caches

Distributed file systems

- Overview
  - File service model
  - case study: Network File System -- NFS
  - case study: AFS
  - comparison NFS <> AFS

Distributed file systems

NFS

- Background and aims
  - first file service product
  - emulate UNIX file system interface
  - de facto standard
    - key interfaces published in public domain
    - source code available for reference implementation
  - supports diskless workstations
    - not important anymore

Distributed file systems

NFS

- Design characteristics
  - client and server modules can be in any node
    Large installations include a few servers
  - Clients:
    - On Unix: emulation of standard UNIX file system
    - for MS/DOS, Windows, Apple, …
  - Integrated file and directory service
  - Integration of remote file systems in a local one: mount ➔ remote mount

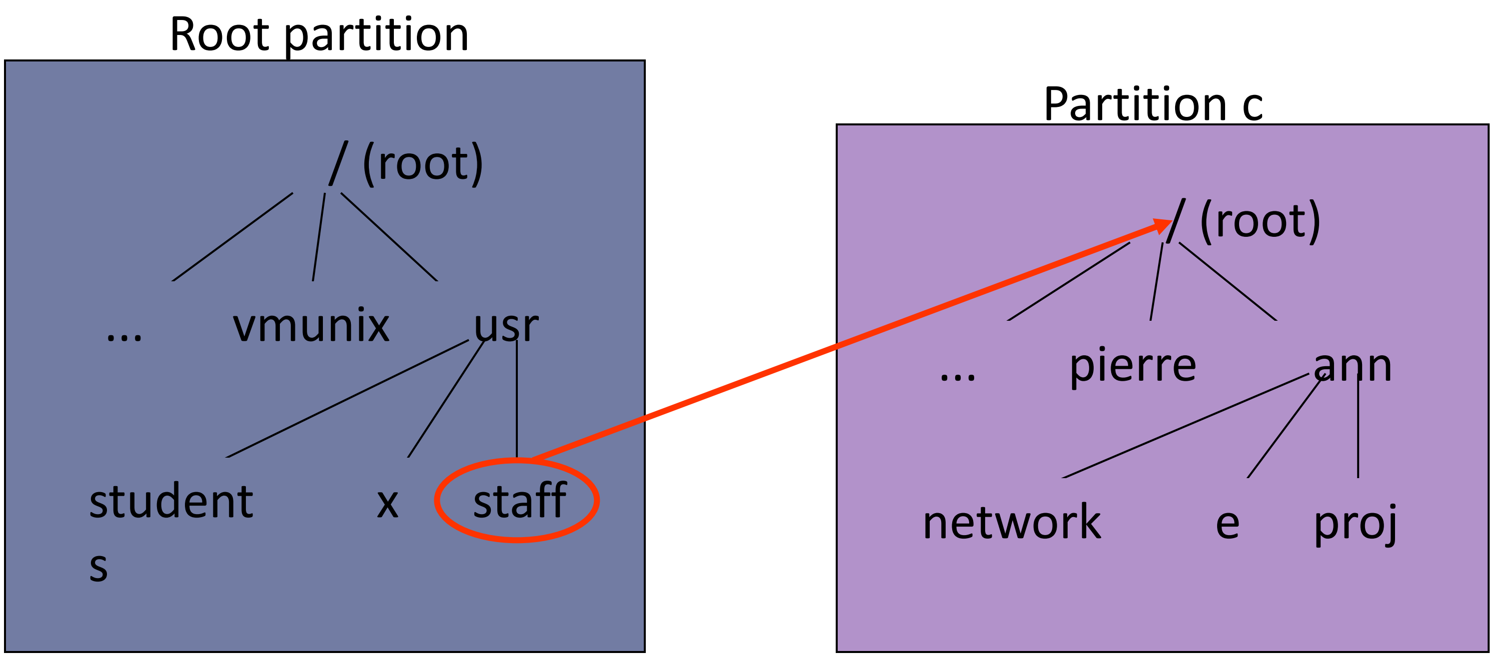
Distributed file systems

NFS: configuration

- Unix mount system call
  - each disk partition contains hierarchical FS
  - how integrate?
    - Name partitions
      a:/usr/students/john
    - glue partitions together
      - invisible for user
      - partitions remain useful for system managers

