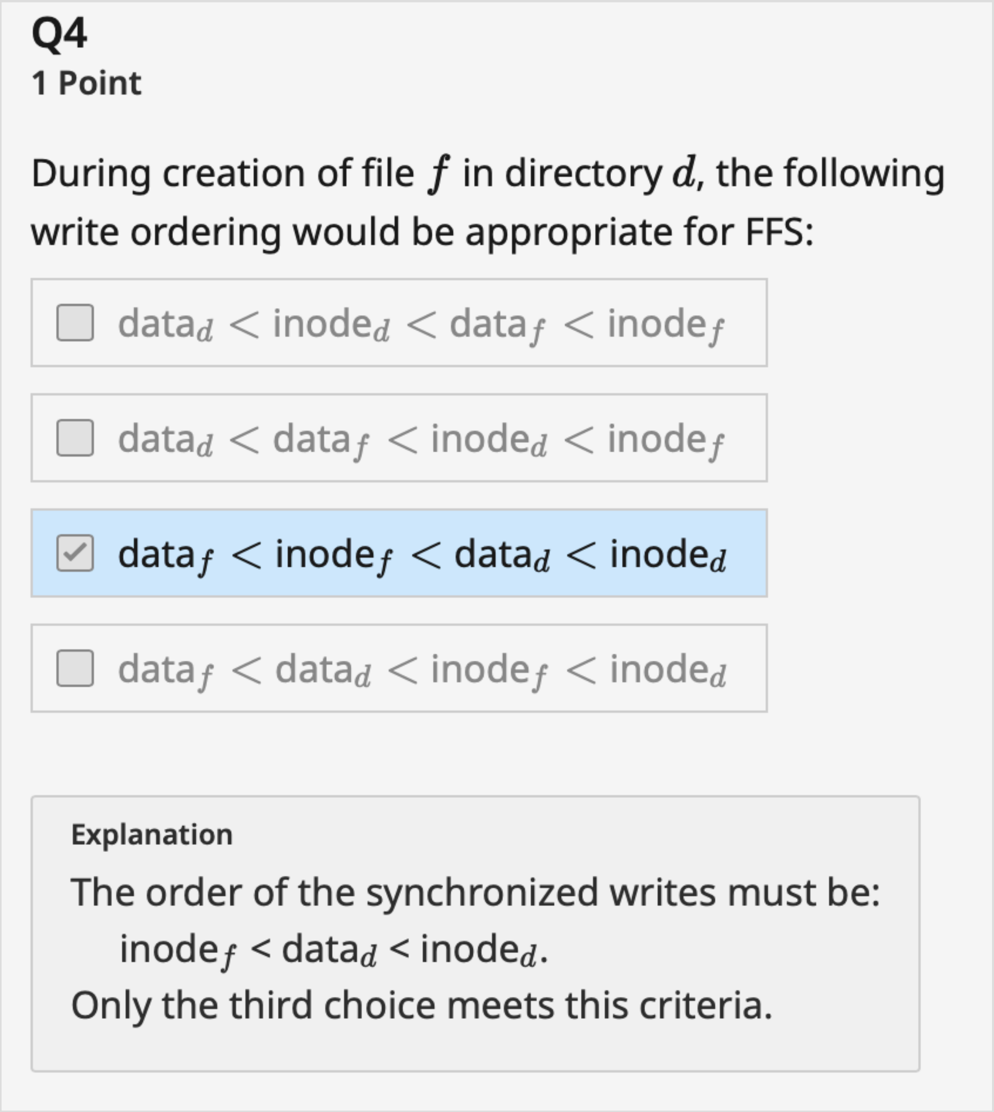
Distributed Systems

- *48 - Communication Basics*
- *49 - NFS*
- *50 - AFS*
- GFS

RH 9 *review*



Q4

1 Point

During creation of file f in directory d, the following write ordering would be appropriate for FFS:

☐ $\text{data}_d < \text{inode}_d < \text{data}_f < \text{inode}_f$

☐ $\text{data}_d < \text{data}_f < \text{inode}_d < \text{inode}_f$

☑ $\text{data}_f < \text{inode}_f < \text{data}_d < \text{inode}_d$

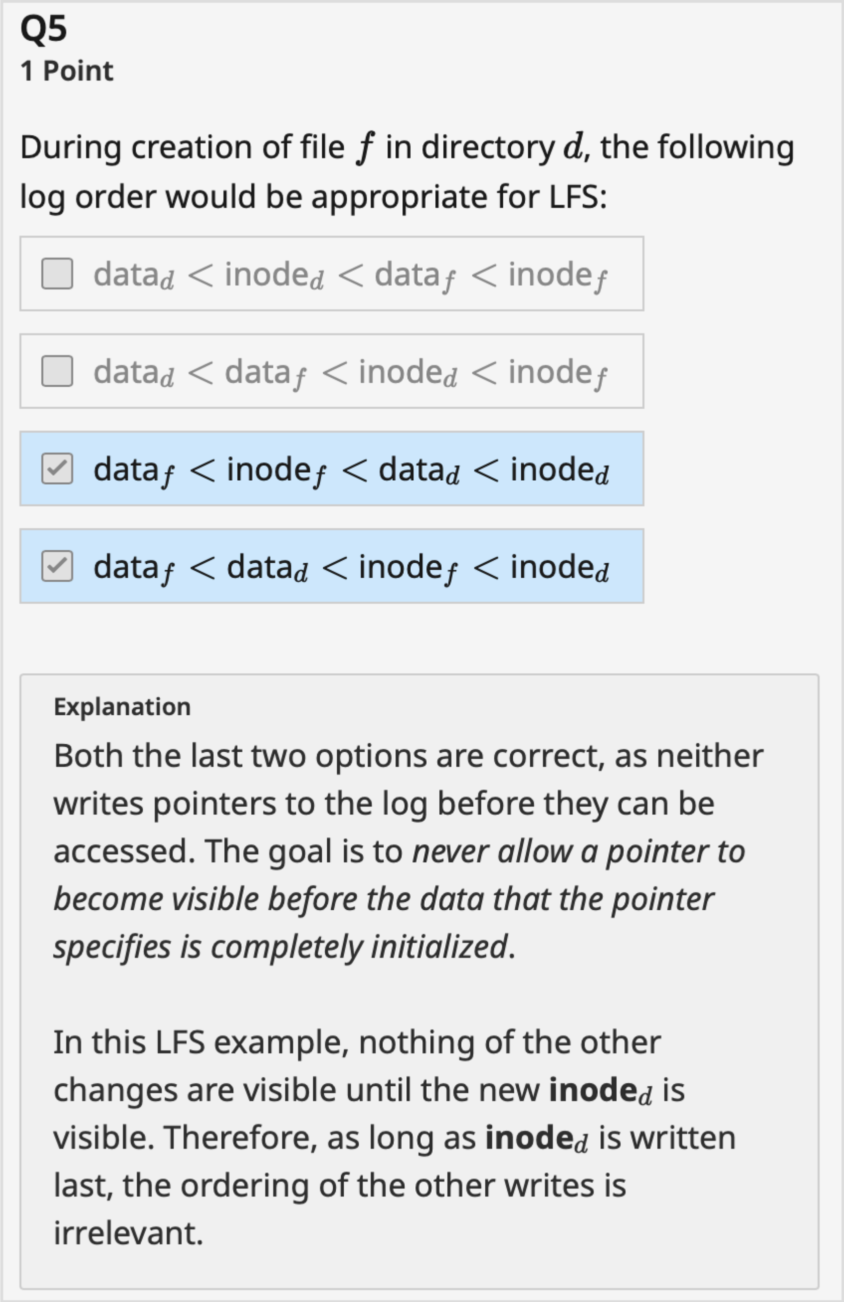
☐ $\text{data}_f < \text{data}_d < \text{inode}_f < \text{inode}_d$

