DAS 839 NoSQL Systems

Distributed File System Principles

Vinu Venugopal
ScaDS.ai Lab, IIIT Bangalore

Distributed Approach

- The primary driver of interest in NoSQL has been its ability to run databases on a large cluster.

- Difficult and expensive to scale up—buy a bigger server to run the database on

- A more appealing option is to scale out—run the database on a cluster of servers.

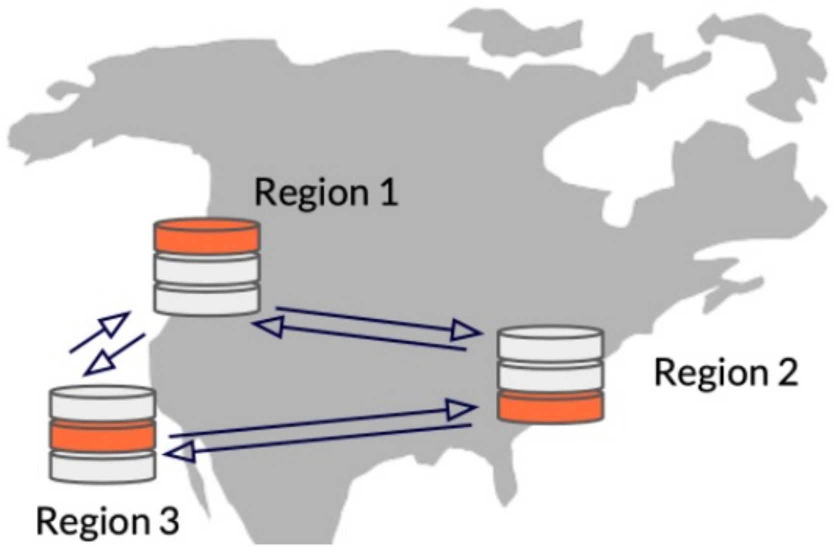
- <u>Running over a cluster introduces complexity—so it's not something to do unless the benefits are compelling.</u>

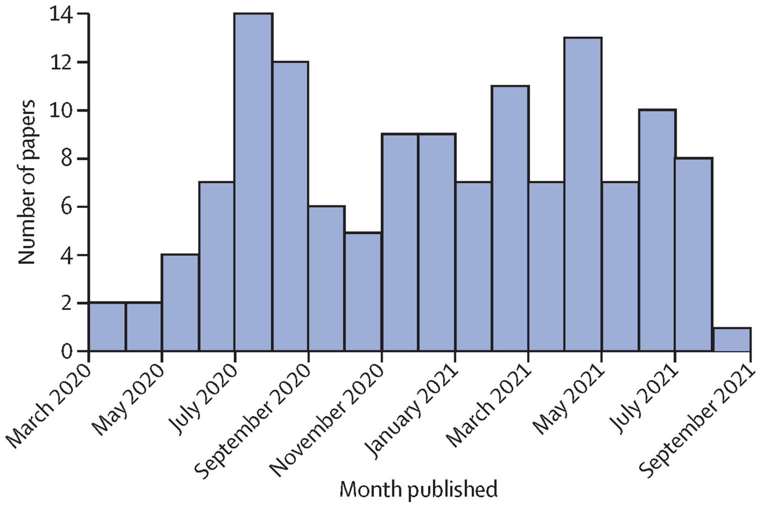
Distributed Approach

- Even though it is a cheaper option, distributed approaches are often challenging:
  - Data is not residing on a single machine
  - File system is distributed
  - How to divide what (data/process) is a concern.
  - Who will take care of sharding and when.
  - Who will take care of correctness of the the output?
  - How can you perform computation over the distributed data?

Distributed Approach

- Create a histogram depicting the frequency of COVID-19 cases across various months within a year.
- This data is sourced from RTPCR tests conducted at numerous locations worldwide, stored locally but accessible through internet connection.

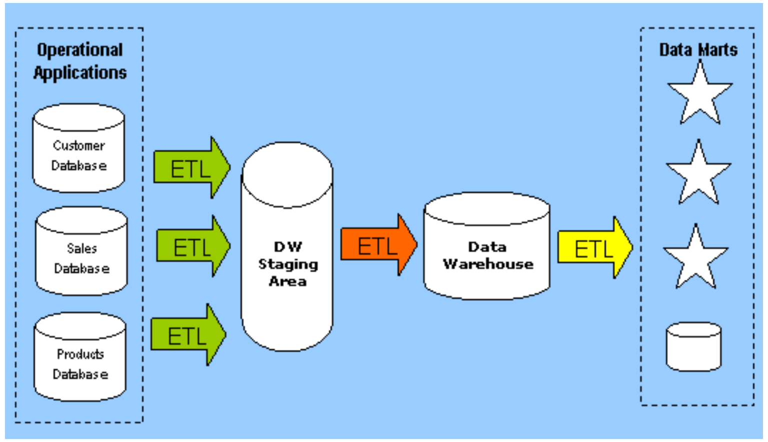




Distributed Approach

- Create a histogram depicting the frequency of COVID-19 cases across various months within a year.
- This data is sourced from RTPCR tests conducted at numerous locations worldwide, stored locally but accessible through internet connection.

Solution: In a data warehousing scenario, it's common to conduct ETL processes to consolidate data into a centralized location. Nevertheless, this approach does not involve distributed solutions.



Distributed Approach

- Create a histogram depicting the frequency of COVID-19 cases across various months within a year.
- This data is sourced from RTPCR tests conducted at numerous locations worldwide, stored locally but accessible through internet connection.

Solution: In a data warehousing scenario, it's common to conduct ETL processes to consolidate data into a centralized location. Nevertheless, this approach does not involve distributed solutions.

Distributed solution: We must group the data by year and month on each individual machine locally. This entails conducting partial aggregation and grouping operations on all machines, followed by transmitting the results across the network to a central machine for global aggregation and grouping operations.

Distributed Approach

- How we shard the data is very important. How to perform computation on the partitioned data is also important.

- From a historic point of view, there are many ad-hoc architectures for sorting and grouping a file/data in a distibuted setting.

- Also, there are generic computation paradigms like “MapReduce” exists.

- First thing first, DFS!

Physical and Logical Views

- Not new! Concepts in distributed file systems
- Where the basic idea is to **separate the logical from the physical storage**.
- The **virtual file system** (layer) enables clients to access all files as if they were stored locally.
- Example:
- /usr/files/1.txt → (192.168.0.0, /dev/sda1/local/1.txt)
- Some historically important DFS approaches:
  - NFS, AFS, CODA, GFS, HDFS

Terminologies: Distributed File System

- **File Service**
  - A specification of what the file system offers to its clients
- **File server**
  - An implementation of a file service
  - Typically, run on one or more systems
- **File access types**
  - Explicit access
  - Transparent access
- **File access models**
  - Upload/download model
  - Remote access model

DFS: “Managing files in a distibuted setting”

File Access Types

- **Explicit access**

- **Transparent access**

File Access Types

- **Explicit access**
  - Client initiates a connection and **accesses remote resources by their host name and file location**.
  - Typical examples: ftp, ssh, telnet
  - Early days of UNIX, no need for anything special.
  - Horribly inflexible and slow, need something better.

- **Transparent access**

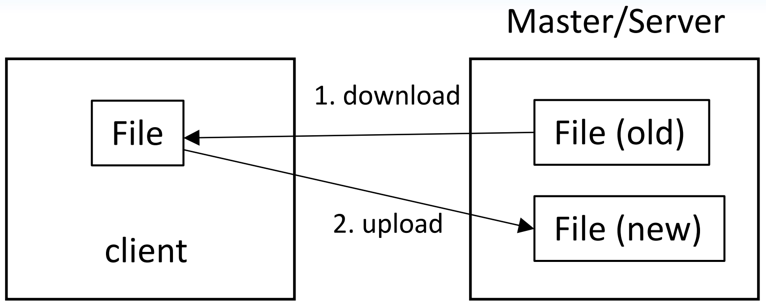
File Access Types

- **Explicit access**
  - Client initiates a connection and **accesses remote resources by their host name and file location**.
  - Typical examples: ftp, ssh, telnet
  - Early days of UNIX, no need for anything special.
  - Horribly inflexible and slow, need something better.

- **Transparent access**
  - Client **accesses** remote resources **just as local ones**.
  - **Feature** of a **true** distributed file system.

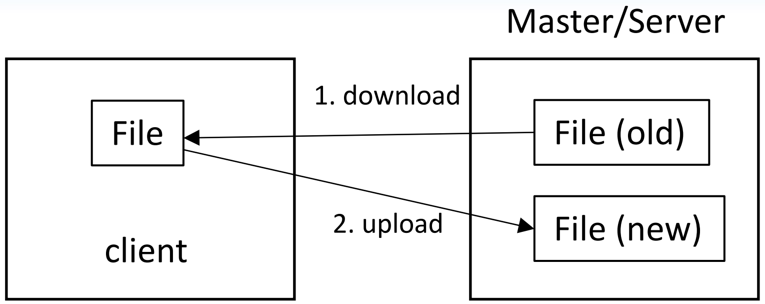
A client does not really need to know the physical address of the server. There would be a middle layer that would take care of the finding the address etc..

File Access Models



- **Upload/download model**
  - **Download**: copy file from server to client.
  - **Upload**: copy file from client to server.

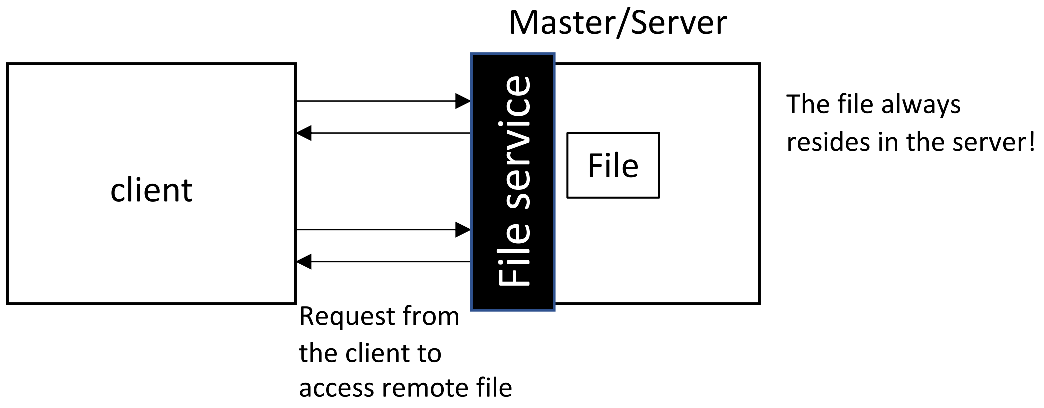
File Access Models



- **Upload/download model**
  - **Download**: copy file from server to client.
  - **Upload**: copy file from client to server.
- **Issues**
  - What if the client does not want to **modify the whole file**?
  - What if the client does not have **enough space**?
  - What if **other clients** want to **access the same file**? (No semantics/polices for handling multiple access to same file.)

