Distributed File Systems

case studies

CS432: Distributed Systems

Spring 2014

Reading

- Chapter 12, 21 [Coulouris ’11]
- Chapter 11 [Tanenbaum ‘06]
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP 2003.

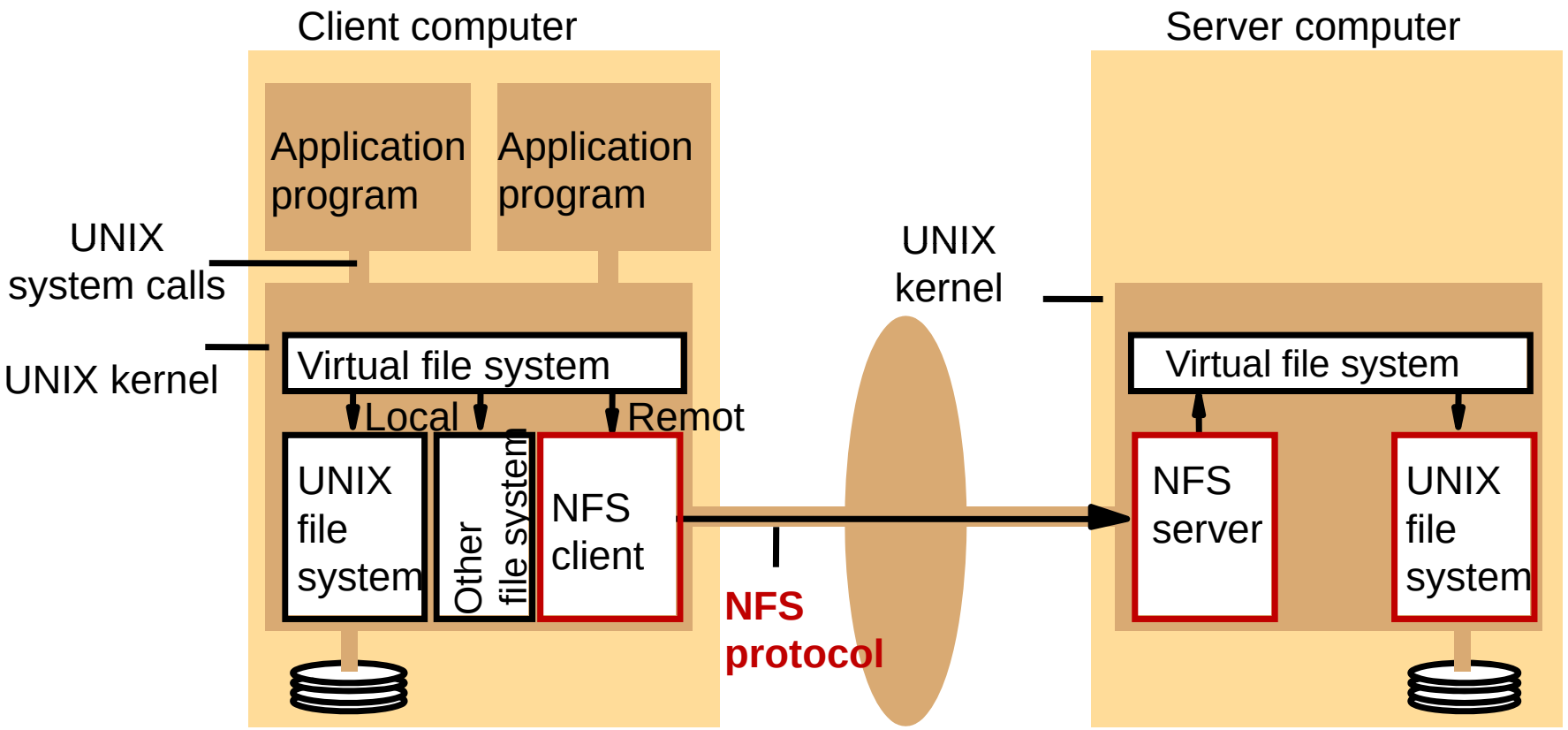
Objectives

- Learn about the following:
  - Review file systems and the main requirements for designing a distributed file system.
  - Famous architecture models of distributed file systems.
- Study the design of three file systems NFS, AFS, and GFS.

Outline

- Introduction.
- Distributed File System Requirements.
- File Service Architecture.
- **Case Studies:**
  - **Sun Network File System (NFS).**
    - Transparency.
    - Client Integration.
    - Access Control and Authentication.
    - Communication via RPC.
    - File System Mounting and Remounting.
    - Caching in NFS.
  - Andrew File System (AFS).
  - Google File System (GFS).

Sun Network File System



NFS V3 Architecture

- **NFS protocol**: a set of remote procedure calls that provide the means for clients to perform operations on a remote file store.
- **NFS server module**: resides in the kernel on each computer that acts as an NFS server.
- **NFS client module**: resides in the kernel on each client computer.
- NFS client module translates client requests referring to remote files to NFS protocol operations and then passes them to the NFS server module.

Sun's RPC

Access Transparency in NFS

- Virtual file system (VFS) is used at the client:
  - Users can access local or remote files without distinction.
  - VFS is part of the UNIX kernel.
  - Function of VFS:
    - Keeps track of the file systems that are currently available both locally and remotely.
    - Distinguishes between local and remote files.
    - Translates between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems.

File Identifiers (File Handles)

- A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file.
- Fields of the file handles:
  - **File system identifier**: a unique number that is allocated to each file system when it is created.
  - **i-node**: a number that identifies and locates the file within the file system.
  - **i-node generation number**: needed because i-node numbers are reused after a file is removed.
- VFS structure: for each mounted file system.
- v-node: for each opened file.
  - Indicates whether the file is local or remote.
  - local➔ reference to i-node. remote ➔ the file handle of the remote file.

