Distributed File Systems

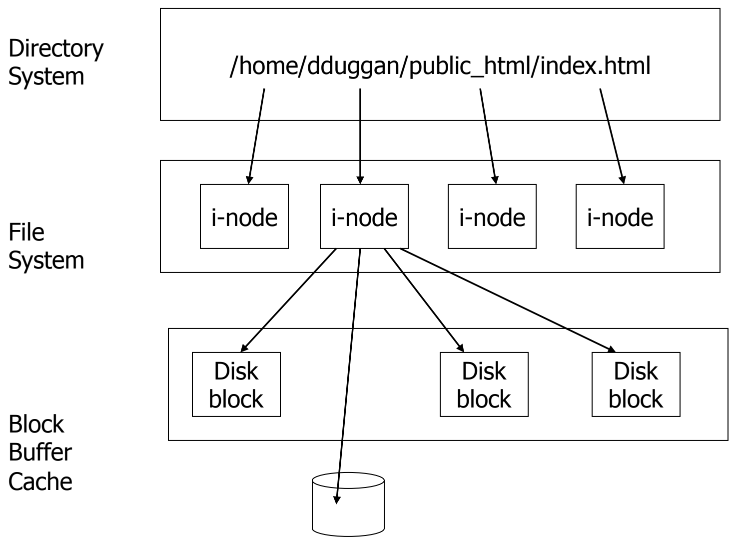
Dominic Duggan
Stevens Institute of Technology

1

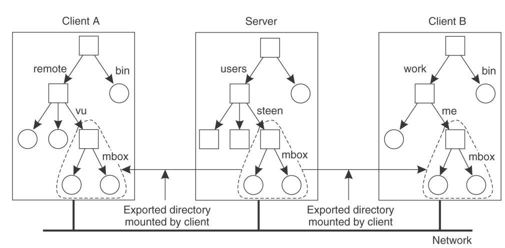
FILE SYSTEMS

2

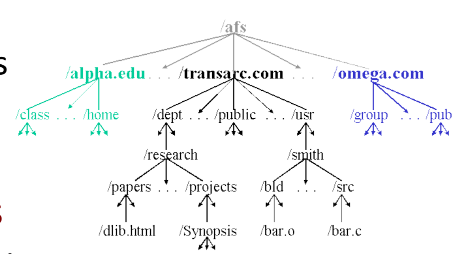
Layers in File Systems



Naming Scheme

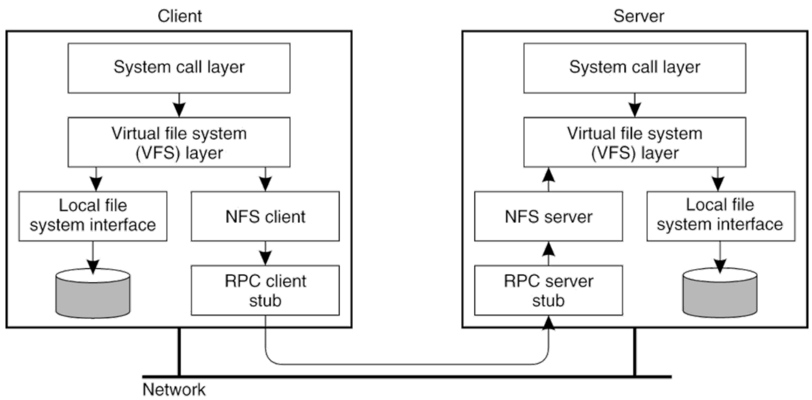


- Mount remote directories to local
  - Coherent **local** directory tree
  - Ex: Unix/Linux with NFS; Windows mapped drives



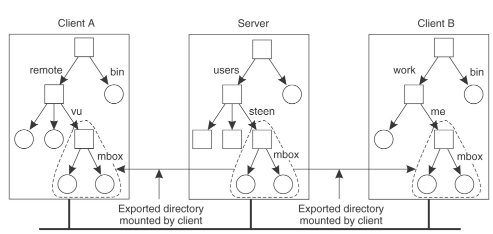
- Total integration of component file systems
  - Single **global** name structure
  - Ex: AFS