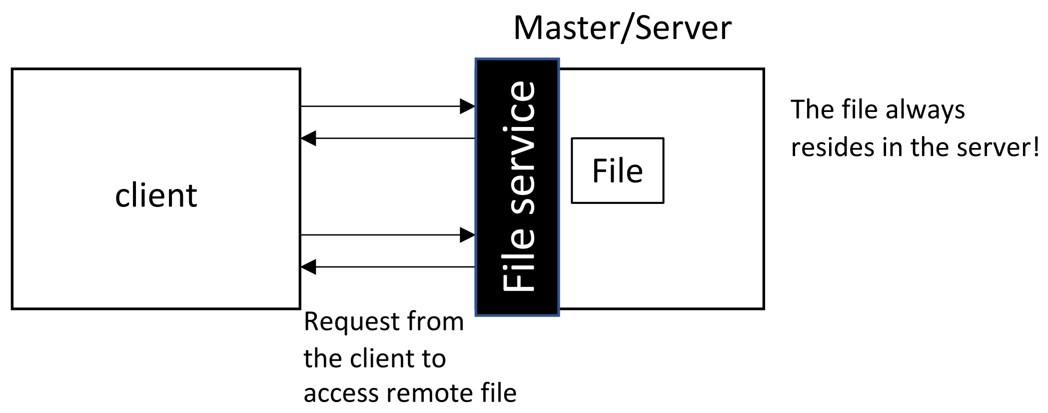
File Access Models

- **Remote-access model**
  - **File service** provides **programming interface** (API) to create(), read(), update(), delete() (CRUD) files or bytes.
  - Same API one would have in a centralized file system.
- **Advantages**
  - Client gets only what’s needed – more flexibility in accessing a file.
  - More logical way to give access to a file – e.g., multiple users can update different parts of the same file.
  - Server can manage coherent view of file system
  - It would be easy to make consistency policies for accessing a file.



File Access Models

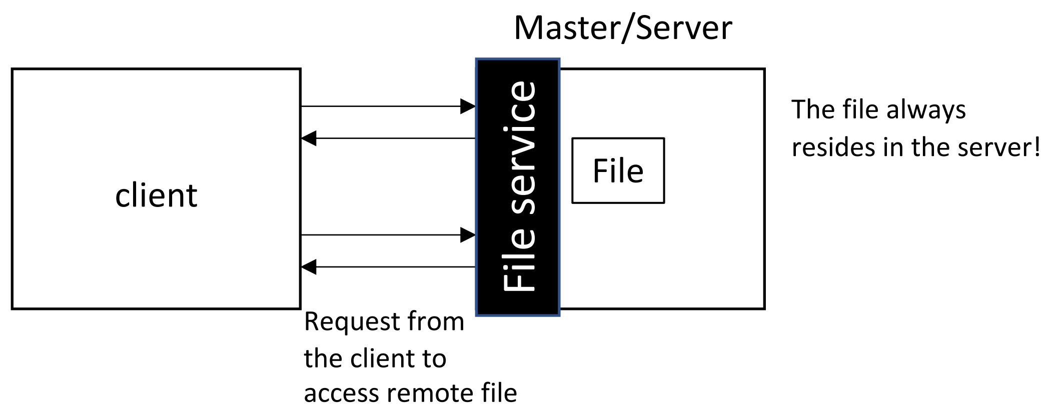
- **Remote-access model**
  - **File service** provides **programming interface** (API) to
    create(), read(), update(), delete() (CRUD) files or bytes.
  - Same API one would have in a centralized file system.
- **Advantages**
  - Client gets only what's needed
  - Server can manage coherent view of file system, …
- **Issues**
  - Possible server and network congestion
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly – e.g., if a client want to do multiple operations on a single file (say, spell checking app)
  - State in server? (Upload/Download model is stateless!)



File Access Models

- **Remote-access model**
  - **File service** provides **programming interface** (API) to

    create(), read(), update(), delete() (CRUD) files or bytes.
  - Same API one would have in a centralized file system.
- **Advantages**
  - Client gets only what's needed
  - Server can manage coherent view of file system
- **Issues**
  - Possible server and network congestion
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly
  - State in server?



What could be a potential solution to this problem?

Distribution Models

- there are two paths to data distribution: **replication** and **sharding**.
- Replication takes the same data and copies it over multiple nodes.
- Sharding puts different data on different nodes.
- You can use either or both of them.
- Replication comes into two forms: master-slave and peer-to-peer.

a. Single Server

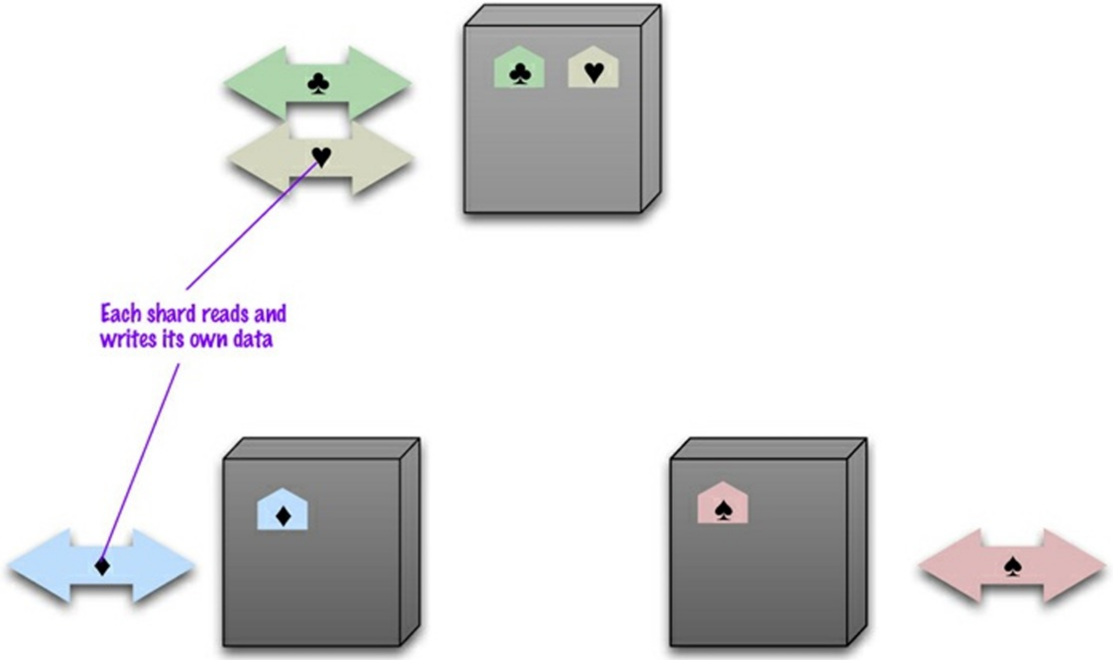
- No distribution at all – simplest approach
- eliminates all the complexities that the other options introduce; it’s easy for operations people to manage and easy for application developers to reason about.

- It make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application
  - For instance, Graph databases

b. Sharding

- Often, a busy data store is busy because different people are accessing different parts of the dataset.
- support horizontal scalability by putting different parts of the data onto different servers—a technique that's called **sharding**

Sharding puts different data on separate nodes, each of which does its own reads and writes.

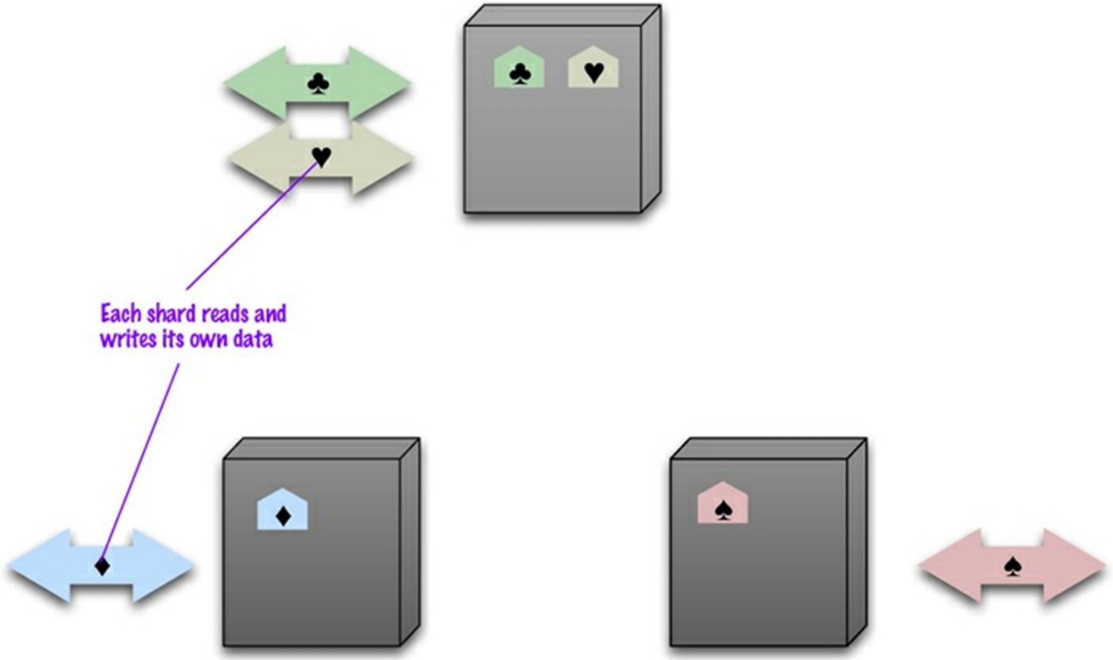


b. Sharding

- In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server.