Explanation

The order of the synchronized writes must be:

$\text{inode}_f < \text{data}_d < \text{inode}_d$.

Only the third choice meets this criteria.



Q5

1 Point

During creation of file f in directory d, the following log order would be appropriate for LFS:

☐ $\text{data}_d < \text{inode}_d < \text{data}_f < \text{inode}_f$

☐ $\text{data}_d < \text{data}_f < \text{inode}_d < \text{inode}_f$

☑ $\text{data}_f < \text{inode}_f < \text{data}_d < \text{inode}_d$

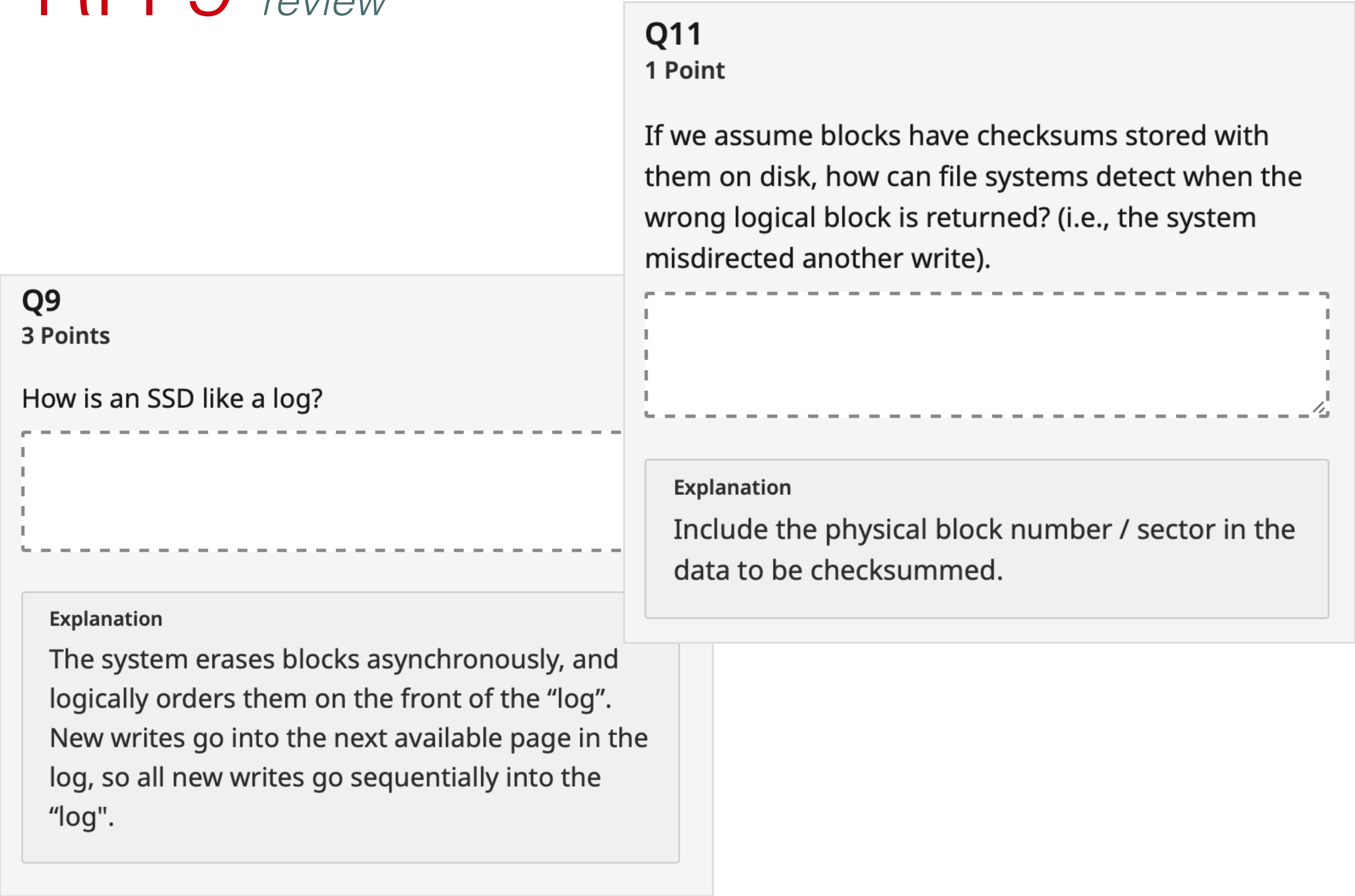
☑ $\text{data}_f < \text{data}_d < \text{inode}_f < \text{inode}_d$

Explanation

Both the last two options are correct, as neither writes pointers to the log before they can be accessed. The goal is to *never allow a pointer to become visible before the data that the pointer specifies is completely initialized*.

