GFS

Emmett Witchel

cs380L

GFS faux quiz (any 2, 5 min):

- Why does GFS not need to hook into the VFS layer?
- Do GFS clients cache data or metadata? Why/why not?
- What is GFS' replication factor? How was it chosen?
- Does GFS support hard links? Why/why not?
- What are some tradeoffs around having the location of GFS chunk replicas be persistent or non-persistent?
- What is the relationship between master RAM capacity and GFS capacity?

(Recall) a (seemingly) very simple problem

My computer

Some other computer

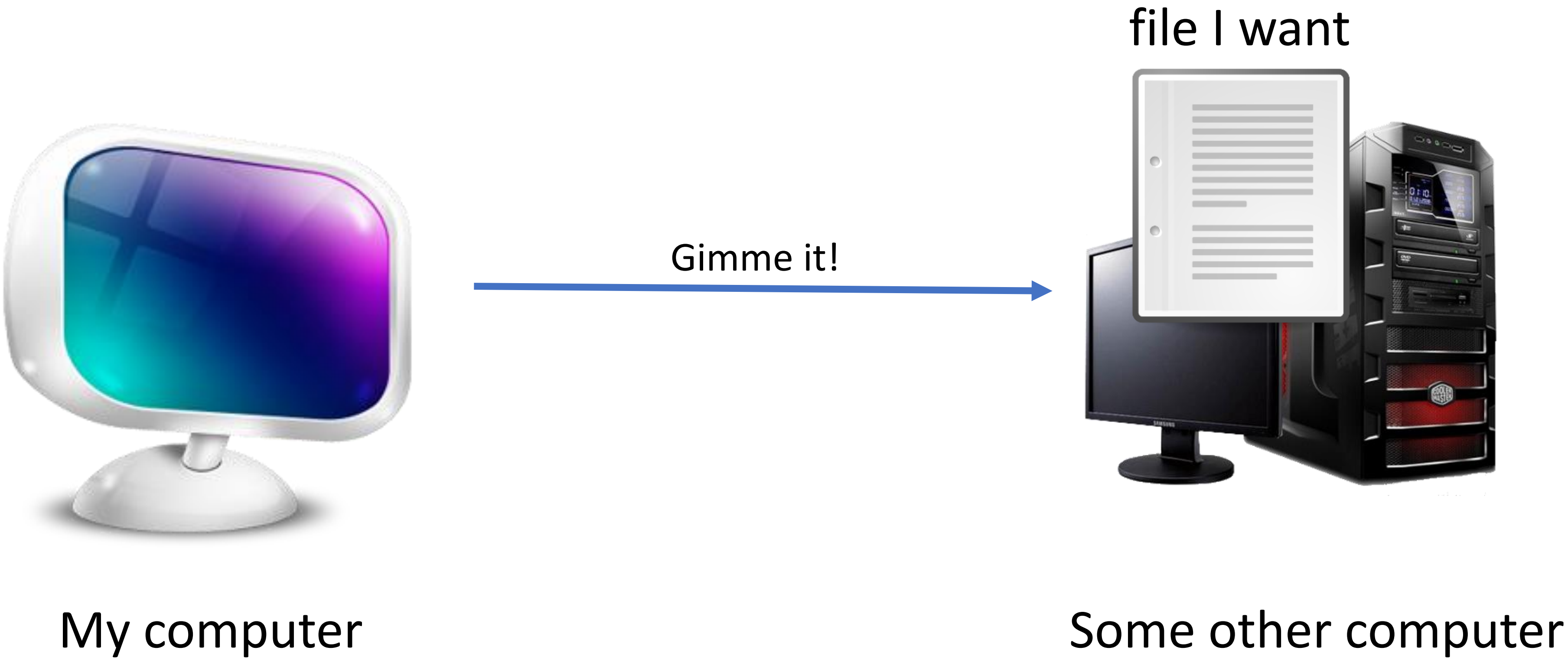
(Recall) a (seemingly) very simple problem

file I want

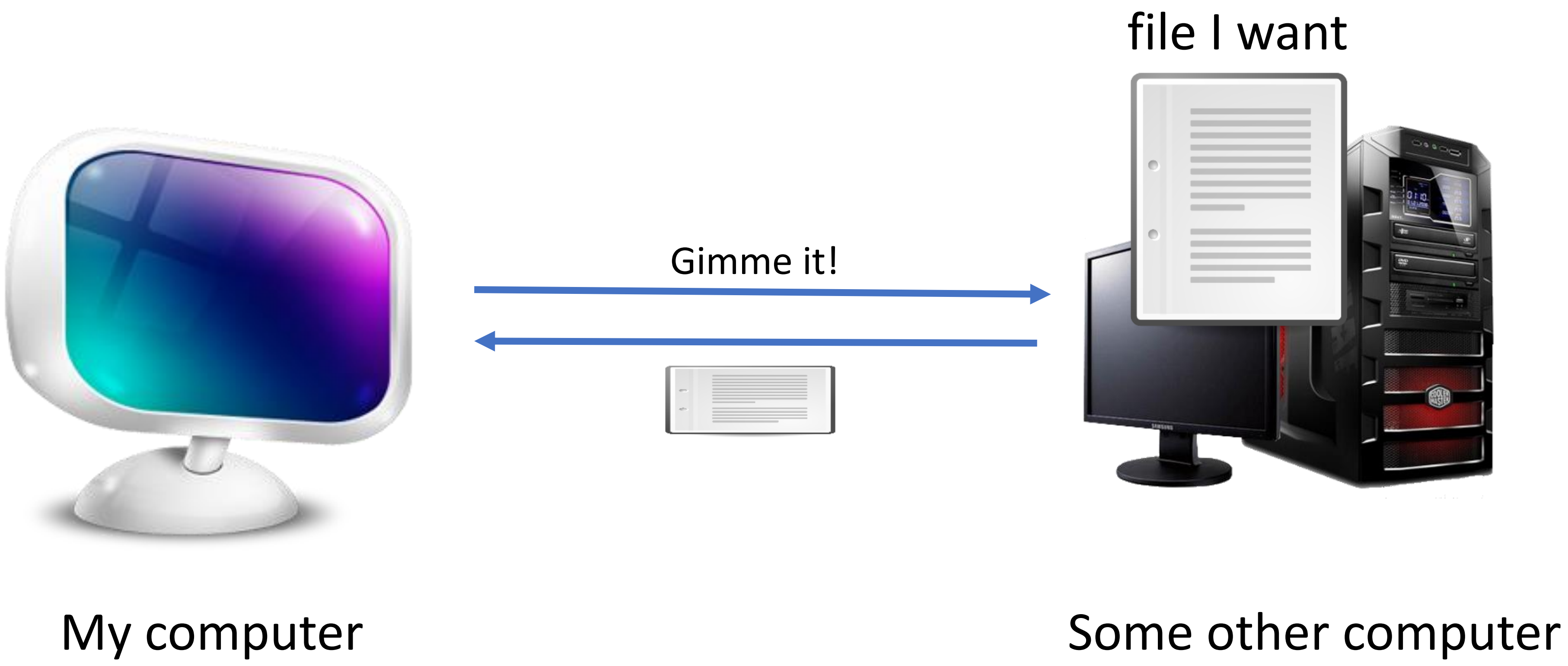
My computer

Some other computer

(Recall) a (seemingly) very simple problem



(Recall) a (seemingly) very simple problem



(Recall) a (seemingly) very simple problem

My computer

Some other computer(s)

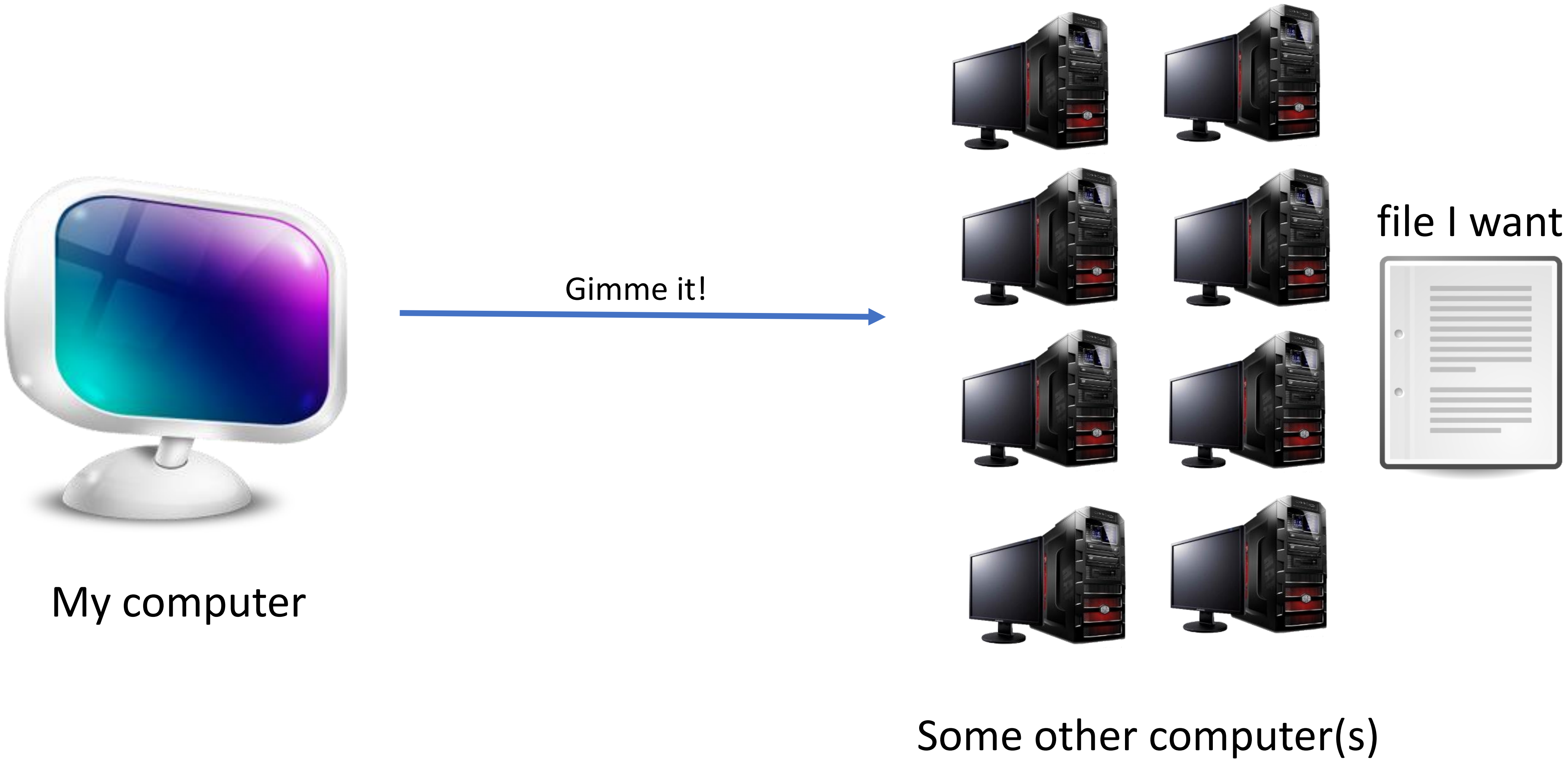
(Recall) a (seemingly) very simple problem

