Distributed Systems (CS 543)

Distributed File System

Dongman Lee

Dept of CS

KAIST

Class Overview

- Introduction
- File Usage Pattern
- DFS Requirements
- Key Techniques
- Case Studies: NFS, AFS, GFS, & HDFS

Introduction

- Definition [Satyanarayanan]
  - distributed implementation of the time sharing file system abstraction
- Requirements
  - sharing of persistent storage and information across machine boundaries
  - access remote files without copying them to a local disk
- Key issues
  - file location
    - access transparency -> location/migration transparency
  - availability
    - replication
    - mobile and spontaneous network

Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓✓	✓	Ivy
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore

Types of consistency:
1: strict one-copy. ✓: slightly weaker guarantees. 2: considerably weaker guarantees.

Files Usage Pattern

- Sizes
  - most files are small
  - absolute value of average file size has increased over time, but it continues to remain small relative to contemporary disk sizes
- Operations
  - read operations on files are much more frequent than write operations
  - files are usually read in *sequence* rather than in random
  - a file is usually read in its entirety once it has been opened
- Functional lifetime
  - time interval between the most recent read and the mote recent write
  - average functional lifetime is short
    - data in files tends to be overwritten often
    - system files shows larger lifetime while temporary files have much shorter lifetime

File Usage Pattern (cont.)

- Sharing
  - most files are read and written by one user
  - shared files are usually modified by one user
  - exceptions:
    - large collaborative project
    - shared database
- Locality of reference
  - if a file is referenced, there is a high probability it will be referenced again in the near future
  - the set of referenced files is a very small subset of all files over short period time
- Type differences
  - system files are rarely written while temp files are not shared and short-lived

Distributed File System Requirements

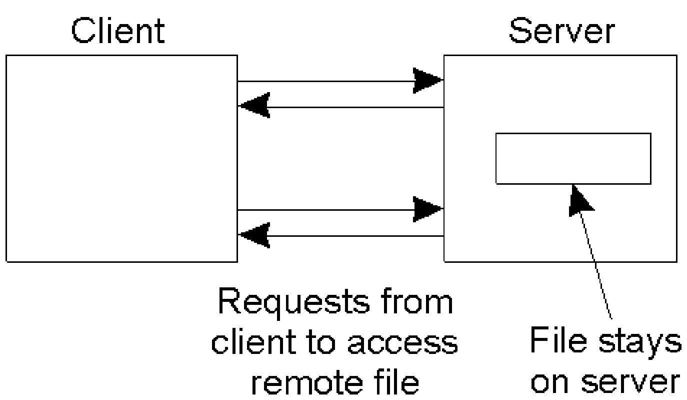
- Transparency
  - access, location, mobility, performance, and scaling transparency
- Concurrent file updates
  - file- or record-level concurrency control
- File replication
  - availability
- Heterogeneity
  - openness
- Fault tolerance
  - stateless (fault-amnesia)
  - stateful (fault-tolerant)

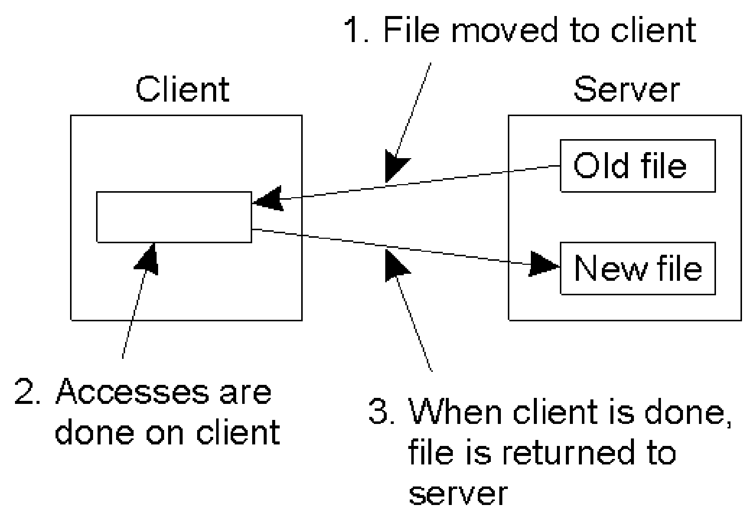
Distributed File System Requirements (cont.)

- Consistency
  - one-copy update semantics
- Security
  - authentication
- Efficiency
  - caching and event-driven update

Key Techniques

- Caching at clients
  - exploit temporal locality of reference
    - high probability of reusing file data after its 1st use
    - meta data (e.g. directory, file status info, location info) do not change frequently





Key Techniques (cont.)

- Caching at clients (cont.)
  - key issue: size of cached units of data
    - individual page of file vs. whole file
  - cache location
    - memory vs. local disk
    - remote operation vs. upload/download model
  - cache coherence
    - client-initiated vs. server-initiated
  - spatial locality
    - adjacent pages are likely to be used soon

Key Techniques (cont.)

- Transfer data in bulk
  - bulk data transfer protocols depend on spatial locality of reference within files for effectiveness
- Hints
  - a piece of information that can substantially improve performance if correct but has no semantically negative consequence if erroneous [Lampson]
  - mostly used for file location information
- Encryption
  - used to prevent unauthorized release and modification of files
  - private (e.g. DES) vs. public (e.g. RSA)

Key Techniques (cont.)

- Mount points
  - the mount mechanism provides a seamless integration of a remote file system into a local file system
  - two approaches
    - client-initiated (e.g. Sun NFS v3): stateless server
      - no central management of mount information
      - server is unaware of where its subtrees are exported
      - fault tolerant
      - globally single shared namespace is difficult to maintain
      - movement of files requires re-mount at associated clients
    - server-initiated (e.g. AFS & NFS v4): stateful server
      - mount information is embedded in the data stored in the servers
      - globally single shared namespace is allowed
      - movement of files can be done transparent to clients