In this LFS example, nothing of the other changes are visible until the new **inode**$_d$ is visible. Therefore, as long as **inode**$_d$ is written last, the ordering of the other writes is irrelevant.

RH 9 *review*



Q9

3 Points

How is an SSD like a log?

Explanation

The system erases blocks asynchronously, and logically orders them on the front of the “log”. New writes go into the next available page in the log, so all new writes go sequentially into the “log".

Q11

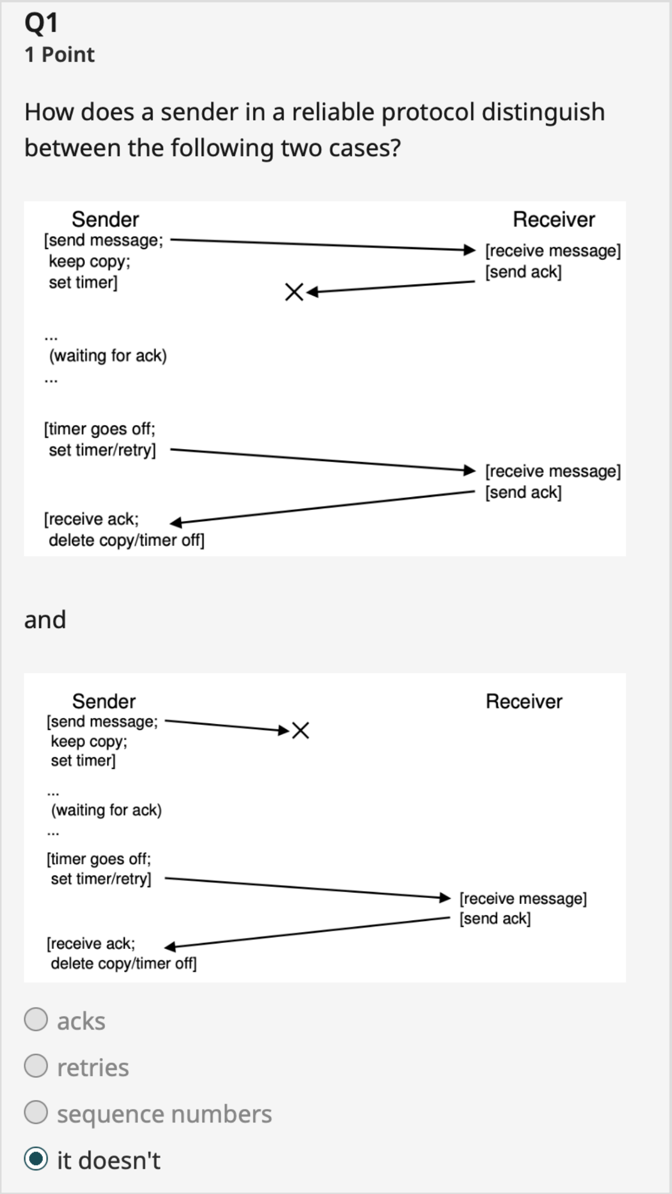
1 Point

If we assume blocks have checksums stored with them on disk, how can file systems detect when the wrong logical block is returned? (i.e., the system misdirected another write).

Explanation

Include the physical block number / sector in the data to be checksummed.

RH 10 *review*

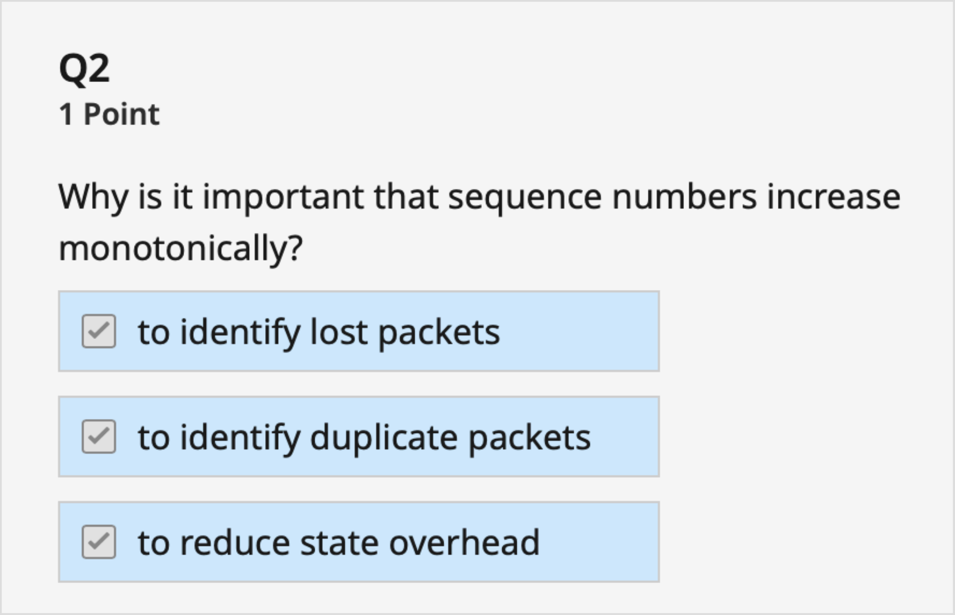


Q1
1 Point

How does a sender in a reliable protocol distinguish between the following two cases?