My computer

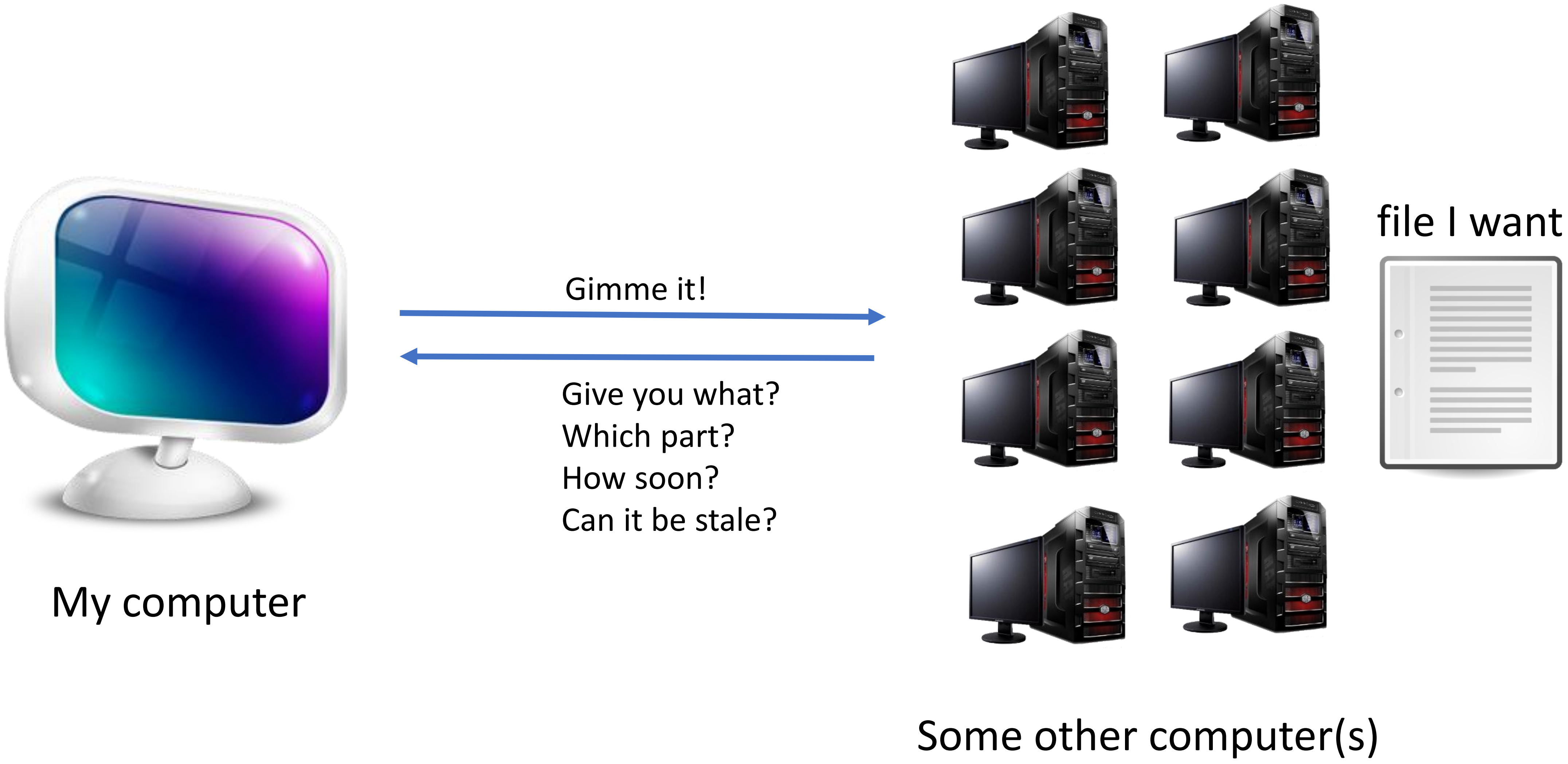
file I want

Some other computer(s)

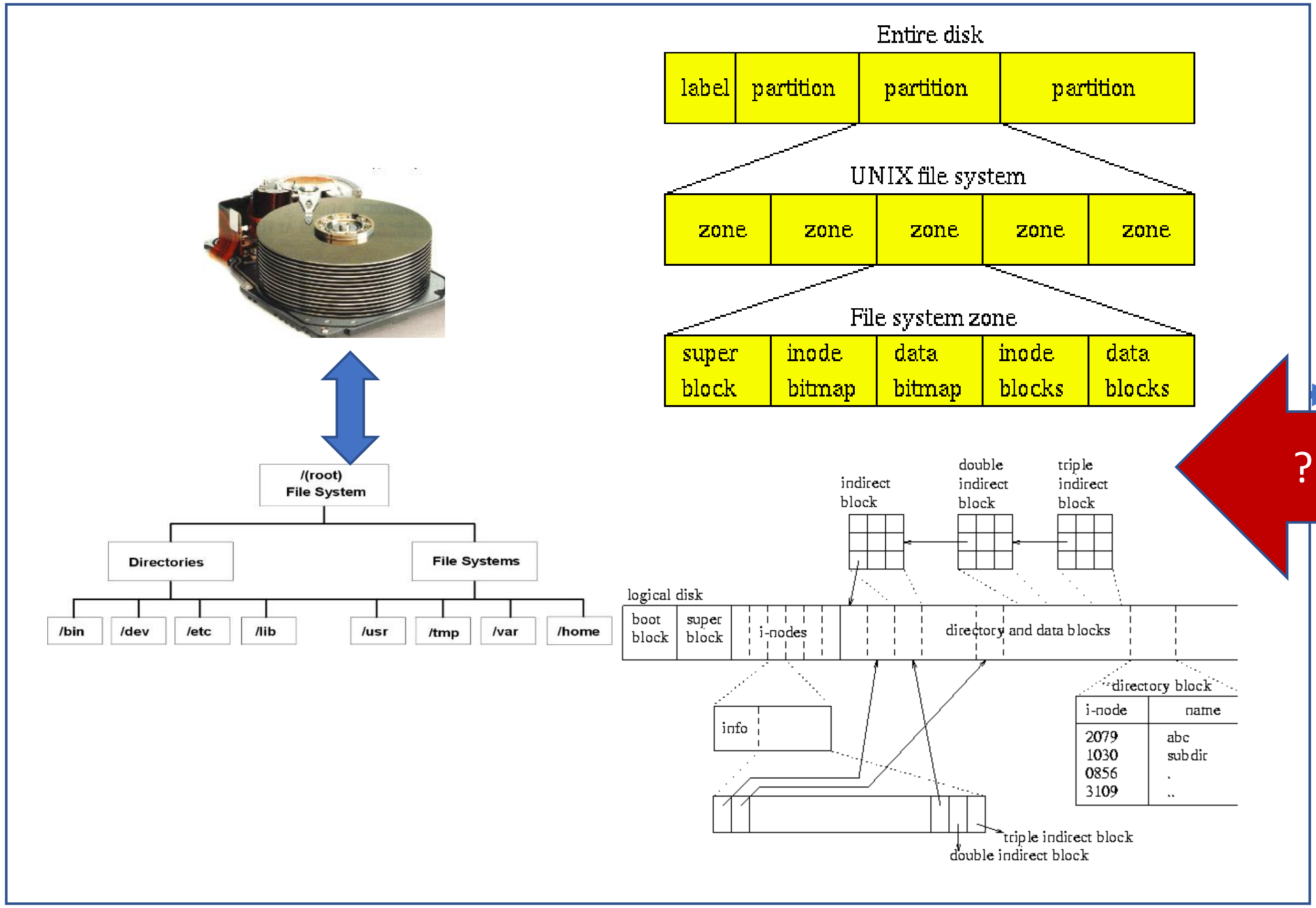
(Recall) a (seemingly) very simple problem