- The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

Sharding puts different data on separate nodes, each of which does its own reads and writes.

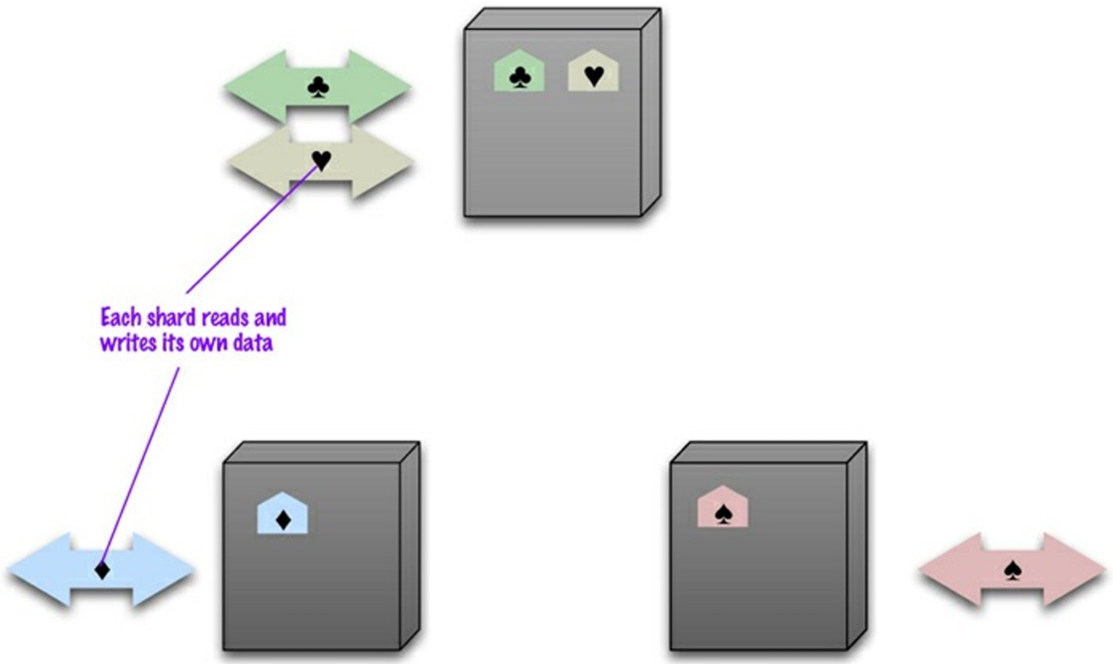


b. Sharding

- What about queries with aggregation?

- data that's accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access.

Sharding puts different data on separate nodes, each of which does its own reads and writes.



b. Sharding

- The whole point of **aggregates** is that we design them to combine data that's commonly accessed together—so aggregates leap out as an obvious unit of distribution.

  If you know that most accesses of certain aggregates are based on a physical location, you can place the data close to where it's being accessed. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center.

- **Keep the load even.** Should try to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load.

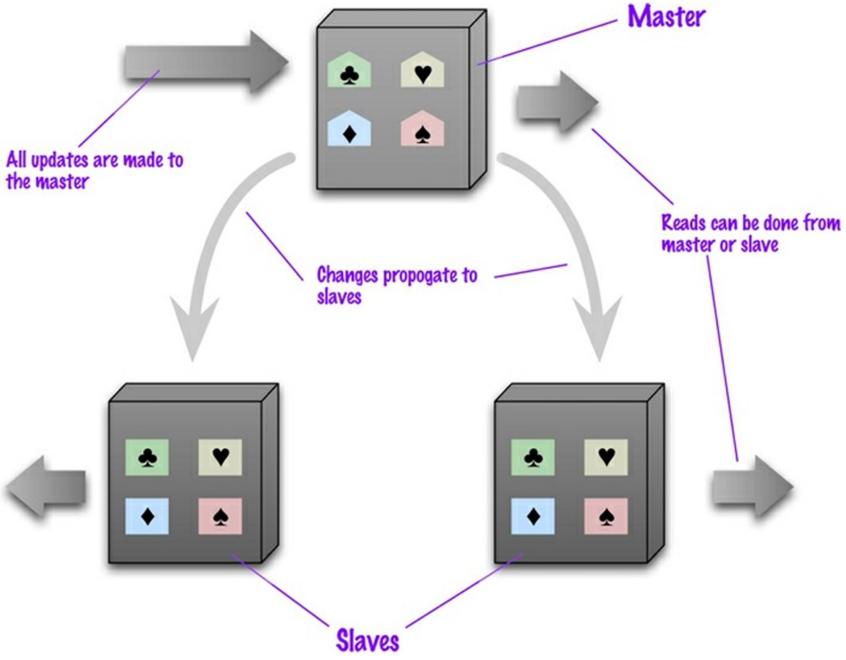
- Sharding can **improve both read and write performance.**
- Sharding **does little to improve resilience** when used alone – a node failure makes that shard's data unavailable

b. Sharding

- Sharding is made much easier with aggregates, it’s still **not a step to be taken lightly**
- Sharding well before you need to—when you have enough headroom to carry out the sharding, and not just turned it on in production.

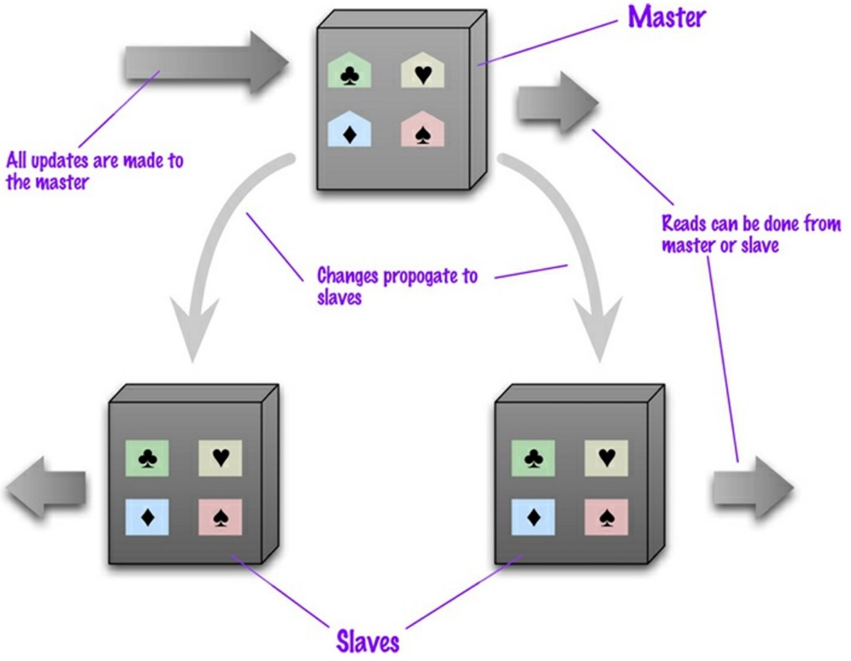
c. Replication – master-slave archi.

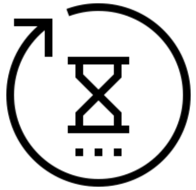
- Replicate data across multiple nodes
- One node acts a master – will be the authoritative source for the data.
- Replication-process synchronizes the slaves with the master.
- Scales well when you have a read-intensive dataset.
- Read-resilience: If a master fails, the slaves can still handle read requests.
- The failure of the master does eliminate the ability to handle writes until the master is restored or having a slave appointed as a master.
- Replication comes with benefits, but <u>can cause inconsistencies</u>



c. Replication – master-slave archi.

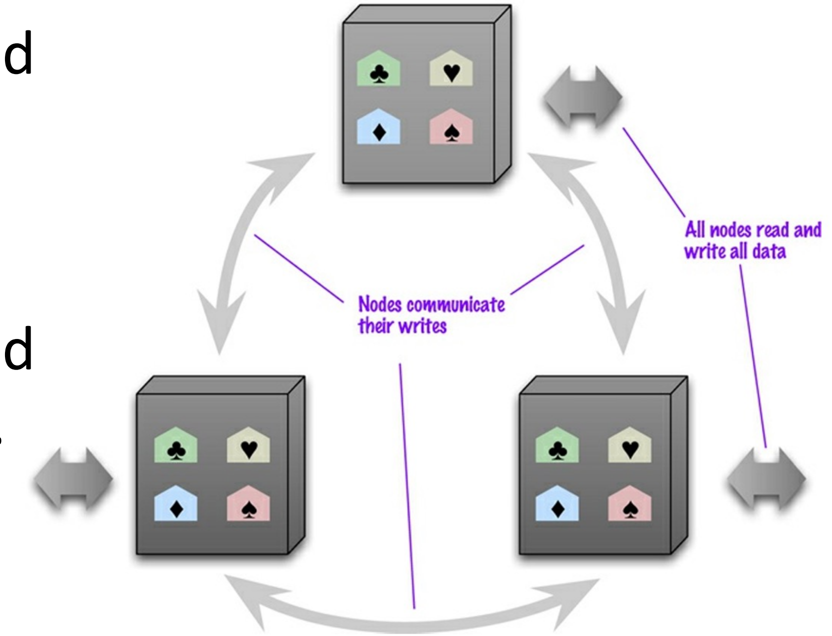
- Replicate data across multiple nodes
- One node acts a master – will be the authoritative source for the data.
- Replication-process synchronizes the slaves with the master.
- Scales well when you have a read-intensive dataset.
- Read-resilience: If a master fails, the slaves can still handle read requests.
- The failure of the master does eliminate the ability to handle writes until the master is restored or having a slave appointed as a master.
- Replication comes with benefits, but <u>can cause inconsistencies</u>





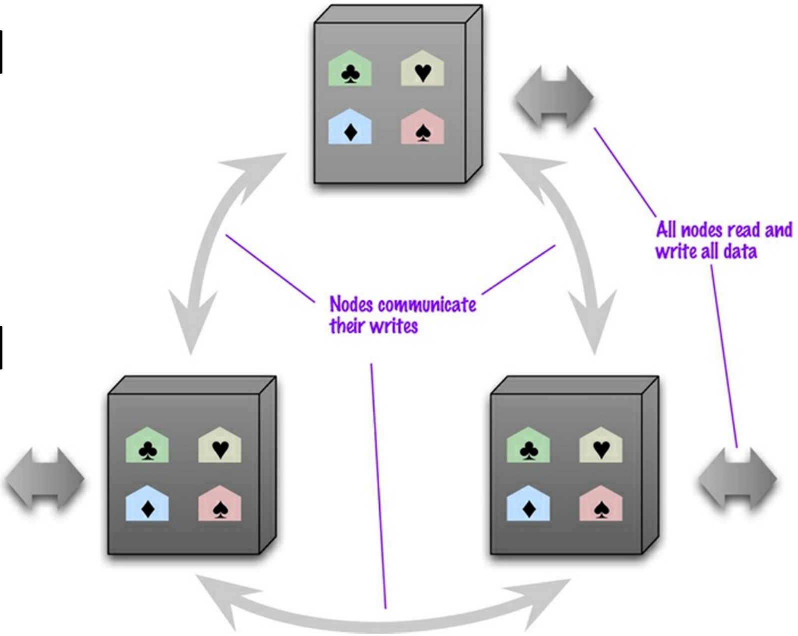
c. Replication – Peer-to-Peer archi.