and

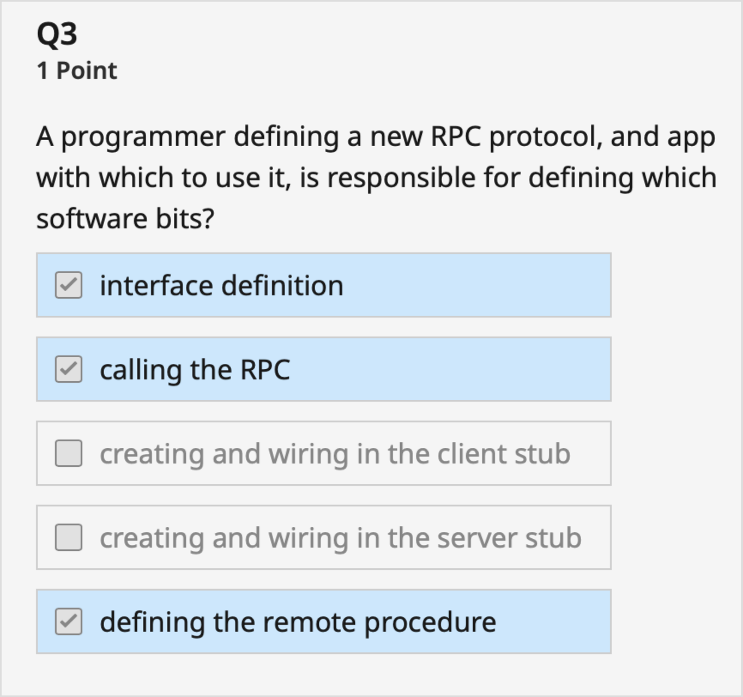
○ acks
○ retries
○ sequence numbers
● it doesn't



Q2
1 Point

Why is it important that sequence numbers increase monotonically?

☑ to identify lost packets
☑ to identify duplicate packets
☑ to reduce state overhead

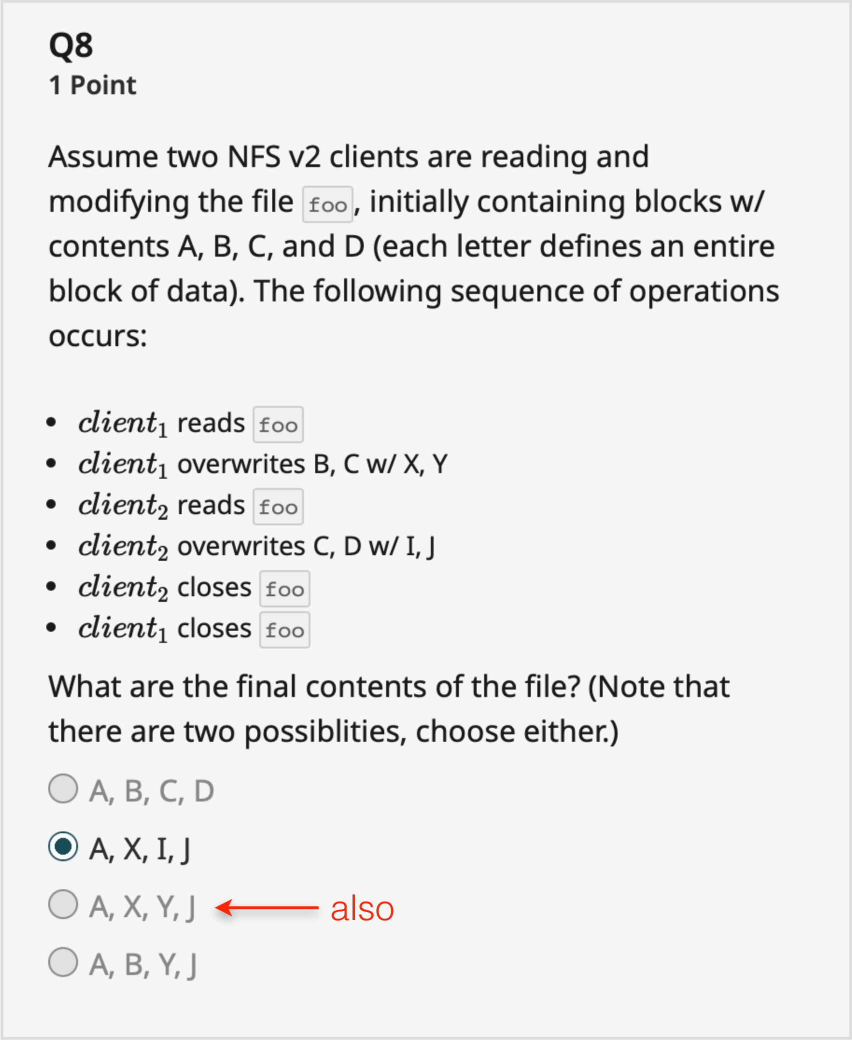


Q3
1 Point

A programmer defining a new RPC protocol, and app with which to use it, is responsible for defining which software bits?

☑ interface definition
☑ calling the RPC
□ creating and wiring in the client stub
□ creating and wiring in the server stub
☑ defining the remote procedure

RH 10 *review*



Q8
1 Point

Assume two NFS v2 clients are reading and modifying the file `foo`, initially containing blocks w/ contents A, B, C, and D (each letter defines an entire block of data). The following sequence of operations occurs:

- $client_1$ reads `foo`
- $client_1$ overwrites B, C w/ X, Y
- $client_2$ reads `foo`
- $client_2$ overwrites C, D w/ I, J
- $client_2$ closes `foo`
- $client_1$ closes `foo`

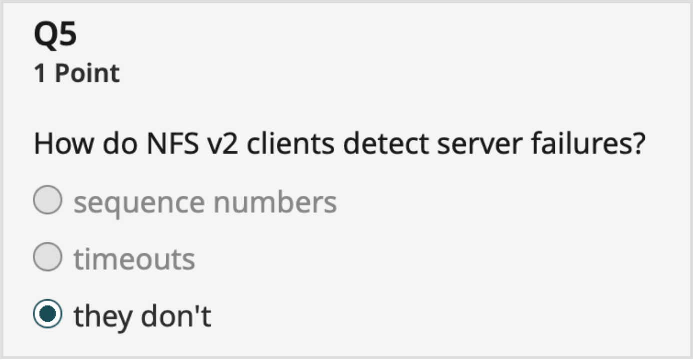
What are the final contents of the file? (Note that there are two possiblities, choose either.)