Location Transparency



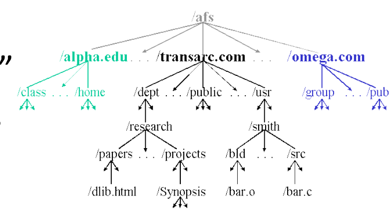
Location Independence (Migration Transparency)

- NFS: Not location independent
  - Server: `export /root/fs1/`
  - Client: `mount server:/root/fs1 /fs1`



Location Independence (Migration Transparency)

- AFS: global volumes
  - Global directory /afs;
  - /afs/cs.stevens.edu/vol1/…; /afs/cs.njit.edu/vol1/
  - File id = <vol_id, vnode #, uniquifier>
  - "Volume location database"
    - vol_id→ server_ip mappings
    - Shared by servers



7

File Server Semantics

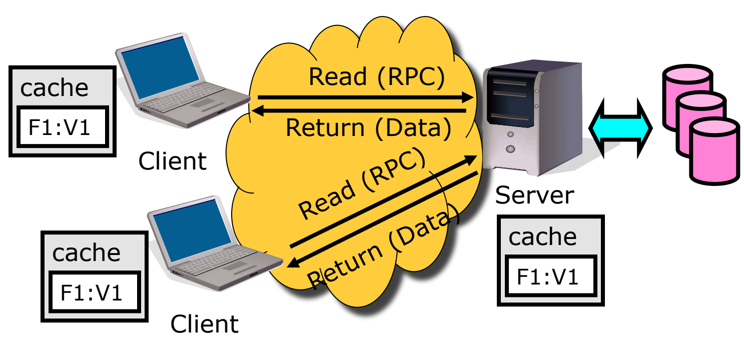
- Stateless
  - + Crash recovery
  - - File locking
  - Ex: NSF v3
- Stateful
  - - Crash recovery
  - + File locking
  - Ex: AFS, NFS v4

8

CACHING POLICIES

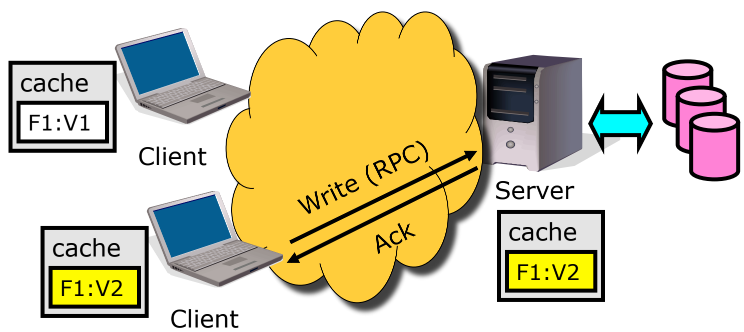
9

File Caching



10

Cache Consistency



Cache Update Policies

- When does the client update the master file?
- Write-through: write data to server ASAP
- Delayed-write: cache, write to server later
  - Better local performance
  - Network I/O
  - Poor reliability
- Write-on-close