- In a master slave replication, master node is a bottleneck and a single point of failure.
- Peer-to-peer has not master node.
- All replicas have equal weight, they can all accept writes and loss of any of them doesn't prevent access to the data store.



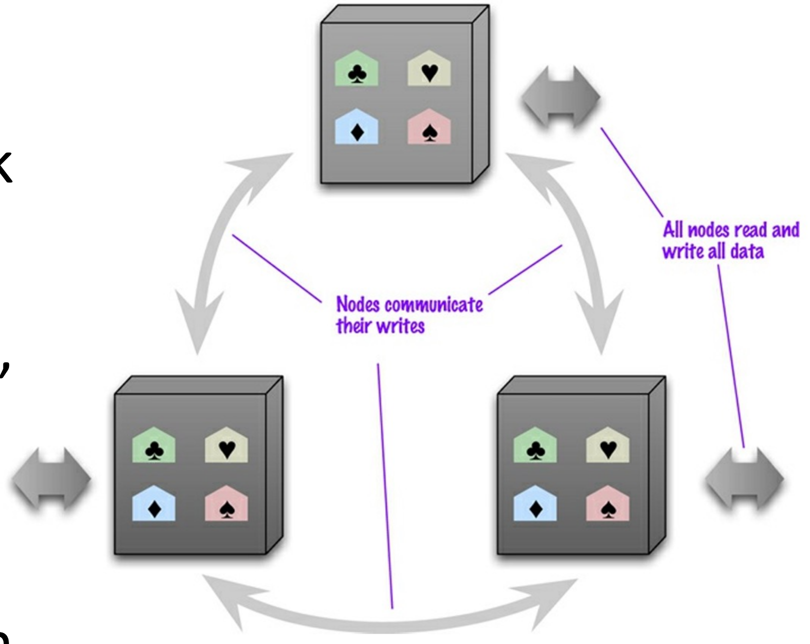
c. Replication – Peer-to-Peer archi.

- In a master slave replication, master node is a bottleneck and a single point of failure.
- Peer-to-peer has not master node.
- All replicas have equal weight, they can all accept writes and loss of any of them doesn't prevent access to the data store.
- Easy scale-out
- Consistency is the biggest complication.
- Write-write conflict – when two users attempt to update the same record at the same time



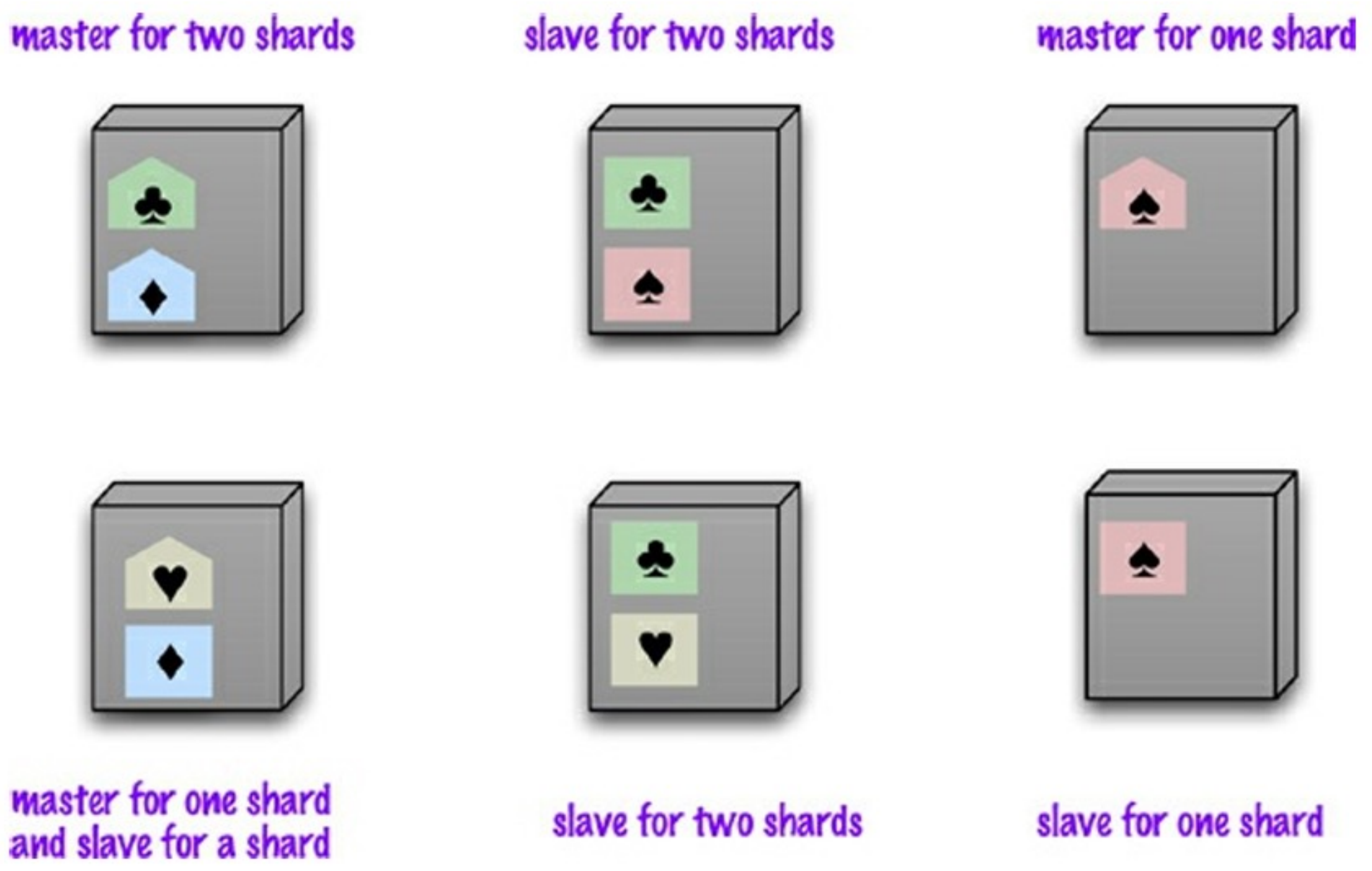
c. Replication – Peer-to-Peer archi.

- Coordinate replicas to avoid conflicts during data writes
- Strong guarantee similar to a master, with increased network traffic for coordination
- Majority agreement among replicas needed for writes, surviving loss of minority replica nodes
- Alternatively, tolerate inconsistent writes
- Develop policies to merge inconsistent writes in certain contexts
- Obtain full performance benefits by writing to any replica
- Points represent a spectrum trade-off between consistency and availability