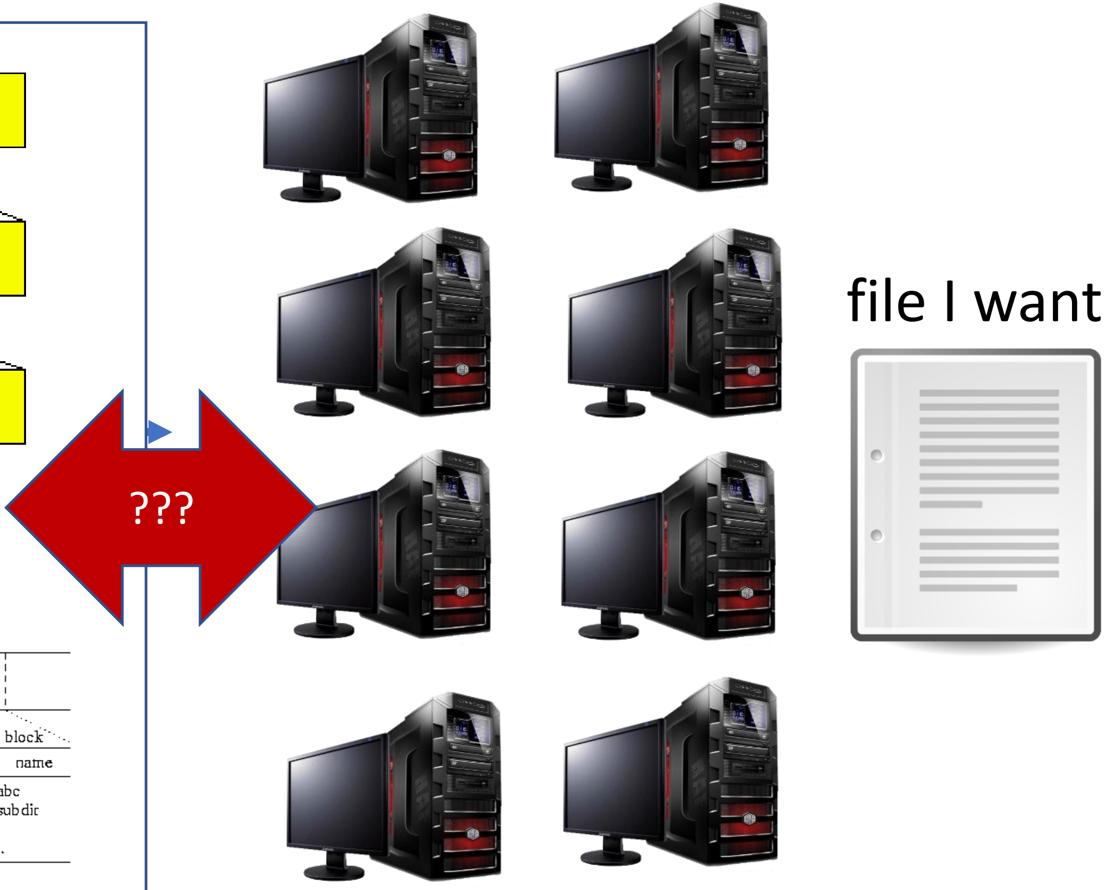


(Recall) a (seemingly) very simple problem



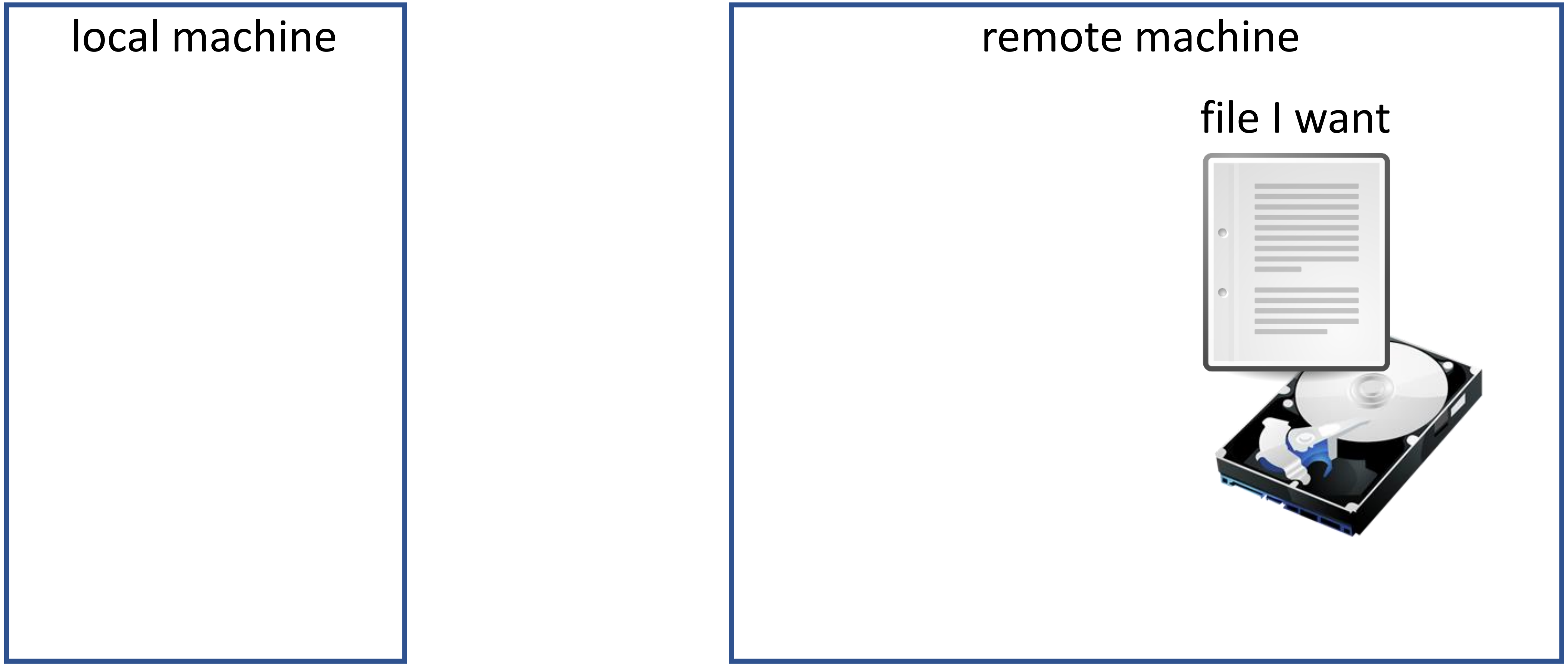
(Recall) a (seemingly) very simple problem



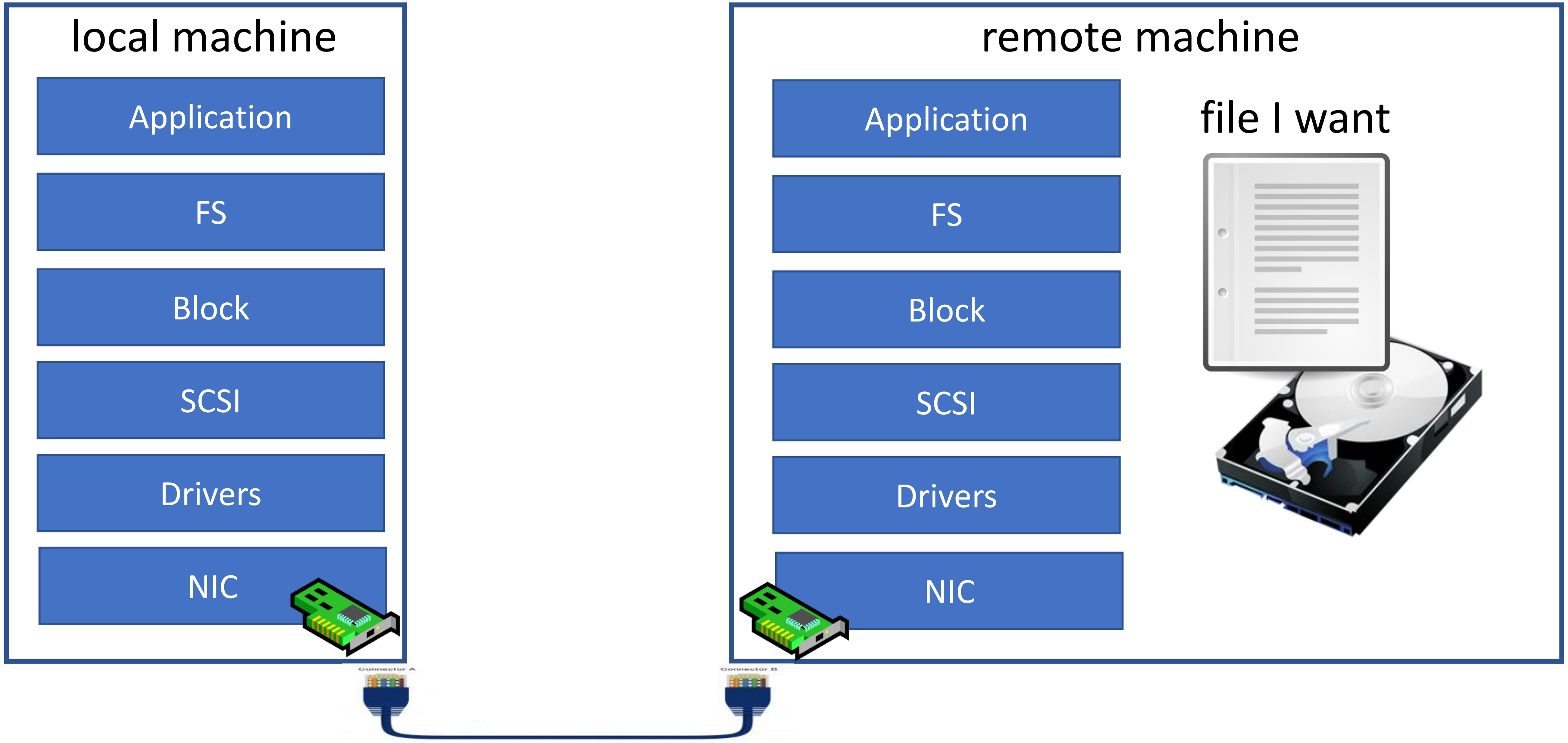


Some other computer(s)

