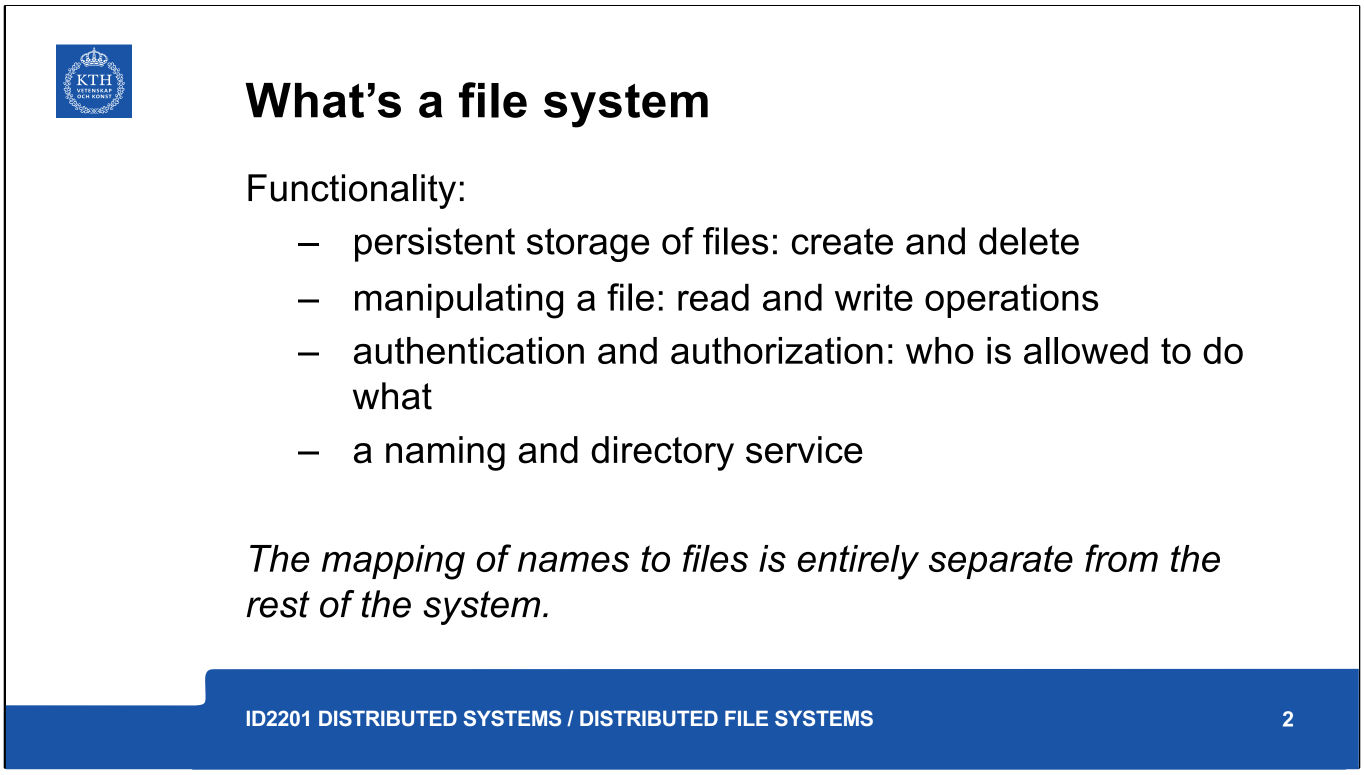
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Distributed file systems

Vladimir Vlassov and Johan Montelius

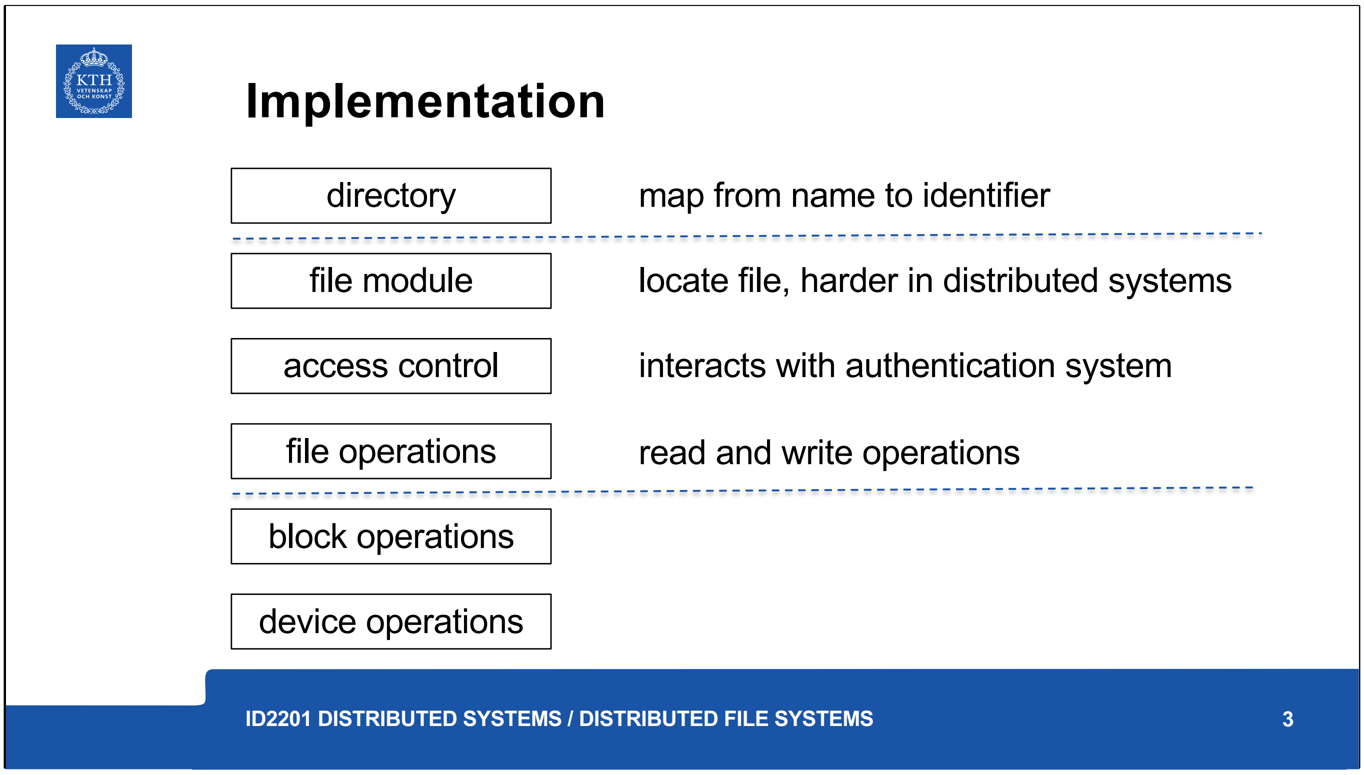


What’s a file system

Functionality:

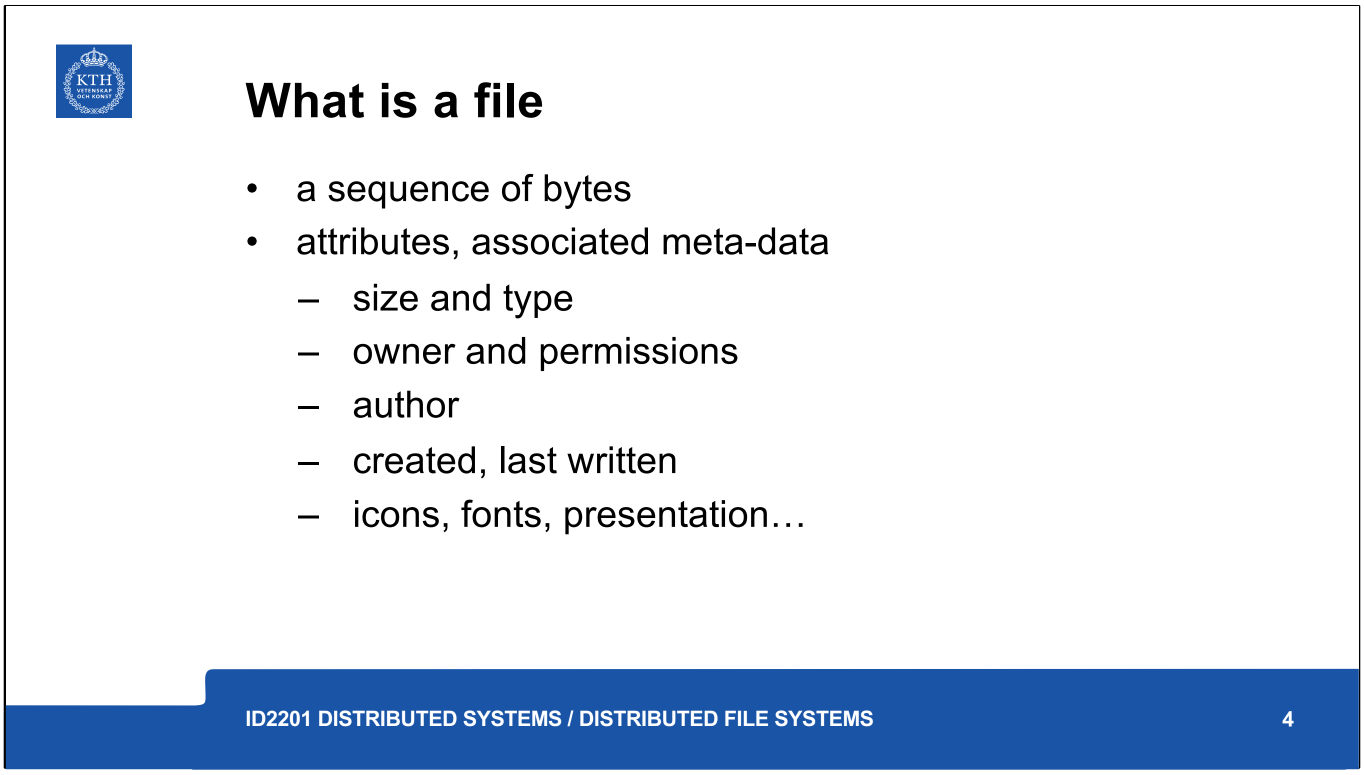
- persistent storage of files: create and delete
- manipulating a file: read and write operations
- authentication and authorization: who is allowed to do what
- a naming and directory service

The mapping of names to files is entirely separate from the rest of the system.