Distributed file systems
NFS: configuration

- Unix mount system call

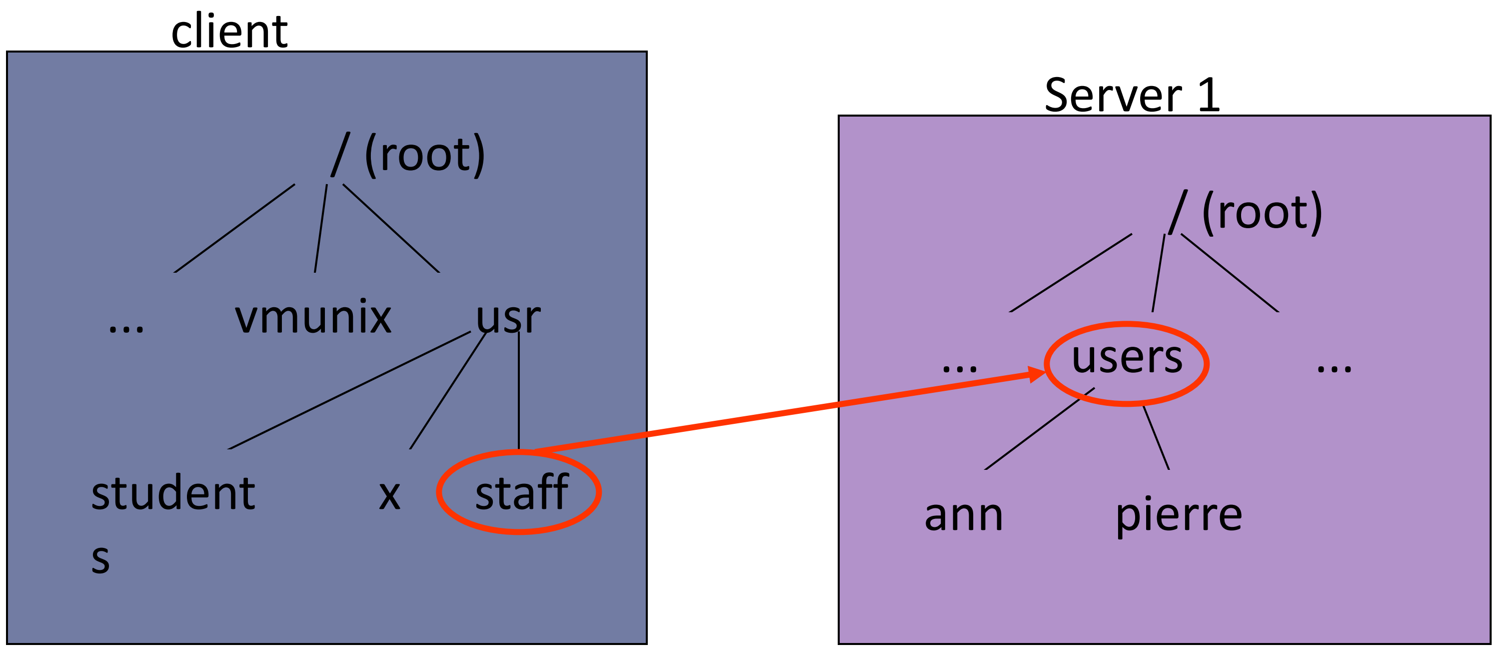


Directory staff ≡ root of c: /usr/staff/ann/network

Distributed file systems

NFS: configuration

- Remote mount



Directory staff ≡ users: /usr/staff/ann/...

Distributed file systems

NFS: configuration

- Mount service on server
  - enables clients to integrate (part of) remote file system in the local name space
  - exported file systems in /etc/exports + access list (= hosts permitted to mount; secure?)
- On client side
  - file systems to mount enumerated in /etc/rc
  - typically mount at start up time

Distributed file systems

NFS: configuration

- Mounting semantics
  - hard
    - client waits until request for a remote file succeeds
    - eventually forever
  - soft
    - failure returned if request does not succeed after n retries
    - breaks Unix failure semantics