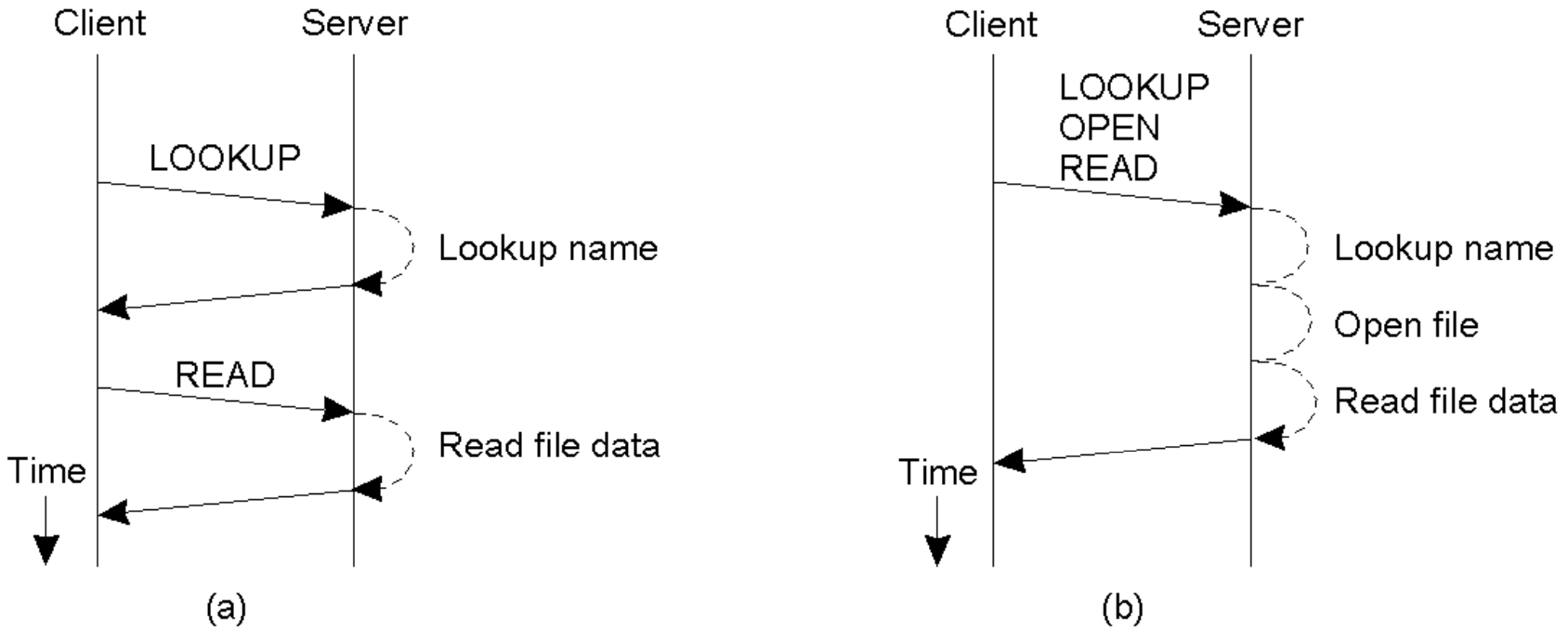
Client Integration

- The NFS client module is integrated into the kernel:
- Files are accessed via UNIX system calls.
- A shared cache of recently used blocks.
  - Same buffer cache is shared with local file system.
  - Consistency problem because of copies cached at clients.
- Encryption key used to authenticate user IDs passed to the server is retained in the kernel.

Access Control and Authentication

- NFS server is stateless:
  - it does not keep files open on behalf of its clients.
  - user's identity is checked on each request.
- Sun RPC calls are used to send requests to the NFS server.
  - User authentication information (user ID, group ID).
  - Checked against the access permission in the file attributes.
- Security loophole:
  - Conventional RPC interface at a well-known port on each host.
  - Client can modify the RPC calls to include the user ID of any user.
  - Solution:
    - DES encryption of the user's authentication information.
    - Integration of Kerberos authentication protocol.

Communication via RPC



a) Reading data from a file in NFS version 3.

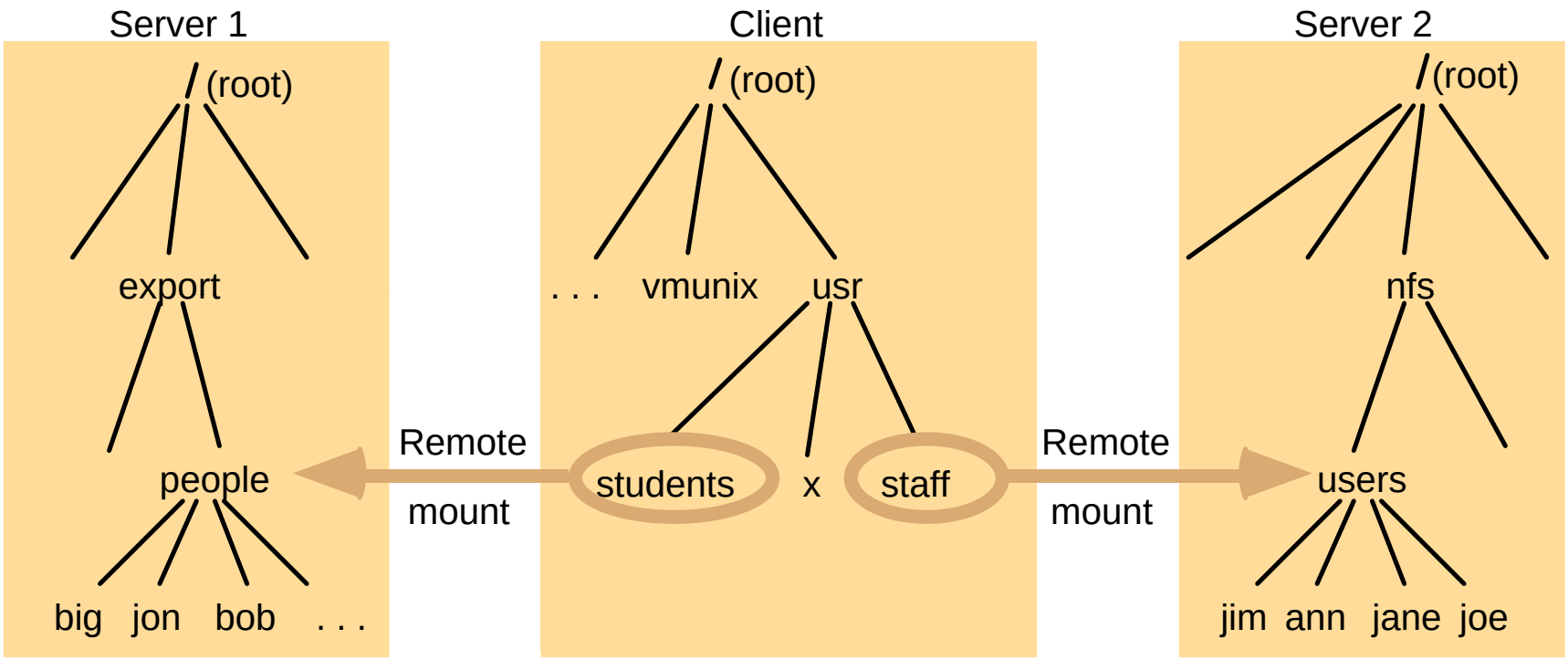
b) Reading data using a compound procedure in version 4. (RPCs can be grouped into a single request)

Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

File System Mounting

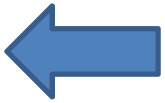
- A **mount service** process at each NFS server.
- The file /etc/exports provides names of local file systems that can be **remotely** mounted.
- Access list: hosts that are permitted to mount the file system.
- Clients use a modified mount command: remote host's name, the pathname of a directory in the remote file system, and the local name with which it is to be mounted.
- Mount Protocol (RPC-based):
  - Given a directory pathname, the file handle of the specified directory is returned (permissions are checked).
  - Location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Example of File System Mounting

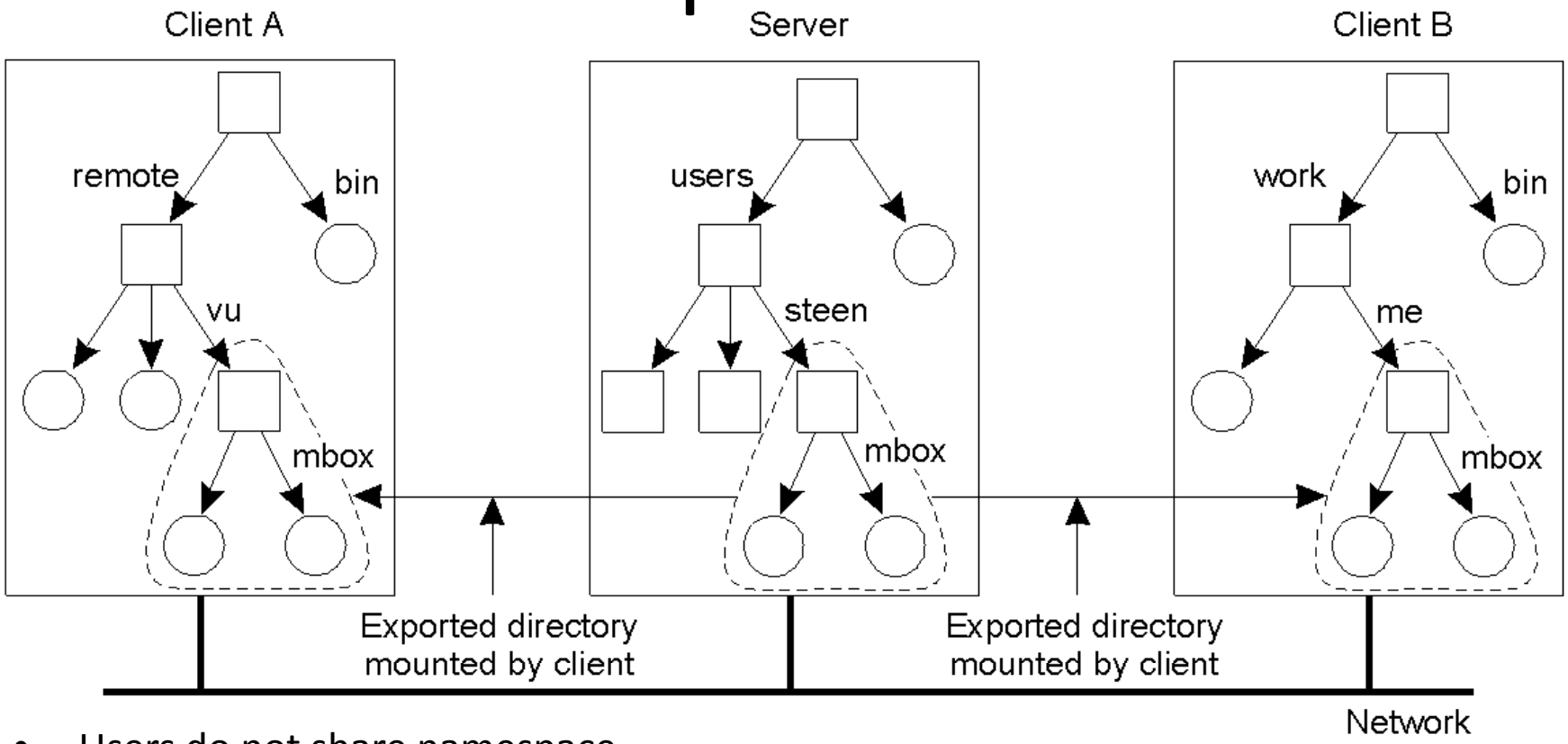


1. Client mounts remote file system in its local file system.
2. Mount part of the remote file system.

Hard-Mounted vs. Soft-Mounted File System

- Hard-Mounted:  More commonly used
  - A user-level process accessing a file is suspended until the request can be completed.
  - If server fails, user-level processes are suspended until the server restarts (requests are retried).
  - Programs are unable to recover gracefully when an NFS server is unavailable for a significant period.
- Soft-Mounted:
  - NFS client module returns a failure indication to user-level processes after a small number of retries.