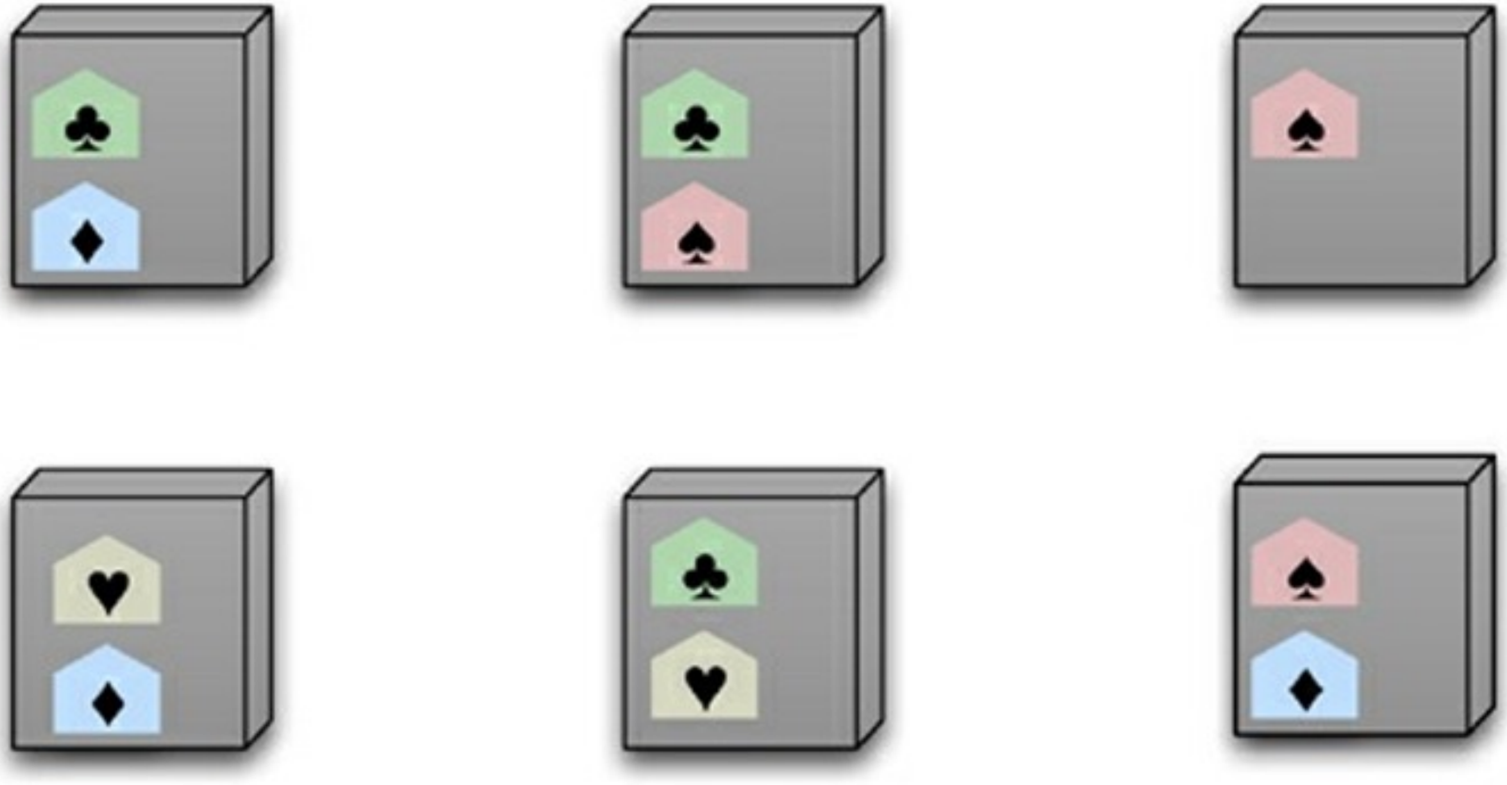
d. Combining Sharding with Replication

Using master-slave replication together with sharding



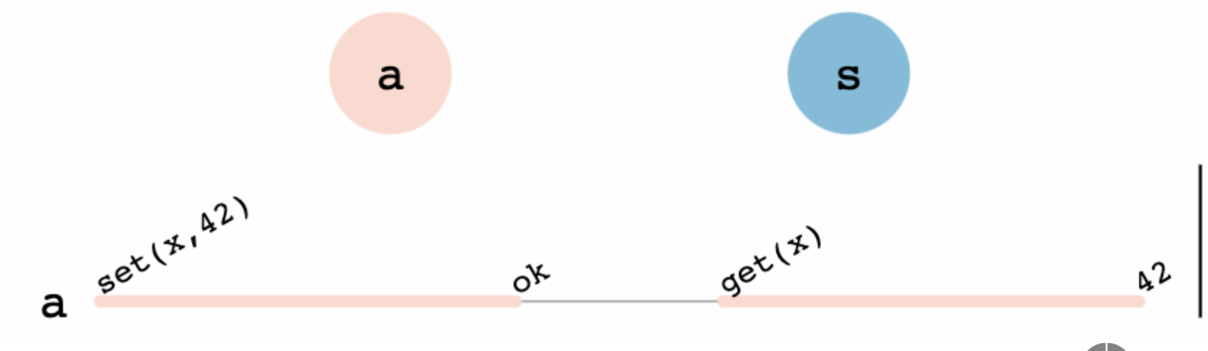
d. Combining Sharding with Replication

Using peer-to-peer replication together with sharding



Replicated Data Consistency

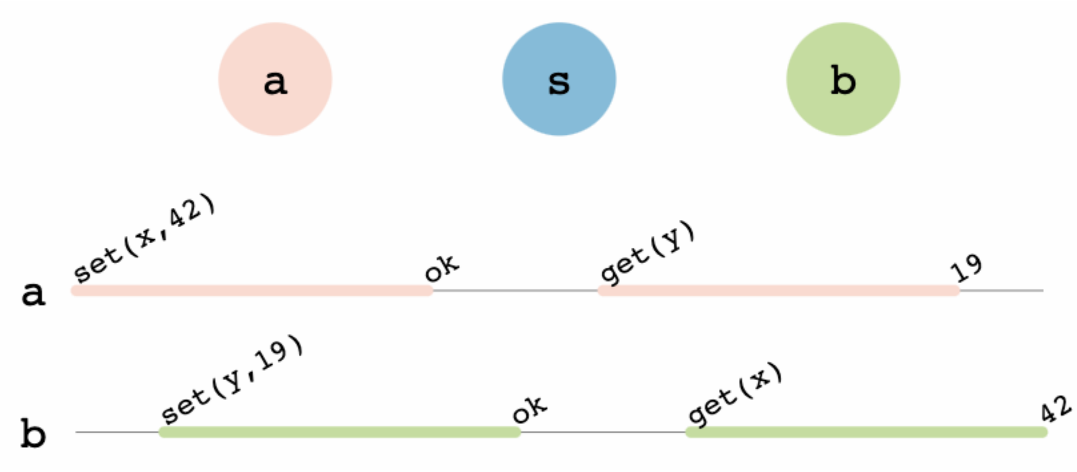
- Consider a non-distributed key-value store running on a single computer.
- A client can issue two commands:
    - get(k) request to retrieve the value associated with key k
    - set(k, v) request to associate the value v with key k



Replicated Data Consistency

- Consider a non-distributed key-value store running on a single computer.

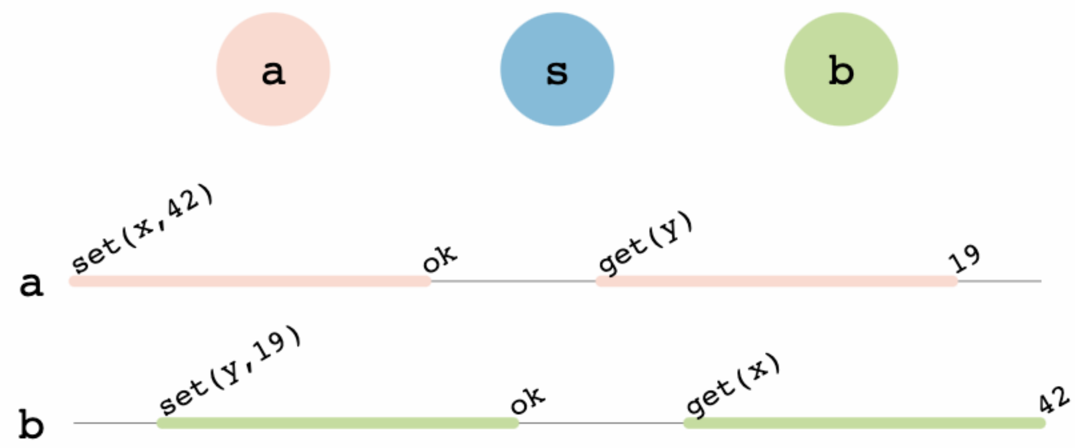
- **Multi-client scenario** where client **a** and client **b** concurrently set and then get a value from the key-value **s**.
- Everything looks good here, until there is a crash.
- If **s** fails, all the data would be lost.



Replicated Data Consistency

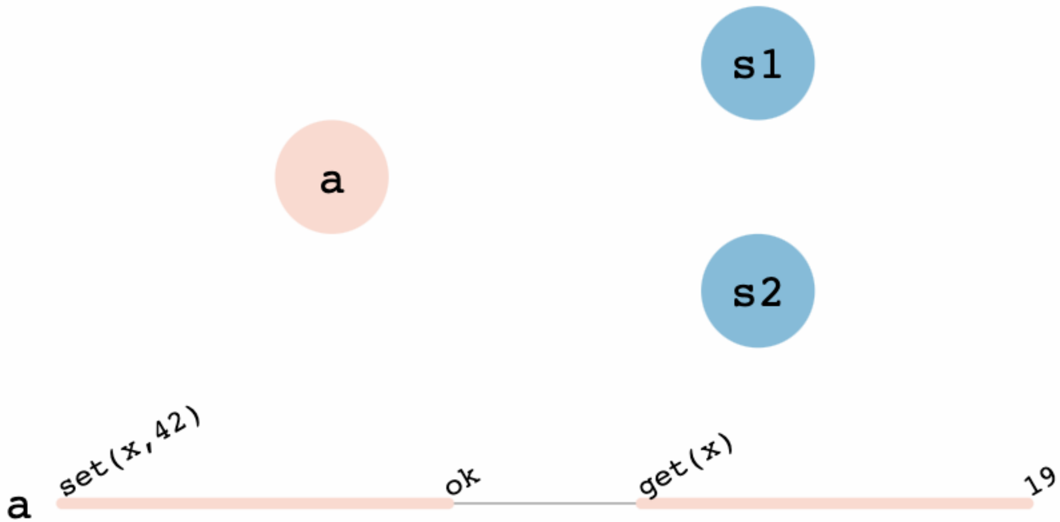
- Consider a non-distributed key-value store running on a single computer.

- **Multi-client scenario** where client **a** and client **b** concurrently set and then get a value from the key-value **s**.
- Everything looks good here, until there is a crash.
- If **s** fails, all the data would be lost.
- In reality, storage systems **replicate data across multiple computers** so that data survives even when any single computer fails.



