Ciências ULisboa

Programação em Sistemas Distribuídos

MEI-MI-MSI
2018/19

4. Advanced Distributed Systems Services

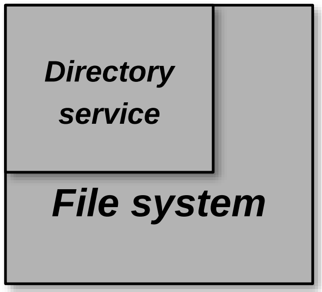
Prof. António Casimiro

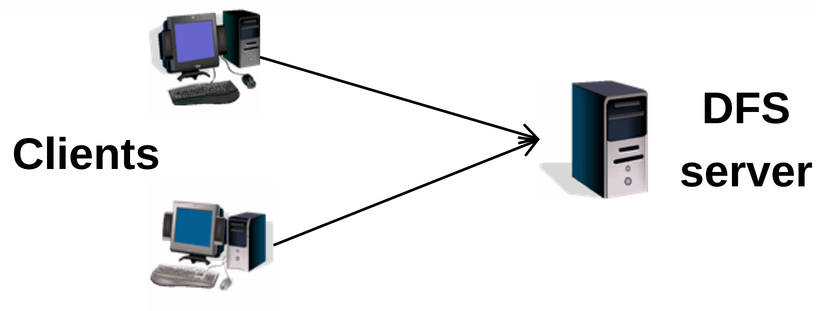
Distributed Systems Services

- **Distributed File Services**
  - **(NFS,AFS,CODA,GFS)**
- Name and Directory Services
  - (X.500)
- Time Services
  - (NTP)

Distributed File Services

Distributed file systems

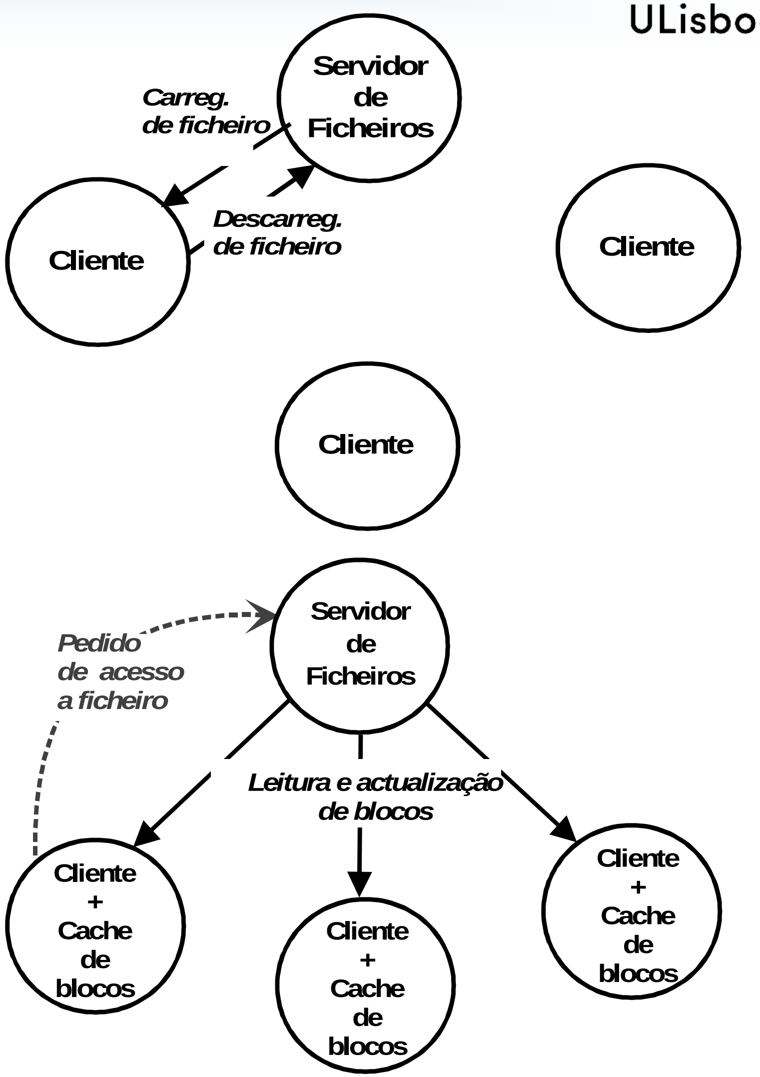




- Design questions:
  - How does a client know which server is storing the file being accessed?
  - The directory service should be distributed or centralised?
  - How many bytes should be sent in each client-server interaction (i.e., which should be the block size)?
  - The cache should be on the client side, server side, or both?
  - If another client has the same file on cache, how to ensure consistency?
  - Should the servers retain information about clients that opened files?

DFS models

- Upload/download
  - Local image (persistent)
  - Good behaviour in large-scale
  - Supports disconnected operation
- Remote access
  - Local volatile cache
  - Stateful servers
  - Weak consistency semantics

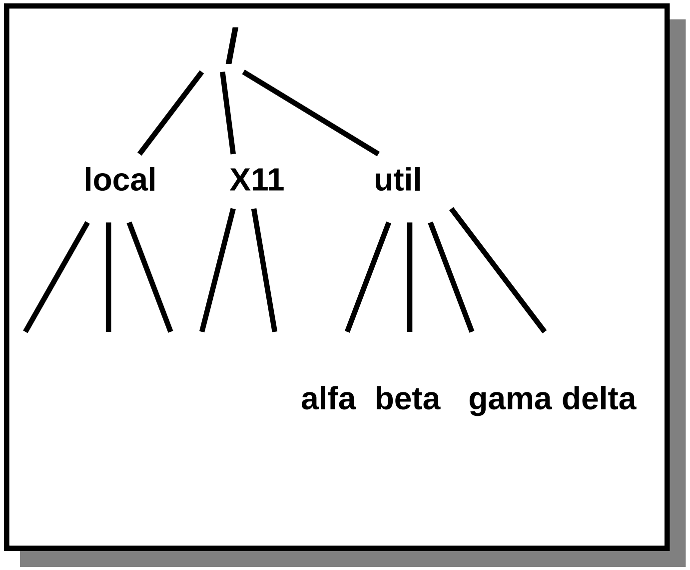


Case studies

- NFS – Network File System (sec. 4.2)
  - Classical DFS that follows the client-server model of remote-access
- AFS – Andrew File System (sec. 4.2)
  - DFS based on upload/download to support more clients in large-scale
- CODA (sec. 4.2)
  - Improvements to AFS to support disconnected operation
- GFS – Google File System (SOSP’03 paper)
  - Large-scale system to store big files that are only updated through append operations