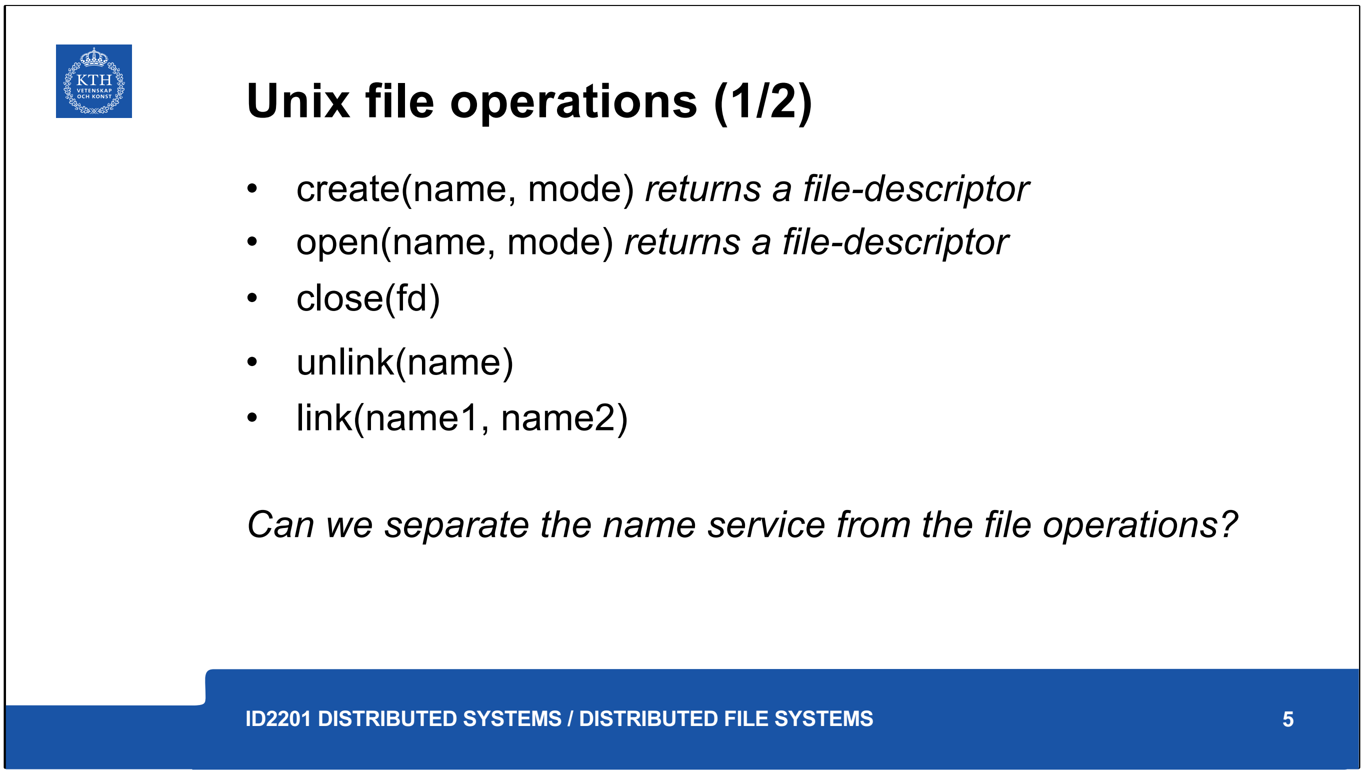
Implementation

directory	map from name to identifier
file module	locate file, harder in distributed systems
access control	interacts with authentication system
file operations	read and write operations
block operations	
device operations	



What is a file

- a sequence of bytes
- attributes, associated meta-data
  – size and type
  – owner and permissions
  – author
  – created, last written
  – icons, fonts, presentation…

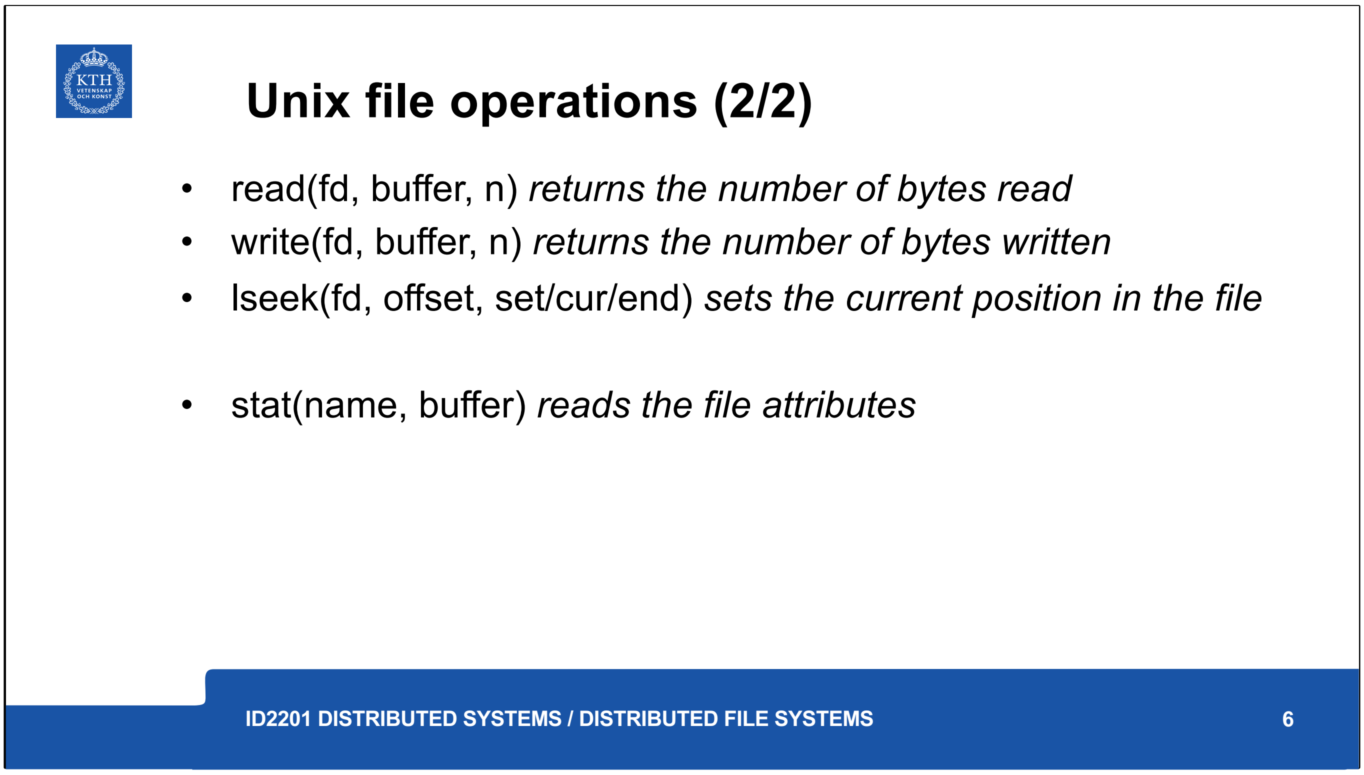


Unix file operations (1/2)

- create(name, mode) *returns a file-descriptor*
- open(name, mode) *returns a file-descriptor*
- close(fd)
- unlink(name)
- link(name1, name2)

Can we separate the name service from the file operations?

unlink(name) - Removes the file name from the directory structure. If the file has no other names, it is deleted.
link(name1, name2) Adds a new name (name2) for a file (name1).
Unix directories are lists of association structures, each containing one filename and one inode number. The file system driver must search a directory looking for a particular filename and then convert the filename to the correct corresponding inode number. Inodes do not contain file names, only other file metadata. Inode holds the metadata for this file and, traditionally, the pointers to the disk blocks where the file resides on the hard drive.
Directories are usually implemented as files. They have an inode and a data area but are typically accessed (at least written to) by special system calls. The data area of the directory file then contains the directory entries.
The standard solution is that some of the inodes in the root directory point to entries that are also directories. In many respects, they are just like files, but the file type indicates the filesystem to interpret them as directories. In other words, every directory is a simple linear list of inode pointers. Some point to leaf nodes in the directory tree (files), and others point to internal nodes (another directory).



Unix file operations (2/2)

- read(fd, buffer, n) *returns the number of bytes read*
- write(fd, buffer, n) *returns the number of bytes written*
- lseek(fd, offset, set/cur/end) *sets the current position in the file*

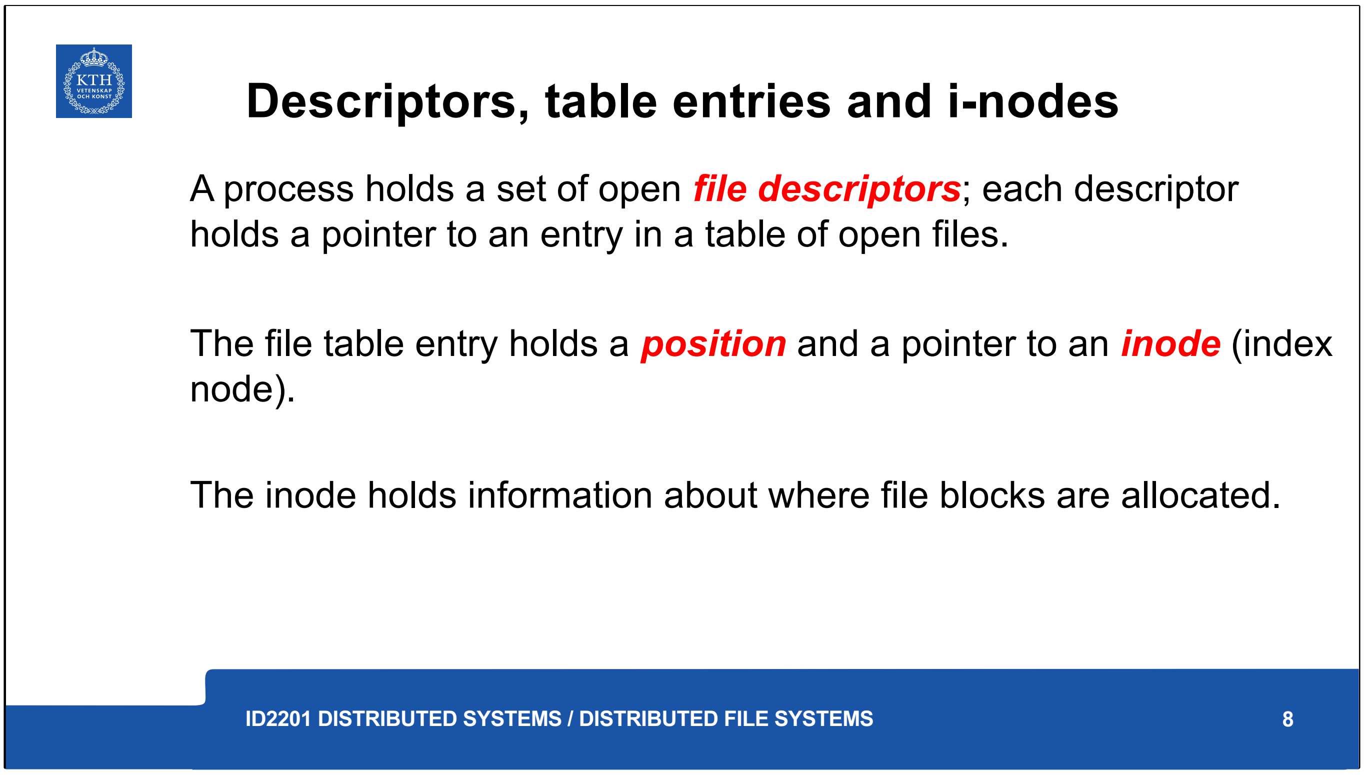
- stat(name, buffer) *reads the file attributes*

long lseek(fildes, offset, whence) -- move read/write file pointer;
whence :
set: 0, the pointer is set to offset bytes.
cur: 1, the pointer is set to its current location plus offset.
end: 2, the pointer is set to the file size plus offset.