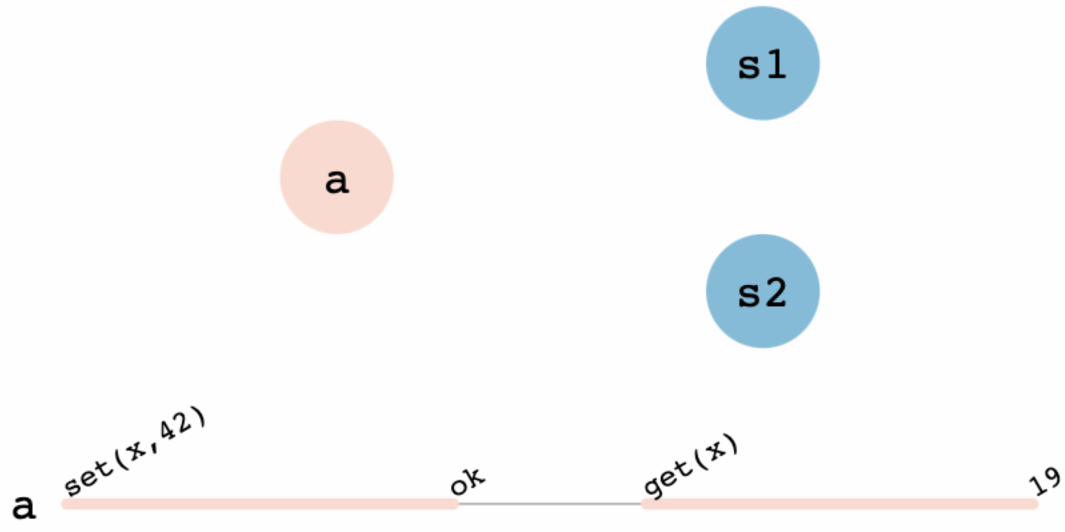
Strong & Weak Consistency

- Replication allows fault tolerance. However, **naively replicated storage system can behave very weirdly.**



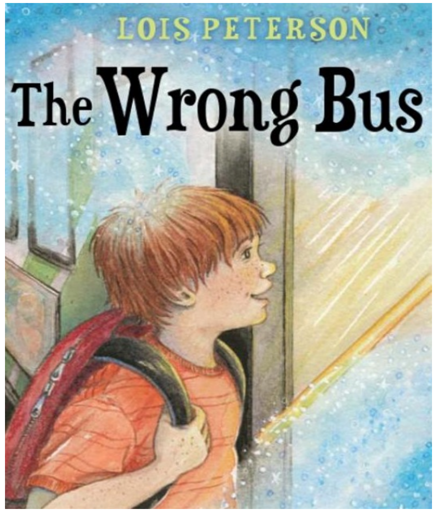
Strong & Weak Consistency

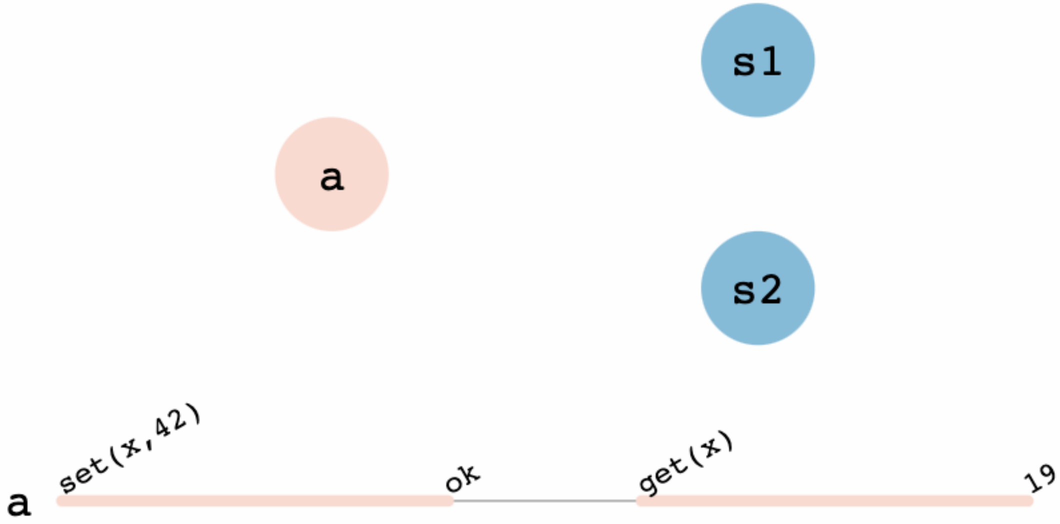
- Replication allows fault tolerance. However, **naively replicated storage system can behave very weirdly.**
- Example:
  - Consider a key-value store replicated across two servers (s1 and s2).
  - If a client-a issues a set (x, 42) request to s1 and then a get (x) request to s2, the get could return something that's not 42!



Strong & Weak Consistency

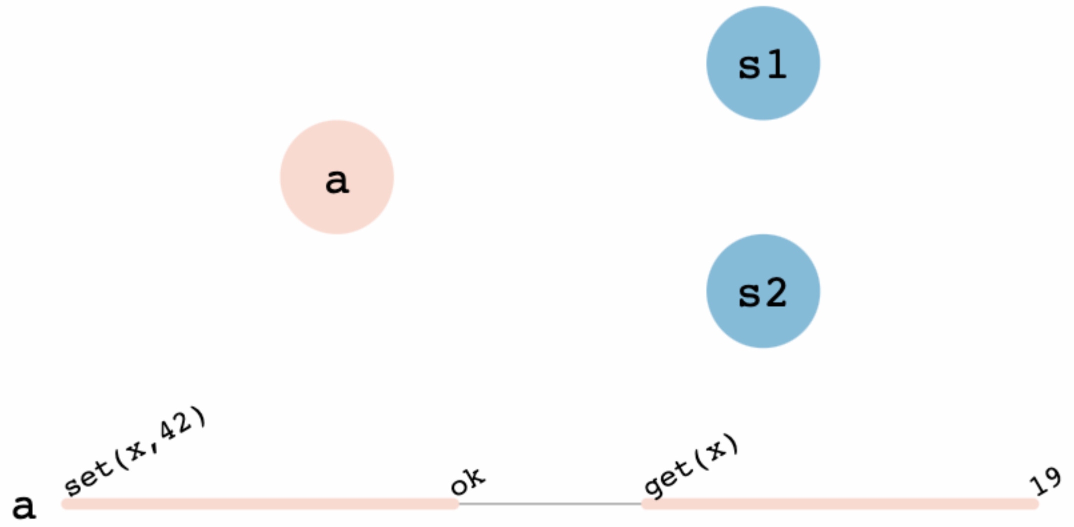
- Replication allows fault tolerance. However, naively replicated storage system can behave very weirdly.
- Example:
  - Consider a key-value store replicated across two servers (s1 and s2).
  - If a client-a issues a set (x, 42) request to s1 and then a get (x) request to s2, the get could return something that's not 42!





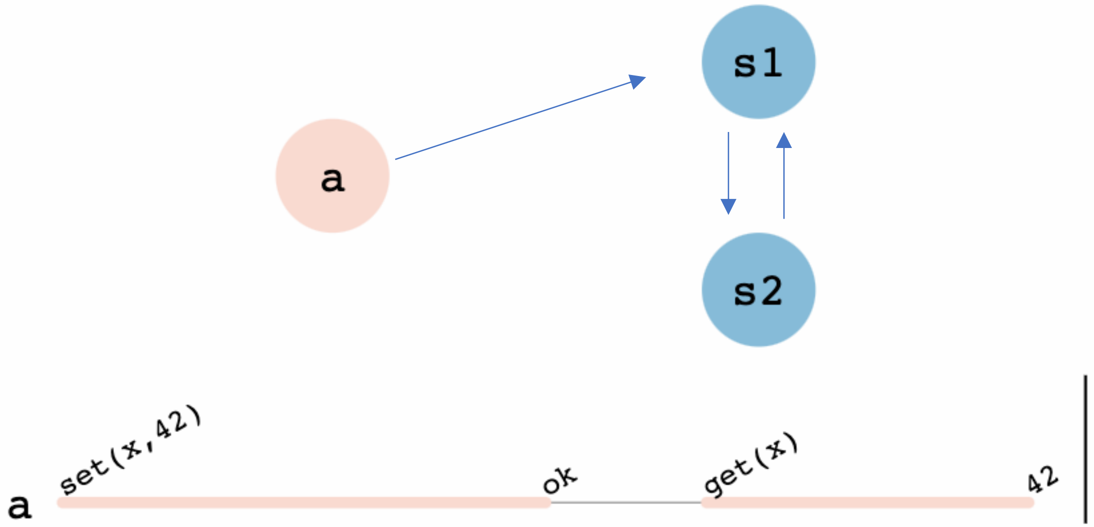
Strong & Weak Consistency

- Replication allows fault tolerance. However, naively replicated storage system can behave very weirdly.
- Example:
  - Consider a key-value store replicated across two servers (s1 and s2).
  - If a client-a issues a set (x, 42) request to s1 and then a get (x) request to s2, the get could return something that's not 42!
  - When a strong system exposes an unbridled number of anomalies like this, we say it is **inconsistent**.
  - When a replicated storage system behaves **indistinguishably** from a storage system running on a single computer, we say it is **strongly consistent**.



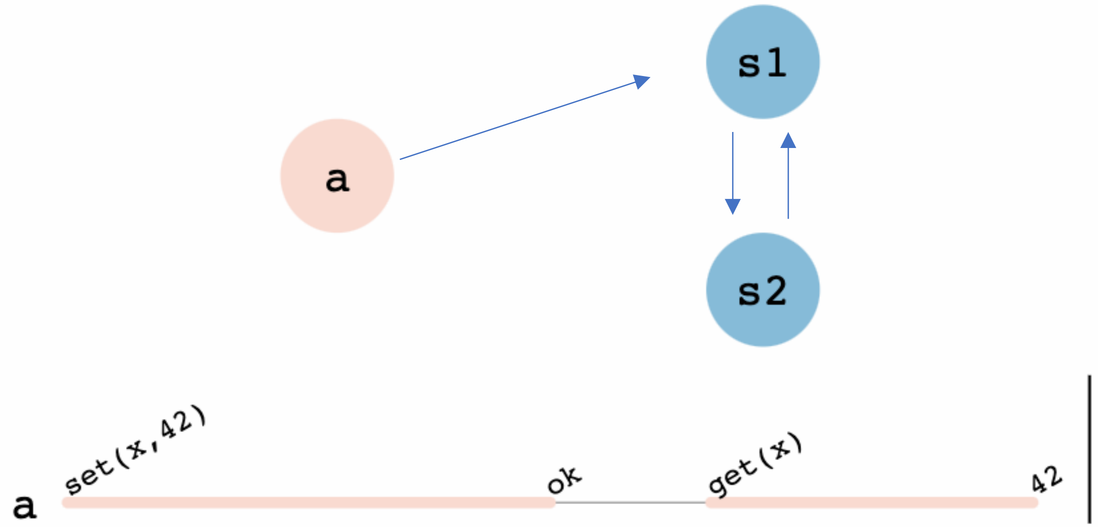
Strong & Weak Consistency

- Replication allows fault tolerance. However, naively replicated storage system can behave very weirdly.
- Example:
  - How to make the system consistent?