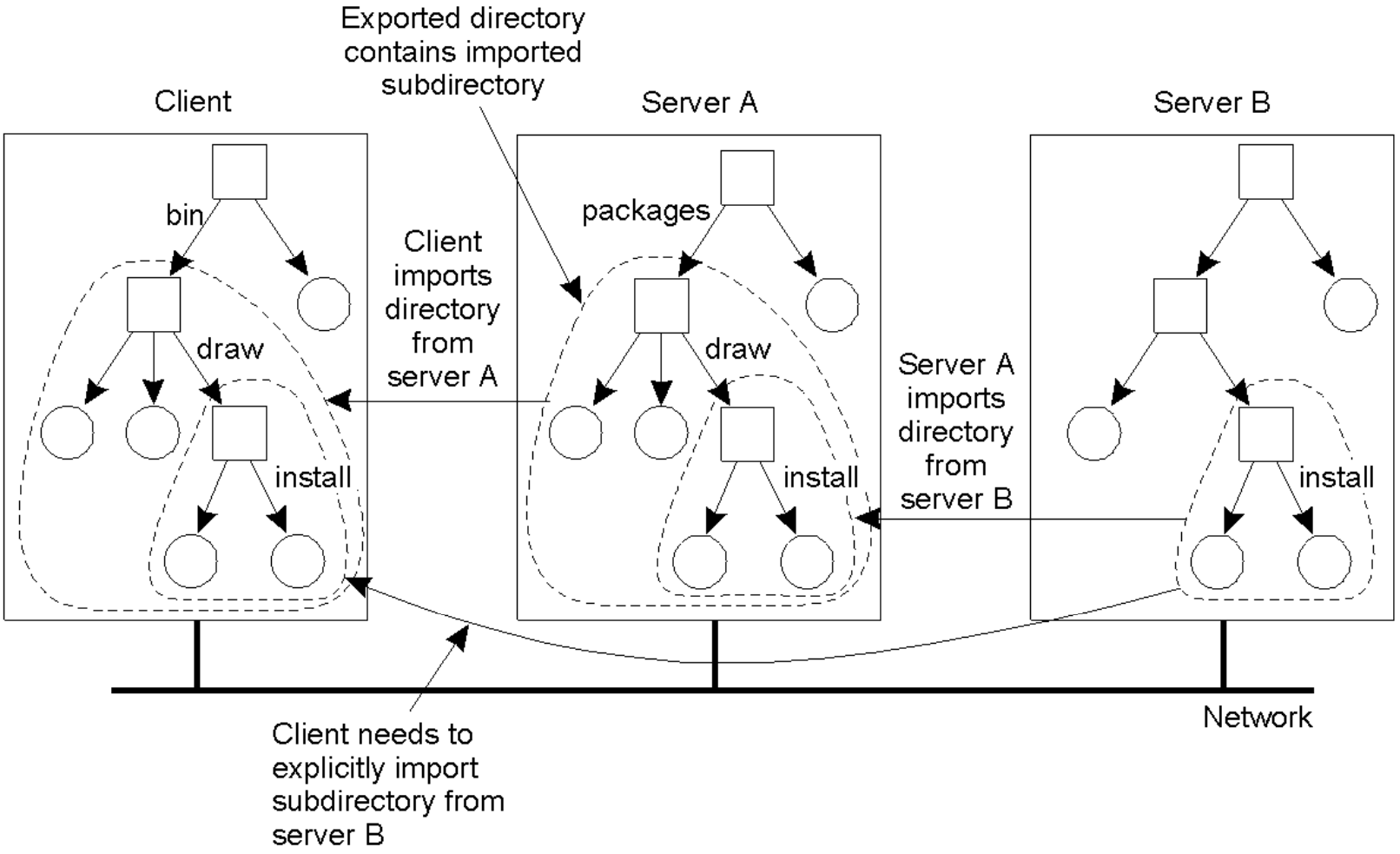
Mounting Same Directory by Multiple Clients



- Users do not share namespace.
- Example: /remote/mbox at A is the same as /work/mbox at B.
- Sol: standardization.

Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Mounting nested directories from multiple servers in NFS