Programming language API

Operating system operations are not always directly available from a high-level language.

Buffering of write operations to reduce the number of system calls.



Descriptors, table entries and i-nodes

A process holds a set of open ***file descriptors***; each descriptor holds a pointer to an entry in a table of open files.

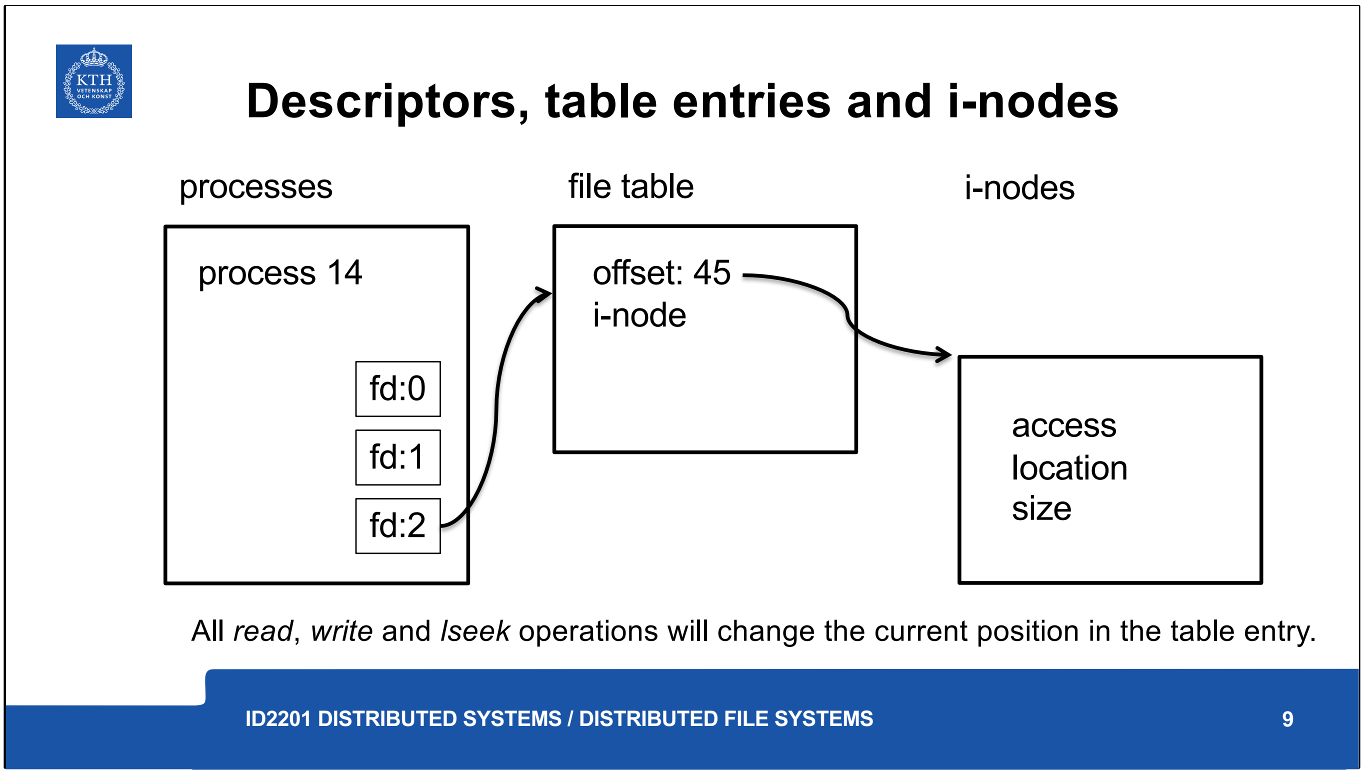
The file table entry holds a ***position*** and a pointer to an ***inode*** (index node).

The inode holds information about where file blocks are allocated.

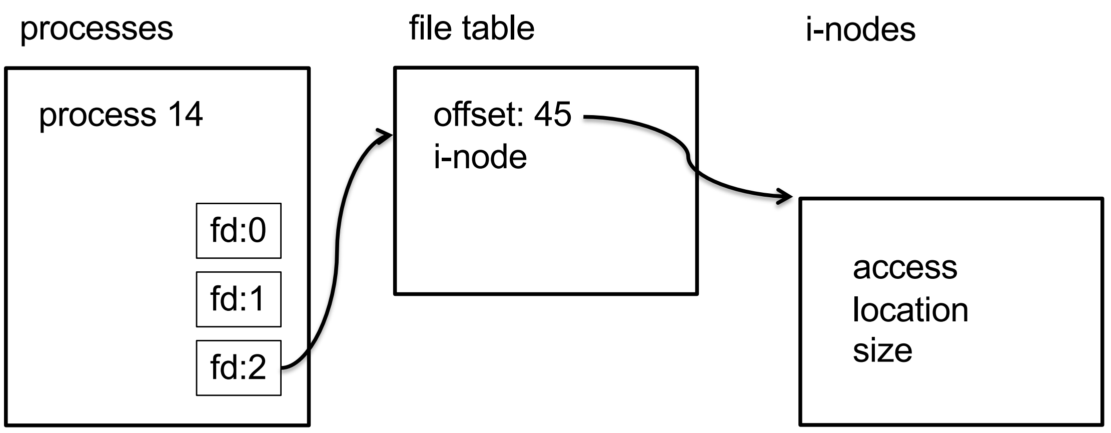
ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED FILE SYSTEMS 8

A file has the following attributes:

- Device ID (this identifies the device containing the file; that is, the scope of uniqueness of the serial number).
- File serial numbers.
- The file mode determines the file type and how the file's owner, its group, and others can access the file.
- A link count tells how many hard links point to the inode.
- The User ID of the file's owner.
- The Group ID of the file.
- The device ID of the file if it is a device file.
- The size of the file in bytes.
- Timestamps tell when the inode itself was last modified (ctime, inode change time), the file content last modified (mtime, modification time), and last accessed (atime, access time).
- The preferred I/O block size.
- The number of blocks allocated to this file.