Distributed file systems

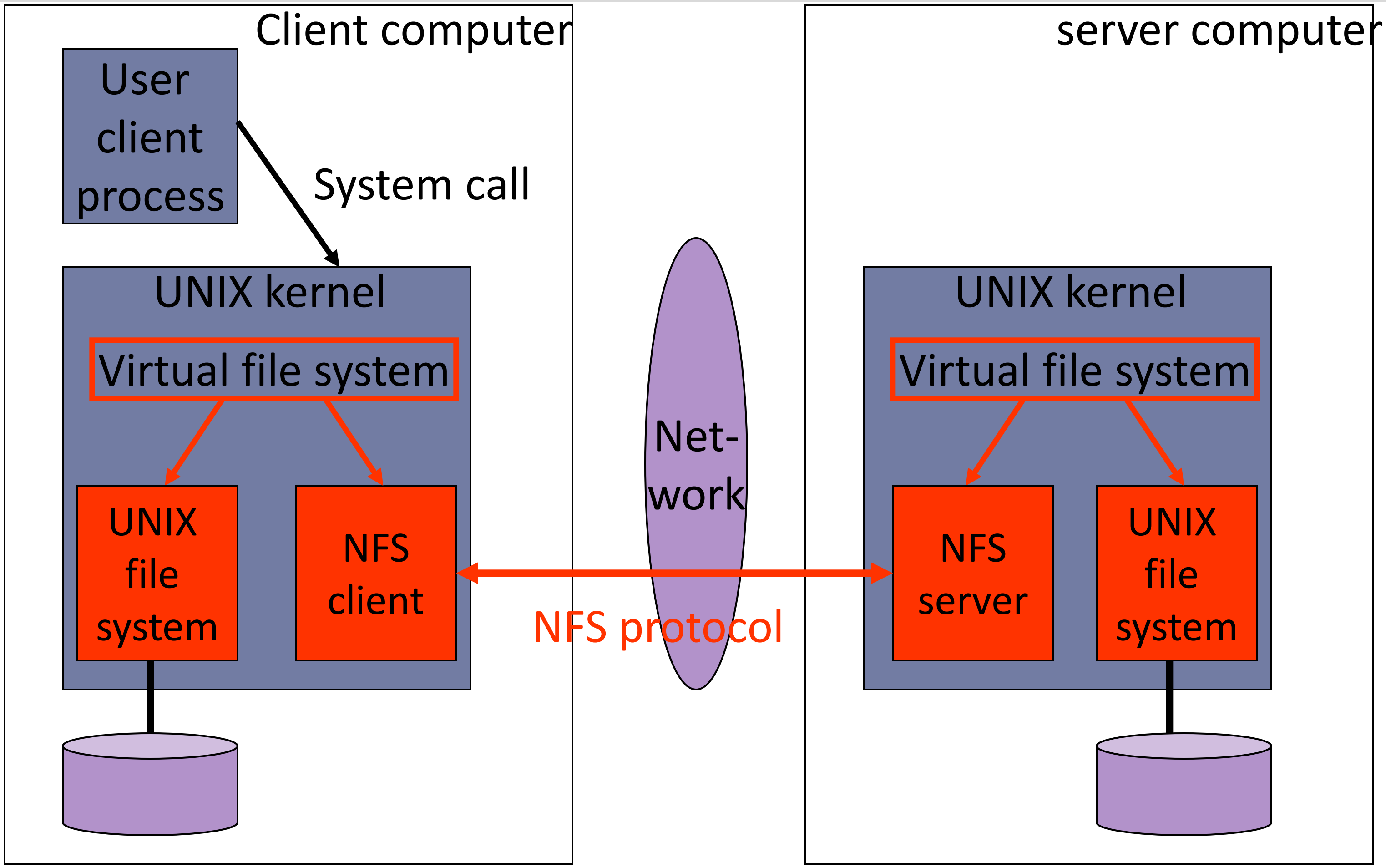
NFS: configuration

- Automounter
  - principle:
    - *empty* mount points on clients
    - mount on first request for remote file
  - acts as a server for a local client
    - gets references to empty mount points
    - maps mount points to remote file systems
    - referenced file system mounted on mount point via a symbolic link, to avoid redundant requests to automounter

Distributed file systems
NFS: implementation

- In UNIX: client and server modules implemented in kernel
- virtual file system:
  - internal key interface, based on file handles for remote files