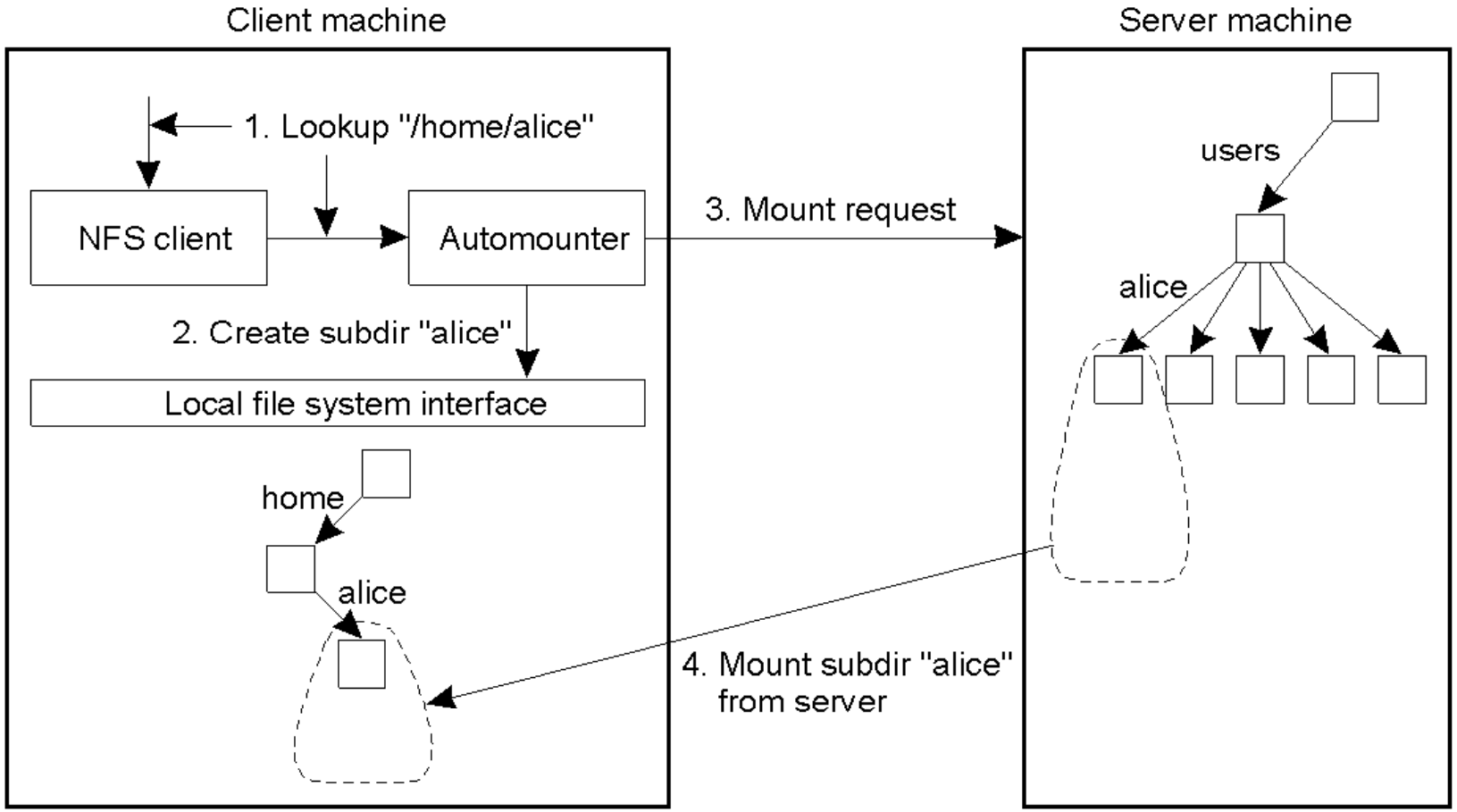


Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Pathname Translation

- Unix use a step-by-step process to translate multi-part file pathnames to i-node.
- Directory name cannot be translated at server.
  - File name may cross a mounting point (directories holding multiple mounted file systems).
- Translation process at the client:
  - Pathnames are parsed.
  - Translation is done iteratively.
  - Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate **lookup request** to the remote server.

Automounting



Mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client.

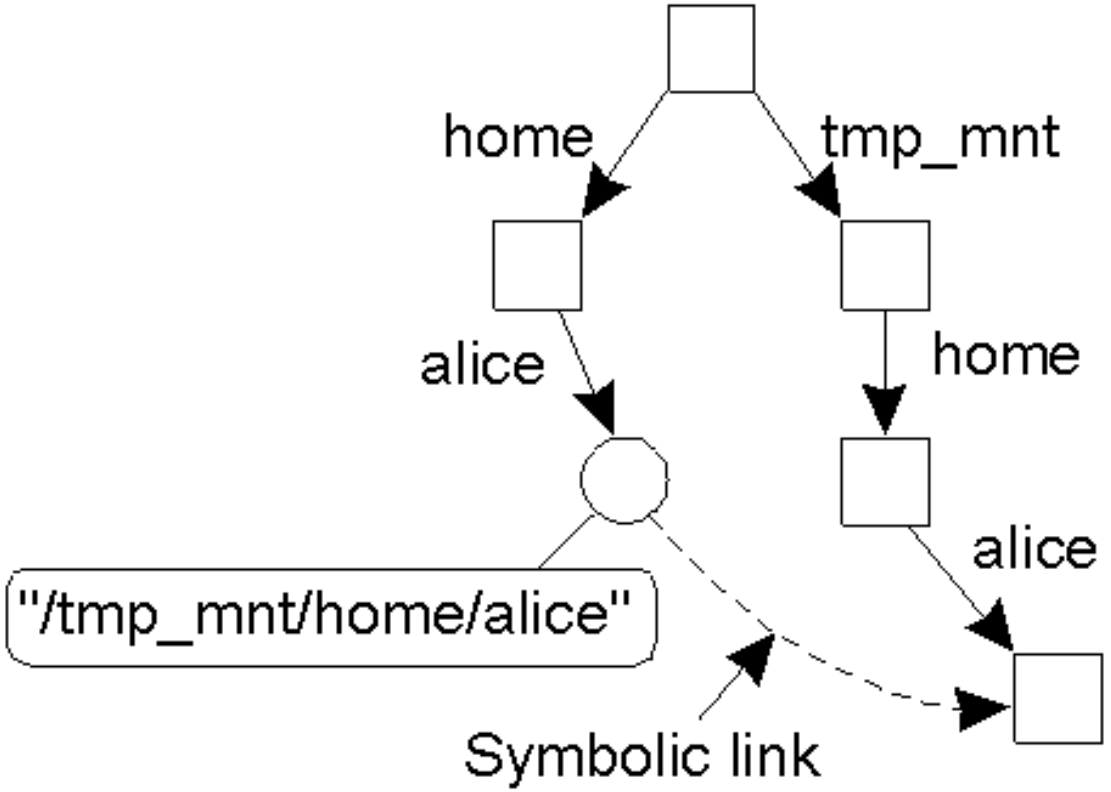
Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Automounting Procedure

- The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers.
- NFS client module attempting to resolve a pathname:
  - NFS client passes to the local automounter a *lookup() request* that locates the required file system in its table and sends a 'probe' request to each server listed.
  - The file system on the first server to respond is then mounted.
  - The mounted file system is linked to the mount point using a symbolic link, so that accesses to it will not result in further requests to the automounter.

Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002

Using Symbolic Links with Automounting



Server Caching in NFS

- Caching in server and client is essential to achieve performance.
- Disk caching as in non-networked file systems.
- Read operations: simple, no problems introduced.
- Write operations: consistency problems.
- Write- through caching:
  - Store updated data in cache and written on disk before sending reply to client .
  - Relatively inefficient if frequent write operations occur.
- Write is persistent when commit operation is performed:
  - Stored only in cache memory.
  - Write back to disk only when commit operation for file received.

More commonly used

Client Caching

- read , write, getattr, lookup and readdir operations are cached.
- Potential inconsistency: the data cached in client may not be identical to the same data stored on the server.
- Timestamp-based scheme used in polling server about freshness of a data object:
  - Tc: time cache entry was last validated .
  - $Tm_{client/server}$: time when block was last modified at the server as recorded by client/ server .
  - t: freshness interval.
  - Freshness condition: at time T

    $[(T - Tc) < t] \vee [Tm_{client} = Tm_{server}]$ (validity condition)

    - if (T - Tc) < t (can be determined without server access), then entry presumed to be valid .
    - if not (T - Tc) < t, then Tm_{server} needs to be obtained by a getattr call.
    - if $Tm_{client} = Tm_{server}$, then entry presumed valid; update its Tc to current time, else obtain data from server and update Tm_{client}

Write Operation of Cached Data

- 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.
- Bio-daemons: perform read-ahead and delayed-write operations.
  - Notified after each read request, and it requests the transfer of the following file block from the server to the client cache.
  - In the case of writing, the bio-daemon sends a block to the server whenever a block has been filled by a client operation.
  - Directory blocks are sent whenever a modification has occurred.
- Bio-daemon processes improve performance:
  - client module does not block waiting for **reads to return** or **writes to commit at the server**.