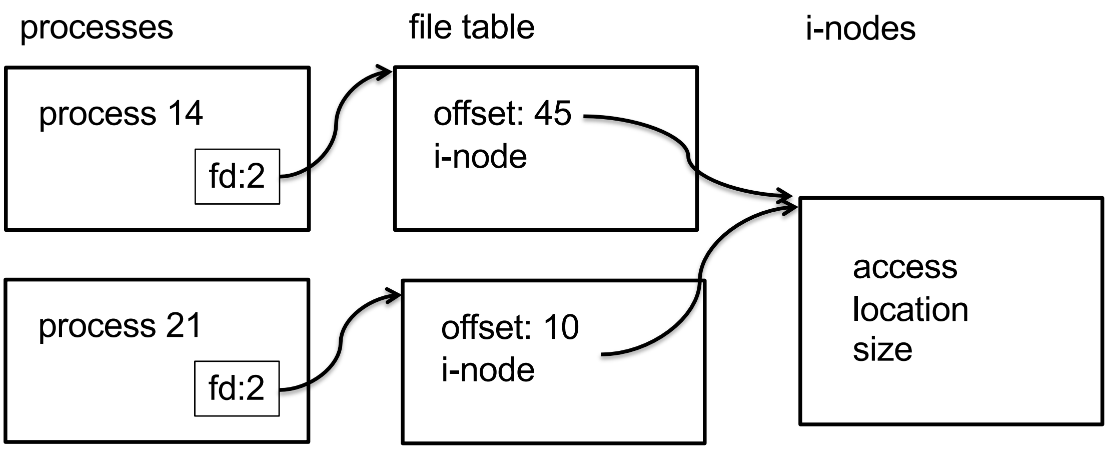
Descriptors, table entries and i-nodes



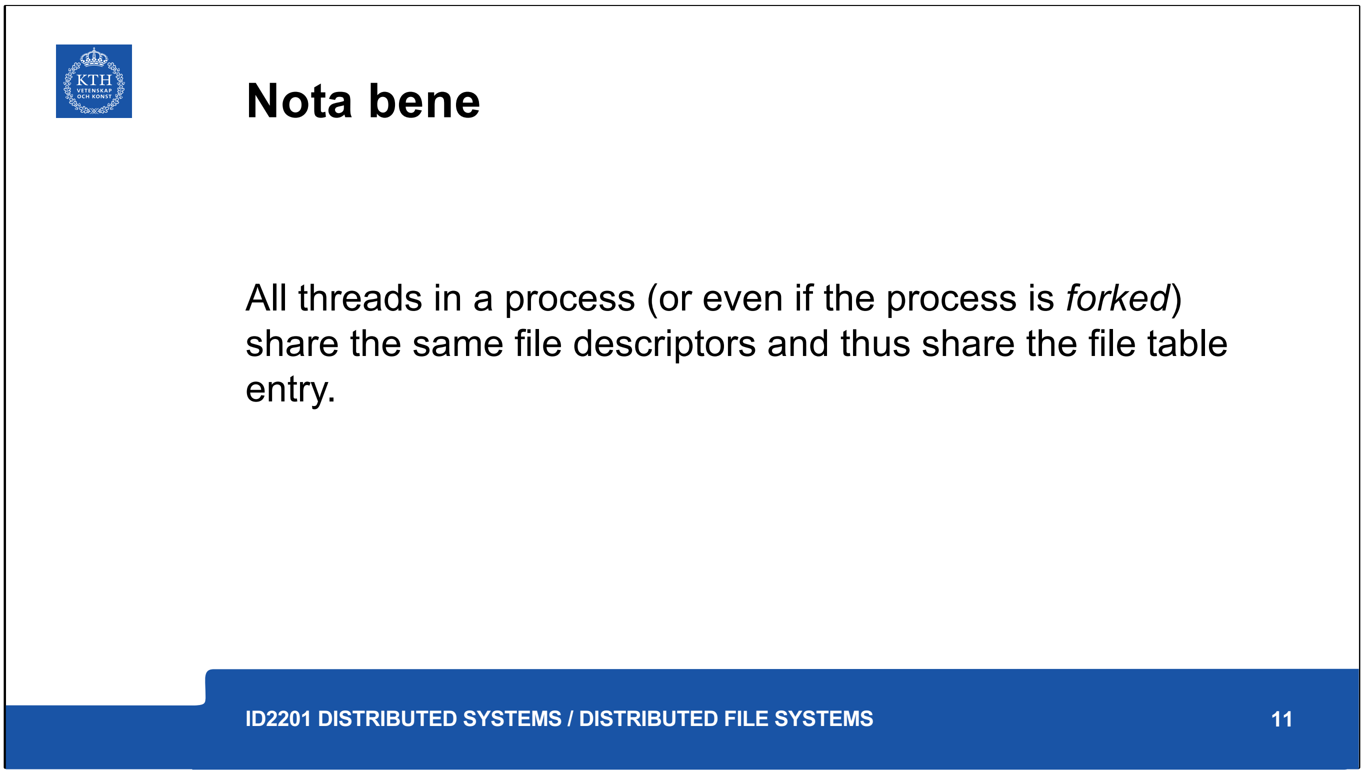
All *read*, *write* and *lseek* operations will change the current position in the table entry.

Inode holds the metadata for this file and, traditionally, the pointers to the disk blocks where the file resides on the hard drive. If the file is fragmented, very large, or both, some of the blocks the inode points to might hold further pointers to other disk blocks. And some of those other disk blocks might also have pointers to another set of disk blocks. This overcomes the problem of the inode being a fixed size and able to hold a finite number of pointers to disk blocks.
That method was superseded by a new scheme that uses “extents.” These record the start and end blocks of each set of contiguous blocks used to store the file. You only have to keep the first block and file length if the file is unfragmented. If the file is fragmented, you have to store the first and last block of each part of the file. This method is (obviously) more efficient.

Two processes open the same file



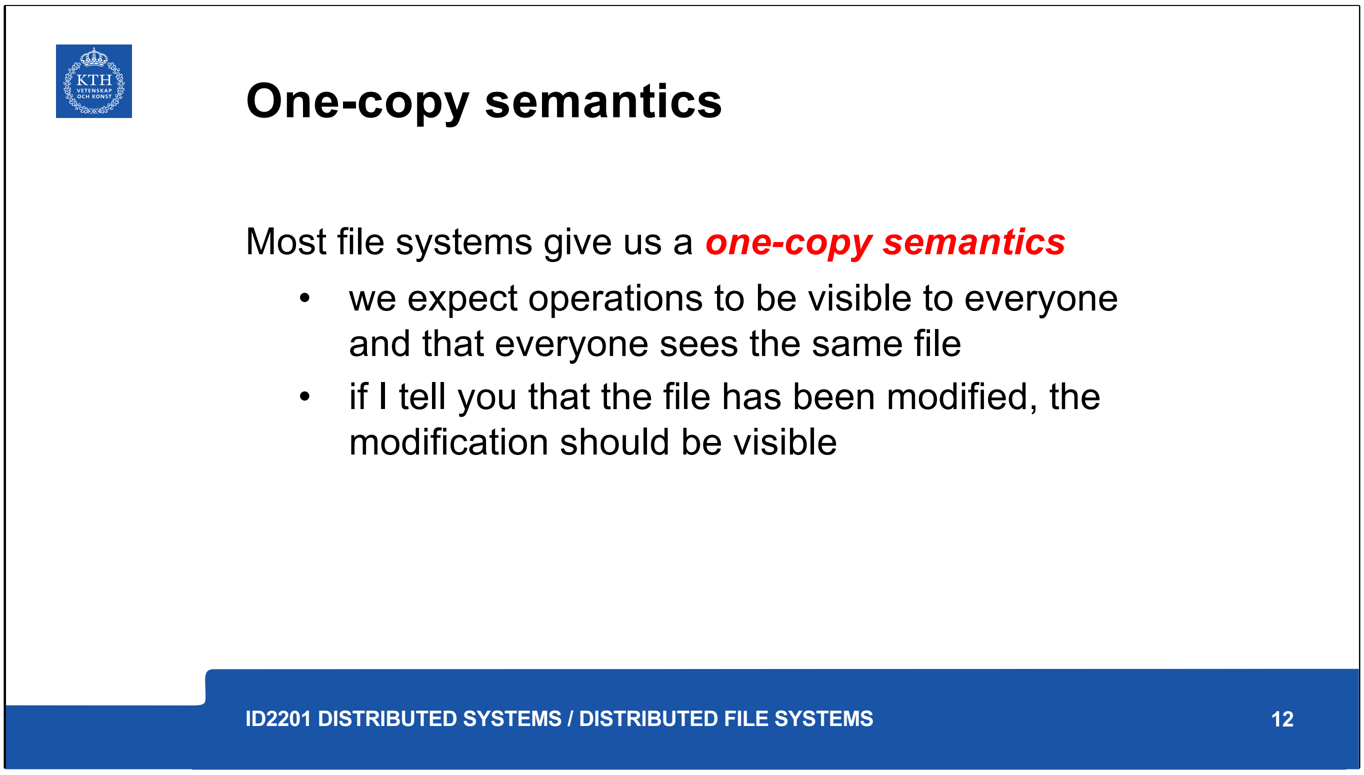
Two processes that open the same file will have independent file table entries.



Nota bene

All threads in a process (or even if the process is *forked*) share the same file descriptors and thus share the file table entry.

Nota bene: observe carefully or take special notice (used in written text to draw attention to what follows).



One-copy semantics

Most file systems give us a ***one-copy semantics***

- we expect operations to be visible to everyone and that everyone sees the same file
- if I tell you that the file has been modified, the modification should be visible

An architecture of a distributed file system

Let's define the **requirements** on a distributed file system.

Transparency - no difference between local and remote files

- access: same set operations
- location: same namespace
- mobility: allowed to move files without changing client side
- performance: close to a non-distributed system

Concurrency - simultaneous operations by several clients

Heterogeneity - not locked into a particular operating system

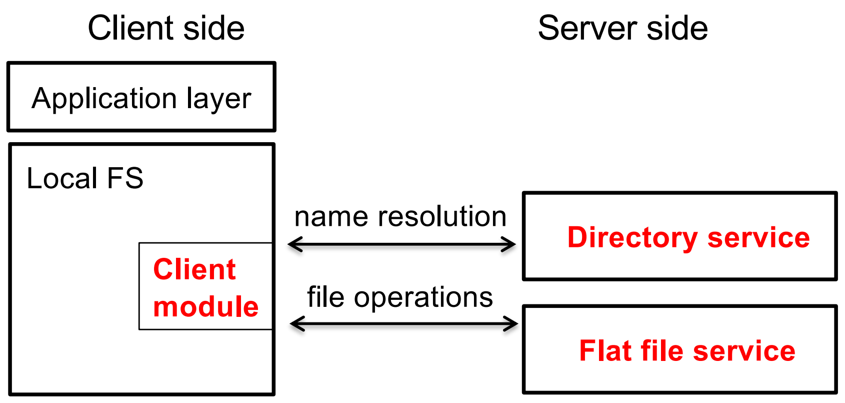
Fault tolerance - independent of clients, restartable, etc

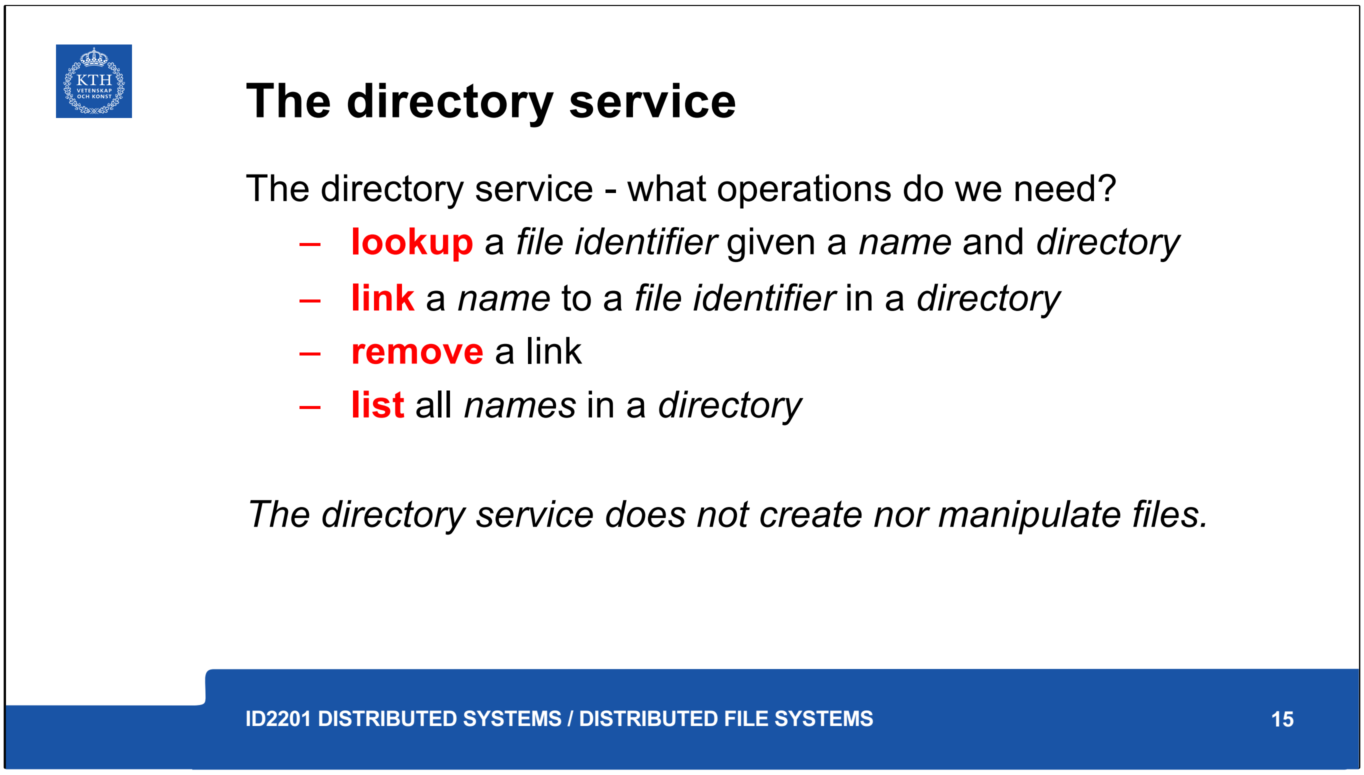
Consistency - one-copy semantics.... or?