File Sharing Semantics

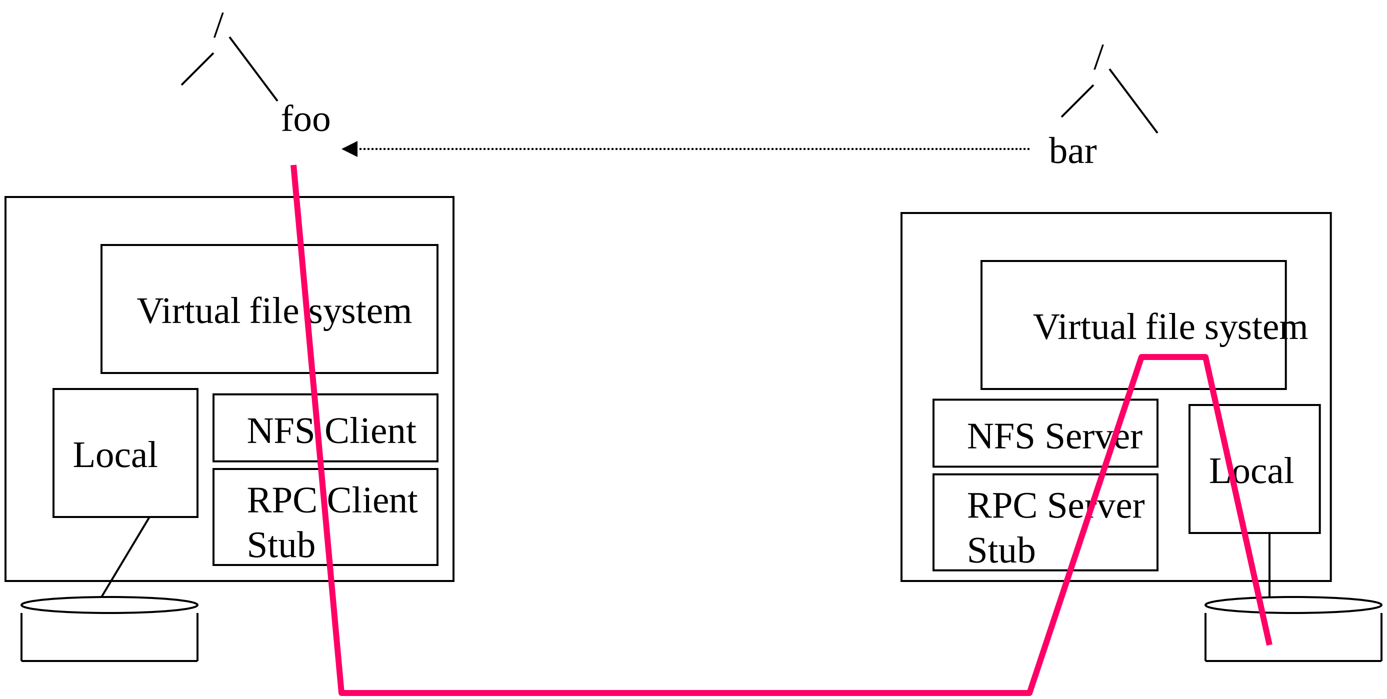
- Unix file semantics: one copy update semantics
  - read after write returns value written
  - system enforces absolute time ordering on all operations and returns most recent value
  - changes immediately visible to all processes
- To achieve these in distributed systems, all accesses must happen at file server => performance degradation
  - session semantics
    - no changes are visible to other processes until the file is closed
  - immutable files
    - no updates are possible, simplifies sharing and replication
  - transaction
    - all operations are atomic

Mobile Support

- Conflict detection and resolution
- Connectivity
- Cache update frequency & amount
- Disconnected operations and reconciliation

Case Study: Network File System

- Goal
  - transparent access to remote files
  - emulation of UNIX single copy update semantics
- NFS structure

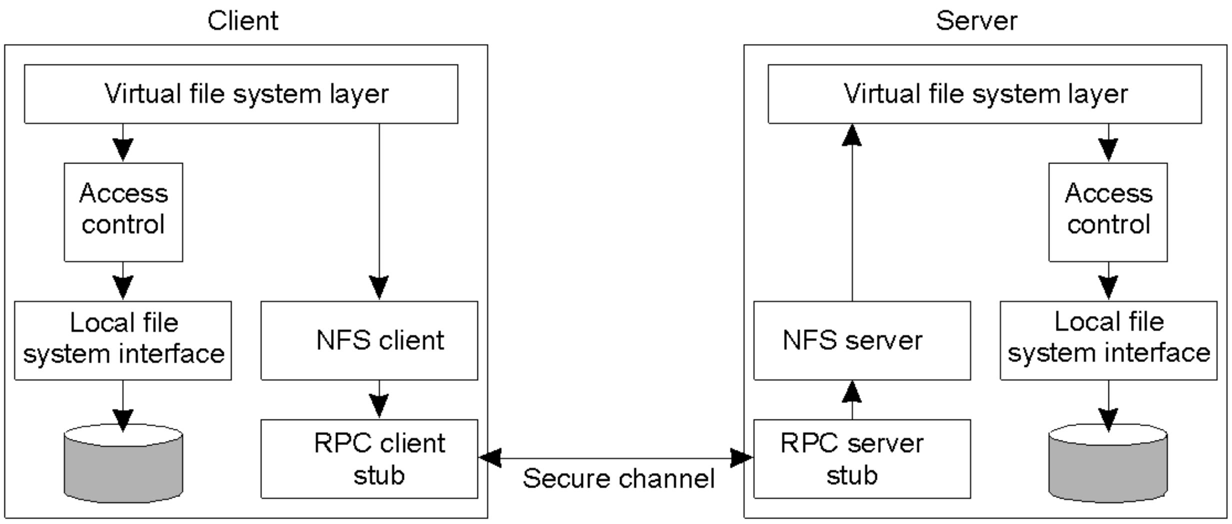


Network File System (cont.)

- Architectural characteristics
  - symmetric client-server architecture
  - implemented in kernel
    - keep the existing file APIs remain intact
  - each client expands its name space by adding remote file systems
    - mounted remote file systems become part of local name space
    - a single name space is not enforced
  - stateful
    - Server maintains state between operations on the same file
  - File operation semantics
    - Create is used for non-regular files while open and close for regular files
  - file cache
    - client and server both provide cache for performance improvement
  - compound procedures
    - Several RPC calls can be put together into a single request