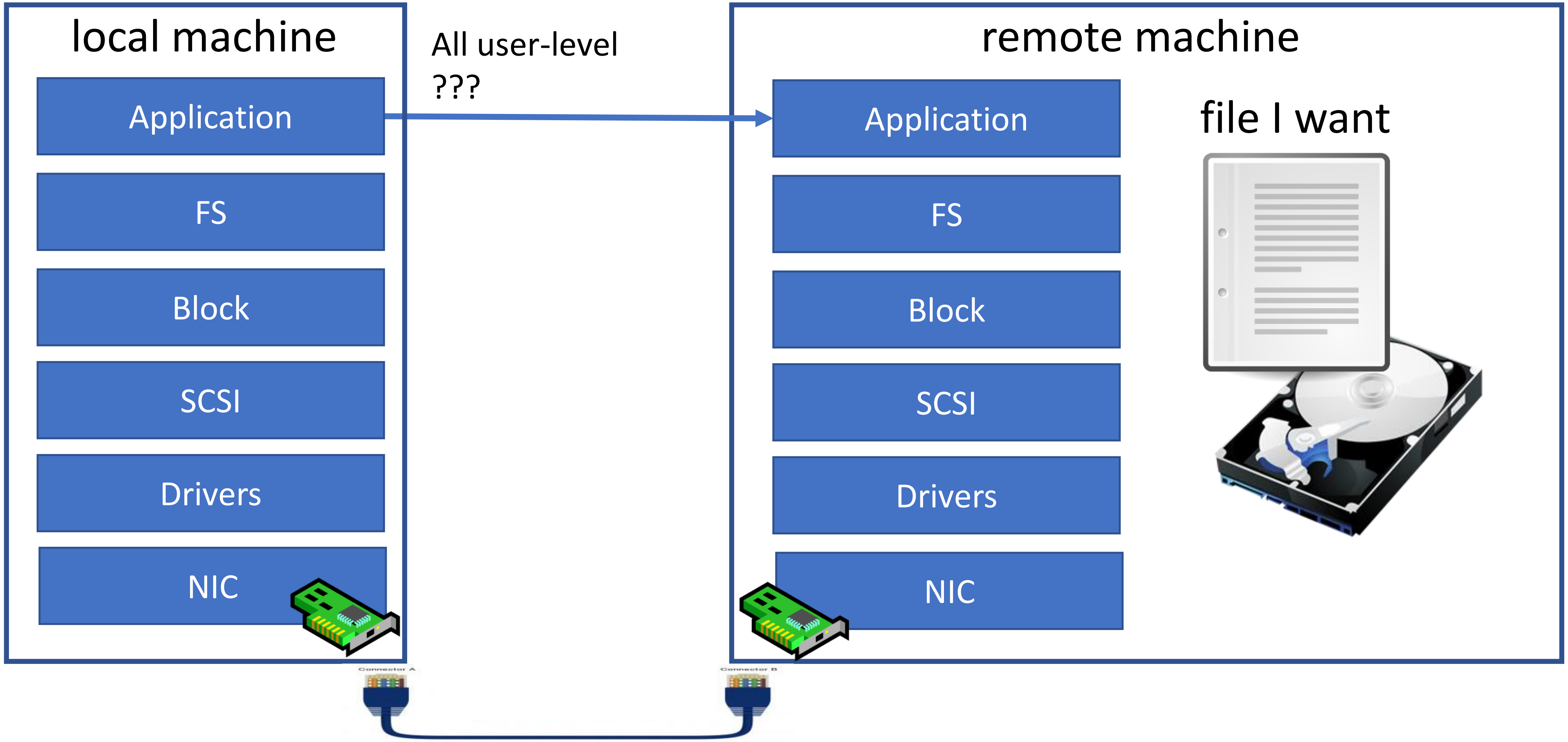
Networked storage design space



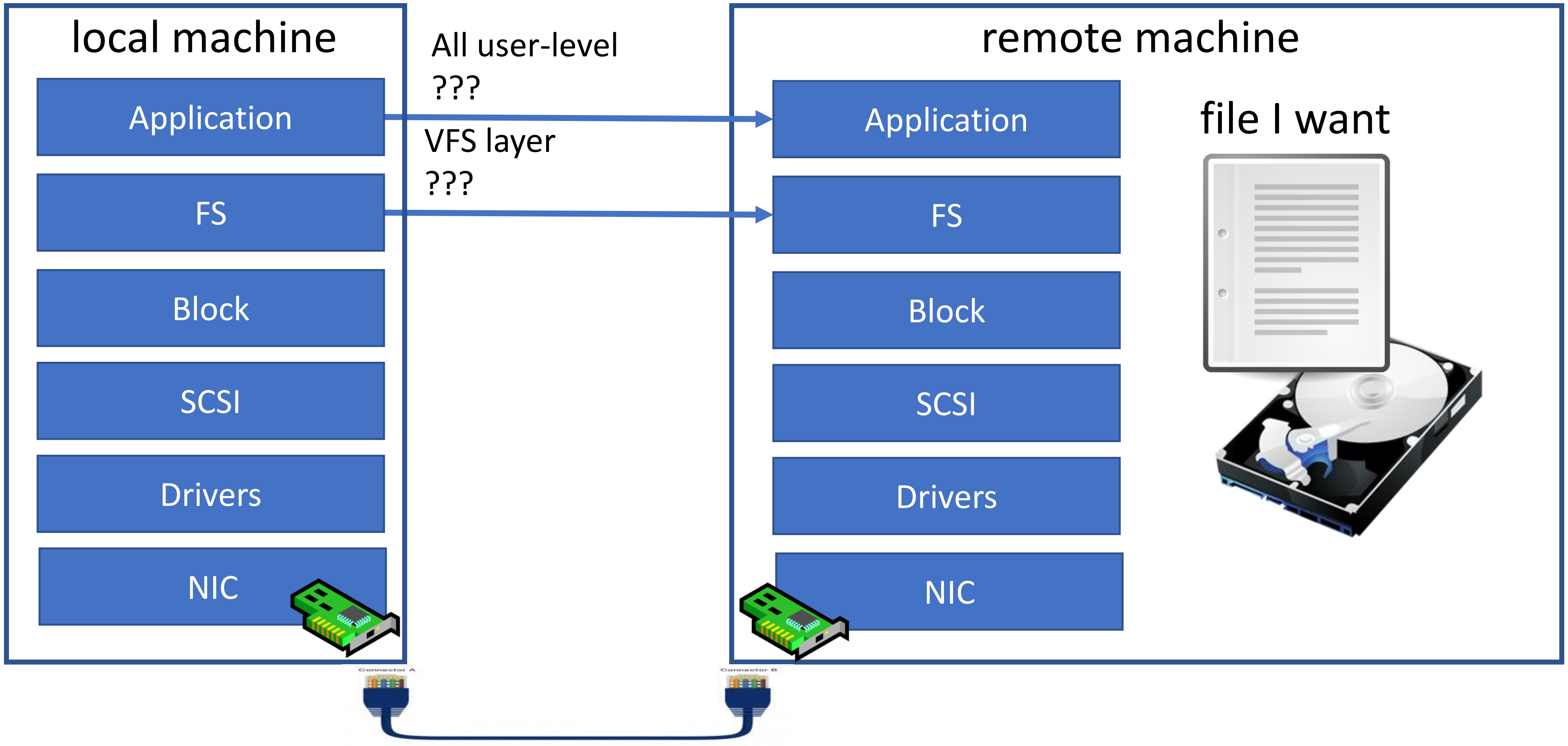
Networked storage design space



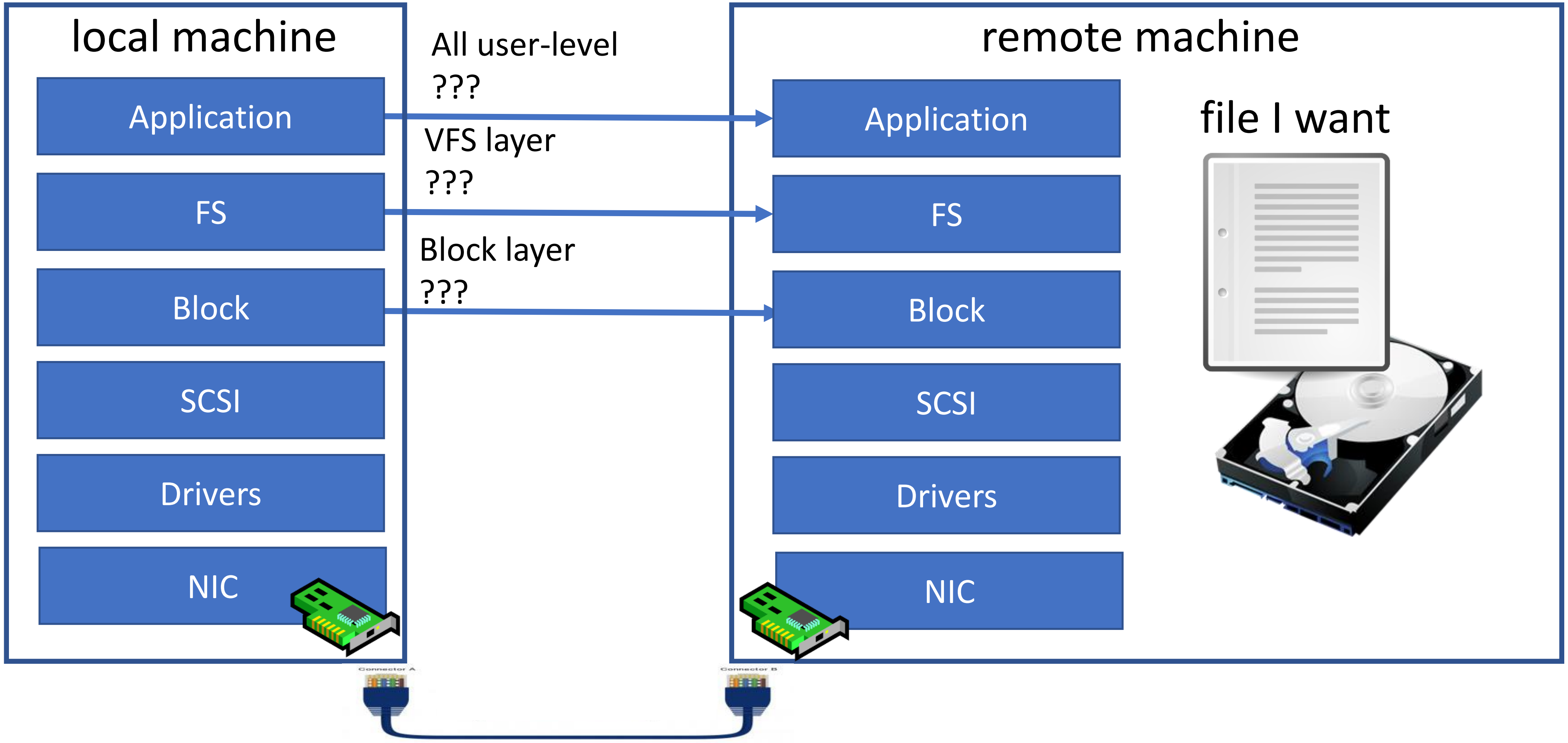
Networked storage design space



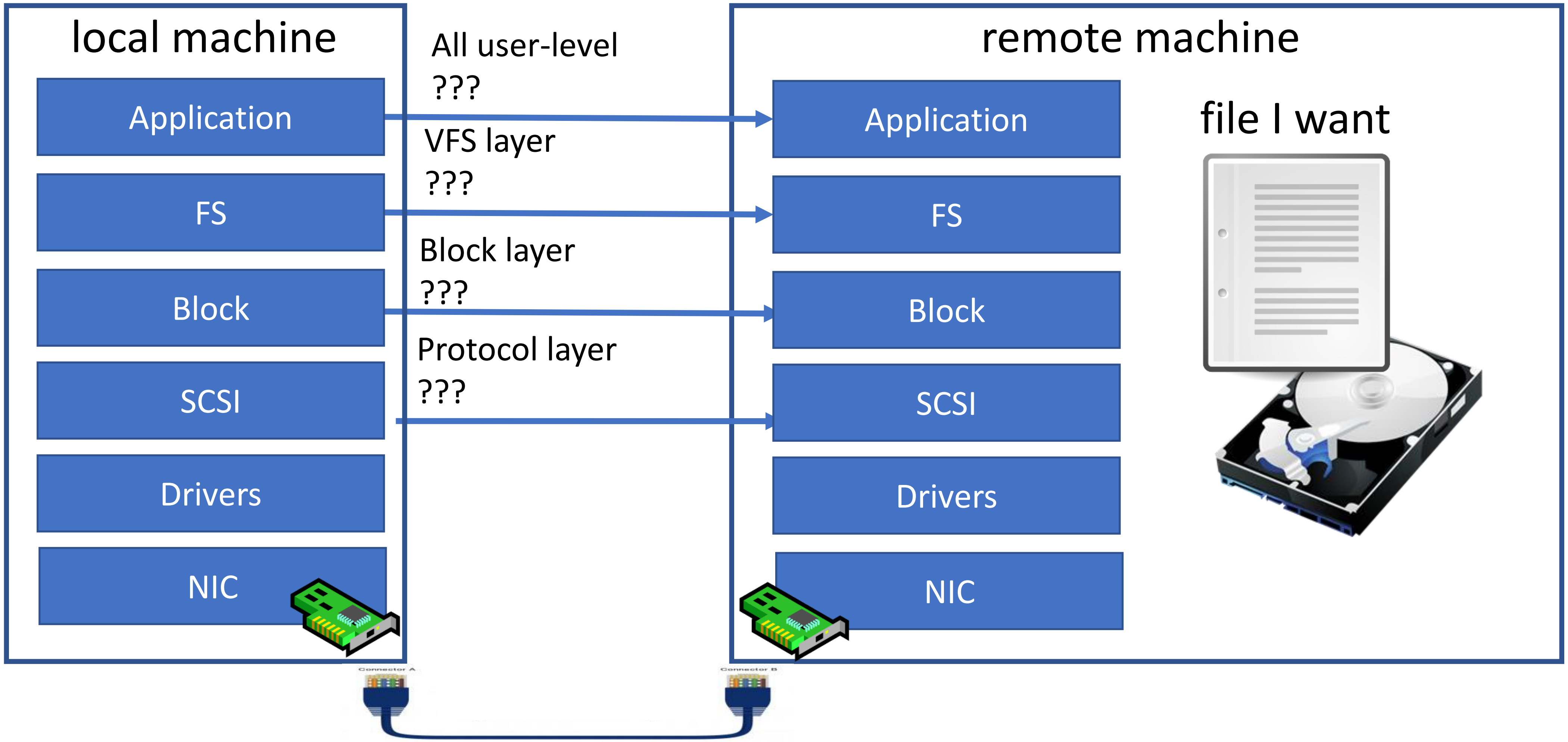
Networked storage design space



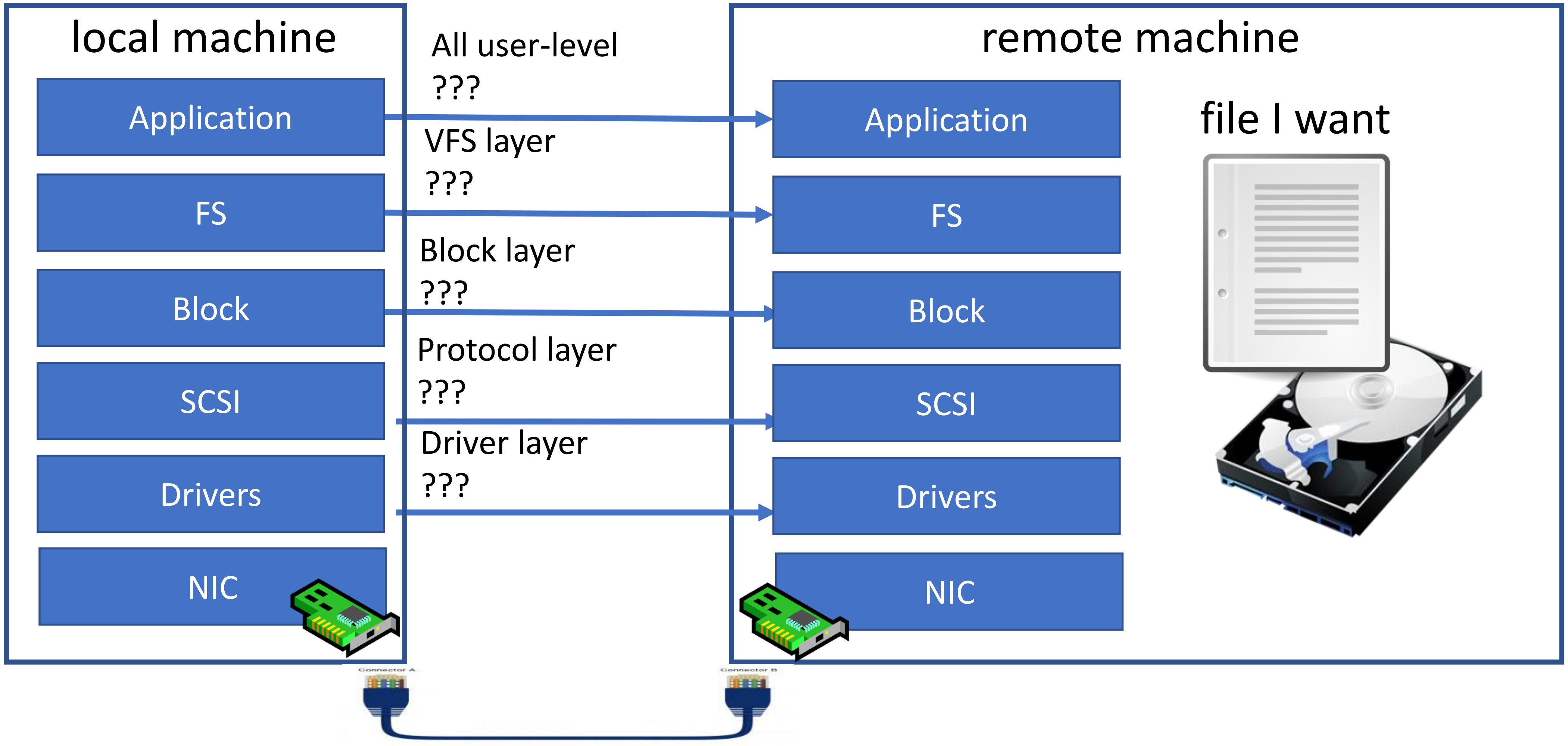
Networked storage design space



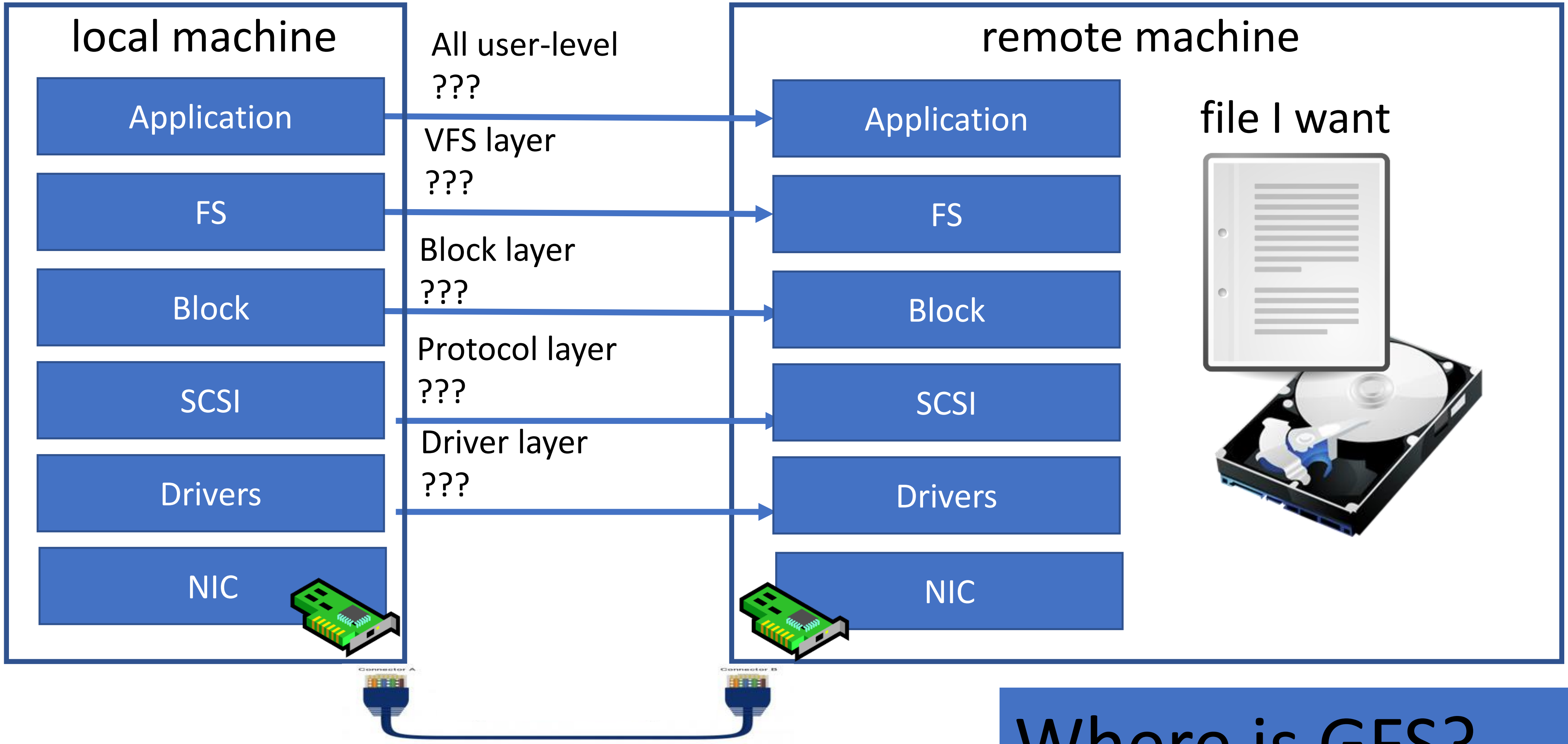