Security - access control, who is allowed to do what

Distributed architecture

Separate the directory service from the file service.





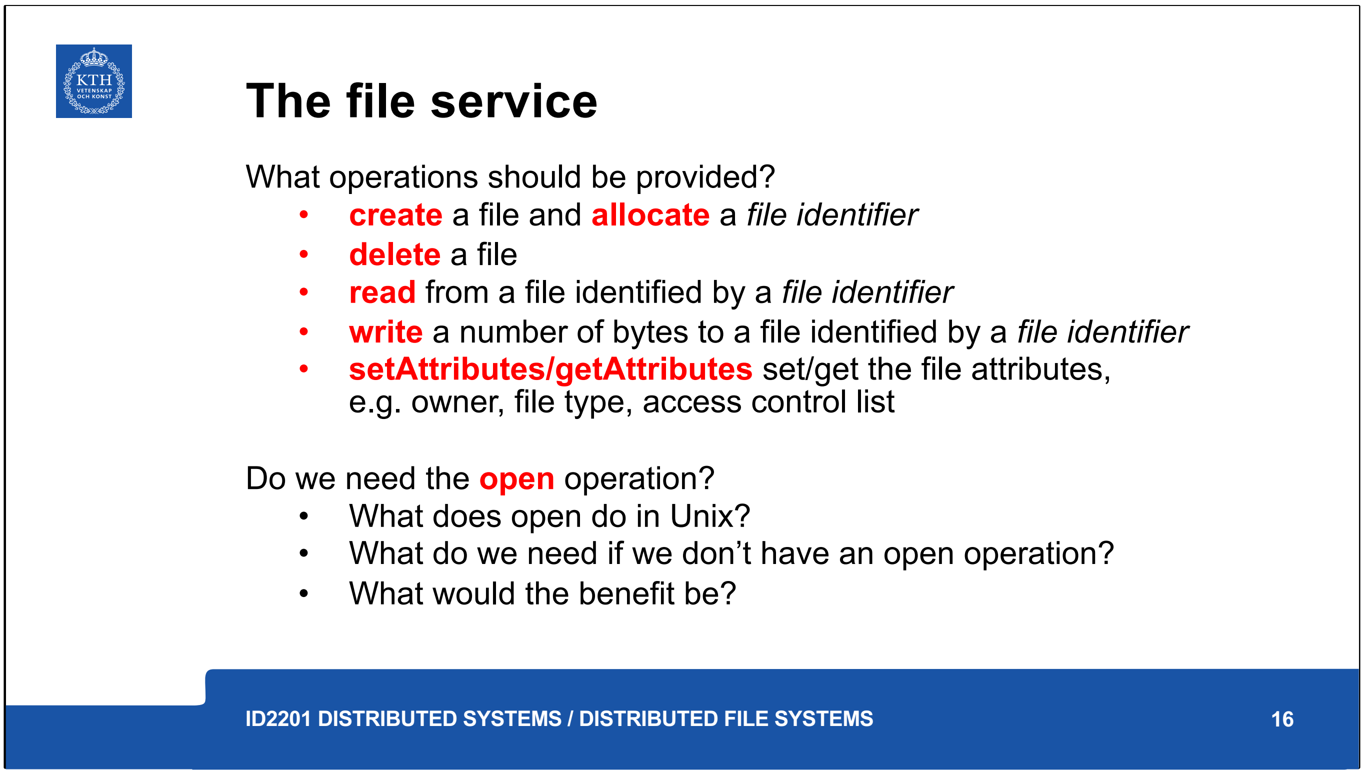
The directory service

The directory service - what operations do we need?

- **lookup** a *file identifier* given a *name* and *directory*
- **link** a *name* to a *file identifier* in a *directory*
- **remove** a link
- **list** all *names* in a *directory*

The directory service does not create nor manipulate files.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a ‘well-known’ UFID (Unique File IDentifier). Multiple names for files can be supported using the AddName (or link)operation and the reference count field in the attribute record.
In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used following a path to ensure that each part of the name, except the last, refers to a directory.



The file service

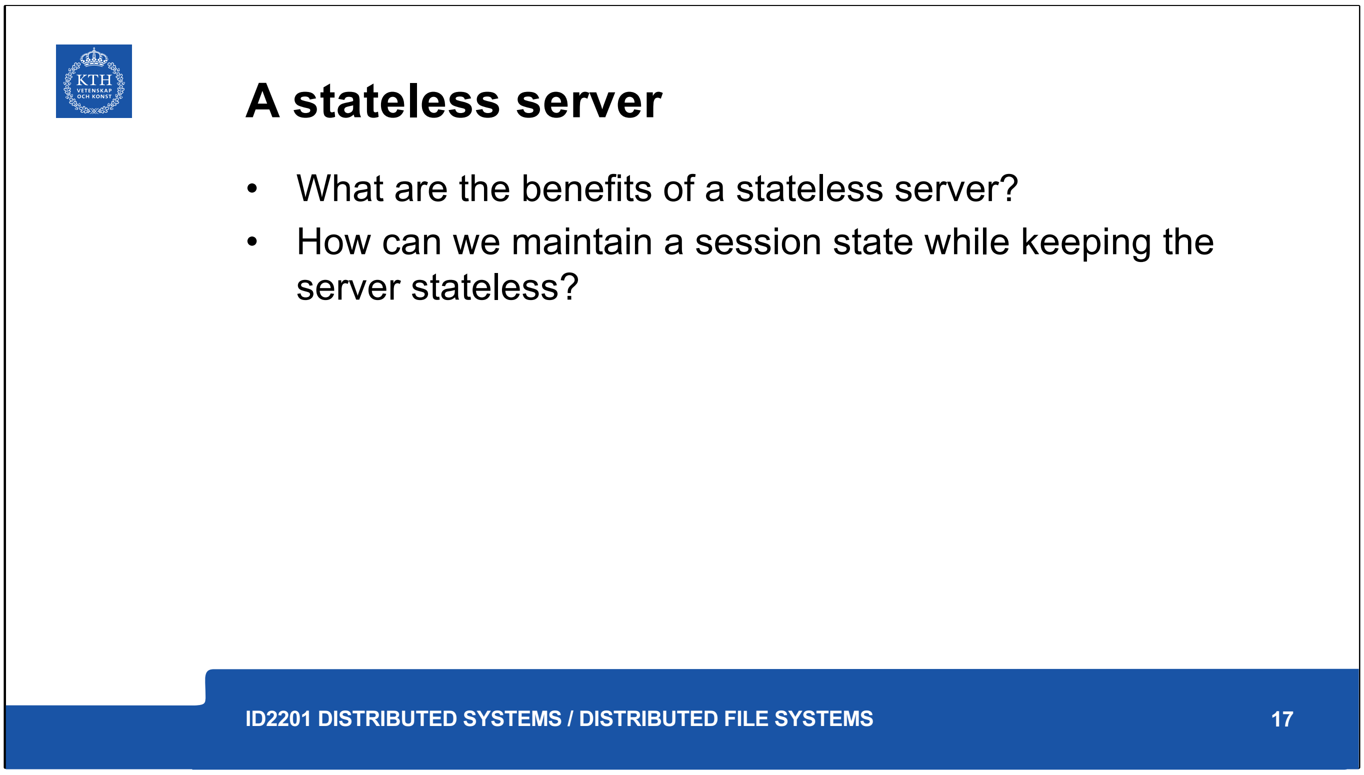
What operations should be provided?

- **create** a file and **allocate** a *file identifier*
- **delete** a file
- **read** from a file identified by a *file identifier*
- **write** a number of bytes to a file identified by a *file identifier*
- **setAttributes/getAttributes** set/get the file attributes, e.g. owner, file type, access control list

Do we need the **open** operation?

- What does open do in Unix?
- What do we need if we don't have an open operation?
- What would the benefit be?

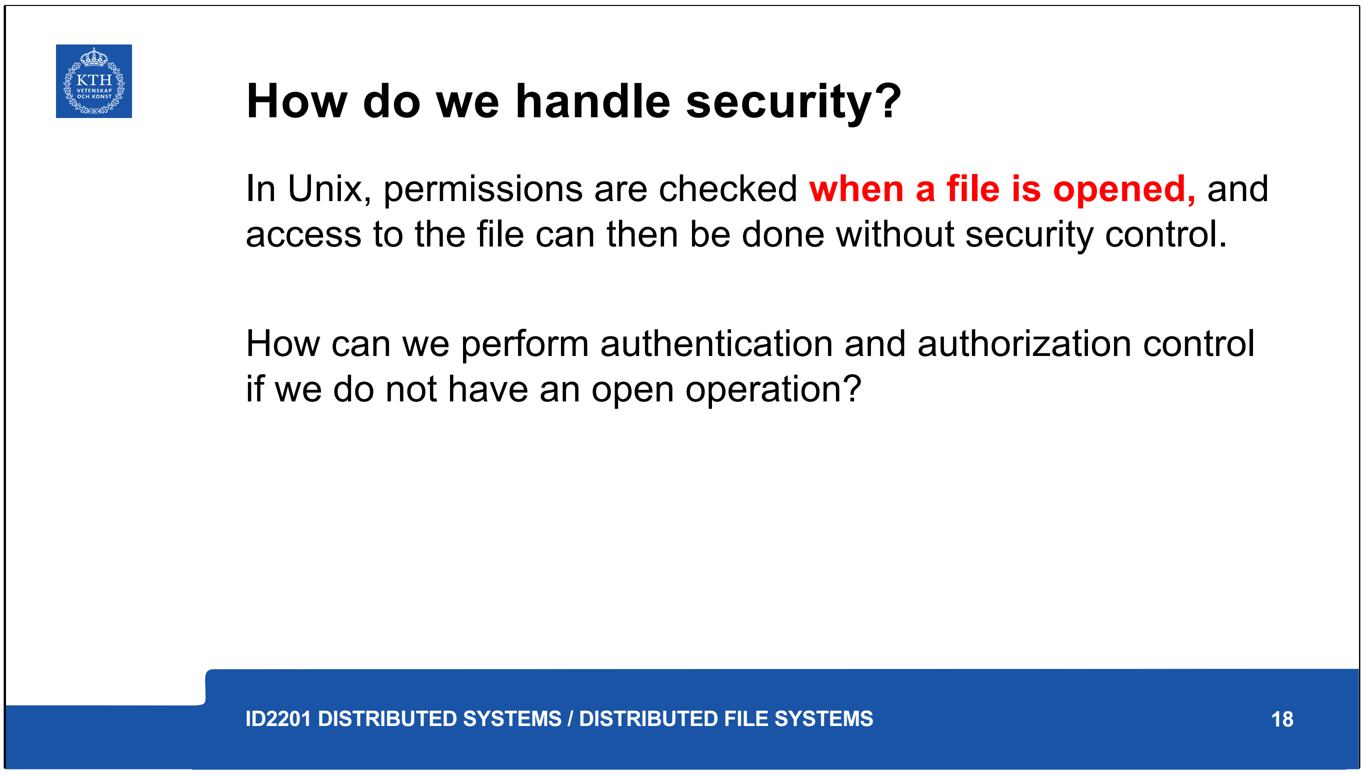
In the UNIX file system, **the user's access rights are checked against the access mode (read or write) requested in the open call** and the file is opened only if the user has the necessary rights. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.
In distributed implementations, access rights checks must be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity must be passed with requests, and the server is vulnerable to forged identities.
Two alternative approaches to have a stateless server can be adopted:
• An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
• (**Most common**) A user identity is submitted with every client request, and access checks are performed by the server for every file operation.
Read(FileId, i, n) -> Data — throws BadPosition: If 1 <= i <= Length(File): Reads a sequence of up to n items from a file starting at item i and returns it in Data.
Write(FileId, i, Data) — throws BadPosition; If 1 <= i <= Length(File)+1: Writes a sequence of Data to a file, starting at item i, extending the file if necessary.