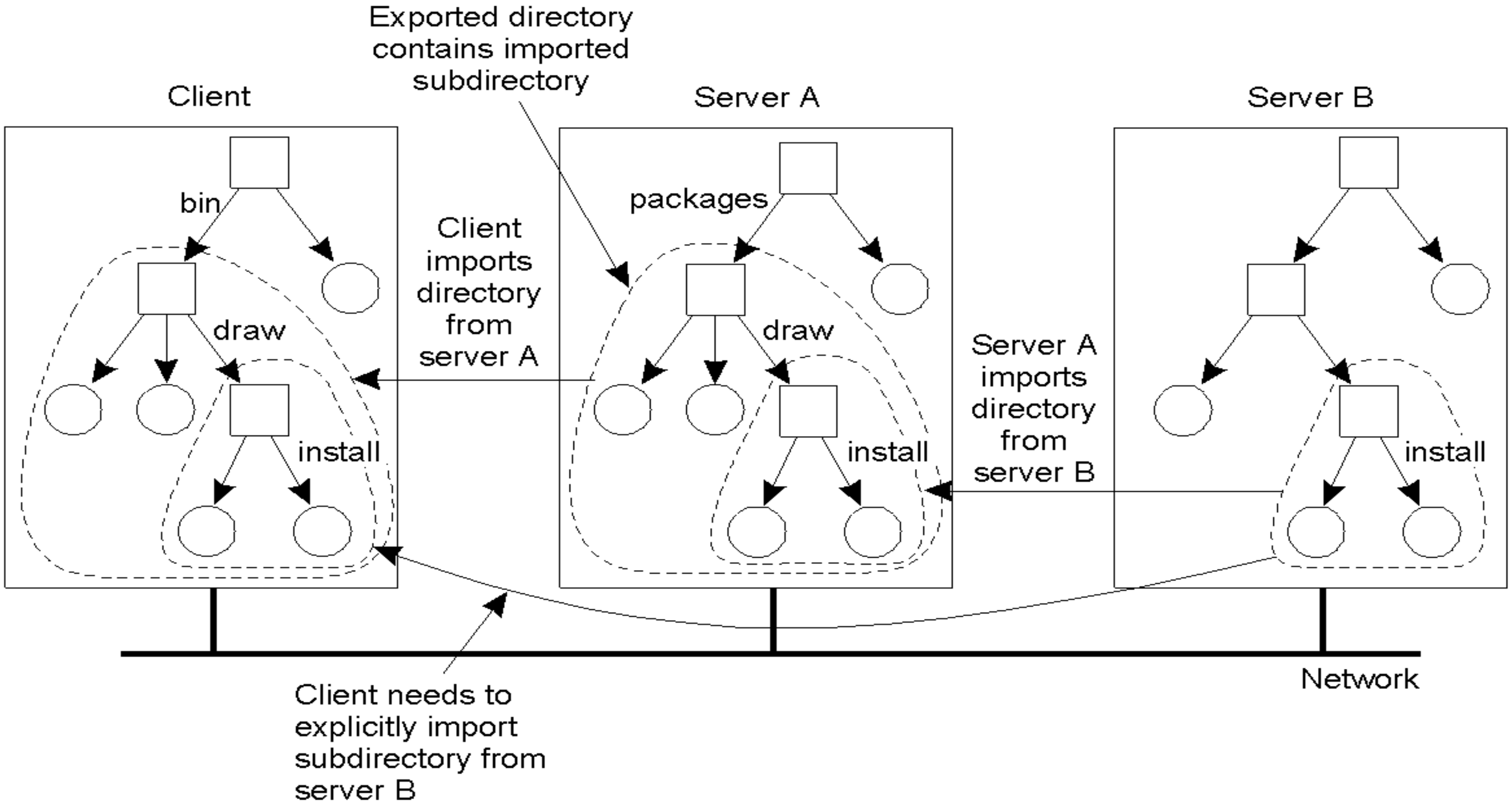
Network File System (cont.)

- Virtual file system (VFS)
  - encapsulation of i-node
  - file handle is being used to identify an i-node at a server
    - file handle = filesystem id + i-node # + i-node gen #
- Security and authentication
  - RPC-level authentication
  - conventional UNIX access control is used



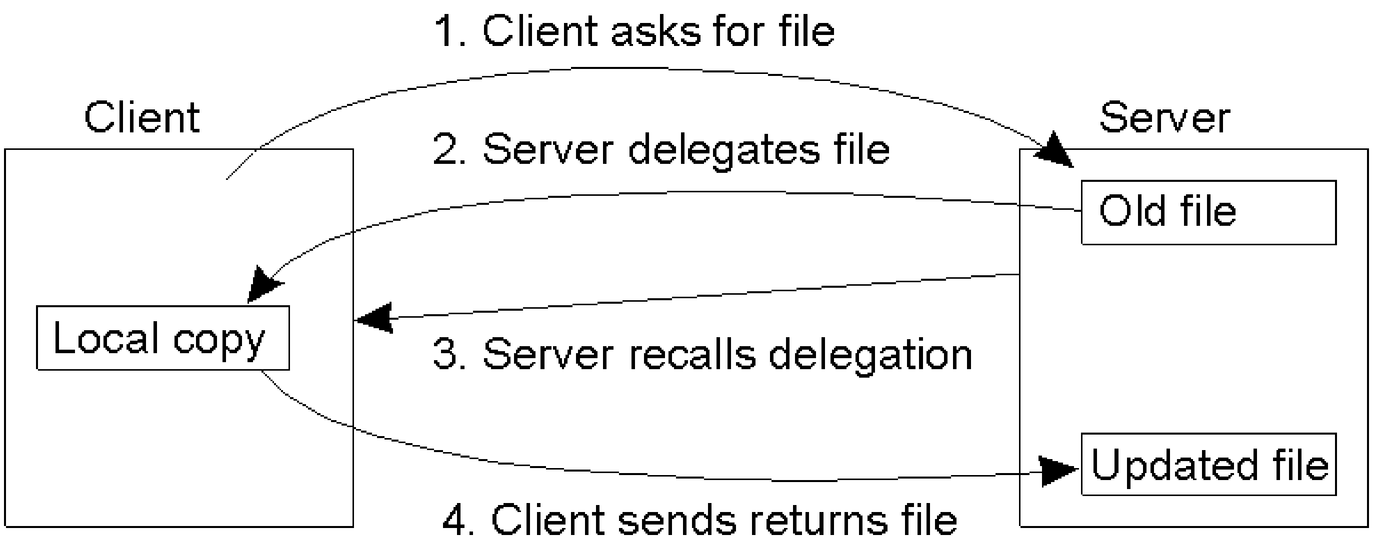
Network File System (cont.)

- Name space expansion
  - Name resolution over a single mount point
    - File names can be resolved recursively