Distributed file systems
NFS: implementation



Distributed file systems
NFS: implementation

- Virtual file system
  - Added to UNIX kernel to make distinction
    - Local files
    - Remote files
  - File handles: file ID in NFS
    - Base: inode number
      - File ID in partition on UNIX system
    - Extended with:
      - File system identifier
      - inode generation number (to enable reuse)

Distributed file systems
NFS: implementation

- Client integration:
  - NFS client module integrated in kernel
    - offers standard UNIX interface
    - no client recompilation/reloading
    - single client module for all user level processes
    - encryption in kernel
- server integration
  - only for performance reasons
  - user level = 80% of kernel level version

Distributed file systems
NFS: implementation

- Directory service
  - name resolution co-ordinated in client
  - step-by-step process for multi-part file names
  - mapping tables in server: high overhead reduced by caching
- Access control and authentication
  - based on UNIX user ID and group ID
  - included and checked for every NFS request
  - secure NFS 4.0 thanks to use of DES encryption

Distributed file systems
NFS: implementation

- Caching
  - Unix caching
    - based on disk blocks
    - delayed write
    - read ahead
    - periodically sync to flush buffers in cache
  - Caching in NFS
    - Server caching
    - Client caching

Distributed file systems
NFS: implementation

- Server caching in NFS
  - based on standard UNIX caching: 2 modes
  - write-through (instead of delayed write)
    - failure semantics ↗
    - performance ↘
  - delayed write
    - Data stored in cache, till commit operation is received
      - Close on client → commit operation on server
    - Failure semantics?
    - Performance ↗

Distributed file systems
NFS: implementation

- Client caching
  - cached are results of
    - *read, write, getattr, lookup, readdir*
  - problem: multiple copies of same data at different NFS clients
  - NFS clients use read-ahead and delayed write

Distributed file systems
NFS: implementation

- Client caching *(cont.)*
  - handling of writes
    - block of file is fetched and updated
    - changed block is marked as dirty
    - dirty pages of files are flushed to server asynchronously
      - on close of file
      - sync operation on client
      - by bio-daemon (when block is filled)
    - dirty pages of directories are flushed to server
      - by bio-daemon without further delay

Distributed file systems
NFS: implementation

- Client caching *(cont.)*
  - consistency checks
    - based on time-stamps indicating last modification of file on server
    - validation checks
      - when file is opened
      - when a new block is fetched
      - assumed to remain valid for a fixed time (3 sec for file, 30 sec for directory)
      - next operation causes another check
    - costly procedure

Distributed file systems
NFS: implementation

- Caching
  - Cached entry is valid

    $(T - T_c) < t$ or $(Tm_{client} = Tm_{server})$
    - T: current time
    - Tc: time when cache entry was last validated
    - t: freshness interval (3 .. 30 secs in Solaris)
    - Tm: time when block was last modified at …
  - consistency level
    - acceptable
    - most UNIX applications do not depend critically on synchronisation of file updates

Distributed file systems
NFS: implementation

- Performance
  - reasonable performance
    - remote files on fast disk better than local files on slow disk
    - RPC packets are 9 Kb to contain 8Kb disk blocks
  - Lookup operation covers about 50% of server calls
  - Drawbacks:
    - frequent *getattr* calls for time-stamps (cache validation)
    - poor performance of (relative infrequent) writes (because of write-through)

Distributed file systems

- Overview
  - File service model
  - case study: NFS
  - case study: Andrew File System -- AFS
  - comparison NFS <> AFS

Distributed file systems

AFS

- Background and aims
  - Base: observation of UNIX file systems (*data driven observations at that time 198X*)
    - files are small (< 10 Kb)
    - read is more common than write
    - sequential access is common, random access is not
    - most files are not shared
    - shared files are often modified by one user
    - file references come in bursts
  - Aim: combine best of personal computers and time-sharing systems

Distributed file systems

AFS

- Background and aims *(cont.)*
  - Assumptions about environment
    - secured file servers
    - public workstations
    - workstations with local disk
    - no private files on local disk
  - Key targets: scalability and security
    - CMU 1991: 800 workstations, 40 servers

Distributed file systems

AFS

- Design characteristics
  - whole-file serving
  - whole-file caching
  - ➔ entire files are transmitted, not blocks
  - client cache realised on local disk (relatively large)
  - ➔ lower number of open requests on the network
  - separation between file and directory service