○ A, B, C, D

● A, X, I, J

○ A, X, Y, J ← also

○ A, B, Y, J



Q5
1 Point

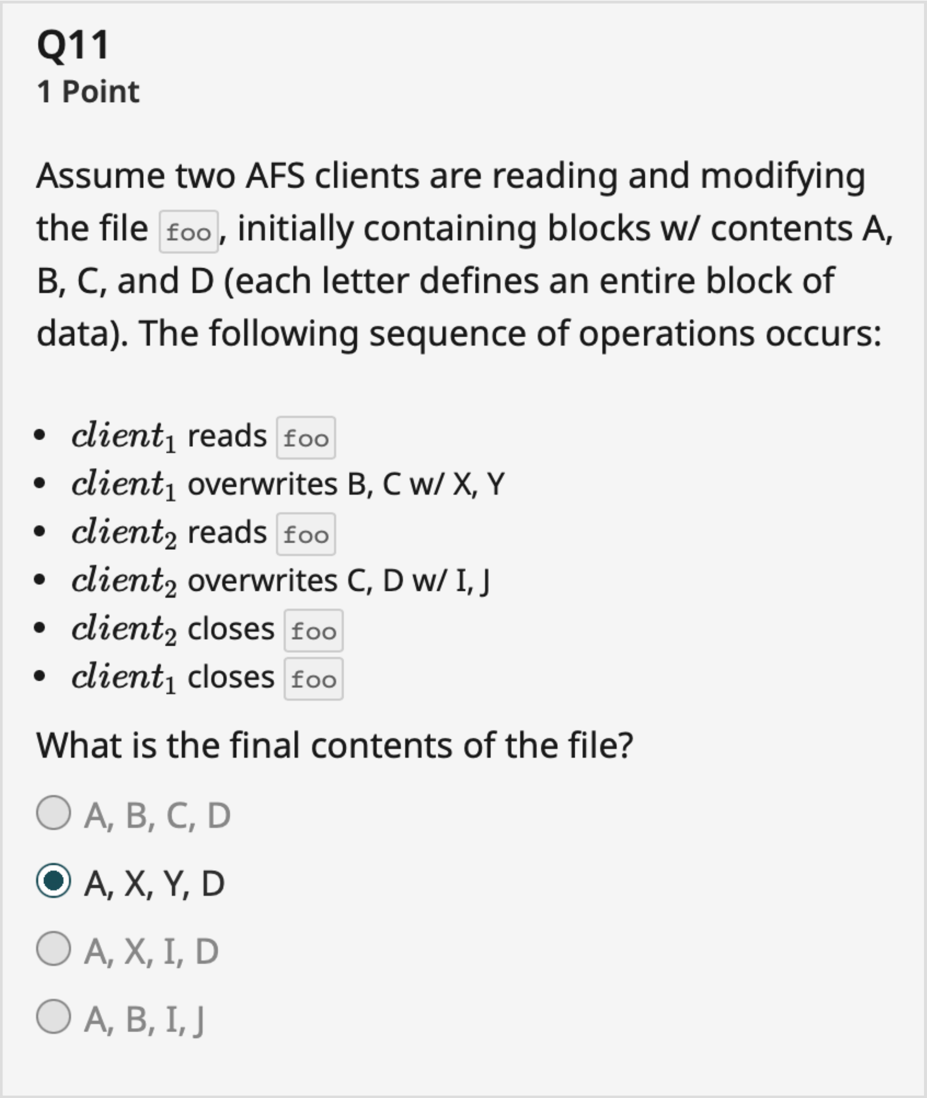
How do NFS v2 clients detect server failures?

○ sequence numbers

○ timeouts

● they don't

RH 10



Q11

1 Point

Assume two AFS clients are reading and modifying the file `foo`, initially containing blocks w/ contents A, B, C, and D (each letter defines an entire block of data). The following sequence of operations occurs:

- $client_1$ reads `foo`
- $client_1$ overwrites B, C w/ X, Y
- $client_2$ reads `foo`
- $client_2$ overwrites C, D w/ I, J
- $client_2$ closes `foo`
- $client_1$ closes `foo`

What is the final contents of the file?