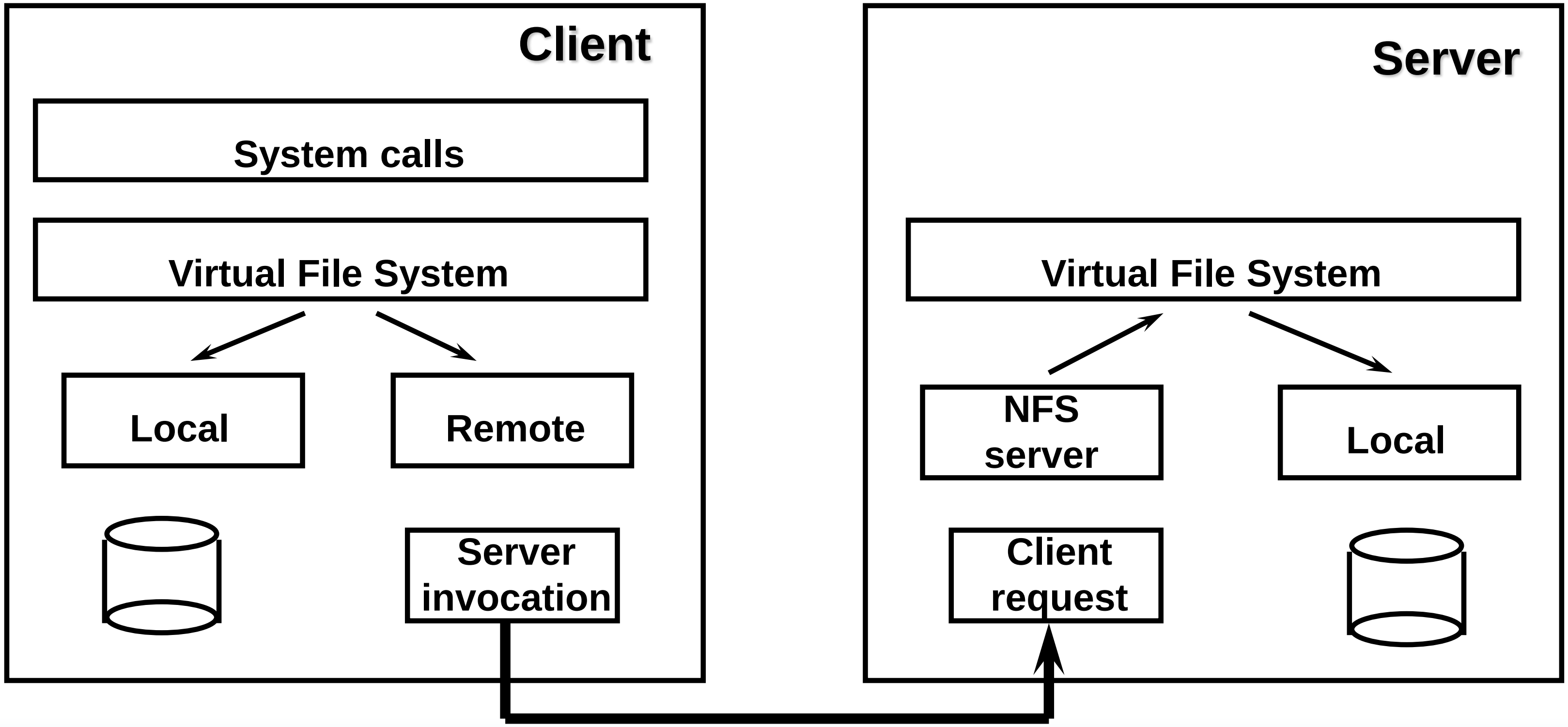
Distributed file systems
Case study: Sun NFS

- NFS: Network File System
  - **Access transparency**:
    - Client interface is the UNIX sys calls
  - **Location transparency**:
    - Remote FS is added (mounted) on the local FS:
      - Remote server exports FS
      - Client (remotely) mounts FS
  - **Failure transparency**:
    - NFS is stateless, all operations are idempotent
  - **Migration transparency**:
    - Supported through mount, whose tables can be updated
  - **Client/server operation**:
    - NFS client and server interact through RPC



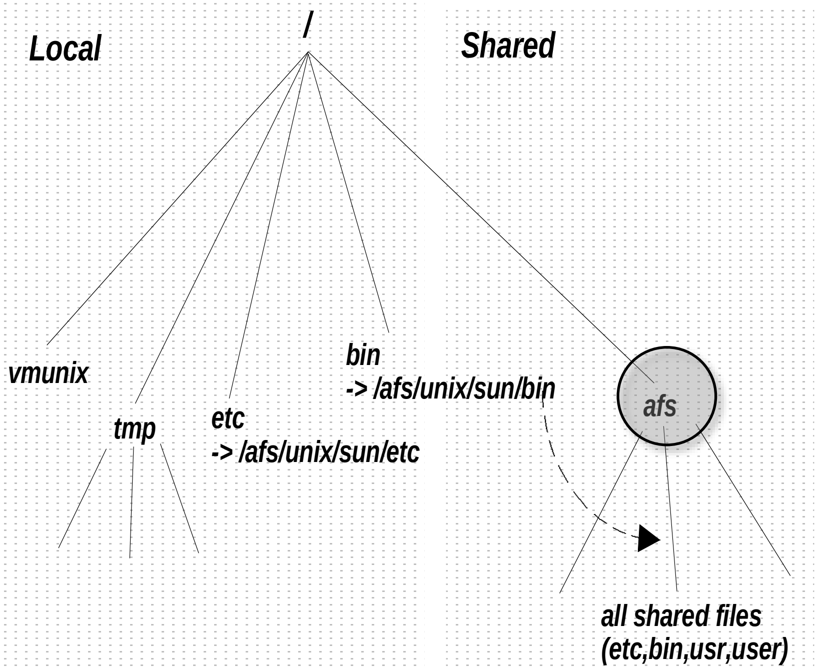
NFS architecture

- Virtual File System (VFS)
- Security and access control (**mount access** and **file access**)
- Mount and automount
- Client and server cache (blocks, performance oriented, weak consistency)