Distributed file systems

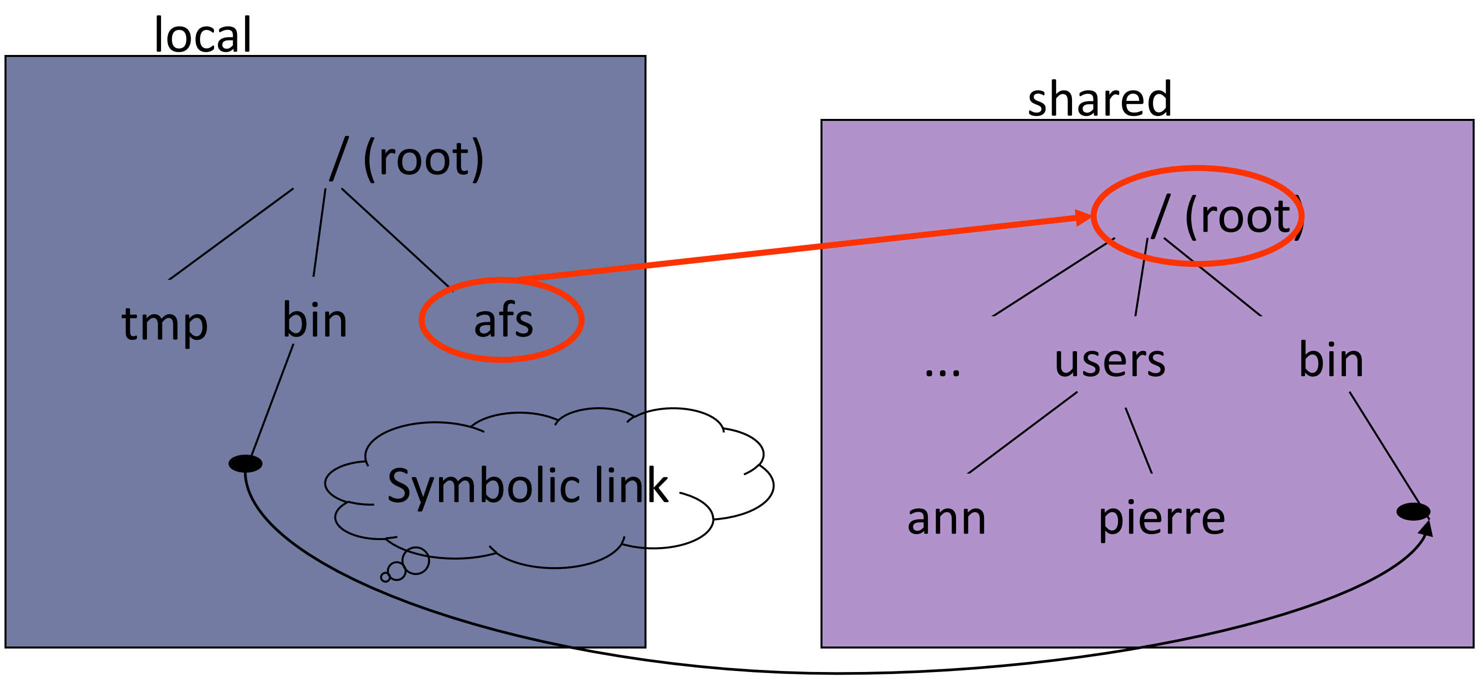
AFS

- Configuration
  - single (global) name space
  - local files
    - temporary files
    - system files for start-up
  - volume = unit of configuration and management
  - each server maintains a replicated location database (volume-server mappings)