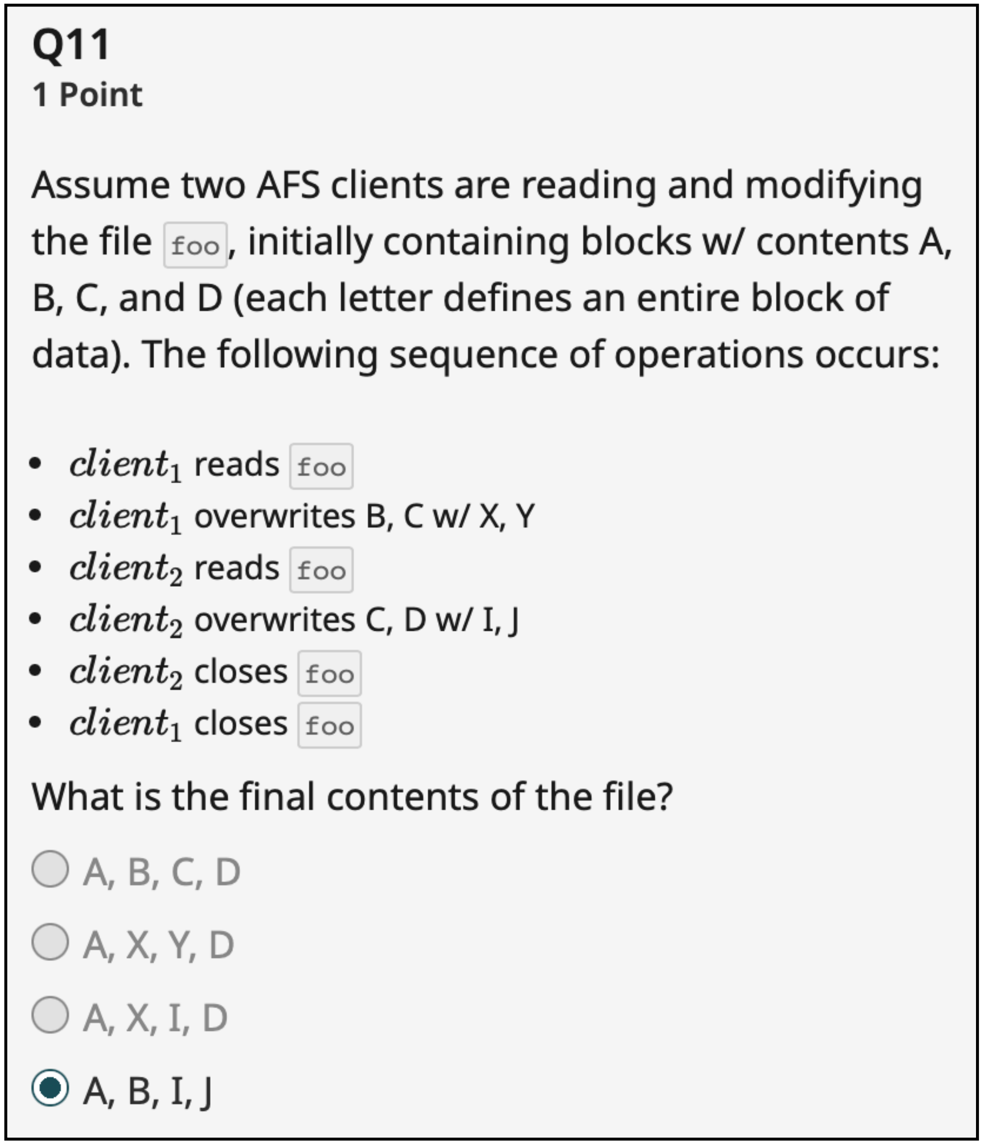
○ A, B, C, D

● A, X, Y, D

○ A, X, I, D

○ A, B, I, J

RH 10



Q11

1 Point

Assume two AFS clients are reading and modifying the file `foo`, initially containing blocks w/ contents A, B, C, and D (each letter defines an entire block of data). The following sequence of operations occurs:

- $client_1$ reads `foo`
- $client_1$ overwrites B, C w/ X, Y
- $client_2$ reads `foo`
- $client_2$ overwrites C, D w/ I, J
- $client_2$ closes `foo`
- $client_1$ closes `foo`

What is the final contents of the file?

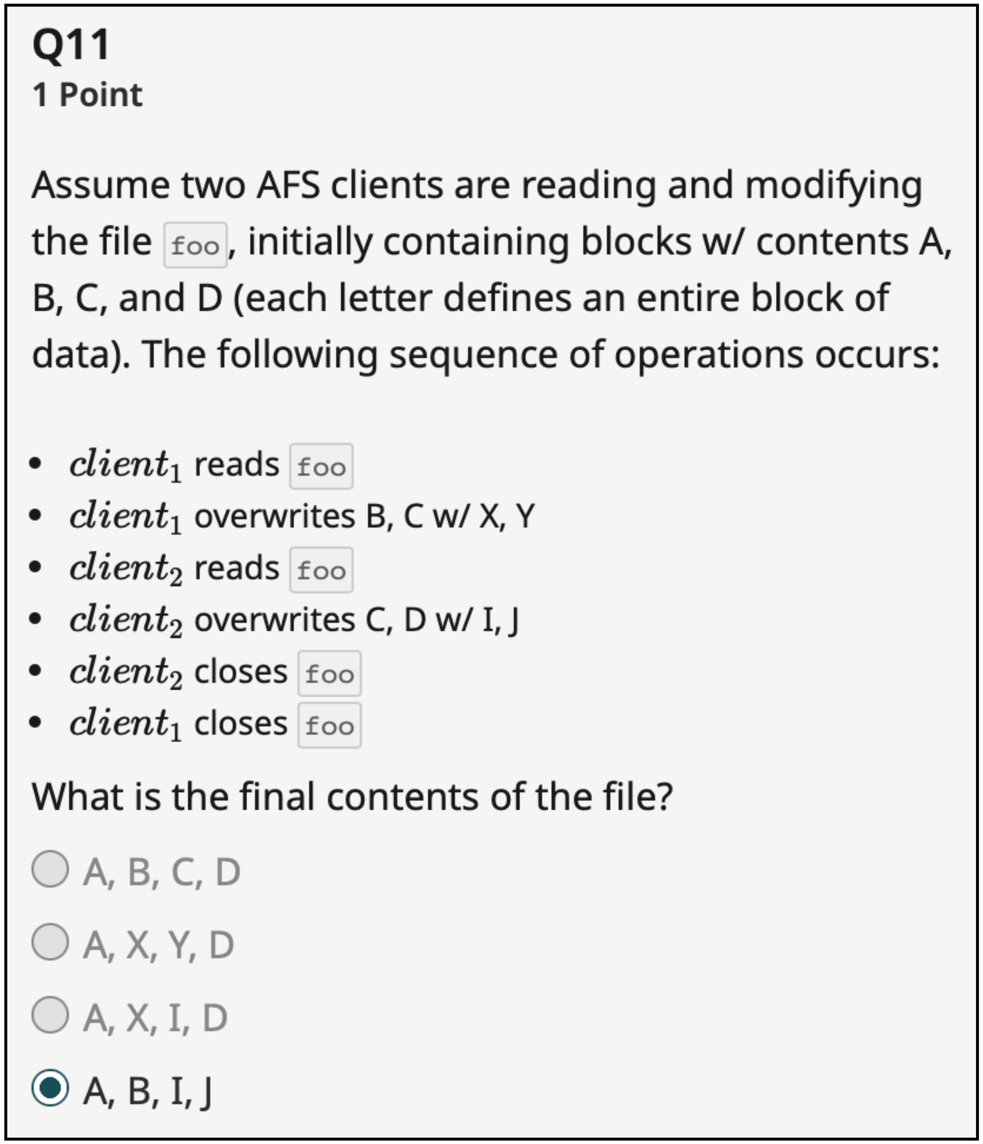
○ A, B, C, D

○ A, X, Y, D

○ A, X, I, D

● A, B, I, J

RH 10



Q11
1 Point

Assume two AFS clients are reading and modifying the file `foo`, initially containing blocks w/ contents A, B, C, and D (each letter defines an entire block of data). The following sequence of operations occurs:

- $client_1$ reads `foo`
- $client_1$ overwrites B, C w/ X, Y
- $client_2$ reads `foo`
- $client_2$ overwrites C, D w/ I, J
- $client_2$ closes `foo`
- $client_1$ closes `foo`

What is the final contents of the file?