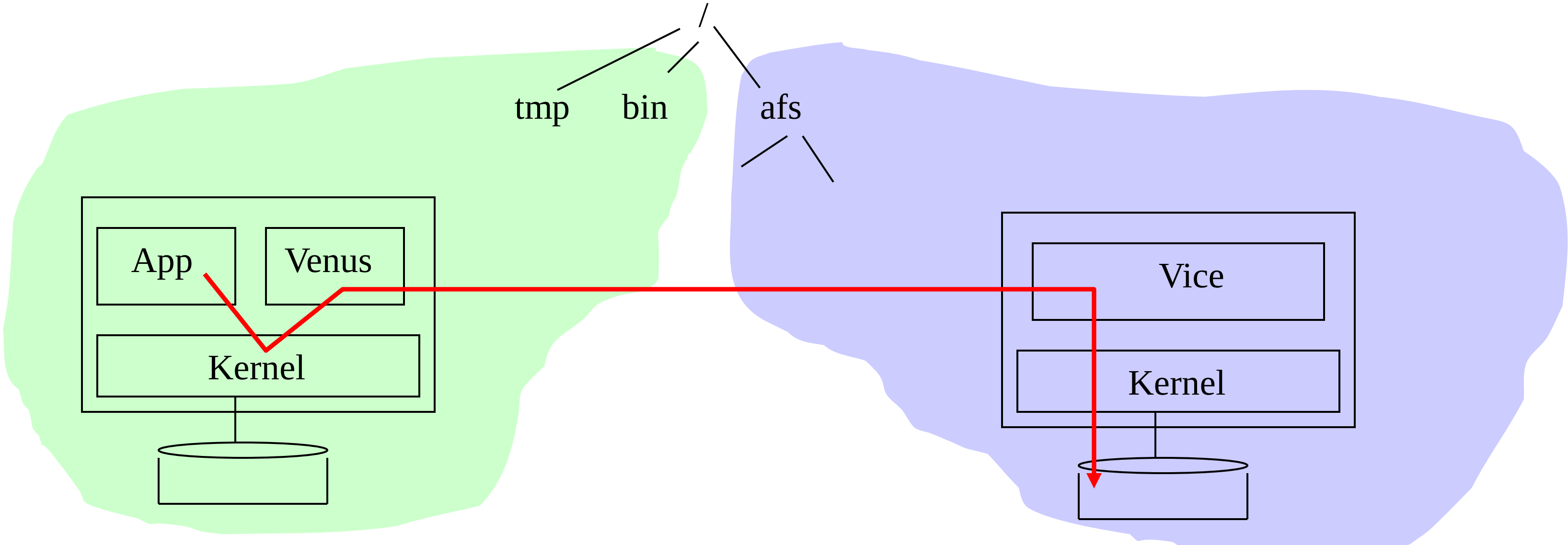
Network File System (cont.)

- Caching
  - Server
    - read: read-ahead cache like conventional UNIX
    - write: write-through to emulate single copy update semantics
      - client must wait for disk write
  - Client
    - Locally written data is flushed back to the server (session semantics)
    - Open delegation



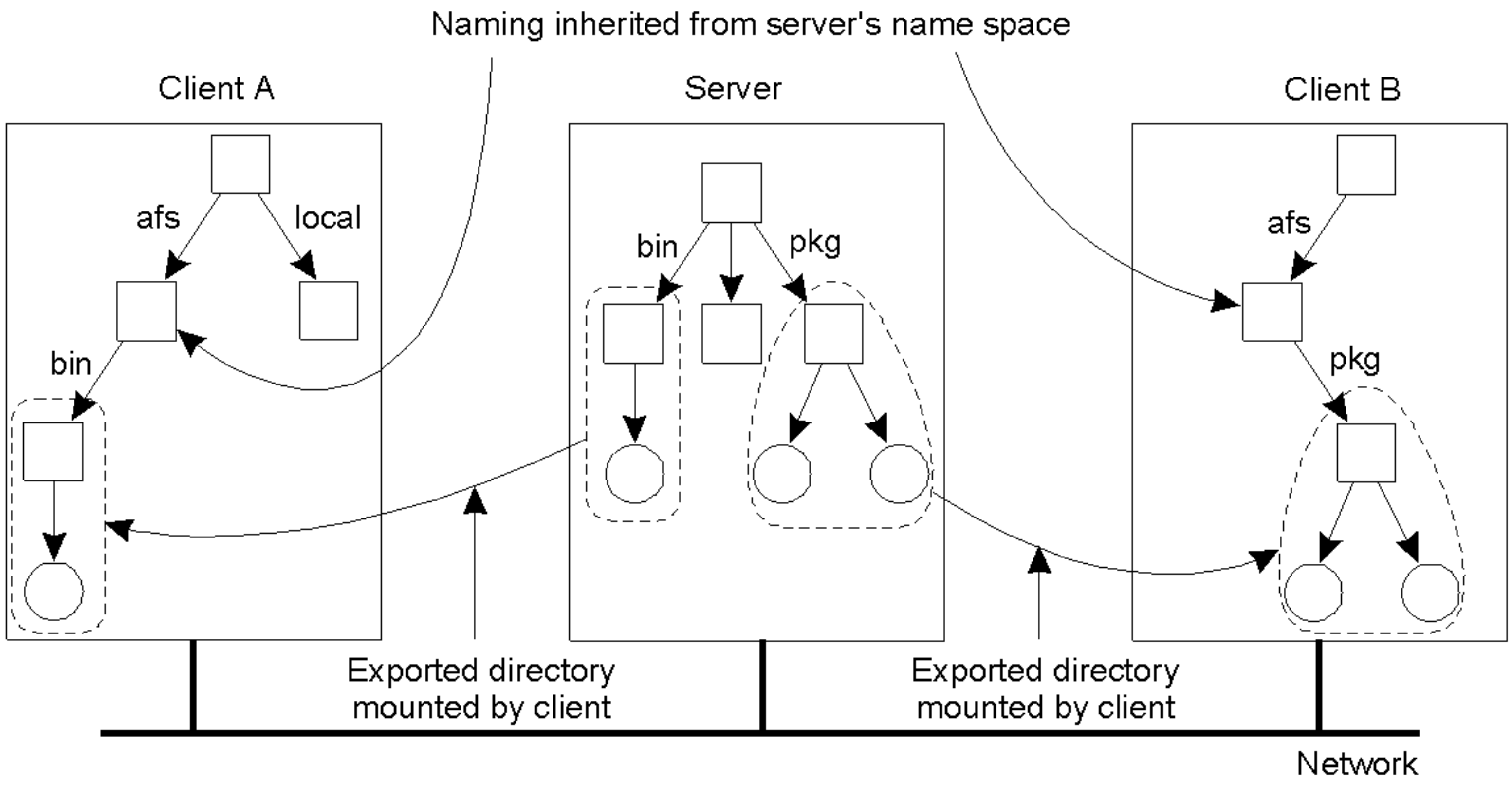
Andrew File System

- Goal
  - large scale (5K-10K clients and servers)
  - security
- Architecture
  - Vice: information sharing back-bone
    - collection of file servers connected by multiple of networks
  - Virtue: client modules



Andrew File System (cont.)

- AFS Name space



Andrew File System (cont.)