Outline

- Introduction.
- Distributed File System Requirements.
- File Service Architecture.
- **Case Studies:**
  - Sun Network File System (NFS).
  - **Andrew File System (AFS).**
    - **Introduction.**
    - **Design Characteristics.**
    - **AFS Architecture.**
  - Google File System (GFS).

Andrew File System (AFS)

- Developed at Carnegie Mellon University (CMU) as a joint work with IBM.
- Compatible with NFS.
- AFS servers hold ‘local’ UNIX files.
- Local Files are referenced by NFS-style file handles (rather than i-node numbers).
- Differences between AFS and NFS:
  - Most important design goal of AFS is scalability.
  - Key strategy for scalability: caching of whole files at client nodes.

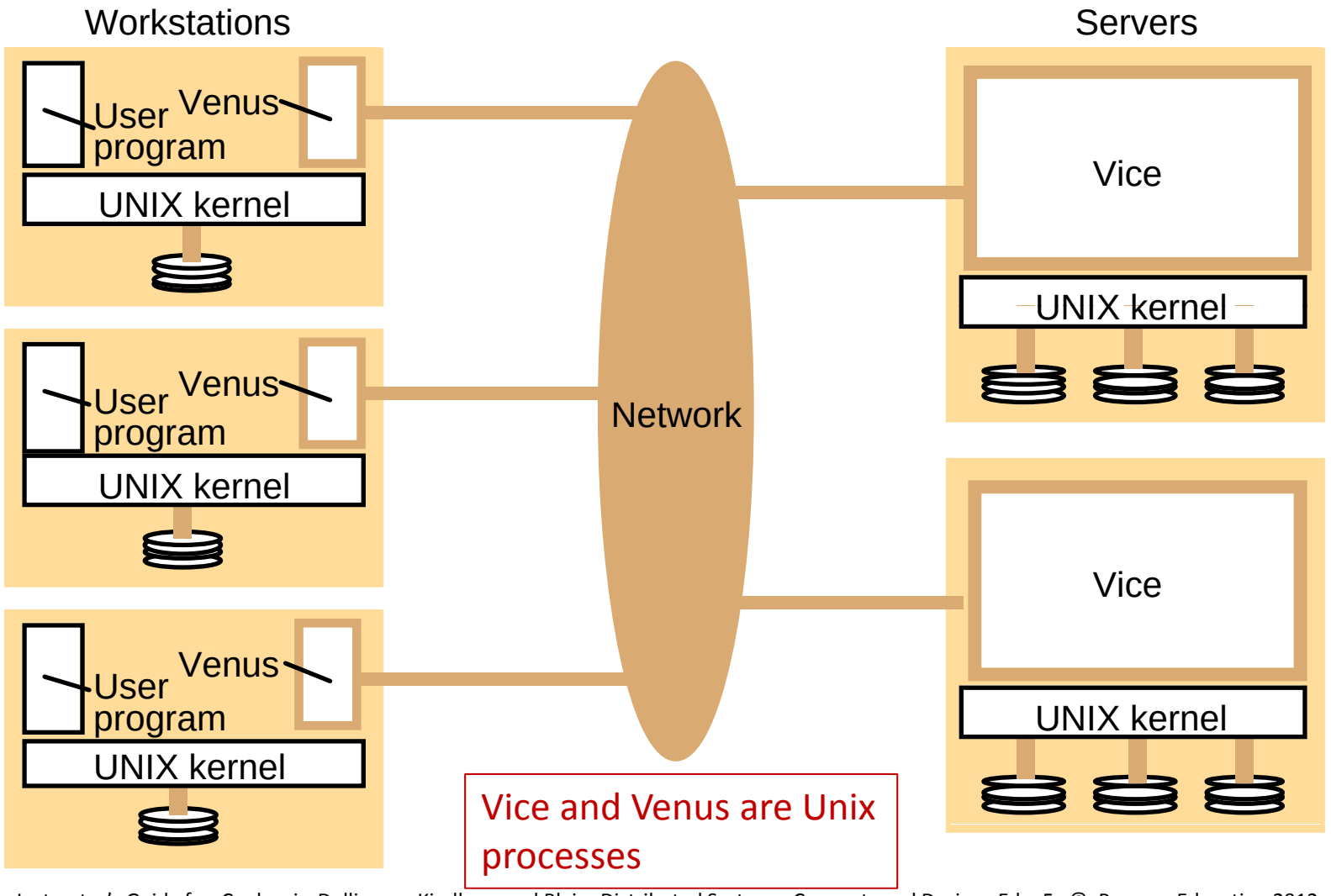
AFS Design Characteristics

- Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers.
  - AFS-3: files larger than 64 kbytes are transferred in 64-kbyte chunks.
- Whole-file caching: file/chunk transferred to a client is cached on local disk.
  - Most recent files are cached (even after the user close them).
  - The cache is permanent (survive reboots).
  - Local copies are used whenever possible.