Distributed file systems
Case study: Andrew File System

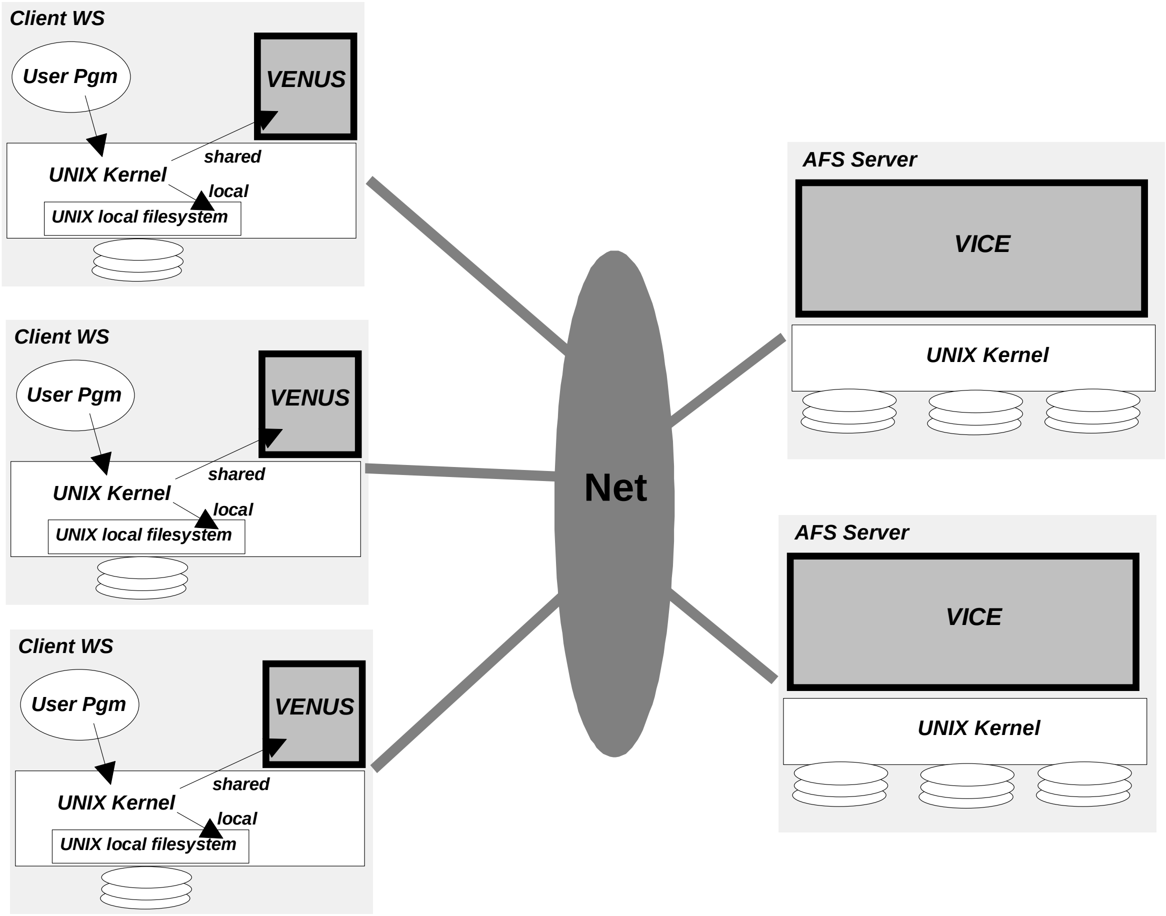
- Scalability of serviced clients
- Based on cells, authentication/ authorization domains (**Kerberos**)
- Cells are composed of volumes (collections of directories and files)
- Entire files are transferred
- **Persistent cache** on client side
- Consistency management based on callbacks
- Replicated read-only files (availability)
- Supports client mobility
  - Users can access their files from anywhere



AFS design principles

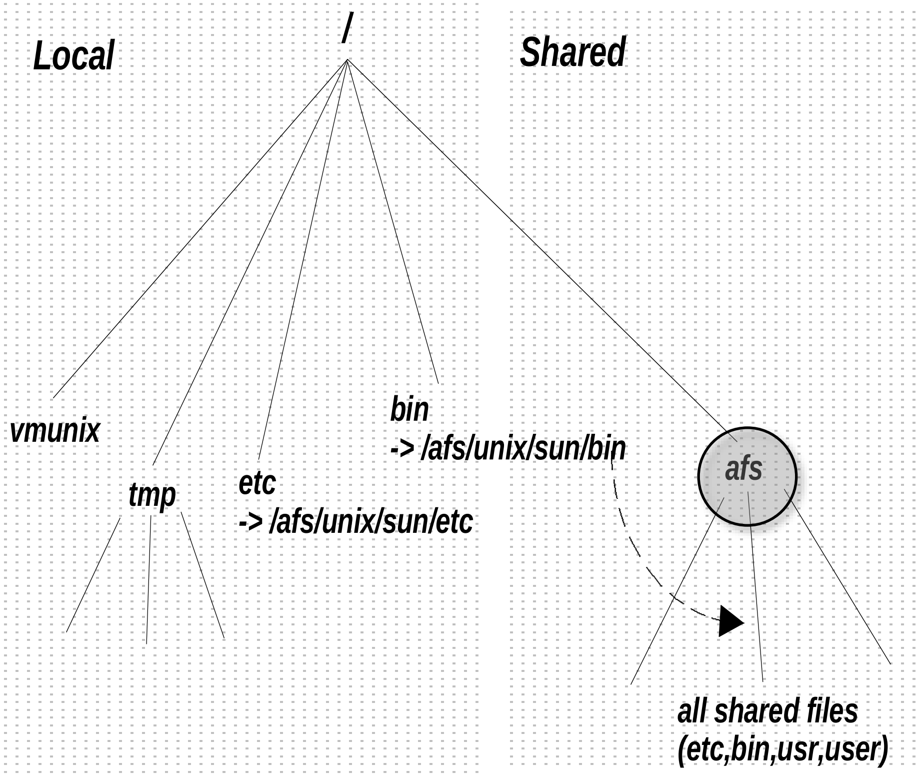
- File access through standard UNIX primitives
- Accessed files are local, stored on disk
- When a file is opened, its entire content is transferred from the server
- When closed, it is kept on a client side persistent cache
- When reopened, local copy is used if possible
- Read/write (RW) files – concurrently opened by several clients, with only one copy on the server side – consistency to be discussed next
- Read/only (RO) files – a file may have many RO copies on any server, but only one RW copy
  - When RW copy is updated, an explicit release command is generated to update RO copies