File Sharing Semantics

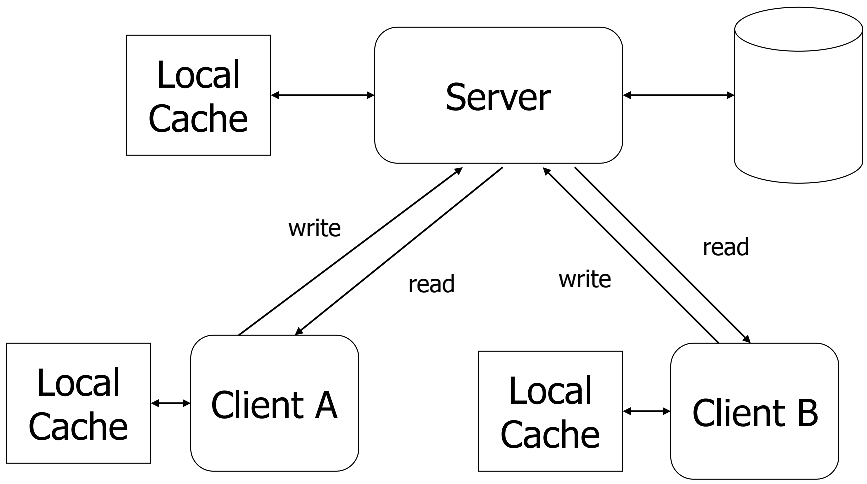
- Sequential Semantics
  - No cache
  - Performance problems
  - Write-through cache
    - Must notify clients holding copies
- Session Semantics
- Immutable Files
- Atomic Transactions

13

NFS: NETWORK FILE SYSTEM

14

Caching in NFS



15

Client Caching

- Client checks validity of cached files
  - File open
  - Periodic polling
- Client responsible for writing out cache
  - Periodic scan, flush of dirty blocks

16

NFS Semantics

- Locking
  - Originally separate (stateless)
  - Stateful for locking since NFS v4
- Unix file semantics not guaranteed
  - E.g., read after write
- Session semantics not even guaranteed
  - Intermediate writes
  - Client may implement close-to-open

17

NFS Implementation

- Remote procedure calls for all operations
  - Originally over UDP
  - Using TCP since NFSv4
- Lost requests are simply re-transmitted
  - At-least-once semantics

18

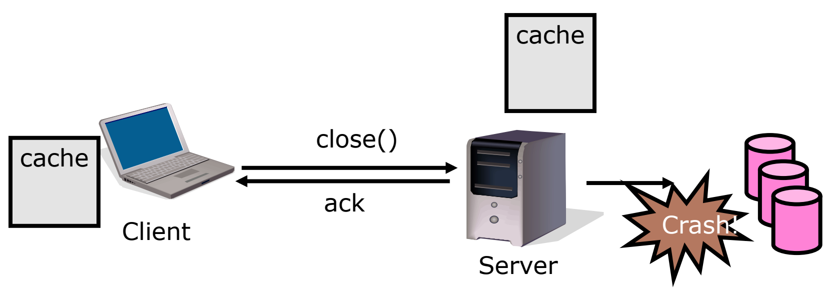
NFS Failure Recovery

- Server crashes transparent to client
  - Each client request self-contained
  - Client retransmits request if crash
    - *"Server not responding, still trying"*

- Client crashes transparent to server

19

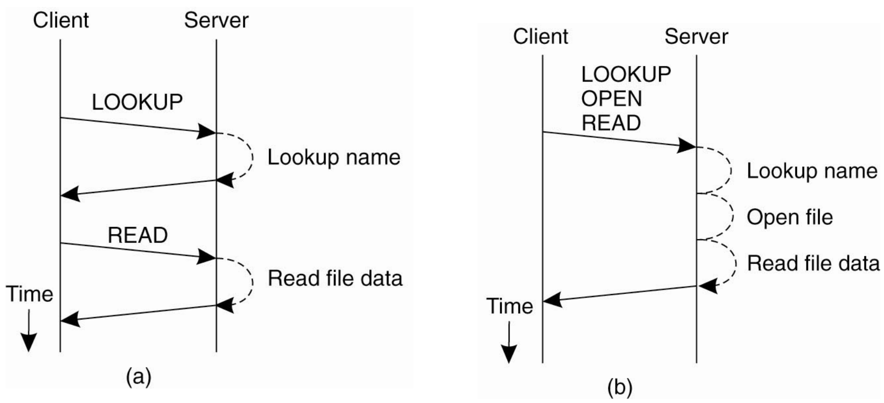
Caching and Failures



- Suppose client implements close-to-open
- Server acks updates
- Server crashes before flushing updates to disk
- Fixes:
  - Server flushes updates before ack
  - NVRAM for server
  - NFS v3: client buffers updates until COMMIT acknowledged