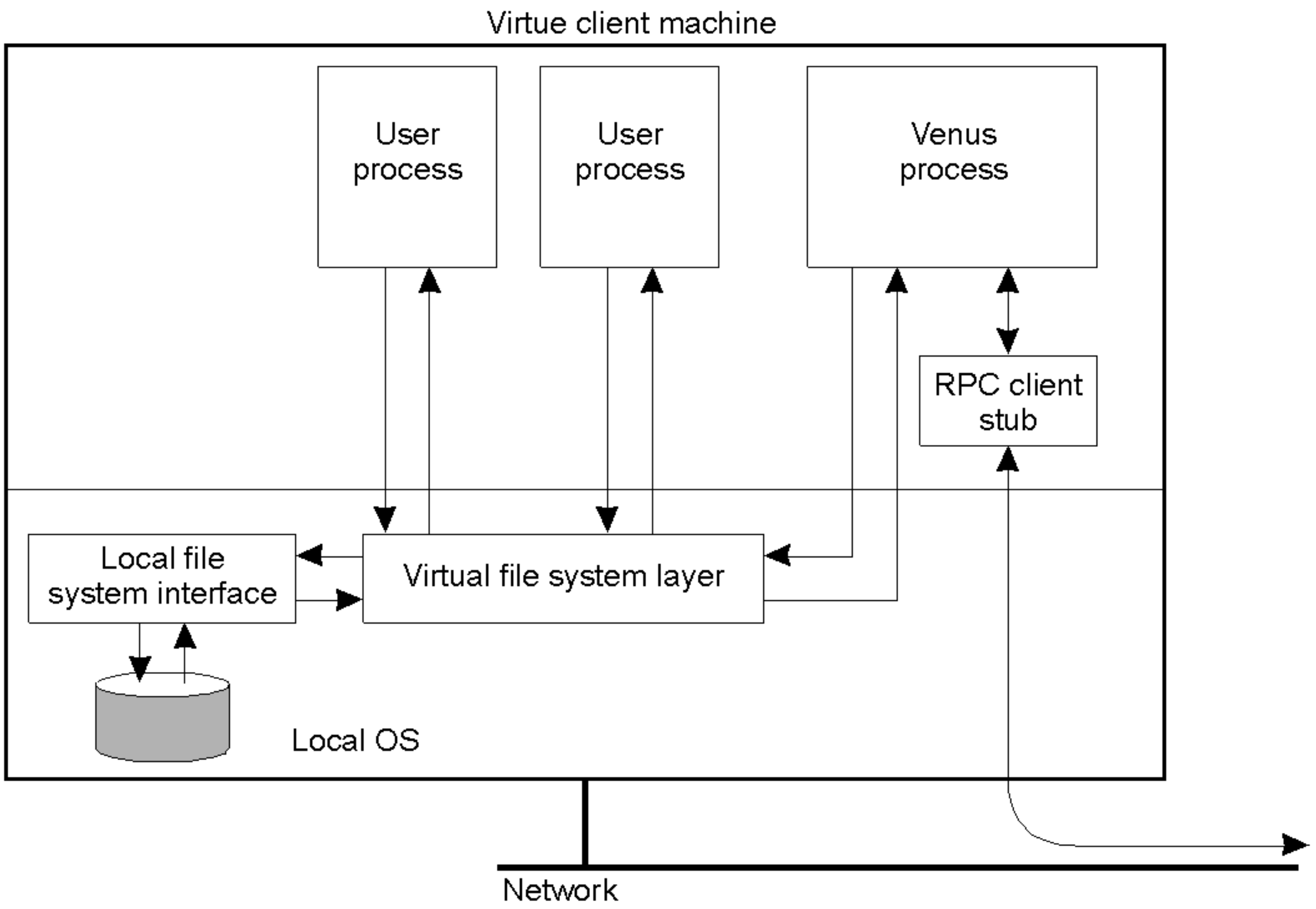
- Architectural characteristics
  - data sharing is simplified
    - server takes care of all the file sharing
    - client caches whole files on its disk
  - user mobility is supported
    - location-independent single file system view (for shared name space) is supported
  - easy system administration
    - all the mounting information is at servers
  - better security is possible
    - servers are physically secure and on which no user programs run
  - client autonomy is improved
    - disservice or relocation of a client imposes no interference to other clients

Andrew File System (cont.)

- Scalability
  - strategy
    - reduce static bindings to a bare minimum
      - no shared data at clients
    - maximize # of active clients
  - servers
    - shadow directory for file status info (e.g. access list)
    - server notifies clients whenever there's change
    - volume forms a name domain
    - volume location database are replicated at each server and cached at clients
    - read-only volume replication
  - clients
    - partial file cache is allowed
    - cache path-name-prefix -> direct access to appropriate server
    - fid = volume # + NFS file handle + uniqufier

Andrew File System (cont.)

- Virtue internal organization



Andrew File System (cont.)

- Implementation of file system calls

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.	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.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
		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.	←	
	Open the local file and return the file descriptor to the application.			
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.

Andrew File System (cont.)

- Cache coherence
  - server initiated update
    - callback promise
      - promise is sent whenever the data changes
      - server must remember promises beyond failures
    - approximation of one copy update semantics
  - update policy at clients: session semantics
    - cache validity is always checked at *open* operation
    - clients check cache validity with the server if a time T expires => to cope with loss of callback messages
    - if a client crashes, it assumes all caches are invalid
    - close operation at client will *replace* the current content of the file at the server

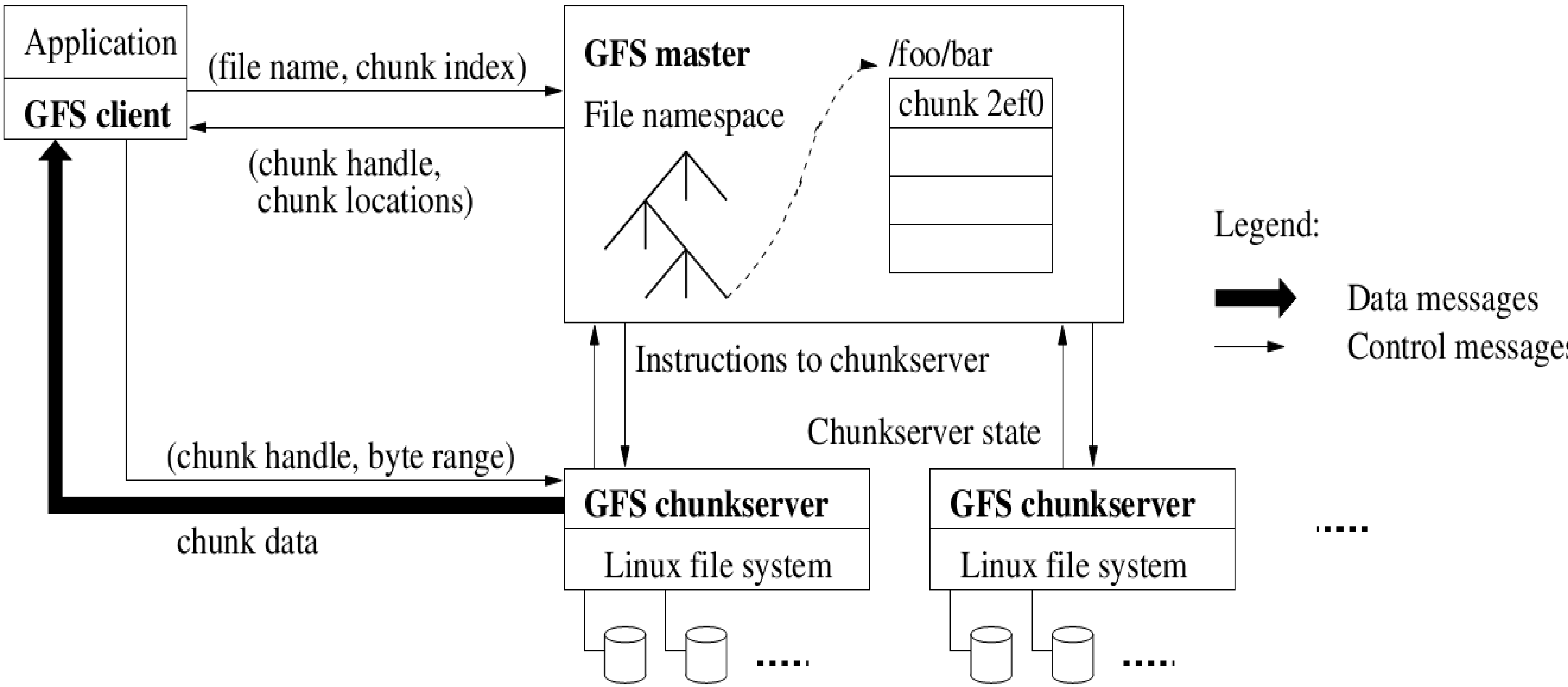
Andrew File System (cont.)