AFS Architecture



Local file structure

- A part of the files is local to the client FS
- Another part is stored on the server and local copies are made on a dedicated partition, which is locally mounted on the local FS
- The dedicated partition is controlled by the AFS client (Venus)
- Path/address translation is performed by a localization server, which is replicated in all AFS servers (Vice)



AFS strategy
Technical assumptions

- Average file size is small (~10KB)
- Reads are more frequent than writes (~6:1)
- Sequential access more frequent than random access
- Most files are of type “one writer/multiple readers”
- Files are access several times in a row, which increases the chances that the copy on cache is still valid

The assumptions were validated in practice through the evaluation of a real distributed file system on the Carnegie-Mellow University in Pittsburgh/USA

AFS strategy
Considered activity types

- Shared read-only repositories, rarely updated
  - Local copies remain valid for a long time
- Shared read-write repositories, infrequent updates by a single writer
  - Local copies are occasionally invalidated
  - Write copy allows local access for each change
- Non-shared read-write repositories, possibly with frequent updates
  - Copy always remains local, whatever the changes, while the user does not move. It may be supported also for nomad users.

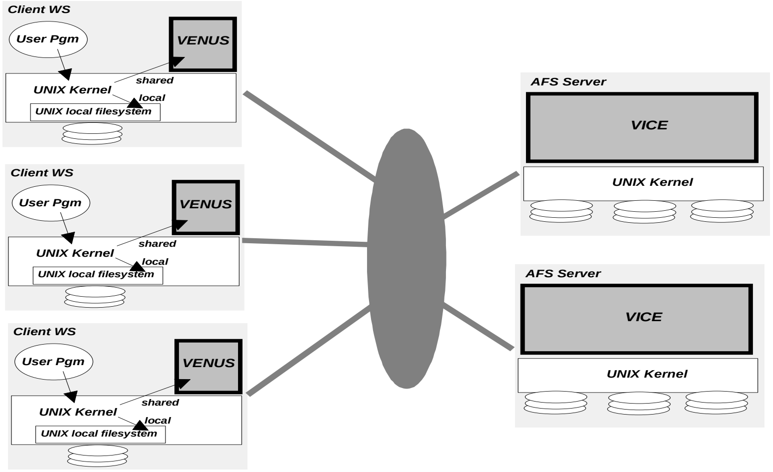
Invariant: one writer/multiple readers per file

AFS in action

- Client opens file, no local copy:
  - Finds the server that holds the file and asks copy
  - Copy is stored in shared AFS space (persistent cache)
  - File is opened like a local file
  - Matching between logical path of file in the client tree (/home/user/foo3) and the position on the shared space (/afs/0798RT56) is made using links
- Client uses the file
  - All operations, including writes, are applied on local copy
- Client closes file
  - Valid copy remains on the client side
  - If local copy was changed, it is copied to the server, which updates the master copy and its timestamp

Consistency in AFS

- When Vice gives a copy to Venus, also give a **call-back promise (CBP)** when another client changes the file
  - CBP is valid: since it is received until said otherwise by the Vice
  - CBP is cancelled: when the Vice receives changed copy it revokes all CBPs for that file on all Venus that had valid copies; when received by the Venus, it invalidates the CBP
- When the Venus want to open a file, it checks the cache:
  - If file not on cache:
    - Requests copy to the Vice
  - If file on cache, checks CBP:
    - If invalid, requests copy to the Vice
    - If valid, uses local copy



Consistency in AFS
Semantics

- Local copy may not be the most recent one
  - AFS-1 would inquire VICE about local CBP whenever a file was opened, to check the validity of the CBP and that would ensure that the most recent copy was used
  - The overhead on large-scale was big, leading to a solution where AFS only checks CBP locally
- A periodic expiration mechanism ensured that too old copies are invalidated
  - A file can only be locally opened if less than T has passed since last interaction between Vice and Venus (caching and file and receiving CBP)
- It is guaranteed that the most recent copy within T is used
  - Compensates loss of call-back
  - Typical value for T = 10 minutes

Consistency in AFS
Failures

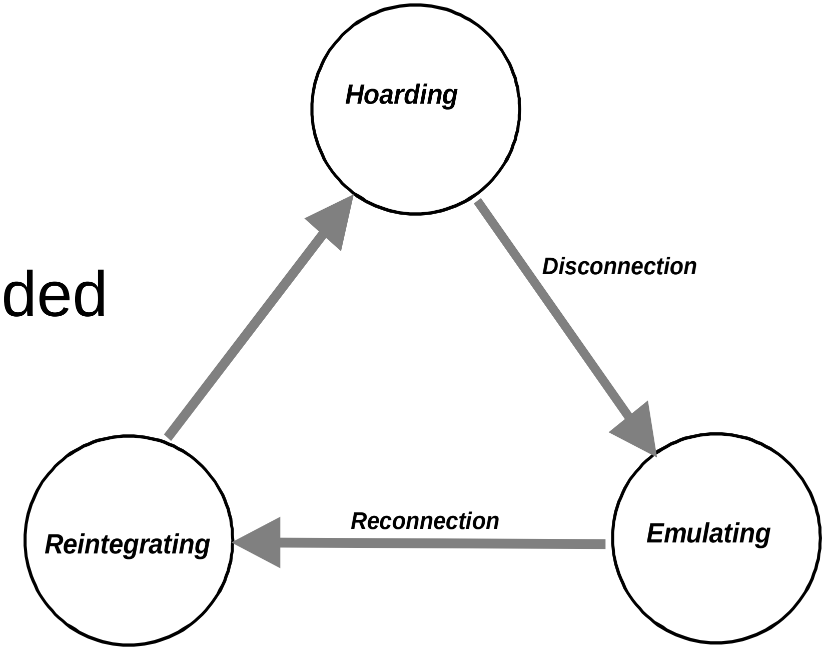
- When the client fails and recovers, Venus does not know how the system is
- It must contact the Vice and request a reconfirmation that the retained CBPs are still valid, sending a timestamp of each corresponding file