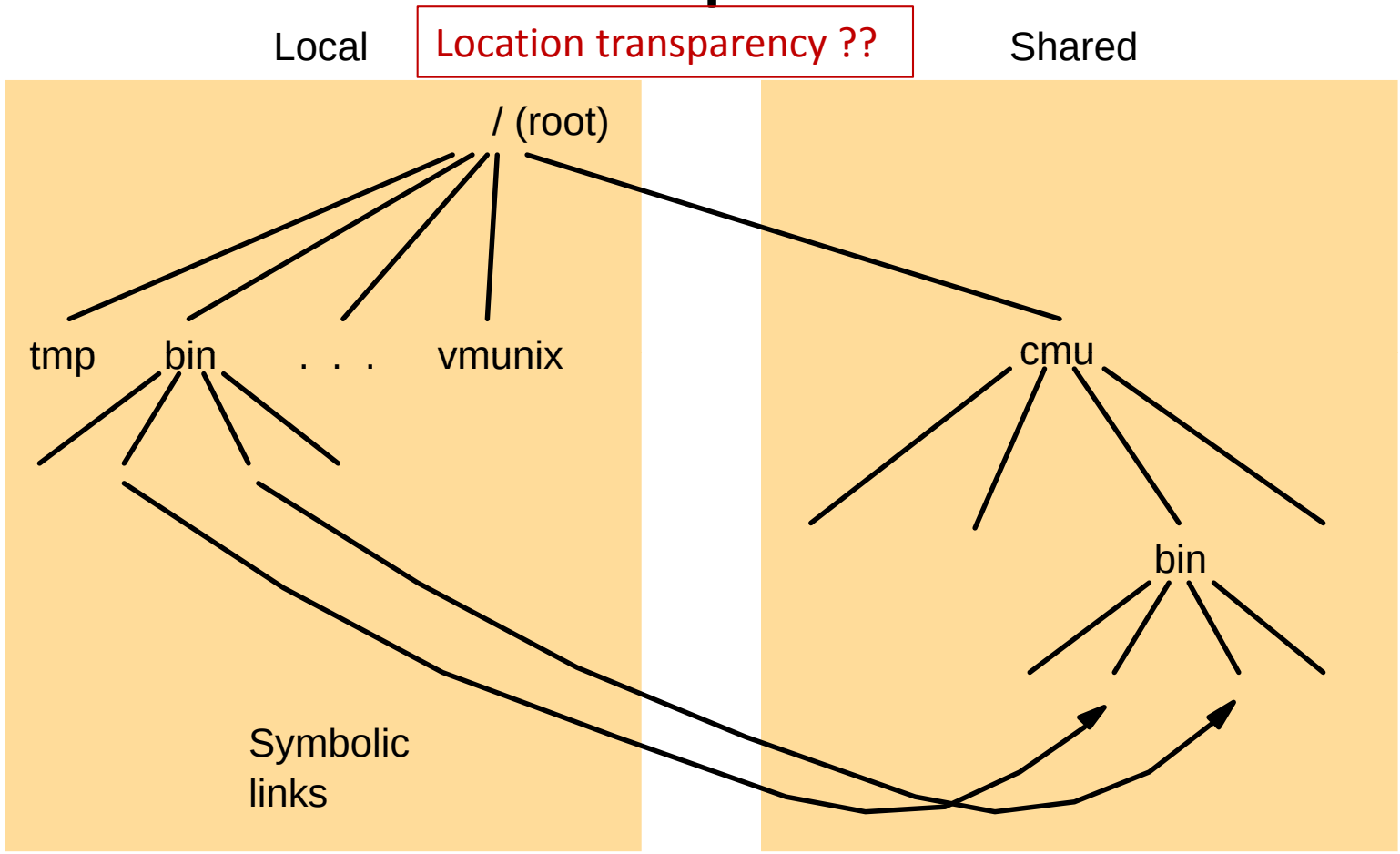
Observed Performance of AFS

- Locally cached copies of shared files that are infrequently updated and files accessed by only a single user remain valid for long periods of time.
- Disk space is allocated for the cache (e.g., 100MB).
- Assumptions based on UNIX workloads:
  - Files are small, less than 10 KB.
  - Read are more common than write operations (x 6).
  - Sequential access is more common.
  - Most files are read and written by only one user. Shared files are updated by one user.
  - Files are referenced in bursts. Recently referenced files have high probability of being referenced again.

AFS Architecture



File Name Space in AFS



System calls referring to shared name space are intercepted and passed to Venus.

Callback Mechanism

- **Callback promise** : a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file.
- States of callback promise: valid, cancelled.
- Callback promise is assigned the “cancelled” state when a file is changed at a workstation, and the vice issues a callback.
- Callback promises in the cancelled state are fetched from the Vice server.
- Callback promises of cached files are checked after a restart or recovery from a failure (timestamps are compared).
- Callback tokens are renewed when opening a file that was not accessed for a time interval greater than ***T***.

Implementation of File System Calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Outline

- Introduction.
- Distributed File System Requirements.
- File Service Architecture.
- **Case Studies:**
  - Sun Network File System (NFS).
  - Andrew File System (AFS).
  - **Google File System (GFS).**
    - **Goals.**
    - **Design Assumptions.**
    - **GFS Architecture.**
    - **Design Choices.**
    - **Write Control and Data Flow.**