20

Compound RPC



21

Open Delegation

- Server may delegate open/close/locking to client
- Operations done locally at client
- Periodic cache checks unnecessary
- Lease and revocation via callback
  - RPC from server to client

22

AFS: ANDREW FILE SYSTEM

23

Andrew File System (AFS)

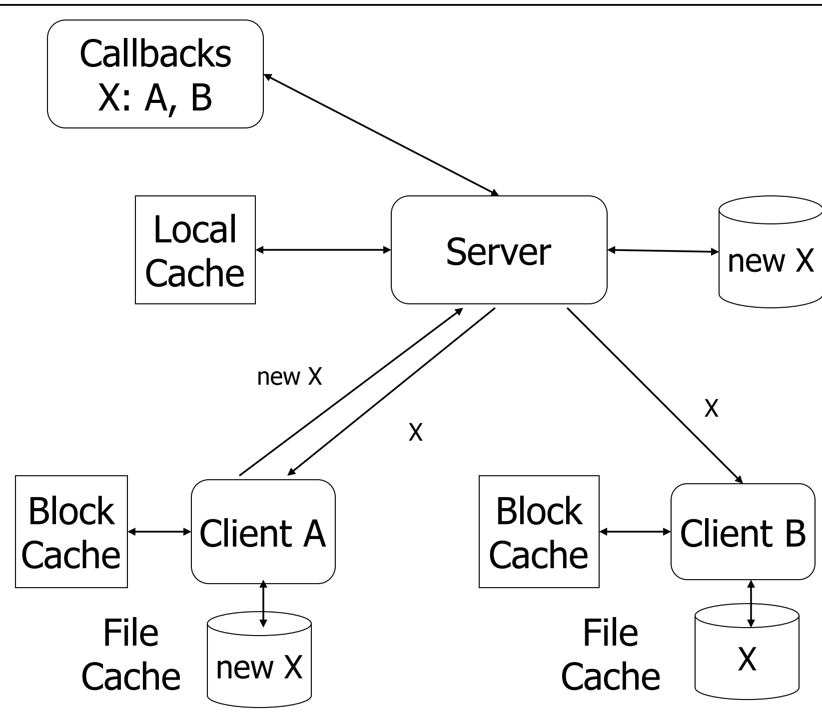
- Stateful
- Single name space
  - Same name anywhere
- Local file caching
  - On workstation disks
  - For long periods of time
  - Originally whole files, now 64K file chunks.

24

Andrew File System (AFS)

- Callbacks on server record clients
  - Poll clients on crash+recovery
- Write-through on close
  - Server updated *only on close*
  - Server informs other clients
  - Clients fetch new version *on next open*
- Session semantics
- Cache files on local disk

25



26

Andrew File System (AFS)

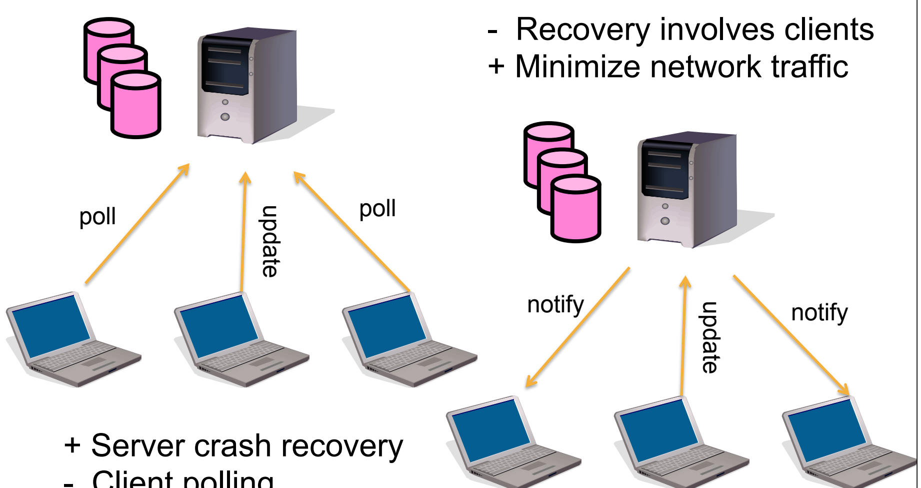
- Reduce message traffic.
  - All operations performed locally
  - No client polling
- On file open()
  - Fetch new copy if callback was received
  - Otherwise use locally-cached copy
- Server crashes
  - Transparent to client if file is locally cached
  - Server must contact clients to find state of files