Consistency in AFS
Conflicts

- AFS semantics is not “one-copy”, i.e., it does not behave like a centralised file system
- When there is more than a client concurrently opening and working on a file, the server will keep the contents of the last close operation
- In addition it is not guaranteed that clients will get copies matching the opening instant
- However, this relaxed consistency is adequate to the expected workloads
- Clients must ensure their own consistency mechanisms, if necessary

Distributed file systems
Case study: CODA File System

- Volume replication (volume storage group)
- Disconnected operation
- Weak consistency
- Reconciliation
- Hoarding: preparation of needed replicas for disconnected operation
- **Advantages**:
  - Continuous availability
  - Support for mobile operation

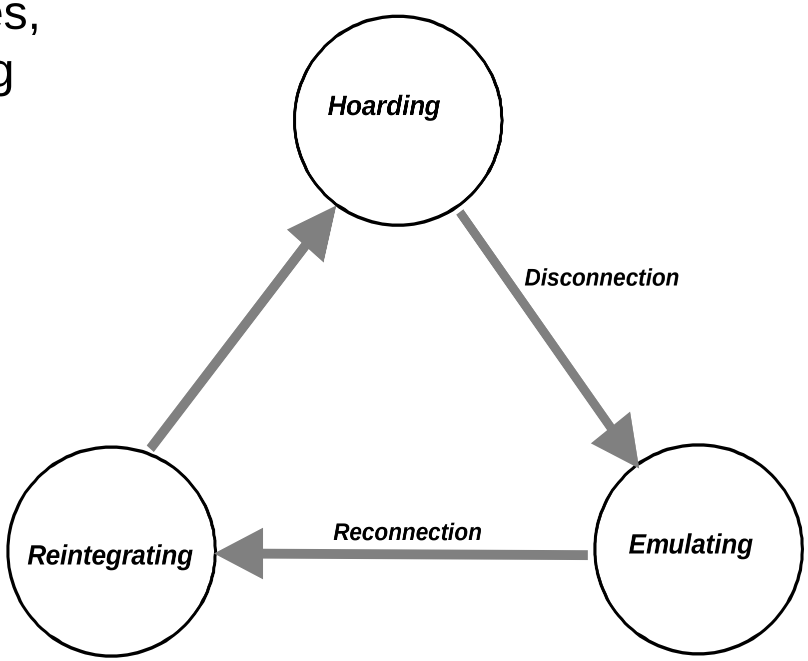


Coda File System - CODA

- Similar to AFS (derives from it), is based on Venus-Vice pairs and keeps scalability and UNIX compatibility objectives of AFS
- Tries to **improve reliability and availability**, regarding:
  - Instability and **network partitions**
  - **Disconnected operation**, e.g., nomad computing
- Introduces RW replication of Vice Volumes
- Constant data availability:
  - Providing the benefits of a shared DFS with the availability of a local FS
- Modifies the caching mechanisms to achieve its goals
- Files accessed by the client are local, stored on disk

CODA in action

- CODA assumed that disconnected operation is the rule, not the exception
- Operation in 3 states:
  - **Hoarding**: client specifies, with specific tool, which files to keep on local cache
  - **Emulation**: client operates on local files, possibly being disconnected, emulating connected operation
  - **Reintegration**: caches are reconciled with the (replicated) master file
- In case some conflict occurs, specific tools will help deciding what to do



CODA in action

- Servers keeping replicas of a file or group of files (e.g., directory) form a **Volume Storage Group (VSG)**
- The reachable replicas form the **Available VSG (AVSG)**, which can change due to partitions and disconnection
- When a file is opened:
  - If one of the replicas is reachable, then operation is like in AFS: file is cached or read from cache, with a valid CBP
- When a file is closed:
  - A copy of the file **is kept valid** on the client
  - If local copy was modified, it is promptly copied to all servers of the AVSG