Networked storage design space



Networked storage design space



Networked storage design space



Where is GFS?

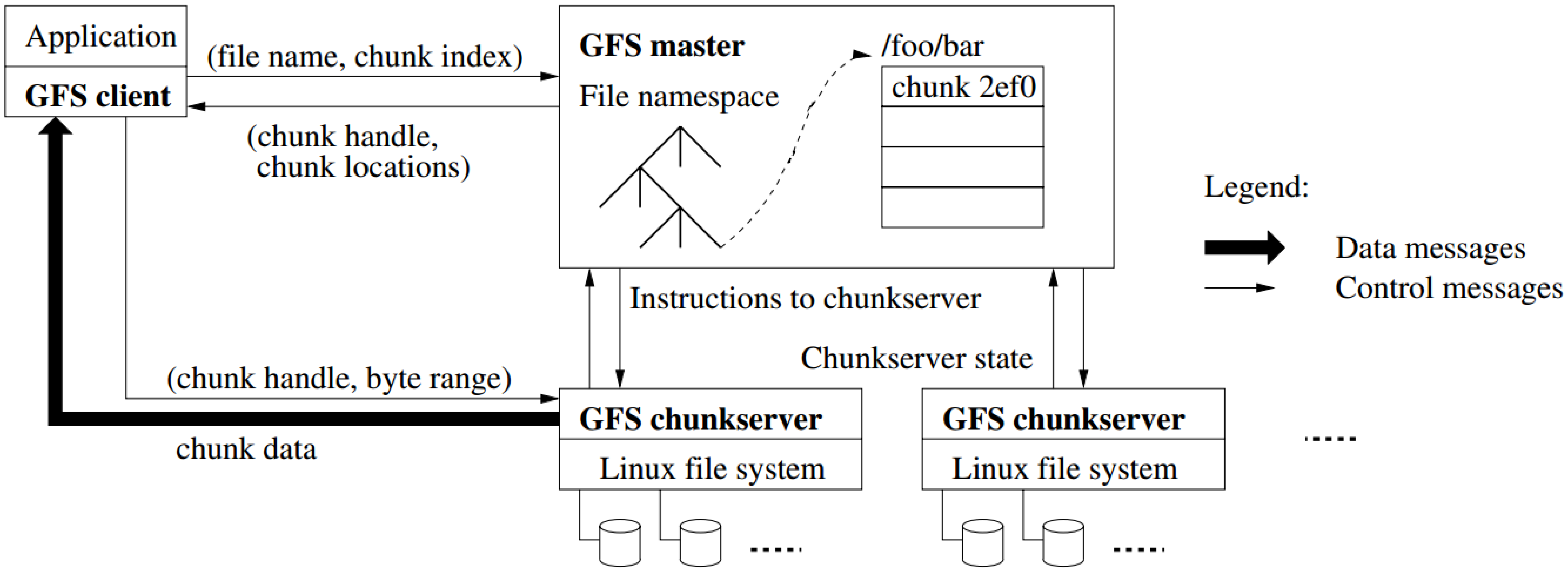
Goals/Design Determinants

- Optimized for:
  - Large files
  - Access bandwidth
  - Sequential reads
  - Appends (atomic!)
- Huge files (multi-GB) common
- Files are only appended and then read
- Snapshot support for files and directories

Why not use an existing file system?