27

NFS

AFS

- Recovery involves clients
+ Minimize network traffic



+ Server crash recovery
- Client polling

28

Based on material by
Alex Moshchuk, University of Washington

GOOGLE FILE SYSTEM (GFS)

29

Motivation

- Massive distributed data store
  - Redundant storage
  - **Massive** amounts of data
  - Cheap and unreliable computers
- Not general purpose
  - Data consistency checking done by application

30

Assumptions

- High component failure rates
- "Modest" number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- *High sustained throughput favored over low latency*

31

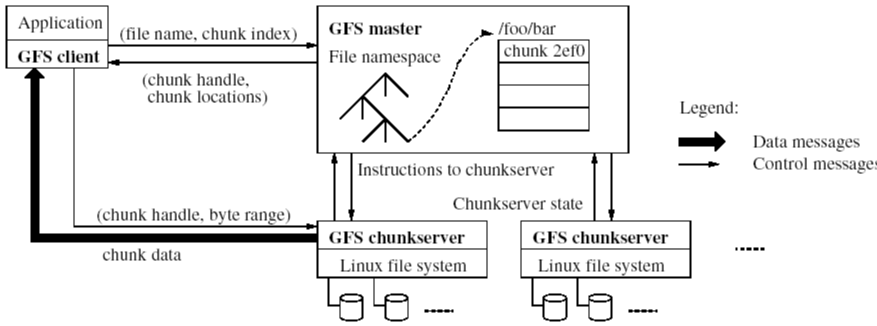
GFS Design Decisions

- Files stored as **chunks**
  - Fixed size (64MB)
- Reliability through **replication**
  - Each chunk replicated across 3+ chunkservers
- Single **master** to coordinate, keep metadata
- No data caching
- Familiar interface, but customize the API
  - Add **snapshot** and **record append** operations

32

GFS Architecture

- Single master
- Multiple chunkservers



…Can anyone see a potential weakness in this design?

33

Single master

- This is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - *Shadow masters*
  - *Minimize master involvement*
    - Use only for metadata
    - large chunk size
    - Delegate authority to primary replicas for data mutations (**chunk leases**)

34

Metadata (1/2)

- Global metadata is stored on the **master**
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
  - Fast, easily accessible

35

Metadata (2/2)

- **Operation log** of critical metadata updates
  - Persistent on master local disk
  - Replicated to backup master servers
  - Checkpoints for faster recovery

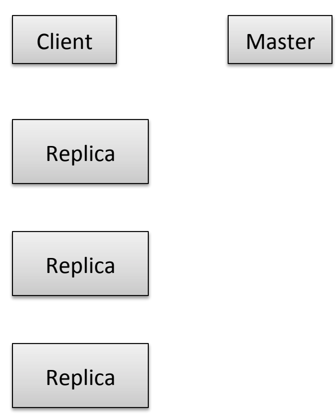
36

GFS: MUTATIONS

37

Mutations

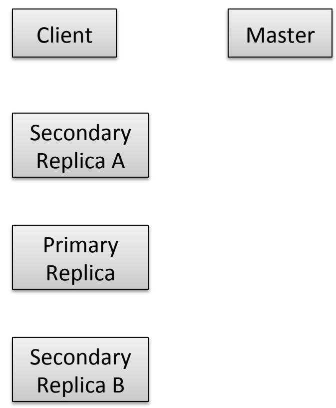
- Mutation = write or append
  - must be done for all replicas
- Goal: minimize master involvement