○ A, B, C, D

○ A, X, Y, D

○ A, X, I, D

● A, B, I, J

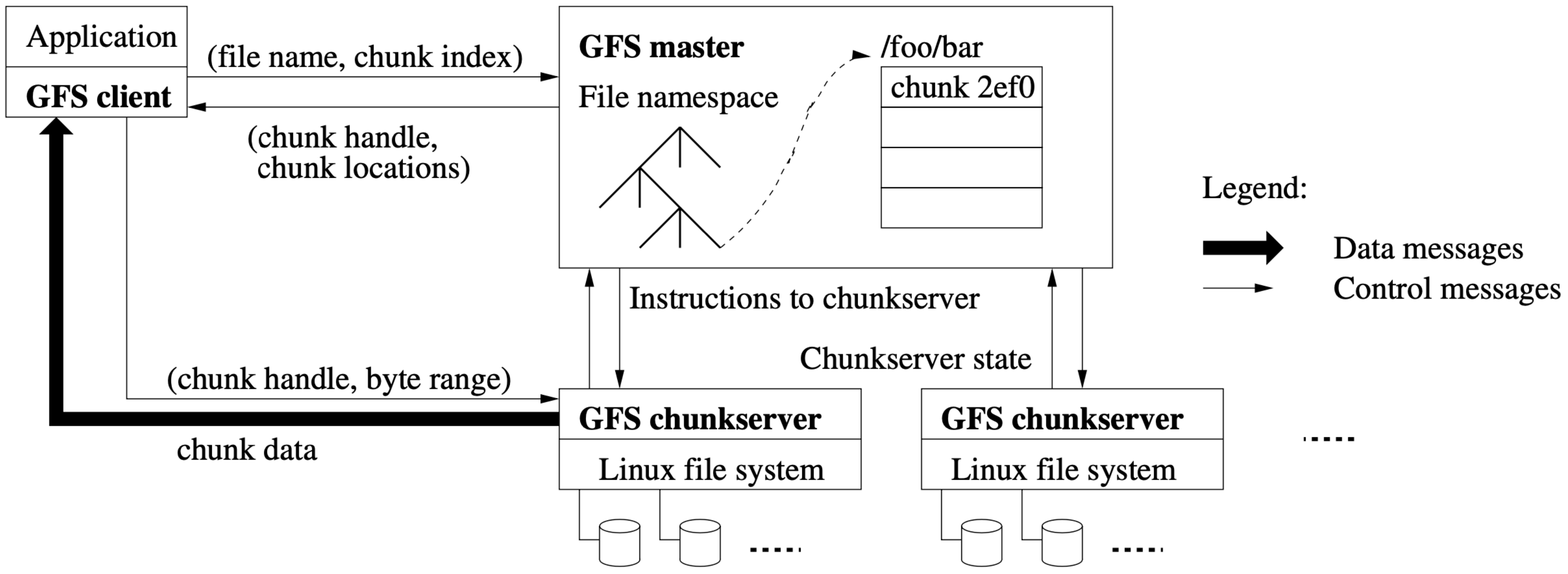
Google File System *v1*

- Needs
  - need to handle *massive* files
  - most mutations are appends
  - co-design w/ applications (*also an advantage*)
- Assumptions
  - built from hundreds, or thousands, of cheap machines
  - failures are the common case
- Features
  - relaxed consistency (*also an advantage*)
  - atomic record append (without locking)
  - no data caches
    - append-only model means re-use not common
    - host operating system does limited caching anyway