Distributed file systems

AFS

- File name space



Directory afs ≡ root of shared file system

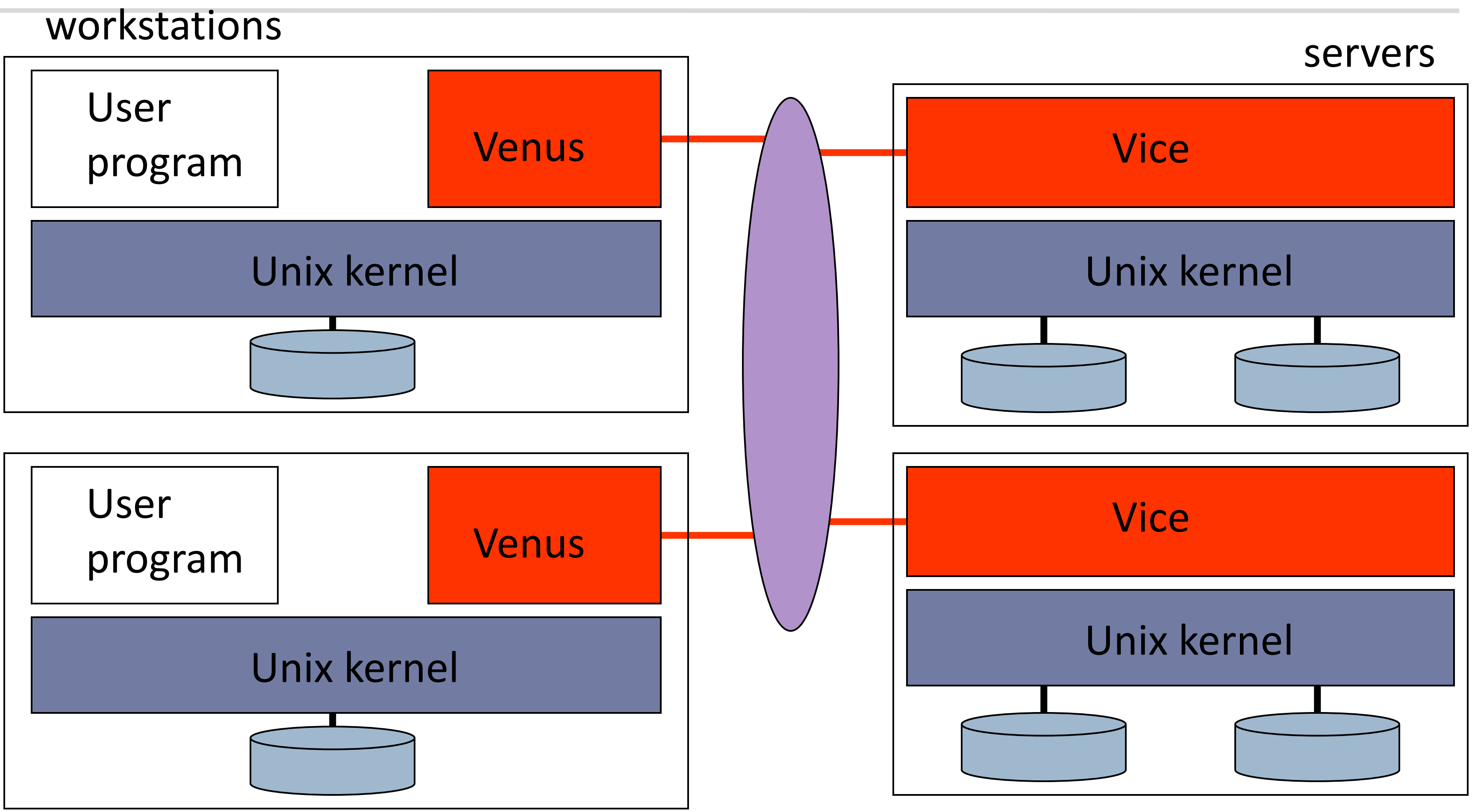
Distributed file systems

AFS

- Implementation
  - Vice = file server
    - secured systems, controlled by system management
    - understands only file identifiers
    - runs in user space
  - Venus = user/client software
    - runs on workstations
    - workstations keep autonomy
    - implements directory services
  - kernel modifications for *open* and *close*