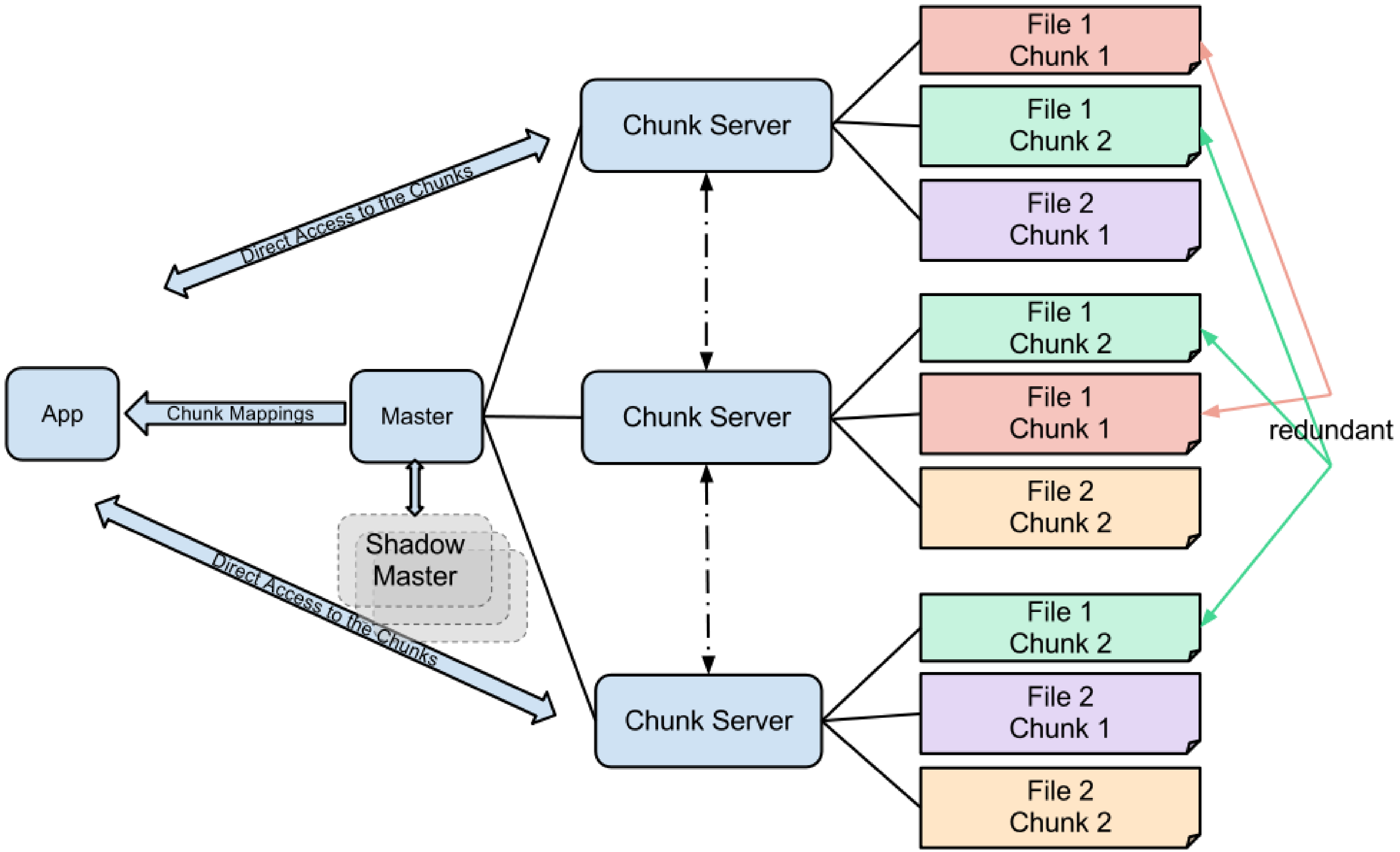
- Large files spread over multiple machines
- Different workload and design priorities
- GFS designed for Google apps/workloads
- Google apps designed for GFS

GFS architecture



- How many masters? Is that enough?
- Master/servers → classic metadata/data distinction
- POSIX semantics not necessary → VFS layer unnecessary

GFS Architecture Revisited



Chunks

- Fixed size (64MB) chunks:
  - easy translation from offset → chunk ID (done by client)
- Lazy chunk allocation justifies large size
  - What is largest source of fragmentation?
- Each chunk can be served by different chunkservers
- Identifier == 64-bit chunk handle
- Client chunk access
  - contact master for chunk server
  - Contact chunk server directly for data
  - No client data cache (why not?)
  - Clients do cache metadata (why?)

Master

Responsibilities

- Metadata storage
- Locking
- Chunkserver communication
- Chunk CRUD, replication, balance
  - Balance capacity/throughput
  - Replicas must cross racks
  - Re-replicate when low redundancy
  - Rebalance chunk locations for load

In memory:

1. File and chunk namespace
   - Changes logged to disk for persistence
   - RW locks for name space management
2. Mapping of files to chunks
3. Locations of chunk replicas.

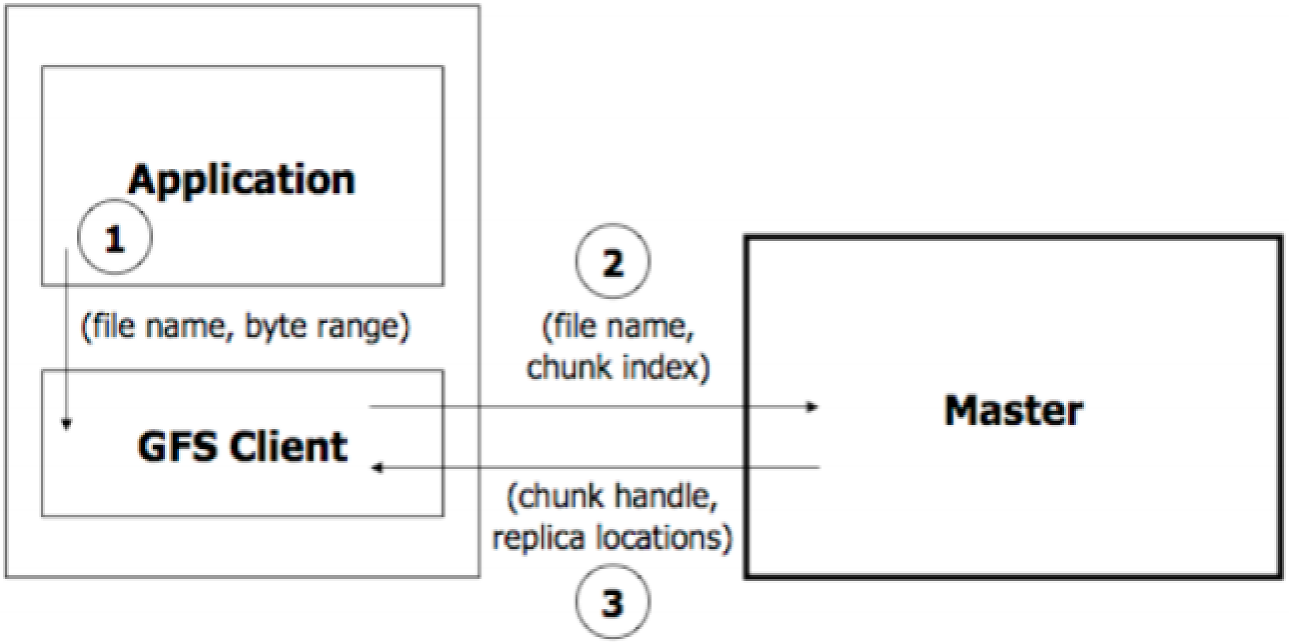
   Why isn't this persisted?

Log is vital: master op log serializes all namespace operations

Namespace mutations are synchronous

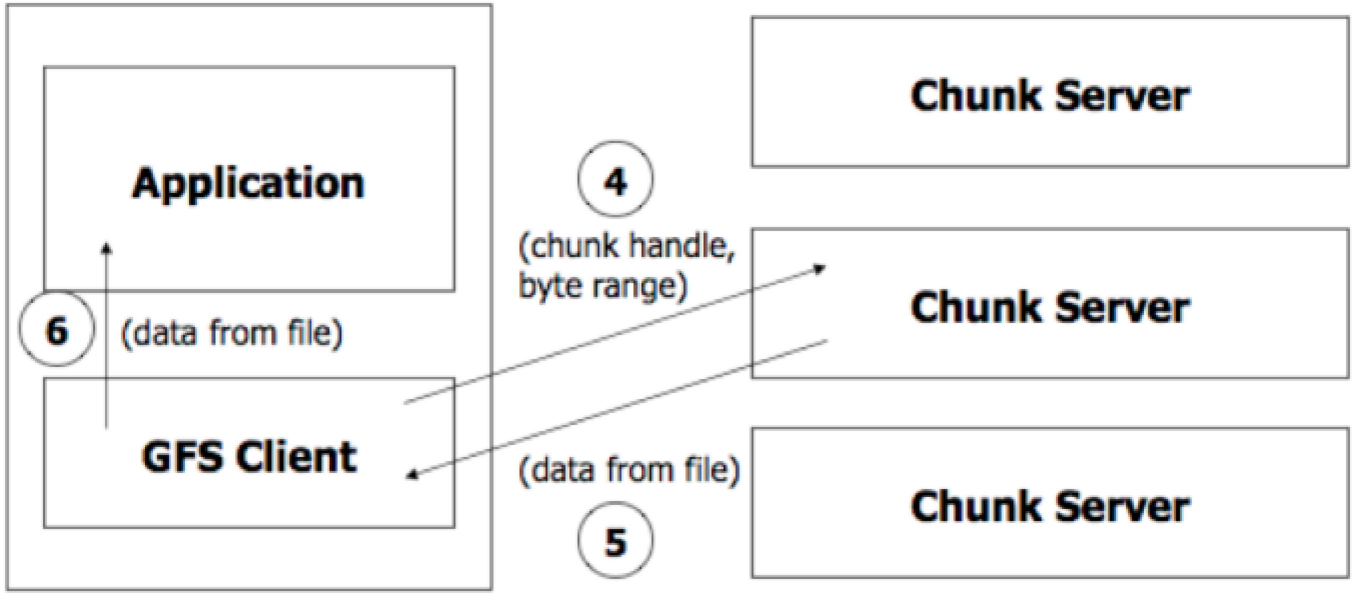
Read Algorithm

1. Application originates the read request
2. GFS client translates request and sends it to master
3. Master responds with chunk handle and replica locations