- Security
  - protection domain
    - group inheritance
    - protection server
  - Kerberos-based authentication
  - file system protection
    - access list for directories: negative rights
    - file level mode bits -> emulates Unix file access control

GFS (Google File System)

- Designed for
  - Component’s Monitoring
  - Storing of huge data
  - Reading and writing of data
  - Well defined semantics for multiple clients
  - Importance of Bandwidth
- Cluster architecture
  - Single Master
  - Multiple Chunk Servers
  - Multiple clients

GFS: Architecture

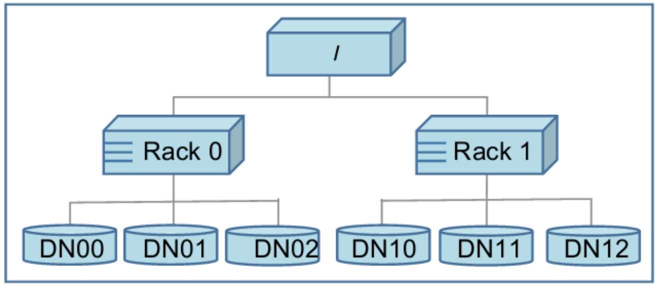


GFS: Master and Chunks

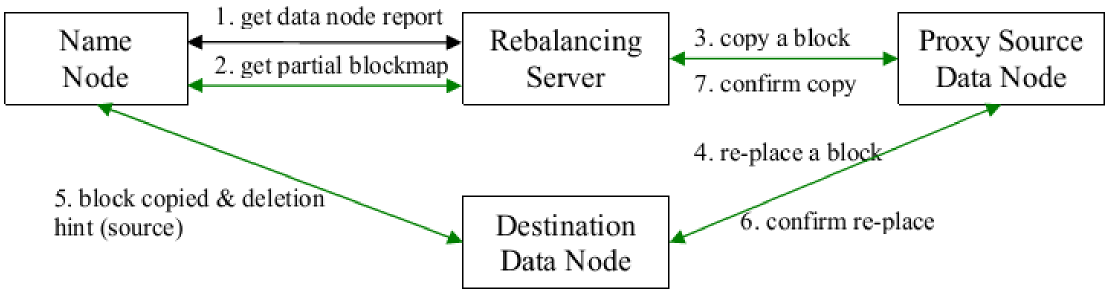
- Single Master
  - Minimal Master Load.
  - Fixed chunk Size.
  - The master also predicatively provide chunk locations immediately following those requested by unique id.
- Chunk Size
  - 64 MB size.
  - Read and write operations on same chunk.
  - Reduces network overhead and size of metadata in the master.

GFS: Replica Management

- Placement policy
  - Minimizing write cost.
  - Reliability & Availability – Different racks
  - Network bandwidth utilization – First block same as writer



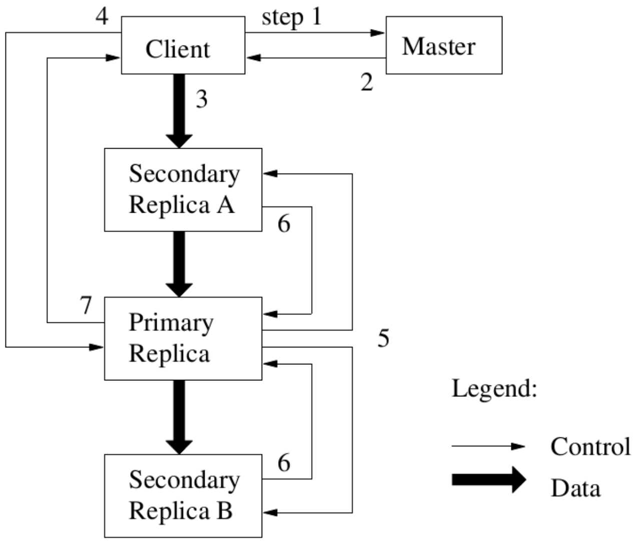
- Data balancing
  - Placing new replicas on chunk servers with below average disk space utilization
  - Master rebalances replicas periodically



GSF: Read & Write

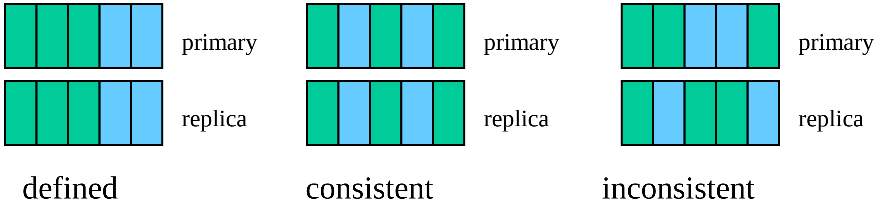
- Data Flow (I/O operations)
  - Leases at primary (60 sec. default)
  - Client read
    - Sends request to master
    - Caches list of replicas

      locations for a limited time.
  - Client Write
    - 1-2: client obtains replica
      locations and identity of primary replica
    - 3: client pushes data to replicas
      (stored in LRU buffer by chunk servers holding replicas)
    - 4: client issues update request to primary
    - 5: primary forwards/performs write request
    - 6: primary receives replies from replica
    - 7: primary replies to client



GFS: Consistency Model

- Write
  - Large or cross-chunk writes are divided buy client into individual writes.
- Record Append
  - GFS's recommendation (preferred over write).
  - Client specifies only the data (no offset).
  - GFS chooses the offset and returns to client.
  - No locks and client synchronization is needed.
  - Atomically, at-least-once semantics.
  - Client retries faild operations.
  - Defined in regions of successful appends, but may have undefined intervening regions.
- Application Safeguard
  - Insert checksums in records headers to detect fragments.
  - Insert sequence numbers to detect duplications.