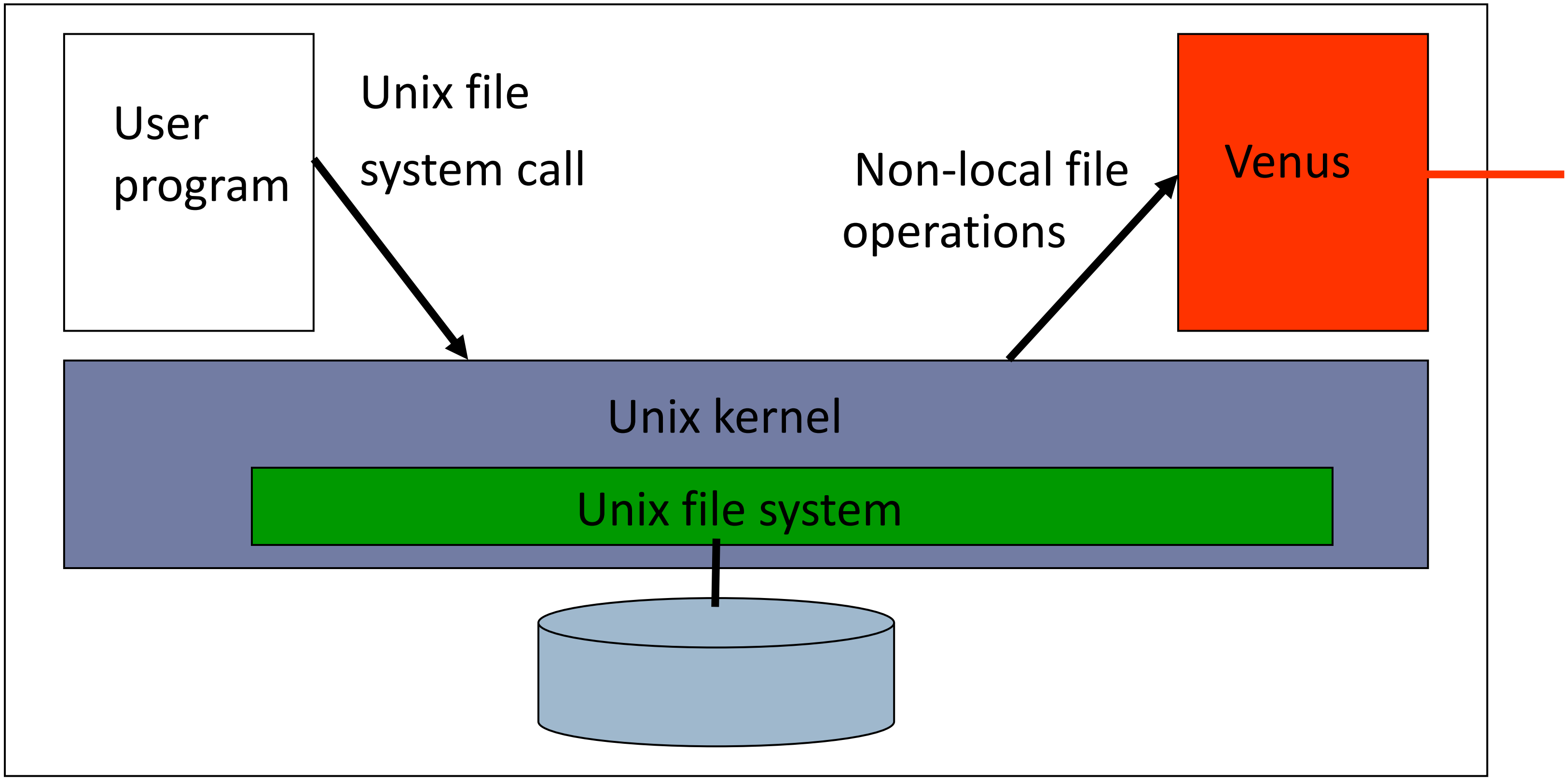
Distributed file systems

AFS



Distributed file systems

AFS



Distributed file systems

AFS

- Implementation of file system calls
  - open, close:
    - UNIX kernel of workstation
    - Venus (on workstation)
    - VICE (on server)
  - read, write:
    - UNIX kernel of workstation

Distributed file systems

AFS

- Open system call

  open(FileName,…)

kernel — if FileName in shared space /afs then pass request to Venus

venus — if file not present in local cache OR file present with invalid callback then pass request to Vice server

network

vice — transfer copy of file and valid callback

place copy of file in local file system and store FileName

open local file and return descriptor to user process

Distributed file systems

AFS

- Read Write system call

kernel

read(FileDescriptor,…)

perform normal UNIX read op local copy of file

….