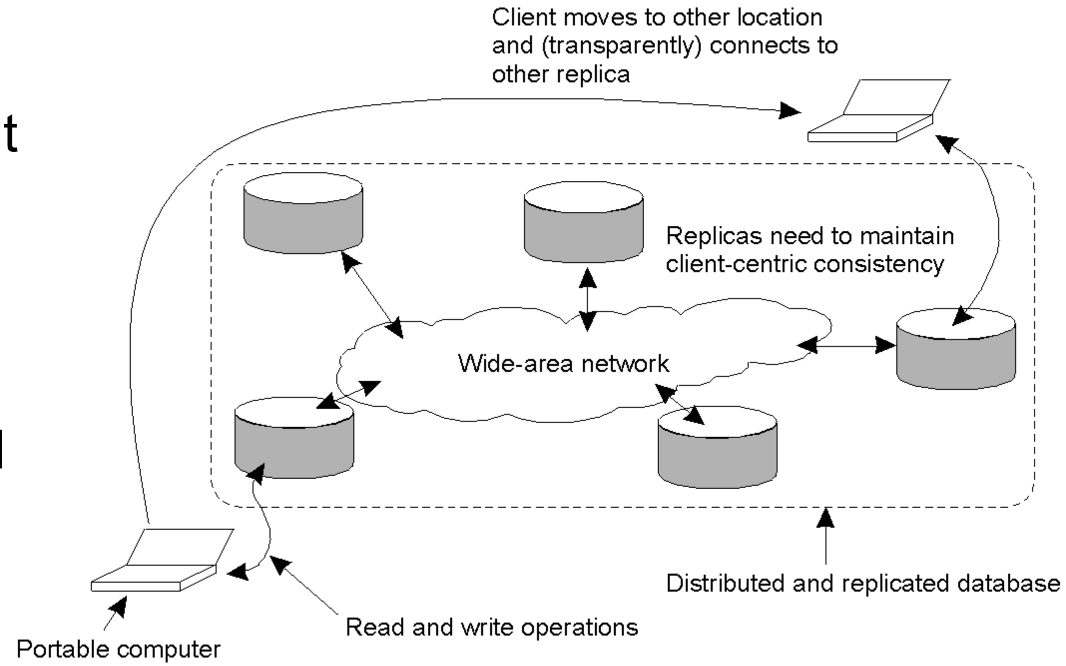
CODA strategy
Technical assumptions

- The same as in AFS:
  - Average file size is small (~10KB)
  - Reads are more frequent than writes (~6:1)
  - Sequential access more frequent than random access
  - Most files are of type “one writer/multiple readers”
  - Files are access several times in a row, which increases the chances that the copy on cache is still valid
- ... and…
  - Consistency in CODA resides on the group of replicas holding a file
  - Therefore, a file on cache is temporary and must be frequently reconciled
  - Updates made to disconnected caches refer to files of a single owner (single writer) who knows how to handle conflicts between the cache and the server

CODA strategy
Considered activity types

- The same as in AFS, and:
  - Read/write (RW) repositories of a single user, who is a nomad user

- Copy is always local to the client for any kind of operations, while he doesn't move, and is updated on the server when closed
- When the user moves (and disconnects) the copy may diverge, but it is considered that no other user will change it on the server
- When it reconnects, the reconciliation is automatic



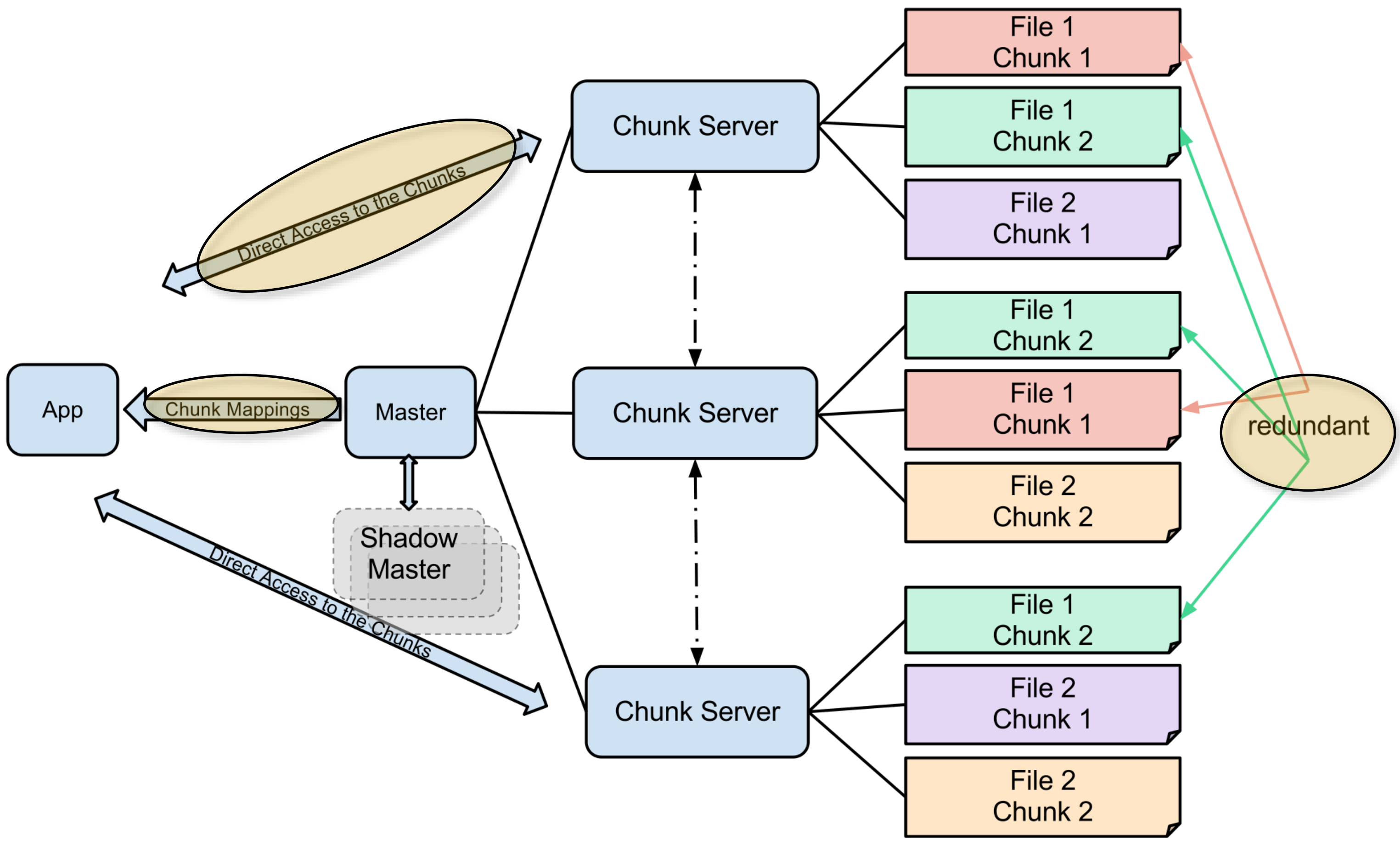
Consistency in CODA

- CODA allows the creation of divergent copies of a file
  - Caches of a file may be changed, opened and closed many times, while disconnected
- CODA has an **optimistic semantics**, because although allowing divergent copies, assumes they are not frequent
- However, it has mechanisms to detect conflicts
  - To reconcile changes in the best possible way, file modifications are recorded in a vector of logical timestamps (CODA Version Vector, CVV), which allows recording the causal order of updates, and which is maintained by clients and servers
  - When reconciled, versions are compared, allowing to choose the more recent one with a causal relation and identifying conflicts in the case of concurrent versions. In this case, the user decides