Strong & Weak Consistency

- Replication allows fault tolerance. However, naively replicated storage system can behave very weirdly.
- Example:
  - How to make the system consistent?
  - If **s1** propagates the effect of the **set(x, 42)** command to **s2** before responding to a, then **s2** will correctly return 42 when it receives a **get(x)** request.
  - By doing this, the system implements strongly consistency.



Strong & Weak Consistency

- “Strongly consistent” is not well defined.
- It might be used colloquially to express a general notion that a storage system doesn’t act weirdly.
- More formally, it might be used as a synonym for a very formally defined form of consistency like **linearizability** (that we will see later)
- Implementing strong consistency is both challenging and costly – the algorithm used to implement strong consistency are often complex.
- Strong consistency is fundamentally at odds with low-latency and availability. For instance, when we make our key-value store strongly consistent, the set (x, 42) request took longer than it did when the key-value store was inconsistent.
- For this reason, system architects opt **weak consistency** instead of strong consistency.
- Weakly consistent storage systems do not behave indistinguishably from storage systems running on a single computer.

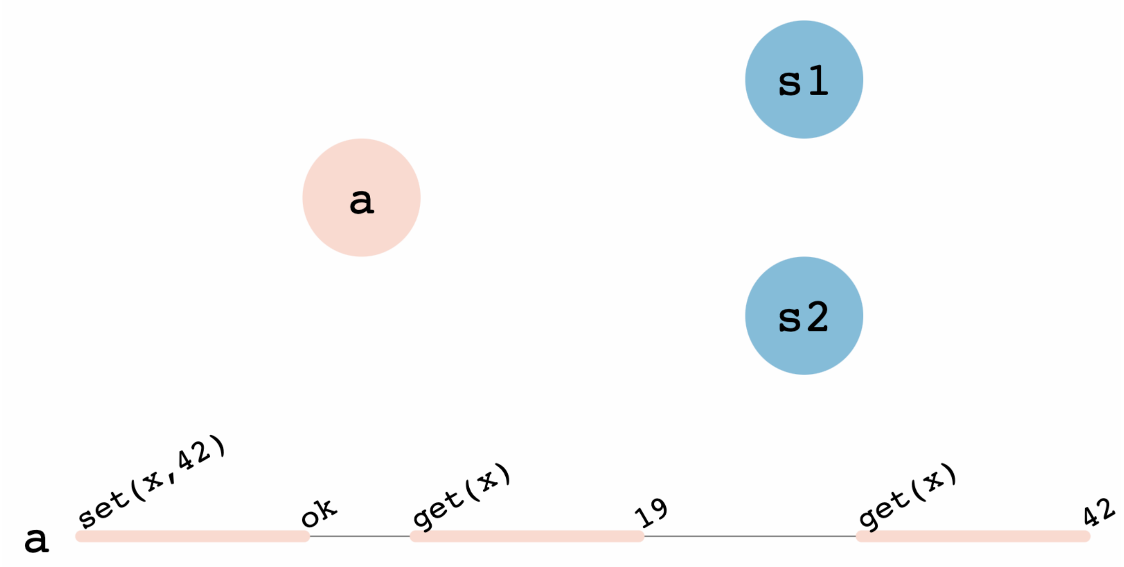
Weak Consistency

Eventual consistency

- One of the weakest of all weak consistencies
- It guarantees that if all clients stop issuing requests for a while, then all the system's replicas will converge to the same state.

Example:

- Each server in the key-value store buffers write (set) requests and propagates them to the other servers every so often.
- A get(x) requests following a set(x,42) request can return something other than 42.
- But, if a client waits long enough, eventually a get(x) request will return 42.



Consistency Models

- There are a buffet of flavors (or models) of weak consistency
- Each consistency model exposes various degrees of inconsistency with various performance characteristics.
- Six consistency models for a replicated key-value store

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Bounded Staleness	See all "old" writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.

Consistency Models

- There are a buffet of flavors (or models) of weak consistency
- Each consistency model exposes various degrees of inconsistency with various performance characteristics.
- Six consistency models for a replicated key-value store

Guarantee	Consistency	Performance	Availability
Strong Consistency	excellent	poor	poor
Eventual Consistency	poor	excellent	excellent
Consistent Prefix	okay	good	excellent
Bounded Staleness	good	okay	poor
Monotonic Reads	okay	good	good
Read My Writes	okay	okay	okay

Consistency Models

- **Strong Consistency.** With a strongly consistent key-value store, a **get(x1,...,xn)** request is guaranteed to return the most recently written values of every key from x1 to xn.
  - In other words, a read observes the effect of all previously completed writes.

Consistency Models

- **Strong Consistency.** With a strongly consistent key-value store, a **get(x1,...,xn)** request is guaranteed to return the most recently written values of every key from x1 to xn.
  - In other words, a read observes the effect of all previously completed writes.

- **Eventual Consistency:** With an eventually consistent key-value store, a **get(x1,...,xn)** request is guaranteed to return values v1,...,vn where vi is any previously written value of key xi. With our assumption that writes are eventually propagated to all replicas, if clients stop issuing write requests for a while, reads will (typically) return the most recently written values.
  - An eventually consistent read can return any value for a data object that was written in past.
  - Such a read can **return results from a replica that has received an arbitrary subset of writes** to the data object being read

Consistency Models

- **Consistent Prefix.** Recall our assumption that writes are executed in the same order on all replicas. With a key-value store guaranteeing consistent prefixes, a get request is guaranteed to return values that are consistent with some prefix of this sequence of writes. Note that for get requests reading a single value, consistent prefix is equivalent to eventual consistency.
  - A reader is <u>guaranteed to observe an ordered sequence of writes</u> starting with the first write to a data object.
  - For example, a read may be answered by a replica that receives writes in order from a master replica **but has not yet received an unbounded number of recent writes.**

Consistency Models

- **Bounded Staleness.** With a key-value store guaranteeing bounded staleness, a get(x1,...,xn) request is guaranteed to return values v1,...,vn where vi is **some value that key xi took on during the last t minutes for some fixed t.**
  - Ensures that read results are <u>not too out-of-date</u>
  - Staleness is defined by a time period, say 5 minutes.
  - The system will guarantee that a read operation will return any values written more than T minutes ago.
  - Alternate definitions in terms of number of missing writes or even the amount of inaccuracy exist.

Consistency Models

- **Monotonic Reads.** With a key-value store guaranteeing monotonic reads, a client's initial read of value x is only guaranteed to return some previously written value of x (this is equivalent to eventual consistency). However, <u>each subsequent read of x by the same client is guaranteed to return the same value of x or a more up-to-date value of x compared with the previous read of x</u>. (avoid getting into the wrong train scenario!!)
- This is a property that applies to a sequence of read operations that are performed by a given storage system client.
- Also know as "**session guarantee**"
- With monotonic reads, a client can read arbitrarily stale data, as with eventual consistency we will ensure that the data will up-to-date over time.
- With monotonic read, if the client issues a read operation and then later issues another read to the same object(s), the 2nd read will return the same value(s) or the results of later writes.

Consistency Models

- **Read My Writes.** With a key-value store guaranteeing read my writes, if a client writes a value v to key x, then any subsequent reads of x by the same client will return v or a more recently written value of x.
- This is a property that applies to a sequence of operations performed by a single client.
- It guarantees that the effect of all writes that were performed by the client are visible to the client’s subsequent read.
- If a client writes a new value for a data object and then reads this object, the read will return the value that was last written by the client (or some other value that was later written by a different client).

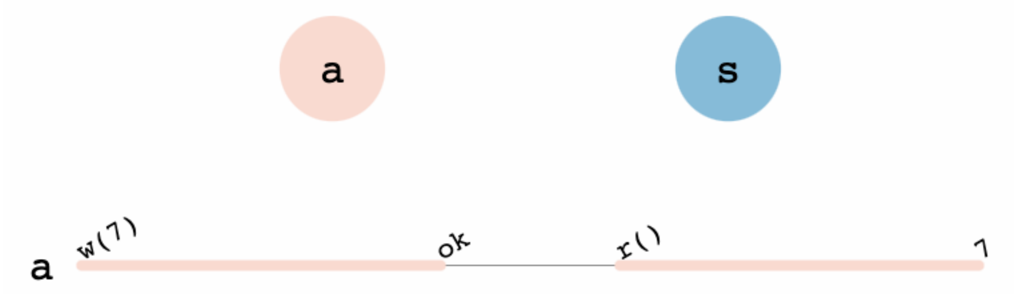
Linearizability

- We established (rather informally) that a distributed storage system is strongly consistent if it behaves indistinguishably from a storage system running on a single computer.
- **Linearizability:** a formalism of strong consistency initially proposed by Maurice Herlihy and Jeannette Wing in 1990.
- Consider a simple storage system, a register, that stores a single value.
- A client can issue:
  - w(x) request to write x to the value
  - r(x) to read the value of x

Linearizability

Setting assumptions…

- A client-a writes the value 7 into the register running on a single computer



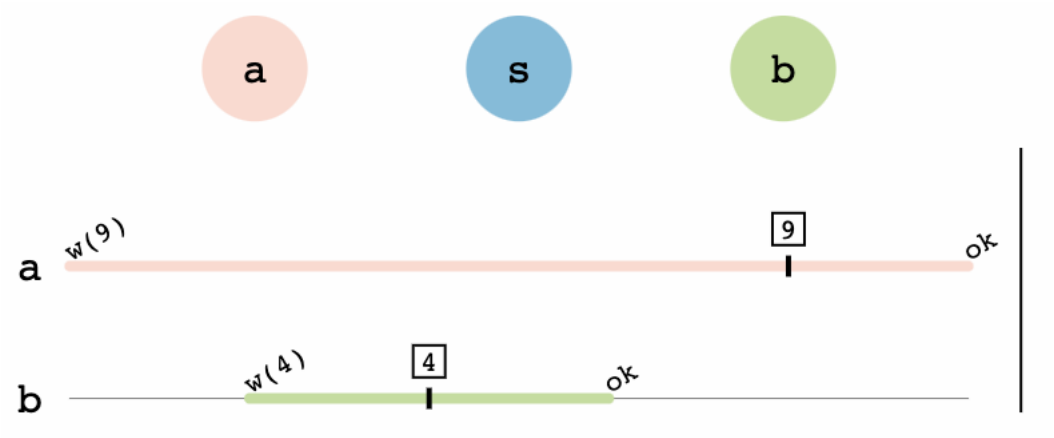
Assume that the register can instantaneously process the request send an “ok”.