38

Mutations

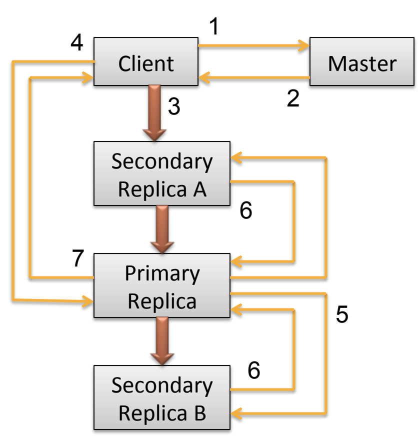
- **Lease** mechanism:
  - Primary replica



39

Mutations

- **Data flow** decoupled from **control flow**:
  - Master: meta-data only
  - Primary replica arranges data transfers
  - Data path has no splitting



40

Atomic record append

- Client specifies data
- GFS appends it to the file atomically *at least once*
  - *GFS picks the offset*
  - works for concurrent writers
- Used heavily by Google apps
  - e.g., Multiple-producer/single-consumer queues

41

GFS: DATA CONSISTENCY

42

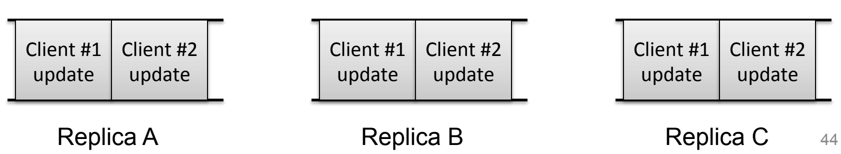
Metadata Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Log defines global order

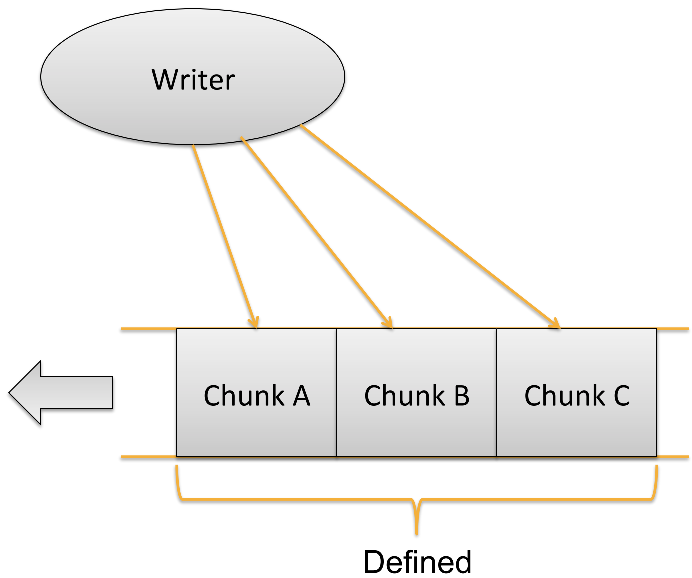
Data Consistency Model

- "Consistent" = all replicas have the same value

- Changes to data are **ordered** by a primary
  - All replicas will be "consistent"
  - But concurrent writes may be interleaved
    - Interleaving of compound updates