Distributed file systems
Case study: Google File System

- Used in Google datacentres:
  - Thousands of “common” machines
  - Failures are the rule, not the exception: in a datacentre with 2000 machines, there will be nearly 10 crashes per day
  - Big files that keep growing (many GB; after creation, writes are more common than appends)
- Organization (separation between data and meta-data)
  - **Master**: server that controls meta-data of the file system (directory service)
  - **Chunk server**: stores data blocks (typically of 64MB)
- GoogleFS is scalable despite centralized control (for simplicity), and the master is not an absolute bottleneck
  - Chunk servers do the most work (transfer data to/from clients)
  - Data for translating (file,chunk_id)/server are kept in master’s primary memory
  - Different services tend to use different GFS “groups”

Google File System
Overview



Google File System
Strategy

- GFS should be built with commodity hardware
  - Inexpensive disks and machines
- GSF stores a modest number of large files
  - A few million files, each typically 100MB or larger (up to Multi-GB)
    - Big-table, Map-Reduce records
  - Not optimized for small files
- Workloads
  - Large streaming reads (> 1MB) and small random reads (a few KBs)
  - Sequential appends to files by hundreds of data producers
    - Utilizing the fact that files are seldom modified again
- High sustained bandwidth more important than latency
  - Response time for individual read and write is not critical

Google File System
Architecture

- A single master (with shadow masters) and multiple chunk servers
  - Chunk: a fixed size data block (64MB)
  - A GFS file consists of one or several chunks (1G = 1024 MB = 16 chunks)
  - Master regularly asks a list of chunks owned by chunk servers (HeartBeat messages)
- Master is single-point-of-failure and a potential bottleneck
  - Typical Google uses mitigate that, but not for general use
  - Shadow Masters have limited functionality (reads only) so not completely F/T

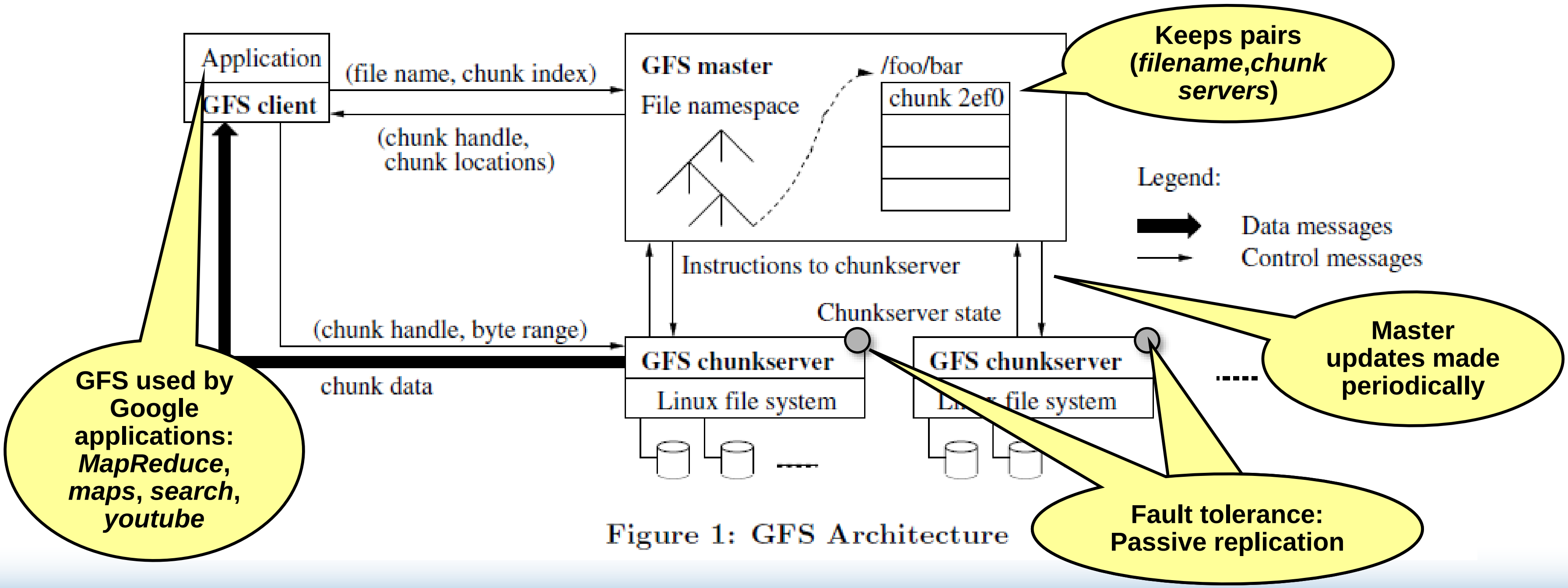
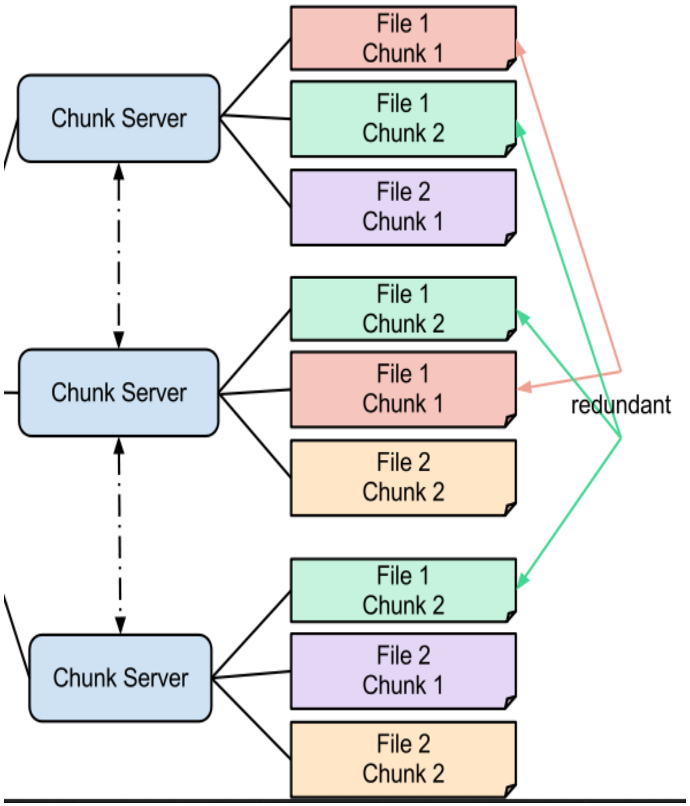