A stateless server

- What are the benefits of a stateless server?
- How can we maintain a session state while keeping the server stateless?

Read and *Write* requests in our interface include **a parameter specifying a starting point** within the file for each transfer, whereas the equivalent UNIX operations do not.
Repeatable operations: **Except Create, the operations are idempotent**, allowing the use of at-least-once RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of Create produces a different new file for each call.
Stateless servers: The interface is suitable for implementation by stateless servers. Stateless servers **can be restarted** after a failure and resume operation without any need for clients or the server to restore any state.
To mimic it in a file server, open and close operations would be needed, and the read-write pointer's value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.
Two alternative approaches to have a stateless server can be adopted:
• An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
• (**Most typical**) A user identity is submitted with every client request, and the server performs access checks for every file operation.
Read(FileId, i, n) -> Data — throws BadPosition: If 1 <= i <= Length(File): Reads a sequence of up to n items from a file starting at item i and returns it in Data.
Write(FileId, i, Data) — throws BadPosition; If 1 <= i <= Length(File)+1: Writes a sequence of Data to a file, starting at item i, extending the file if necessary.

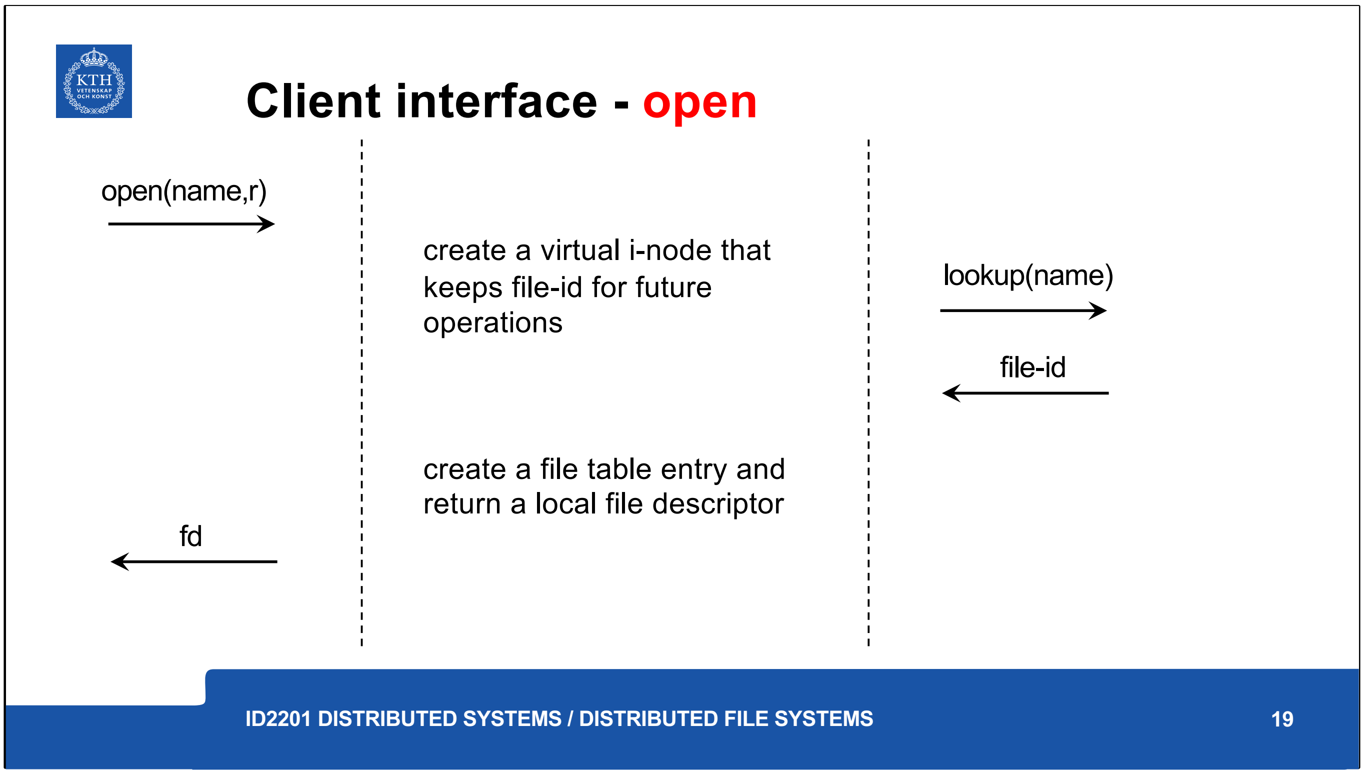


How do we handle security?

In Unix, permissions are checked **when a file is opened,** and access to the file can then be done without security control.

How can we perform authentication and authorization control if we do not have an open operation?

Two alternative approaches to have a stateless server can be adopted:
• An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
• (**Most common**) A user identity is submitted with every client request, and access checks are performed by the server for every file operation.
Each request caries user cridentials.
The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.



Client interface - open

open(name,r) →

create a virtual i-node that keeps file-id for future operations

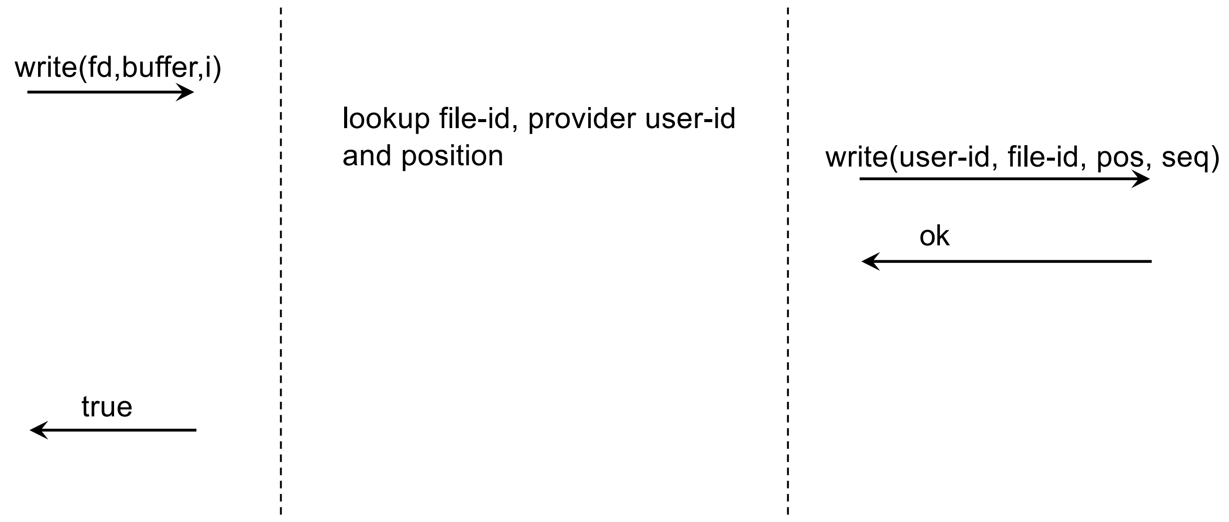
lookup(name) →

← file-id

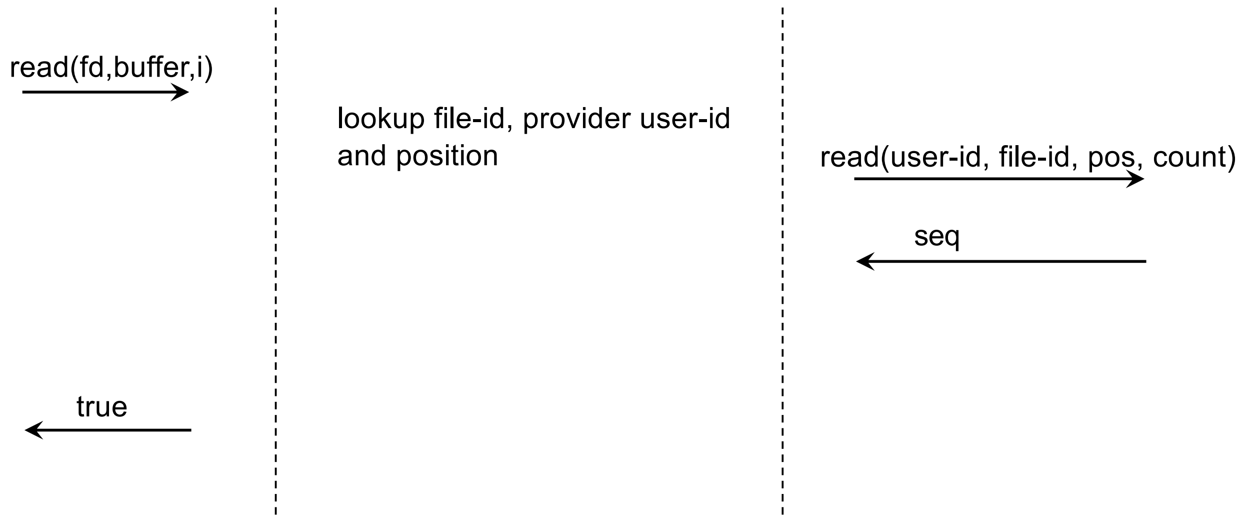
create a file table entry and return a local file descriptor