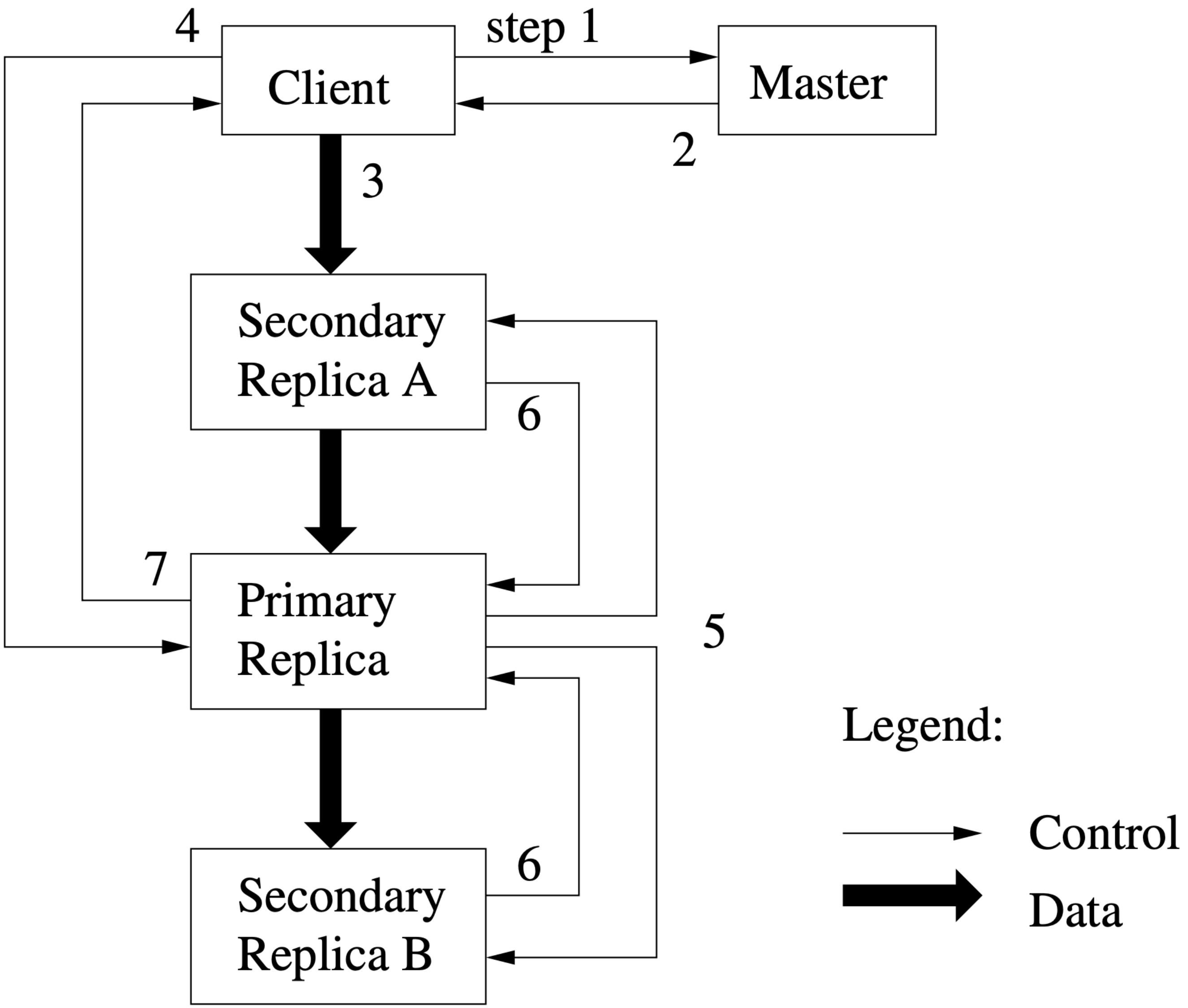
GFS *two types of nodes*

- multiple *chunk servers*
  - hold fixed size *chunks*
  - immutable once written
  - identified by a globally unique 64-bit ID
- coordinator (GFS *master*)
  - single machine holds all *metadata* in memory
    - persistent
      - file and chunk namespaces (think directories)
      - mappings from files to chunks
      - persistent by flushing *operations log* locally, remotely before visible
    - soft state
      - locations of chunk replicas
      - on startup or recovery restore by asking chunkservers
    - total state is 64 bytes for each 64MB chunk
    - background garbage collection, replica reassignment and balancing

GFS *architecture, and read*



GFS pipelined *writes*



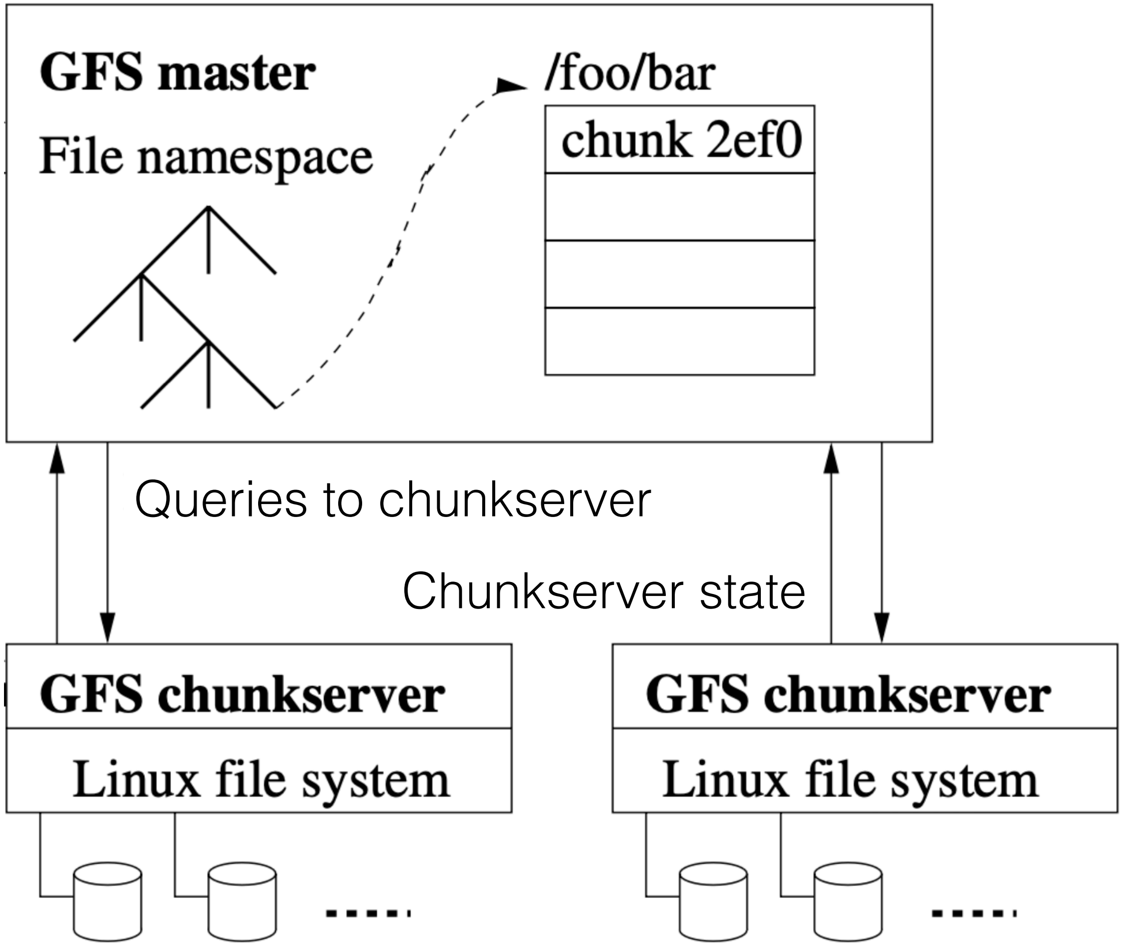
GFS *reliability*

- startup and recovery treated identically:
  - master polls all chunkservers for chunks they cache
  - read namespace info from locally persistent state
- other
  - master has *shadows* that are “almost” up to date
  - chunkservers can flush to disk asynchronously because of replication

GFS *consistency model*

- update consistency
  - file namespace mutations are atomic (handled by master)
  - state of a file region after append can be:
    - *consistent* if clients all guaranteed to see same data
    - *defined* if consistent and *last mutation correct not interleaved*
  - *concurrent updates* may leave system undefined, but consistent
    - all see same data, but may be mingled fragments of updates
    - usually when large writes broken into fragments
    - enough information for *application library* to fix
  - confusing
- cache consistency
  - no caches

During Recovery



GFS *summary*

- System for:
  - very large files (logs, like for web indexing)
  - very large writes
  - reads usually sequential through whole log
- Replication approach:
  - single master
  - multiple chunkservers
  - very simple consistency and recovery
  - single master only involved in lookups, not read or write
- Long-term view:
  - single master was a mistake

yes it’s on the exam