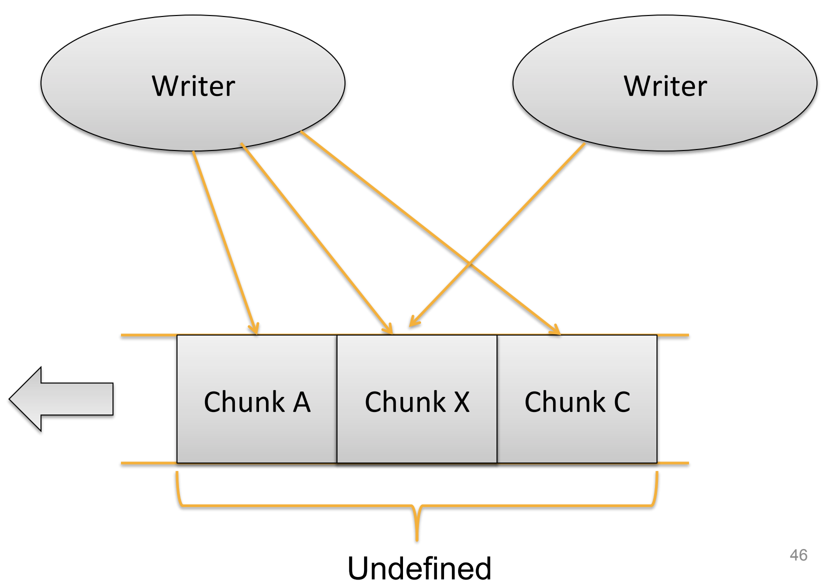


Data Consistency



45

Consistent But Undefined



46

Consistent and Defined

- "Consistent" = all replicas have the same value
- "Defined" = consistent & replica reflects the mutation

- Some properties:
  - Concurrent writes leave region **consistent**
  - Concurrent writes may leave region **undefined**
    - Same corrupted write at all replicas
  - Failed writes leave the region **inconsistent**

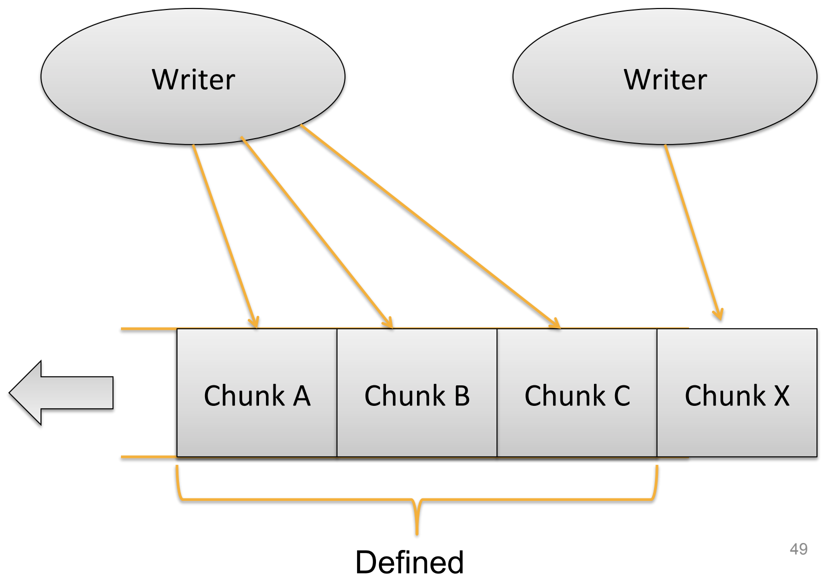
47

Data Consistency Model

	Write
serial success	defined
concurrent success	consistent but undefined
failure	inconsistent

48

Record Append



49

Consistent and Defined

- "Consistent" = all replicas have the same value
- "Defined" = consistent & replica reflects the mutation

- Record append completes *at least* once
  - Offset of append chosen by primary
  - *Applications* must cope with possible duplicates

50

Data Consistency Model

	Write	Record Append
serial success	defined	defined interspersed with inconsistent
concurrent success	consistent but undefined	
failure	inconsistent	

Implications

- Some work has moved into the applications:
  - e.g., self-validating, self-identifying records
- Namespace updates atomic and serializable
  - Single master server

Fault Tolerance

- High availability
  - fast recovery
    - master and chunkservers restartable in a few seconds
  - chunk replication
    - default: 3 replicas
  - shadow masters
- Data integrity
  - checksum every 64KB block in each chunk

53

Conclusion

- How to support large-scale processing workloads on commodity hardware
  - design for frequent component failures
  - optimize for huge files that are mostly appended and read
  - go for simple solutions (e.g., single master)

54