Distributed file systems

AFS

- Close system call

	close(FileDescriptor,…)
kernel	close local copy and inform Venus about close
venus	if local copy is changed then send copy to Vice server
network	
vice	Store copy of file and send callback to other clients holding callback promise
	…

Distributed file systems
AFS

- Caching
  - callback principle
    - service supplies “callback promise” at open
    - “promise” (state) is stored with file in client cache
  - callback promise can be *valid* or *cancelled (terminology from original papers; in fact, a boolean value indicating the validity of of the file in client-side cache).*
    - initially valid
    - server sends message to cancel callback promise
      - to all clients that cache the file
      - whenever file is updated

Distributed file systems

AFS

- Caching maintenance
  - when client workstation reboots
    - cache validation necessary because of missed messages
    - cache validation requests are sent for each valid promise
  - valid callback promises are renewed
    - on *open*
    - when no communication has occurred with the server during a period T

Distributed file systems

AFS

- Update semantics
  - guarantee after successful *open* of file **F** on server **S**:
    latest(F, S, 0)
    or
    lostcallback(S, T) **and** incache(F) **and** latest(F, S, T)
  - no other concurrency control
    - 2 copies can be updated at different workstations
    - all updates except from last close are (silently) lost
    - <> normal UNIX operation

Distributed file systems

AFS

- Performance: impressive <> NFS
  - benchmark: load on server
    - 40% AFS
    - 100% NFS
  - whole-file caching:
    - reduction of load on servers
    - minimises effect of network latency
  - read-only volumes are replicated (master copy for occasional updates)

Notice: Andrew has been optimised for a specific pattern of use!!

Distributed file systems

- Overview
  - File service model
  - case study: NFS
  - case study AFS
  - comparison NFS <> AFS

Distributed file systems

NFS <>AFS

- Access transparency
  - both in NFS and AFS
  - Unix file system interface is offered
- Location transparency
  - uniform view on shared files in AFS
  - in NFS
    - mounting freedom
    - same view possible if same mounting; discipline!

Distributed file systems

NFS <>AFS

- Failure transparency
  - NFS
    - no state of clients stored in servers
    - idempotent operations
    - transparency limited by soft mounting
  - AFS
    - state about clients stored in servers
    - cache maintenance protocol handles server crashes
    - limitations?

Distributed file systems

NFS <>AFS

- Performance transparency
  - NFS
    - acceptable performance degradation
  - AFS
    - only delay for first open operation on file
    - better than NFS for small files

Distributed file systems

NFS <>AFS

- Migration transparency
  - limited: update of locations required
  - NFS
    - update configuration files on all clients
  - AFS
    - update of replicated database

Distributed file systems

NFS <>AFS

- Replication transparency
  - NFS
    - not supported
  - AFS
    - limited support; for read-only volumes
    - one master for exceptional updates; manual procedure for propagating changes to other volumes

Distributed file systems

NFS <>AFS

- Concurrency transparency
  - not supported (in UNIX)
- Scalability transparency
  - AFS better than NFS

Distributed file systems

- Overview
  - File service architecture
  - case study: NFS
  - case study AFS
  - comparison NFS <> AFS

Distributed file systems

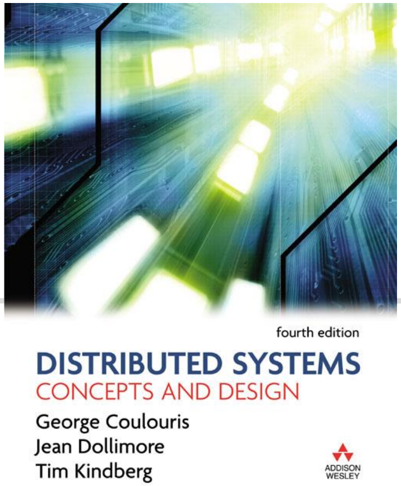
- Conclusion beyond the specific of the case study (this often applies)
  - Standards <> quality
  - evolution to standards and common key interface
    - AFS-3 incorporates Kerberos, vnode interface, large messages for file blocks (64Kb)
  - network (mainly access) transparency causes inheritance of weakness: e.g. no concurrency control
  - evolution mainly performance driven

What if?

- Files are read-only, immutable
- Files are append-only?
- Files are relatively/extremely large?
- Devices are extremely poor? (e.g. certain categories of IoT)

… *everything has been thought off before☺.*

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP



KATHOLIEKE UNIVERSITEIT LEUVEN

Distributed Systems:

File Systems

Questions?