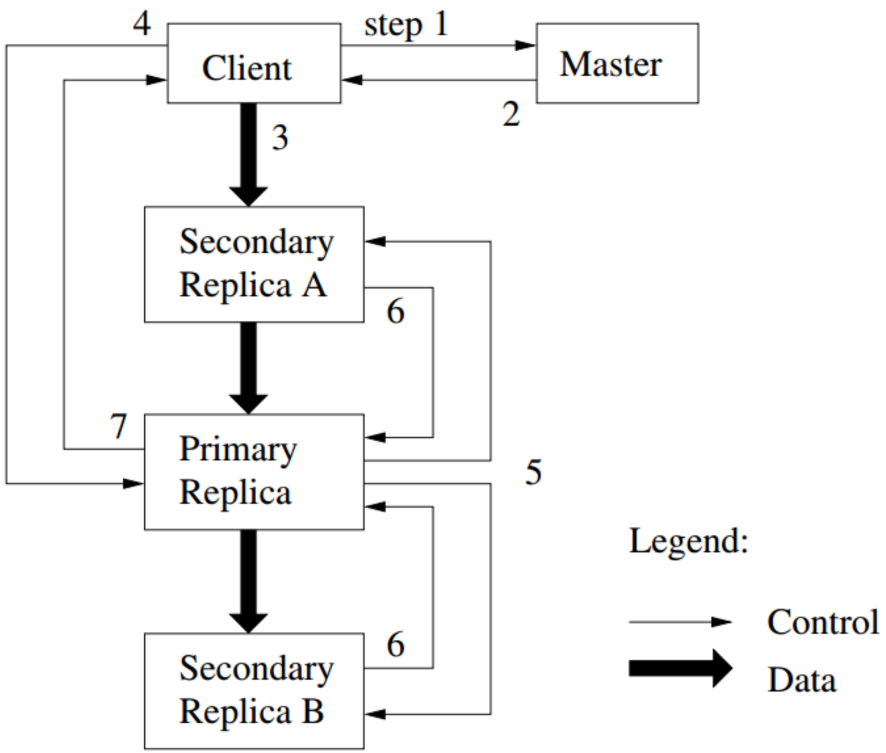


Read Algorithm

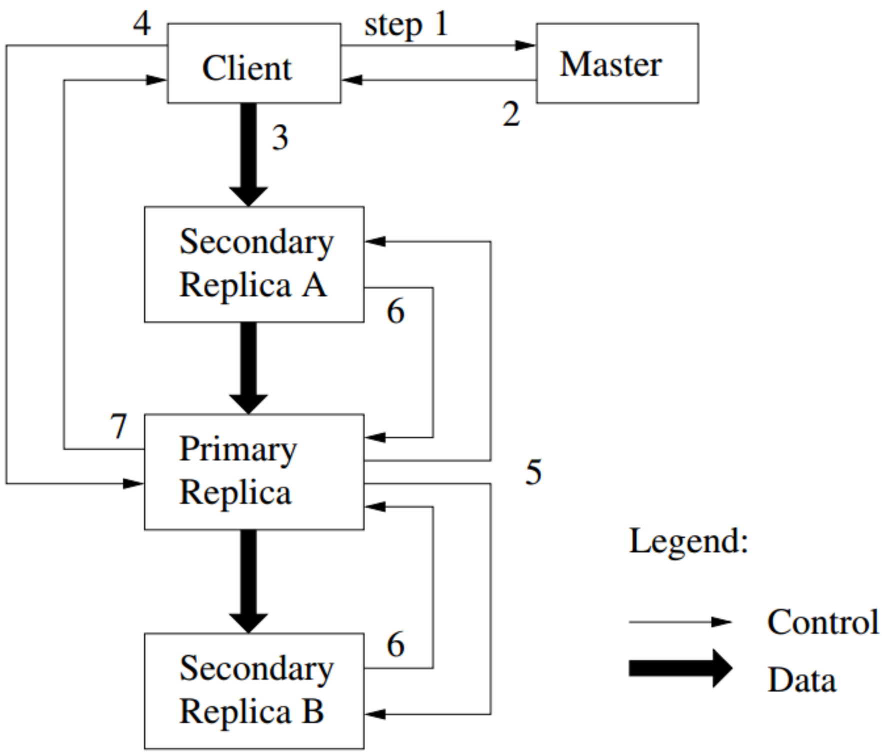
4. Client picks a location and sends the request
5. Chunkserver sends requested data to the client
6. Client forwards the data to the application



Write control and data flow

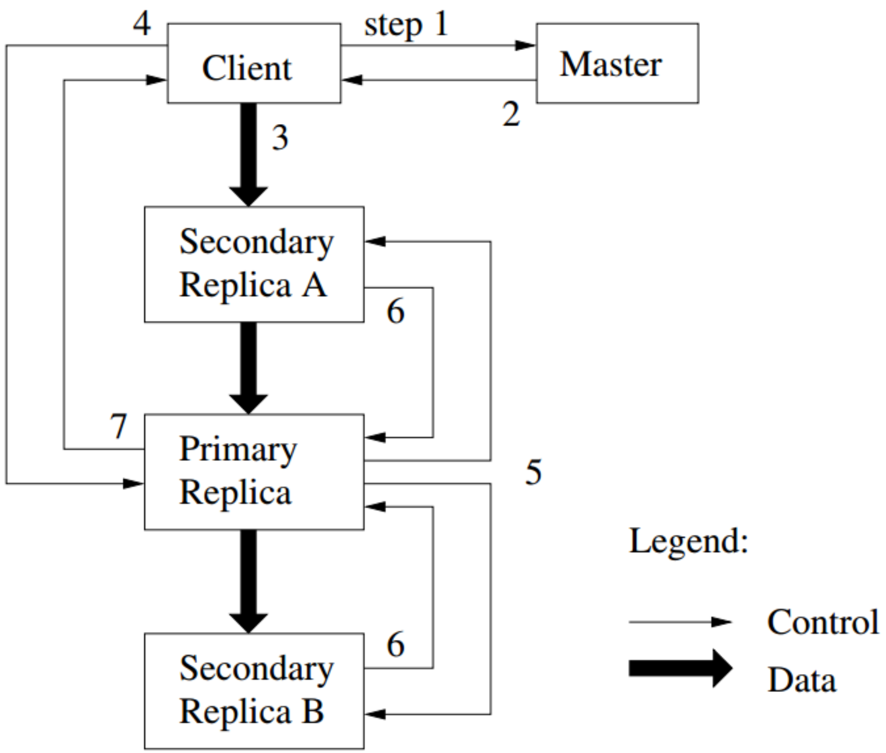


Write control and data flow



- What is #1 asking?
- What is in reply #2?
  - Why ok to cache it?
- bold #3? (one way?)
- #4? How associated with #3?
- Alternatives for #5?
- What if master fails?

Write control and data flow



- What is #1 asking?
- What is in reply #2?
  - Why ok to cache it?
- bold #3? (one way?)
- #4? How associated with #3?
- Alternatives for #5?
- What if master fails?

- *Data moved in any order*
- *Committed in order set by primary*
- *Durability means writes to multiple racks*
- *Lazy GC of deleted files*
  - *Deleted files renamed*
  - *space reclaimed after 3 days (why?)*
- *Shadow master for fast failover*

Atomic Append

- “At least once” semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Atomic Append

- “At least once” semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?

Atomic Append

- "At least once" semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?
3. No:

Atomic Append

- “At least once” semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?
3. No:
   A. Pad chunk
   B. Secondaries pad chunk
   C. Error → client → retry

Atomic Append

- “At least once” semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?
3. No:
   A. Pad chunk
   B. Secondaries pad chunk
   C. Error → client → retry
4. Yes:

Atomic Append

- “At least once” semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?
3. No:
   A. Pad chunk
   B. Secondaries pad chunk
   C. Error → client → retry
4. Yes:
   A. Append record
   B. Instruct secondaries: append record
   C. Collect secondary resps, → send to client

Atomic Append

- "At least once" semantics
  - GFS picks offset
  - Retry on failure
  - Good for concurrent writes
- Used heavily by Google apps
  - Files that server as MPSC queues
  - Merge multiple results to single file

Do secondaries always succeed in 4.b.?

Algorithm

1. Client push to all replicas
2. Primary: record fits current chunk?
3. No:
   A. Pad chunk
   B. Secondaries pad chunk
   C. Error → client → retry
4. Yes:
   A. Append record
   B. Instruct secondaries: append record
   C. Collect secondary resps, → send to client

When you’re Google…

When you’re Google…

- …You can define your own meaning of “consistent”

When you’re Google…

- …You can define your own meaning of “consistent”
- Region States after mutation

When you're Google…

- …You can define your own meaning of "consistent"
- Region States after mutation

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

When you're Google…

- …You can define your own meaning of "consistent"
- Region States after mutation

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

Consistent: all clients see same data
Defined: All clients see all of the latest write

- App-level checksums for integrity
- Applications tolerate duplicate chunks
- Writes ordered by lease for primary data node

When you’re Google…

- …You can define your own meaning of “consistent”
- Region States after mutation

How can this happen? What does it mean to be defined but inconsistent? How can it happen?

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

Consistent: all clients see same data

Defined: All clients see all of the latest write

- App-level checksums for integrity
- Applications tolerate duplicate chunks
- Writes ordered by lease for primary data node

When you're Google…

- …You can define your own meaning of "consistent"
- Region States after mutation

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

Consistent: all clients see same data
Defined: All clients see all of the latest write

- App-level checksums for integrity
- Applications tolerate duplicate chunks
- Writes ordered by lease for primary data node

When you're Google…

- …You can define your own meaning of "consistent"
- Region States after mutation

<table>
<tr><th></th><th>Write</th><th>Record Append</th></tr>
<tr><td>Serial success</td><td>defined</td><td rowspan="2">defined interspersed with inconsistent</td></tr>
<tr><td>Concurrent successes</td><td>consistent but undefined</td></tr>
<tr><td>Failure</td><td colspan="2">inconsistent</td></tr>
</table>

Consistent: all clients see same data
Defined: All clients see all of the latest write

- App-level checksums for integrity
- Applications tolerate duplicate chunks
- Writes ordered by lease for primary data node

Given the write protocol, how does concurrency arise?