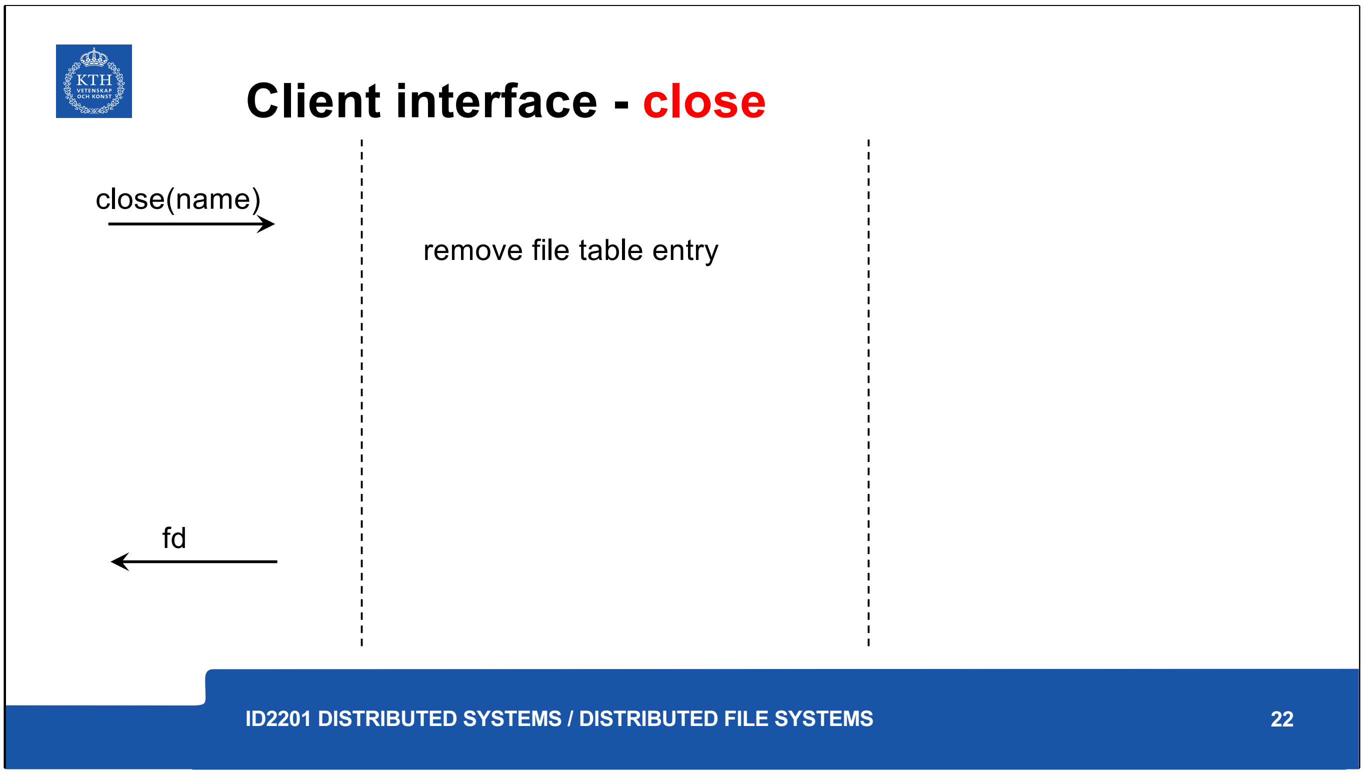
← fd

Client interface - write


write(fd,buffer,i)
lookup file-id, provider user-id and position
write(user-id, file-id, pos, seq)
ok
true

Client interface - read


read(fd,buffer,i)
lookup file-id, provider user-id and position
read(user-id, file-id, pos, count)
seq
true

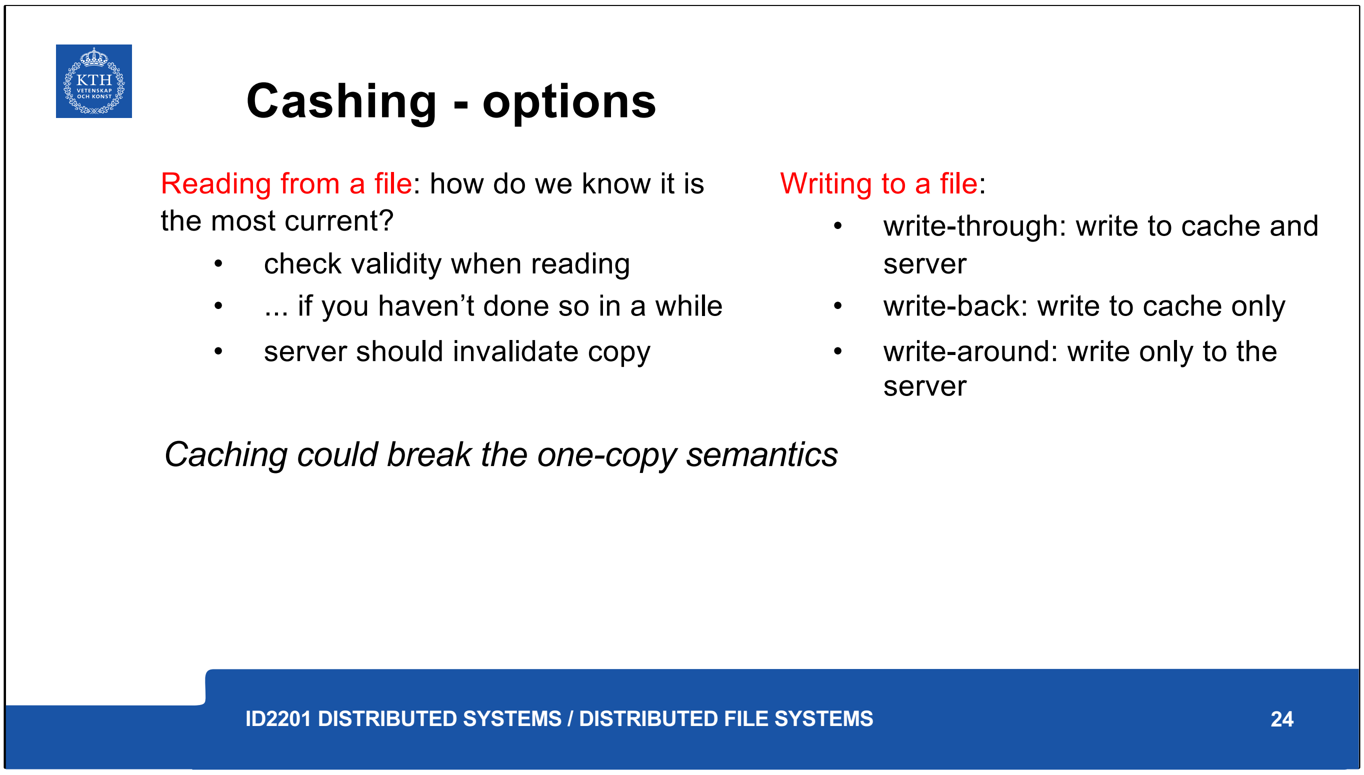


Client interface - close

Performance

Everything would be fine if it were not for performance.

Keep a local copy of the file on the client side.



Cashing - options

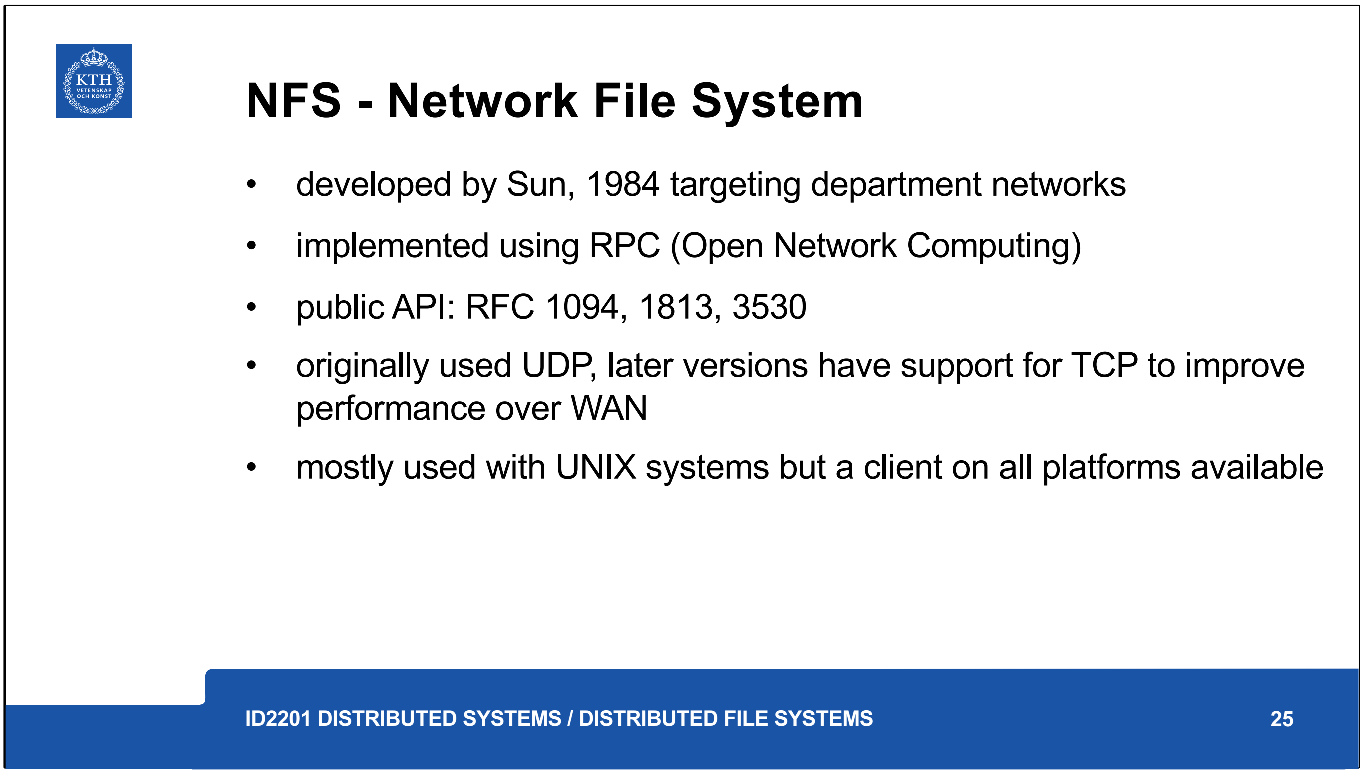
Reading from a file: how do we know it is the most current?