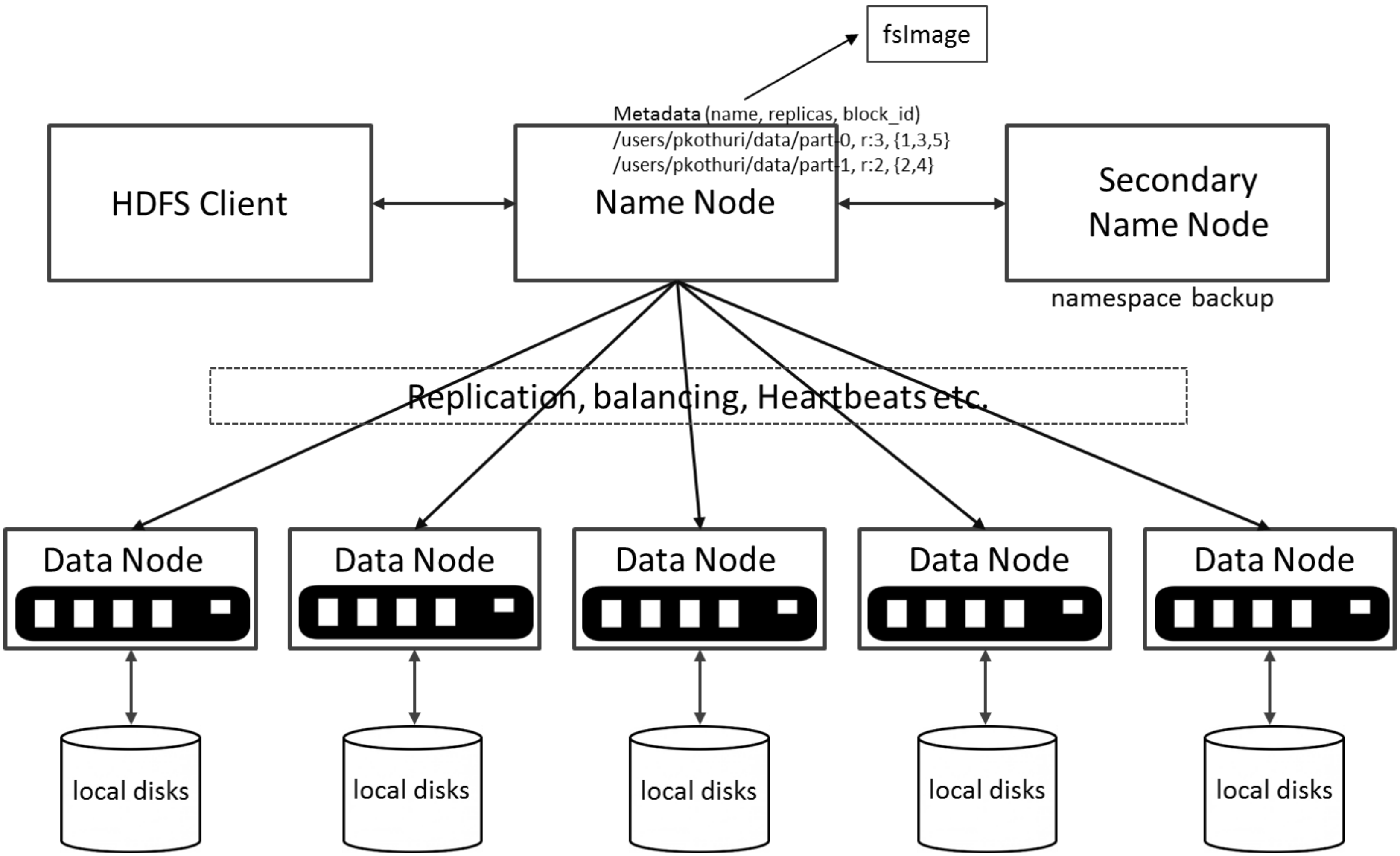


	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

HDFS (Hadoop Dist File System)

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand, leveraging many ideas from GFS (Google File Systems)
- HDFS is the primary distributed storage for Hadoop applications such as Map/Reduce.
- HDFS provides interfaces for applications to move themselves closer to data.
- HDFS is designed to ‘just work’, however a working knowledge helps in diagnostics and improvements.

HDFS:Architecture



HDFS: Components

- There are two (*and a half*) types of machines in a HDFS cluster
- NameNode
  - the heart of an HDFS filesystem
  - maintains and manages the file system metadata. e.g; what blocks make up a file, and on which data nodes those blocks are stored.
- DataNode
  - where HDFS stores the actual data, there are usually quite a few of these.