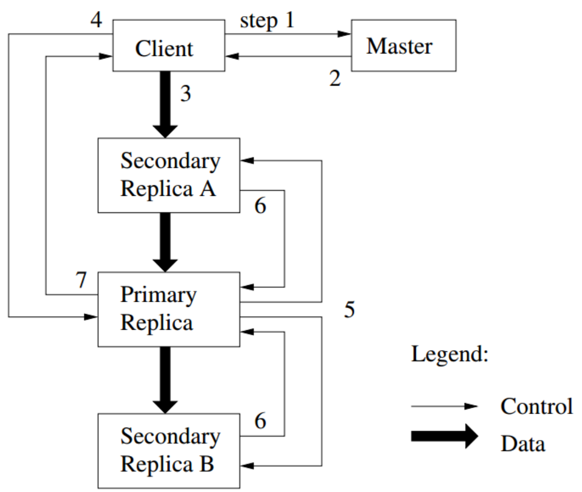
When you're Google…

- …You can define your own meaning of "consistent"
- Region States after mutation

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

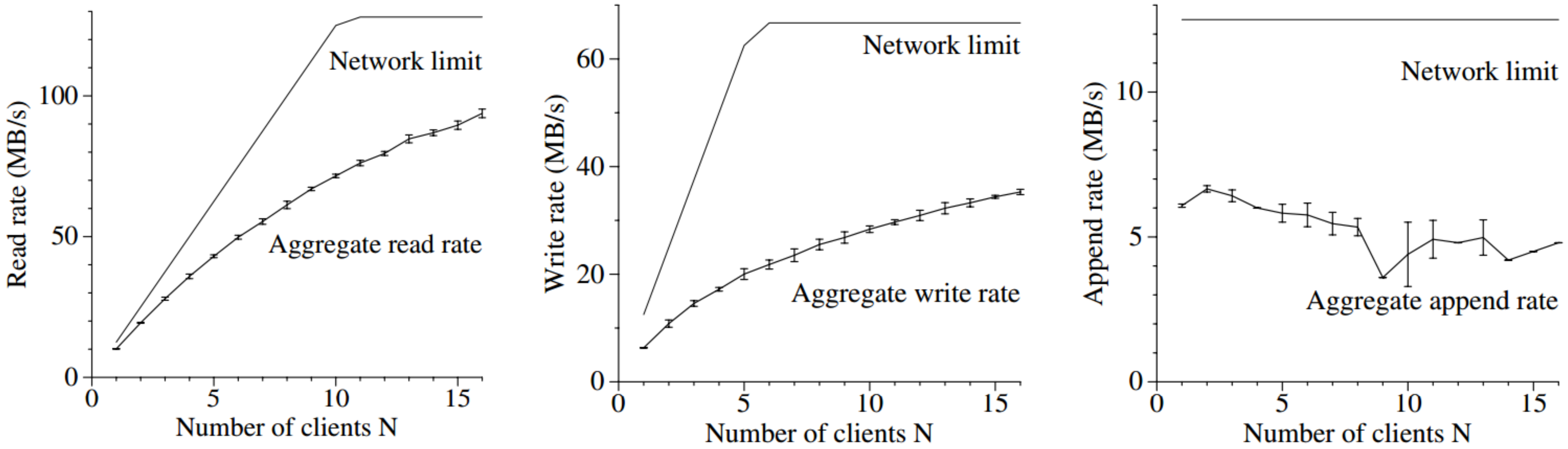
Consistent: all clients see same data
Defined: All clients see all of the latest write

- App-level checksums for integrity
- Applications tolerate duplicate chunks
- Writes ordered by lease for primary data node



Given the write protocol, how does concurrency arise?

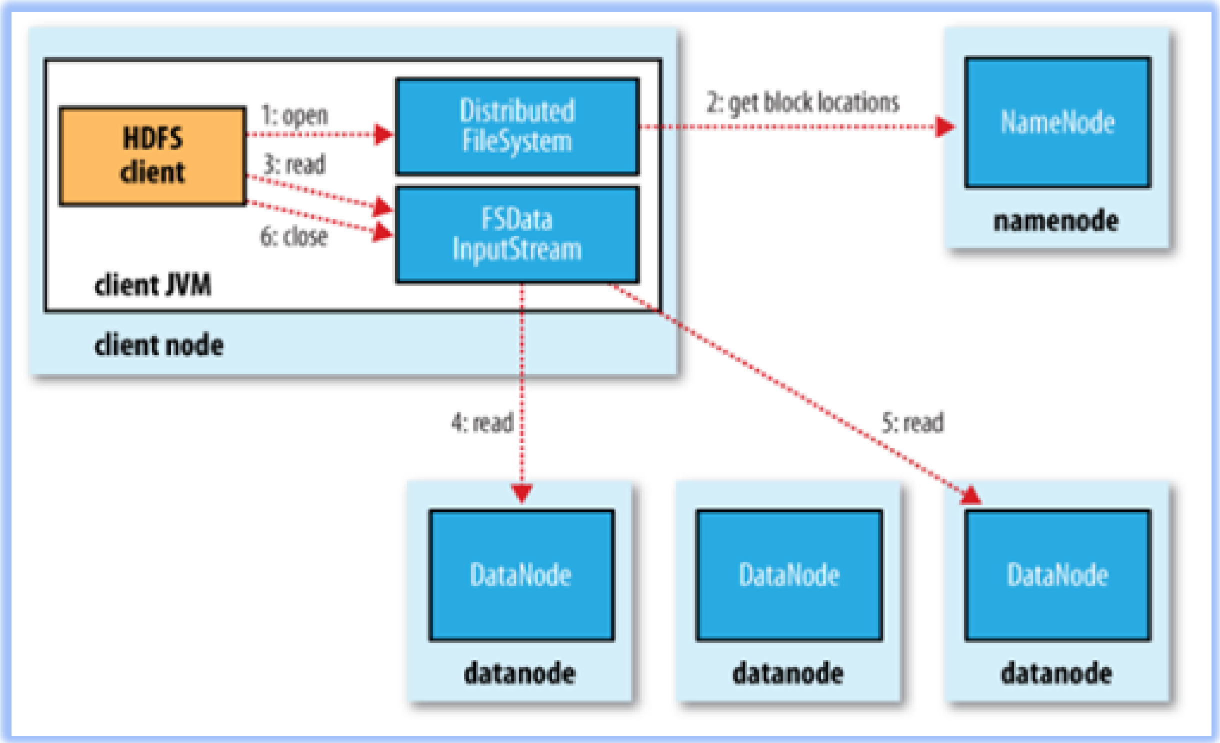
Aggregate Throughput



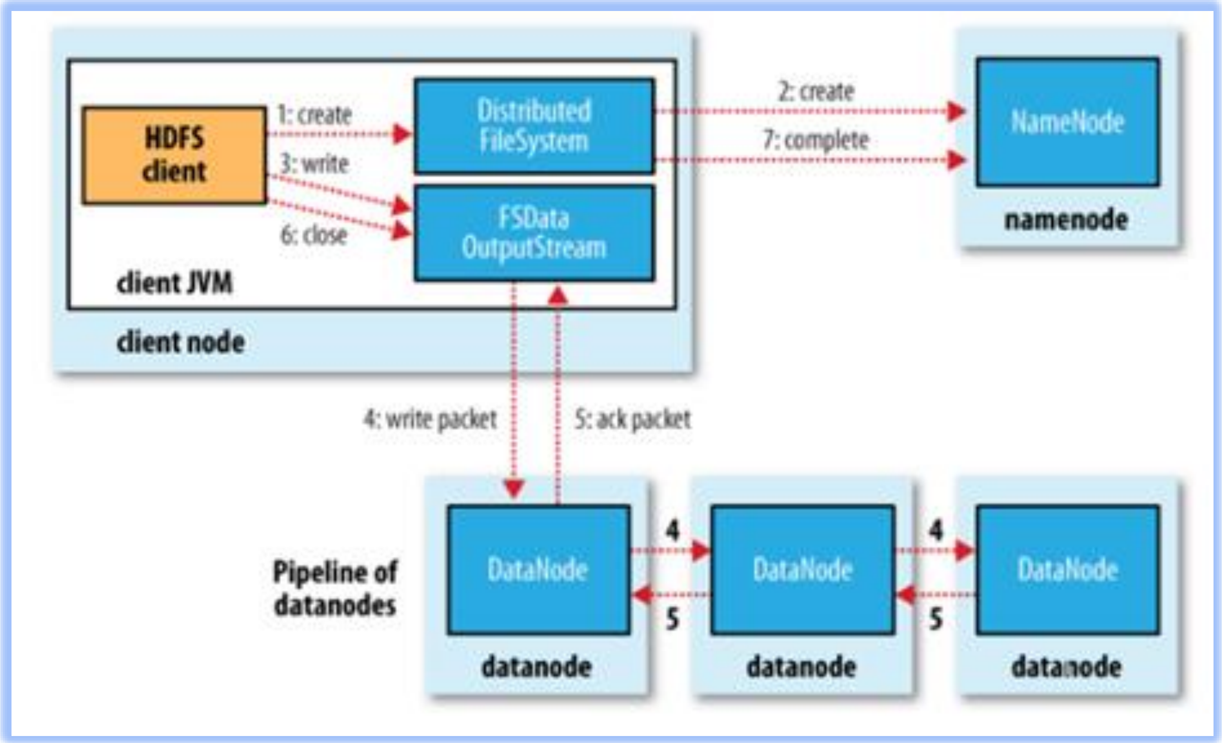
(a) Reads (b) Writes (c) Record appends

HDFS comparison (for hoots)

Read



Write



What differences do you see from GFS?

How might GFS be better/worse?

GFS Evolution

- 64 MB chunks make it hard to support small files (gmail)
  - 1MB new design target
- Master memory limits number of files in a GFS FS
- Trade latency for bandwidth
  - poor choice for user-visible apps (gmail)
- File content inconsistencies are a pain point
  - What causes this again?
- Support for multiple masters is desirable but difficult
- Erasure coding and/or Reed Solomon: 3x overhead → 2.1x

Distributed FS dimensions

Dimension	Examples	NFS	GFS
Architecture	Central/Distributed		
Naming	Index/DB/Log/…		
API	FUSE/CLI/POSIX		
Fault-detection	Fully-connected/P2P/manual		
System availability	Failover/…		
Data availability	Replication/RAID/…		
Placement	Auto/manual		
Replication	Sync/Async		
Consistency	Lock/WORM/…		

Thoughts on the Master

- Single master → simple → short time to deploy
- Small metadata → fits on one machine (IN RAM)
- Metadata: file id, chunks
- Fast scans (gc, recovery)
- 100s TB → 10s PB → orders magnitude metadata increase
- “Open” talks to master
- MR jobs thousands of jobs come alive simultaneously → all want to open something…
- By 2009, master per cell, multi-masters on cell of chunkservers, application-level partitioning.
- GFS: team of 3 people under 1 year

Exploring the consistency tradeoffs

- Write-to-read semantics too expensive
  - Give up caching, require server-side state, or …
- Close-to-open “session” semantics
  - Ensure an ordering, but only between application `close` and `open`, not all `writes` and `reads`.
  - If B opens after A closes, will see A’s writes
  - But if two clients open at same time? No guarantees
    - And what gets written? “Last writer wins”