Figure 1: GFS Architecture

Google File System
Dependability

- Several replicas for each chunk
  - Default: 3 replicas
- Data Reliability
  - Protection from disk and network failure
- Data Availability
  - Each server rack should have a replica
    - Working set
    - Switch or power circuit failure
- Maximum Network Bandwidth Utilization
  - Pipelined data writes
- Primary replica:
  - Master chooses primary
  - When primary is written (mutation), secondary chunk servers copy mutation to their secondary replicas



Google File System in action

- 1, 2: client asks the master all the replica addresses, including the primary
- 3: client sends data to the nearest chunk server (data flow)
  - The other chunk servers are in the pipeline
  - The received data is buffered in the cache
- 4: client sends a write request to primary
- 5: primary replica decides the offset of the received data in the chunk
  - Assigns serial numbers to all mutations
  - Forwards the write request to the other replicas
- 6: completion messages from secondary replicas
- 7: primary replies to the client
  - When some failure is detected, makes several attempts at steps 3 through 7 before falling back to retries

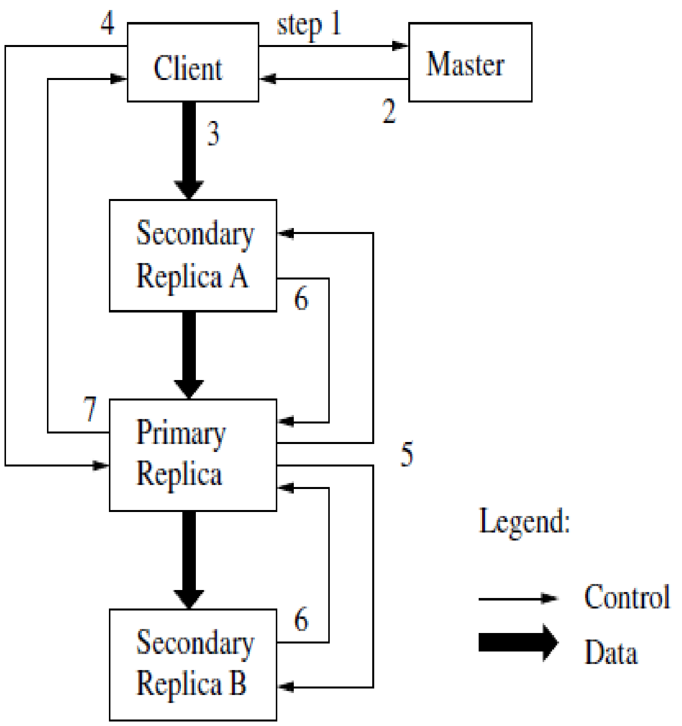


Figure 2: Write Control and Data Flow

Record appends

- Record append is more efficient than rewriting data
  - Pushes the data to all replicas of the last chunk of the file
  - The offset within the chunk is managed by the primary.
    - When the chunk has not enough space, indicates the client to retry appends by using additional chunk (current chunk is padded to maximum size)
- Consistency Model
  - A record append (write) may fail at some replica
    - Inconsistent state
    - Chunk replicas no longer identical
  - The client retries the operation
    - Chunks' state will become consistent
    - Every replica of a chunk has the same data
  - The written region is ***defined***
    - It is *consistent* and the entire written data can be seen by all clients
    - Data offset in a chunk and write order is managed by the primary

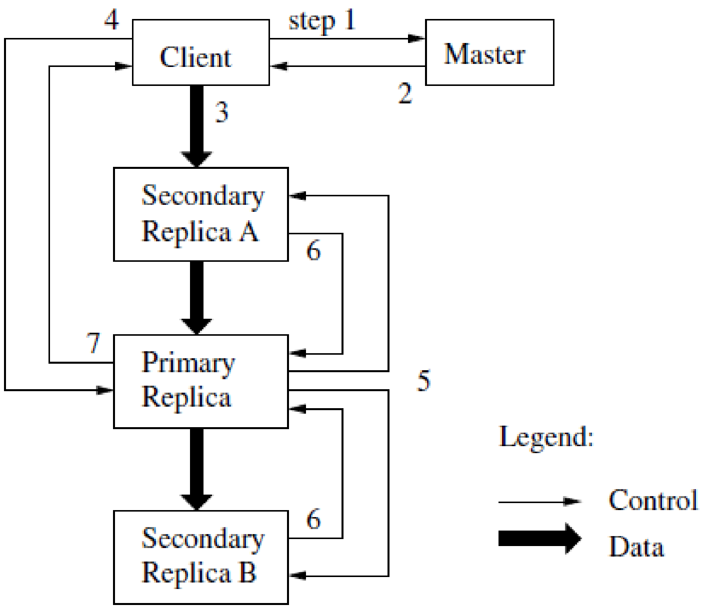


Figure 2: Write Control and Data Flow

Master operation

- GFS has no directory (i-node) structure
  - Simply uses directory-like file names: /foo, /foo/bar
    - Thus listing files in a directory is slow
- Concurrent Access
  - Read lock on a parent path, write lock on the leaf file name
    - Protect delete, rename and snapshot of in-use files
- Rebalancing
  - Places new replicas on chunk servers with below-average disk space utilizations
- Re-replication
  - When the number of replicas falls below 3 (or user-specified threshold)
    - The master assigns the highest priority to copy (clone) such chunks
  - Spread replicas of a chunk across racks

Garbage collection

- File deletion
  - Rename the file to a hidden name (deferred deletion)
  - The master regularly scans and removes hidden files, if they existed for more than three days
    - HeartBeat messages inform chunk servers of deleted chunks

- Stale replica detection
  - Version number is assigned for each chunk
    - Increases when the master grants a new lease of the chunk

Fault tolerance

- Fast Recovery
  – The master and the chunk server are designed to restore their state in seconds no matter how they terminated
  – Servers are routinely shut down just by killing the process
- Master Replications
  – Master has the maps from file names to chunks
  – One (primary) master manages chunk mutations
    - Several shadow masters are provided for read-only accesses
      – Snoop operation logs and apply these operations exactly as the primary does
- Data Integrity
  – Corruption of stored data
    - High temperature of storage devices causes such errors
  – Checksums for each 64KB in a chunk
    - Chunk servers verify the checksum of data before sending it to the client or other chunk servers