Goals – Shared with Previous Distributed File Systems

- Performance
- Scalability
- Reliability
- Availability

Goals - GFS

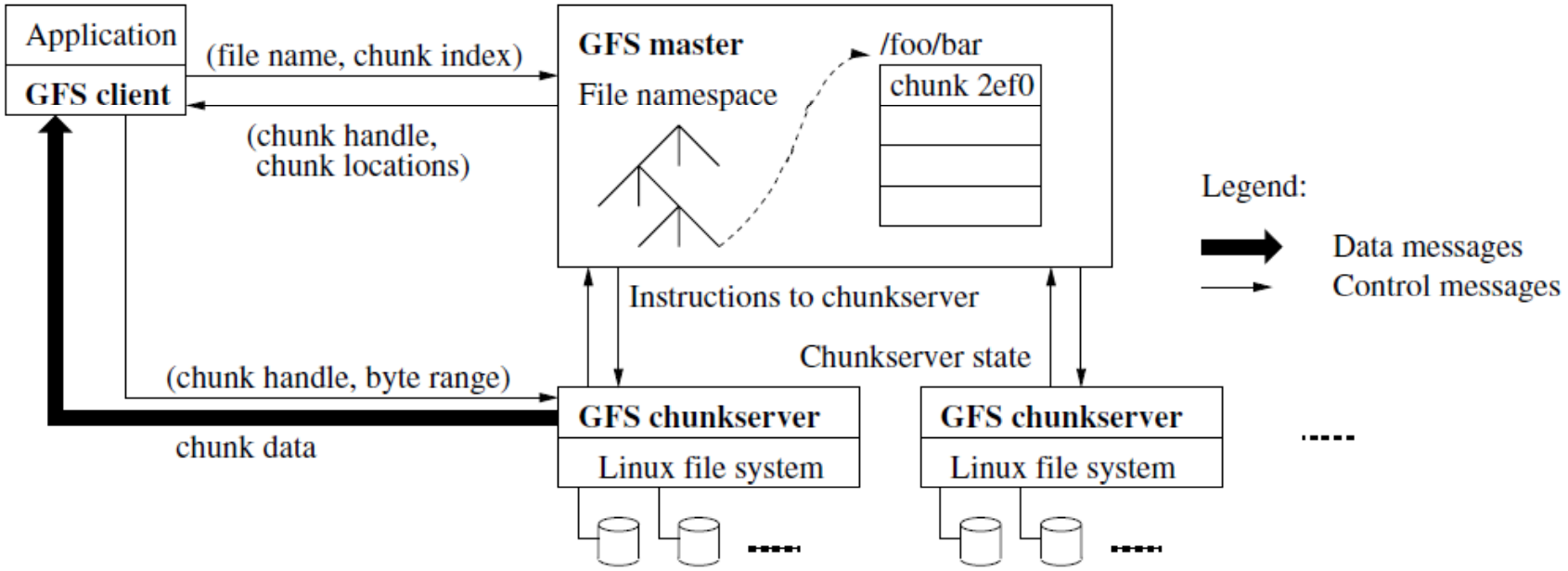
- Provides error detection, fault tolerance, and automatic recovery.
- Revisit design assumptions and parameters (e.g. block size).
- Files mainly change because of appending new data.
- Co-designing the application and file system.
- Runs on inexpensive commodity hardware.
- Delivers high aggregate performance to a large number of clients.

Multiple GFS clusters (2003)
1000 storage nodes
300 TB disk storage
Hundreds of clients

Design Assumptions

- File system is built from many inexpensive components – frequent failures.
- File sizes: 100 MB, multi-GB, and small files.
- Workload consists of two types of reads:
  - Large streaming reads
  - Random reads
- Writes in the workload:
  - Many large sequential writes that append data to files
  - Small writes at arbitrary locations (supported but not efficient)
- Concurrent writes (appending) to the same file
- High bandwidth: process data in bulk in a high rate

GFS Architecture



- Fixed-size chunks, 64-bit chunk handle
- Chunks are replicated on multiple chunkservers (3 replica)
- Master maintains file system metadata (files-chunks mapping, chunk location,..)
- Master and chunkservers communicate through ***HeartBeats***
- **No caching of files.**

Design Choices

- Single master.
- Chunk size, 64MB.
- Metadata stored in server's memory.
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of each chunk's replicas
- Guaranteed consistency:
  - File namespace mutations are atomic.
  - It is guaranteed that after a series of mutations to the same chunk, its status is defined
  - Only appends are allowed
  - Master detects failures of chunks

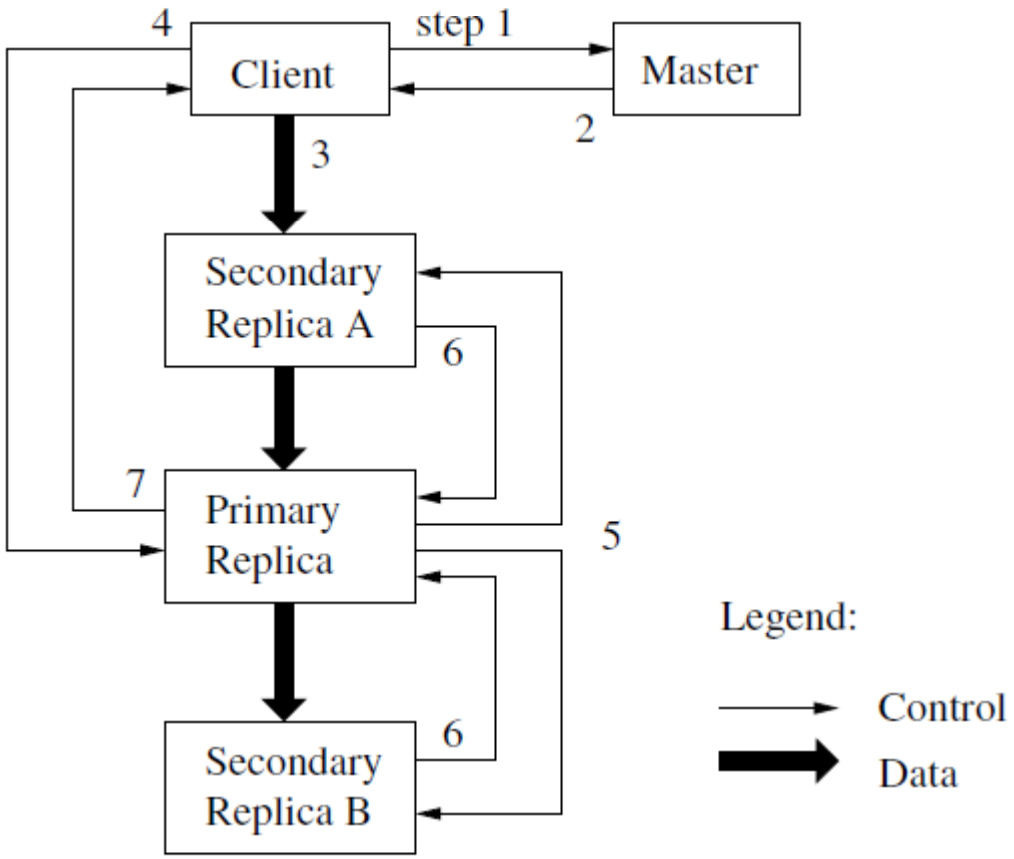
Metadata at the Master

- Types of metadata
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of each chunk's replicas
- Metadata is stored in memory
  - Advantage: master operations are fast
  - Disadvantage: the number of chunks is limited
- Chunk locations
  - Information pulled from chunkservers at startup
  - Maintained through HeartBeats
- Operation log to keep critical metadata changes

Leases and Mutation Order

- The master grants a chunk lease to one of the replicas (primary)
- The primary picks a serial order for all mutations to the chunk
- All replicas follow the order chosen by the primary

Write Control and Data Flow



Thank You