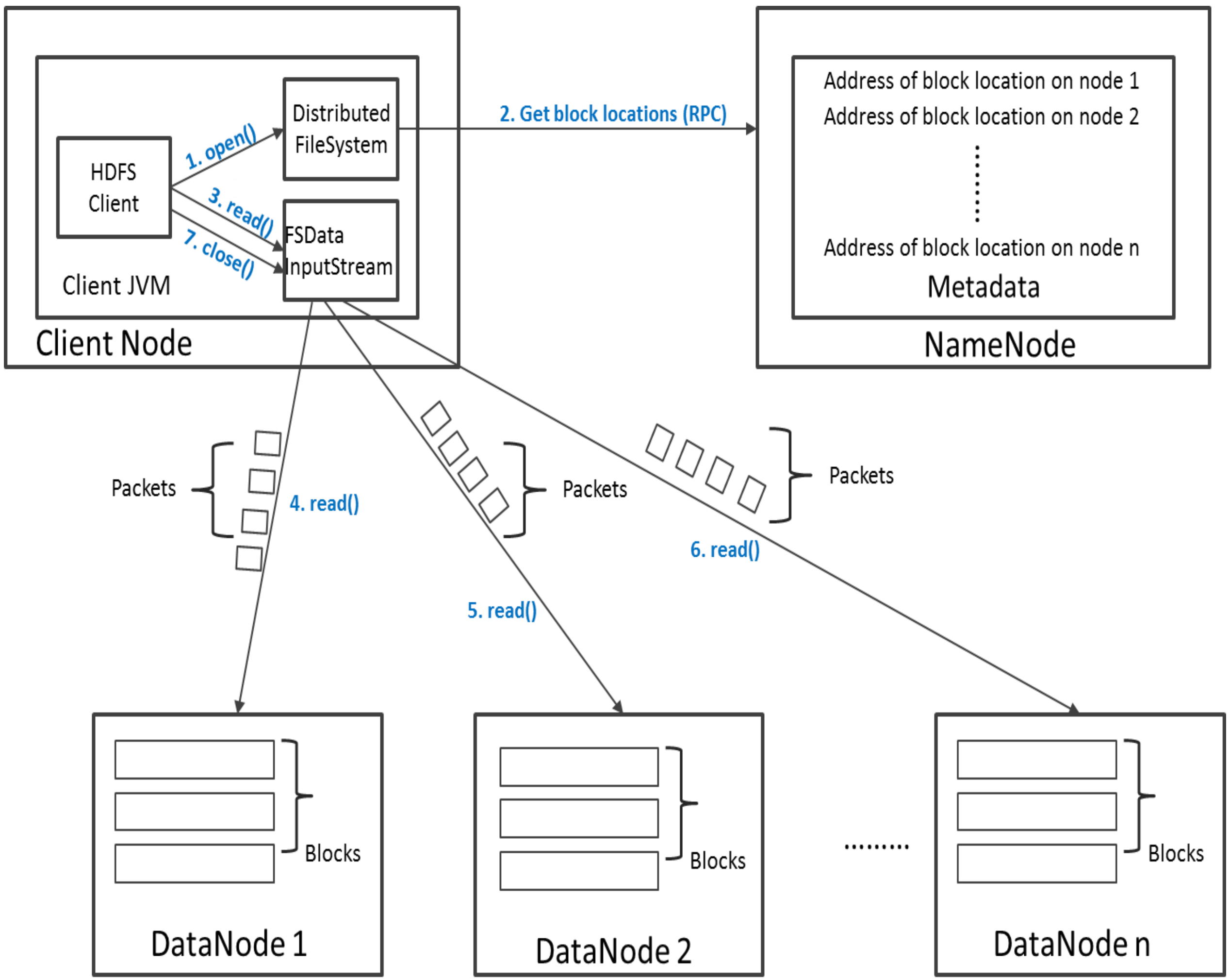
HDFS: Key Features

- Failure tolerant - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
- Scalability - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
- Space - need more disk space? Just add more DataNodes and re-balance
- Industry standard - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)
- HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access

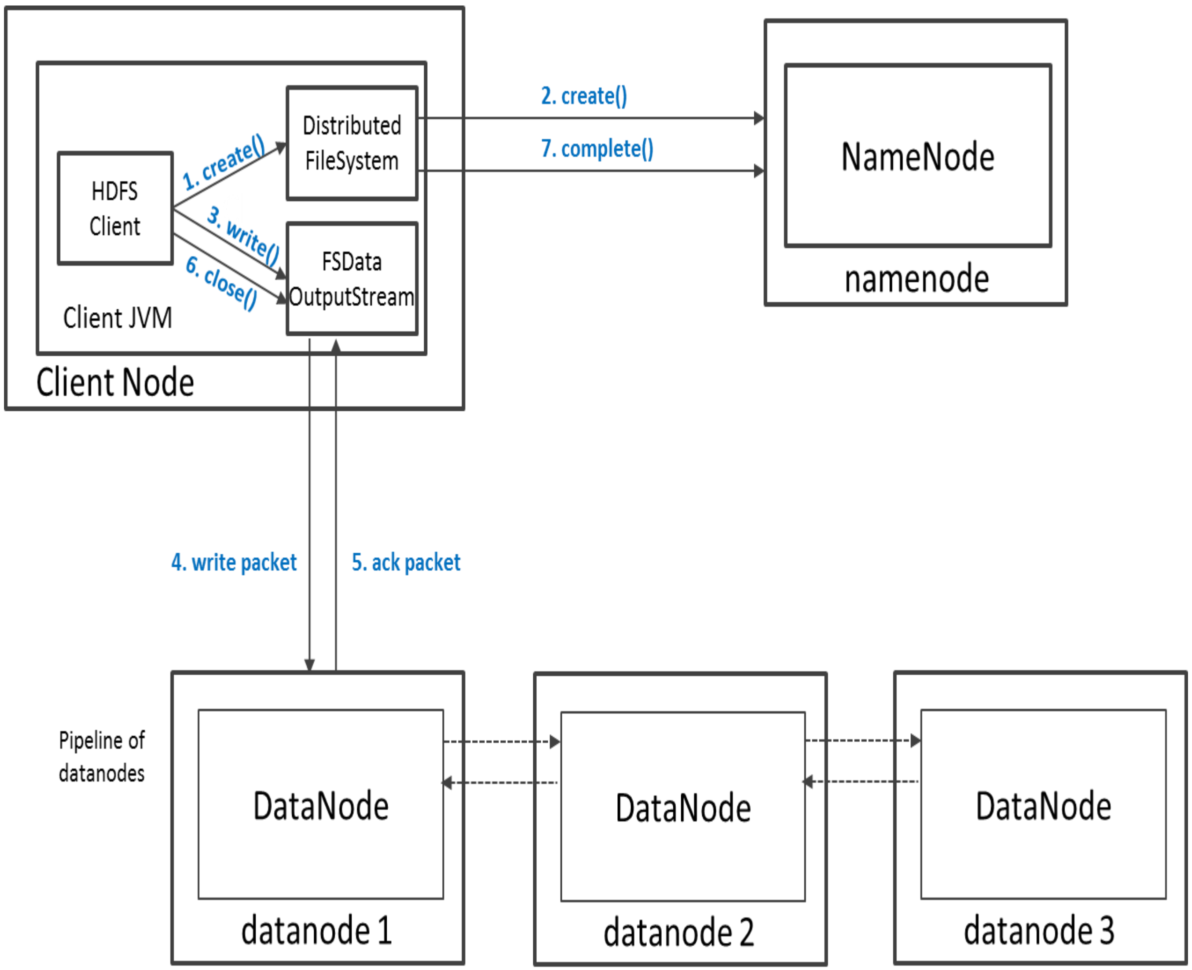
HDFS – Data Organization

- Each file written into HDFS is split into data blocks
- Each block is stored on one or more nodes
- Each copy of the block is called replica
- Block placement policy
  - First replica is placed on the local node
  - Second replica is placed in a different rack
  - Third replica is placed in the same rack as the second replica

HDFS: Read Operation



HDFS: Write Operation



HDFS: Security

- Authentication to Hadoop
  - Simple – insecure way of using OS username to determine hadoop identity
  - Kerberos – authentication using kerberos ticket
  - Set by hadoop.security.authentication=simple|kerberos
- File and Directory permissions are same like in POSIX
  - read (r), write (w), and execute (x) permissions
  - also has an owner, group and mode
  - enabled by default (dfs.permissions.enabled=true)
- ACLs are used for implemention permissions that differ from natural hierarchy of users and groups
  - enabled by dfs.namenode.acls.enabled=true