- check validity when reading
- ... if you haven't done so in a while
- server should invalidate copy

Writing to a file:

- write-through: write to cache and server
- write-back: write to cache only
- write-around: write only to the server

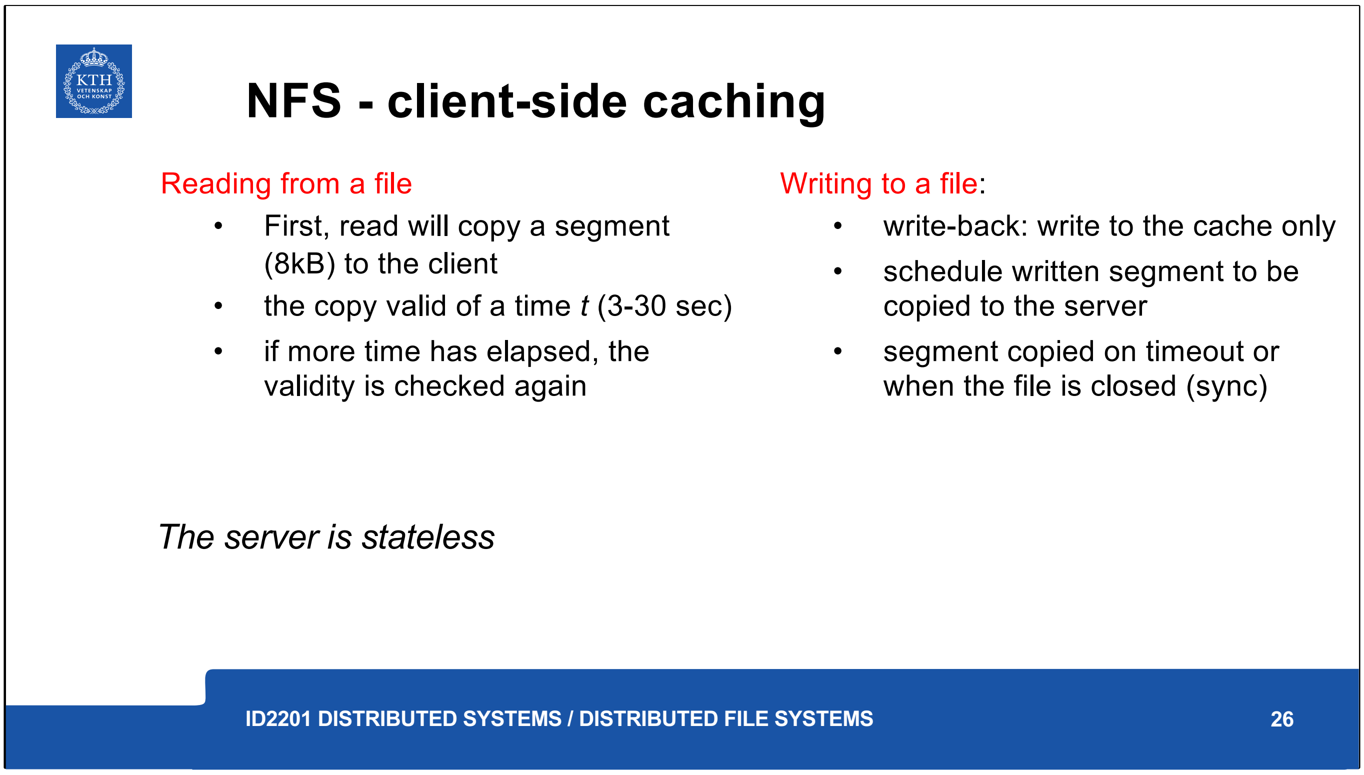
Caching could break the one-copy semantics



NFS - Network File System

- developed by Sun, 1984 targeting department networks
- implemented using RPC (Open Network Computing)
- public API: RFC 1094, 1813, 3530
- originally used UDP, later versions have support for TCP to improve performance over WAN
- mostly used with UNIX systems but a client on all platforms available

NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.
The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the filesystems currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module, or the service module for another file system).



NFS - client-side caching

Reading from a file
- First, read will copy a segment (8kB) to the client
- the copy valid of a time *t* (3-30 sec)
- if more time has elapsed, the validity is checked again

Writing to a file:
- write-back: write to the cache only
- schedule written segment to be copied to the server
- segment copied on timeout or when the file is closed (sync)

The server is stateless

ID2201 DISTRIBUTED SYSTEMS / DISTRIBUTED FILE SYSTEMS 26

The NFS client module caches the results of read, write, getattr, lookup, and readdir operations to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency (validity) of the cached data they hold.
Each cache entry has two timestamps:
A timestamp-based method is used to validate cached blocks before they are used.
Each data or metadata item in the cache is tagged with two timestamps: (1) Tc is the time when the cache entry was last validated. (2) Tm is the time when the block was last modified at the server. Formally, the validity condition at time T is: (T – Tc < t) OR (Tmclient= Tmserver), where t is the *freshness* interval
Writes are handled differently. When a cached page is modified, it is marked as 'dirty' and is scheduled to be flushed to the server asynchronously. Modified pages are flushed when the file is closed, or a sync occurs at the client, and they are flushed more frequently if bio-daemons are in use (see below). This does not provide the same persistence guarantee as the server cache, but it emulates the behavior for local writes. Bio (block i/o) daemons support read-ahead and delayed-write.

AFS - Andrew File System

- developed by Carnegie Mellon University
- clients for most platforms, OpenAFS (from IBM),
  Arla (a KTH implementation)
- used mainly in WAN (Internet), where the overhead of NFS would be prohibitive
- caching of whole files and infrequent sharing of writable files

AFS - client-side caching

Reading from a file

- copy the whole file from the server (or 64kB)
- receive a *call-back promise*
- The file is valid if the promise exists and is not too old (minutes)

Writing to a file:

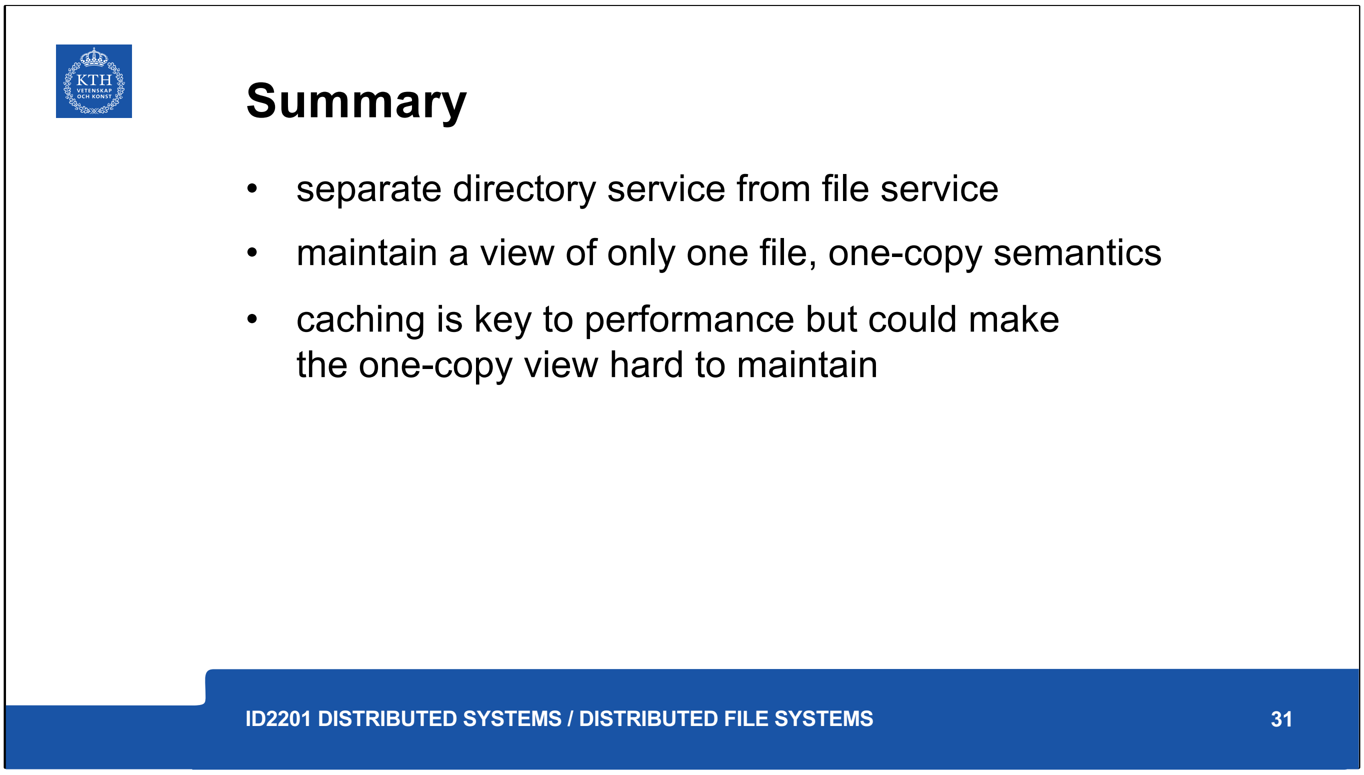
- write-back: write to the cache only
- file copied to the server when closed (sync)
- the server will invalidate all existing promises

SMB/CIFS

- ***Service Message Block (SMB)*** was originally developed by IBM but then modified by Microsoft, now also under the name ***Common Internet File System (CIFS)***.
- not only file sharing but also name servers, printer sharing etc.
- ***Samba*** is an open source reimplementation of SMB by Andrew Tridgell

SMB/CIFS client-side caching

- SMB uses *client locks* to solve cache consistency
- A client can open and lock a file; all read and write operations in the client cache.
- A read-only lock will allow multiple clients to cache and read a file
- Locks can be revoked by the server forcing the client to flush any changes
- In an unreliable or high latency network, locking can be dangerous and counterproductive



Summary

- separate directory service from file service
- maintain a view of only one file, one-copy semantics
- caching is key to performance but could make the one-copy view hard to maintain