Forget about data propagation for the time being.

Linearizability – slow and fast delivery of messages

- Now, let's look at how two clients might interact with such a register
- When the network starts to delay and quicken the delivery of messages.

The client a sends a very slow w(9) request to the register



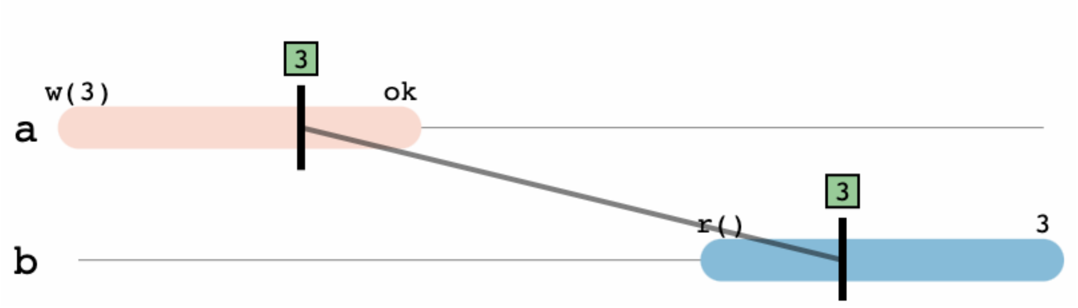
Before the w(9) request has a chance to make it to the register, client b sends a very speedy w(4) request.

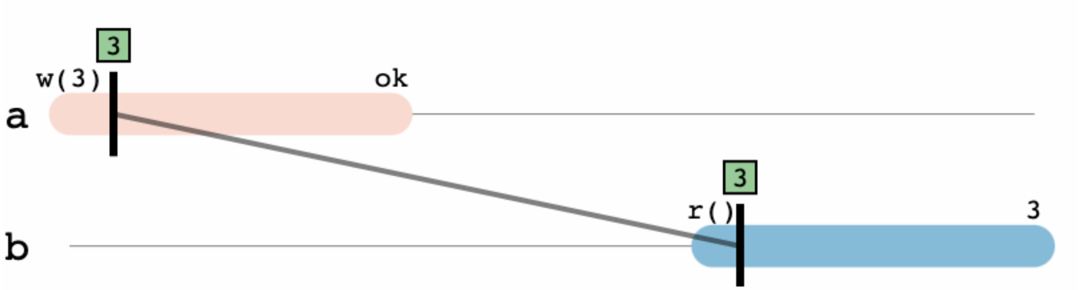
The time at which a message reached a server matters!!

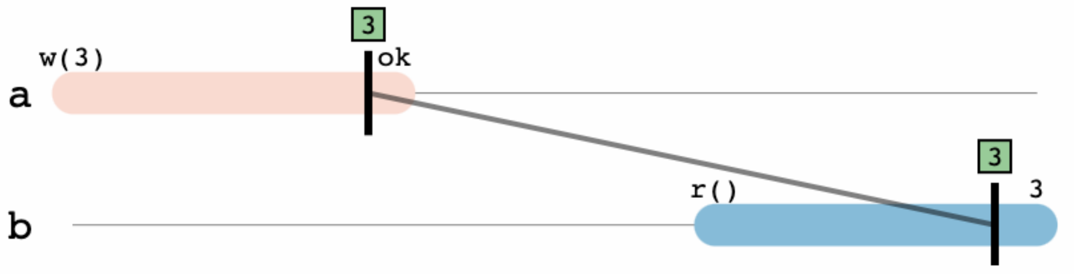
Let's see some examples.

Linearizability – slow and fast delivery of messages

- From a client's point of view, he can only make guesses on when a message has received at the Register.
- Guessable zones! The value of x would be "3" for all the guesses.

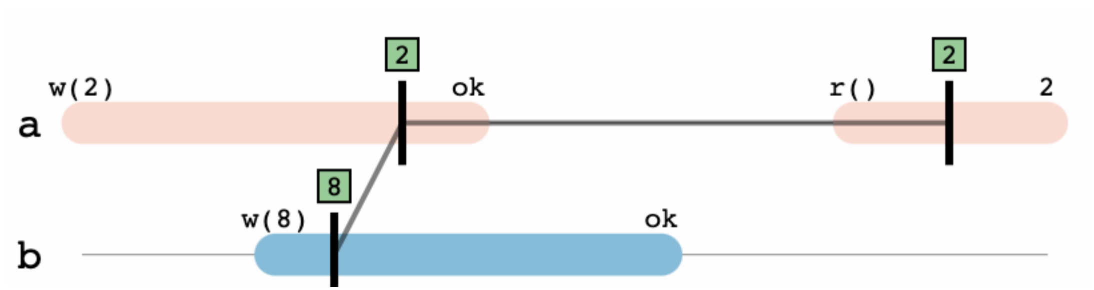


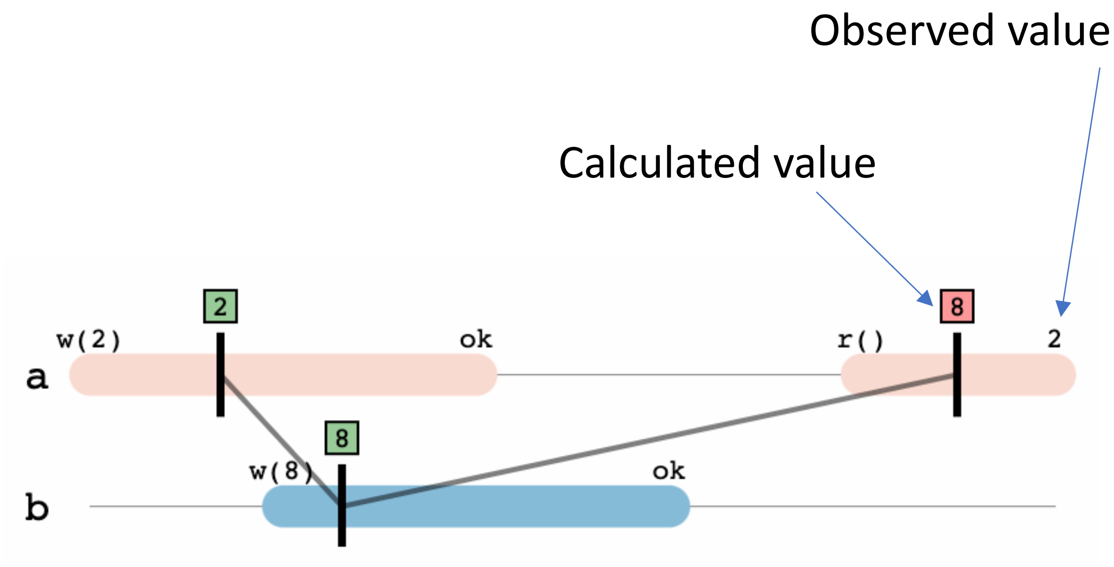




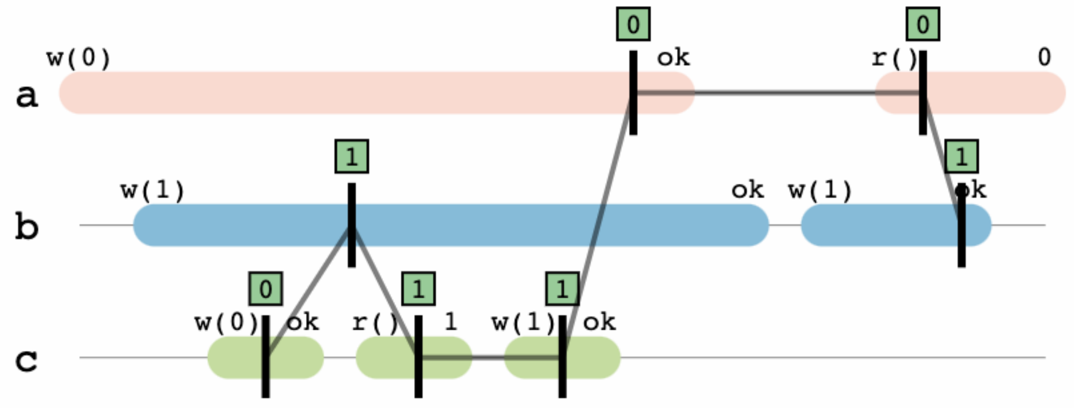
For all these guesses, the value of x would remain the same for r(); however, this is not true for all the cases.

Linearizability – slow and fast delivery of messages

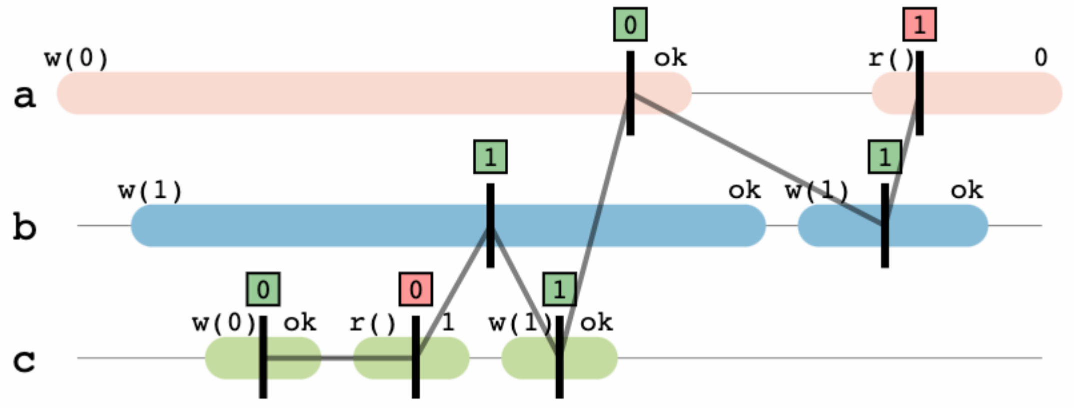




Linearizability – slow and fast delivery of messages

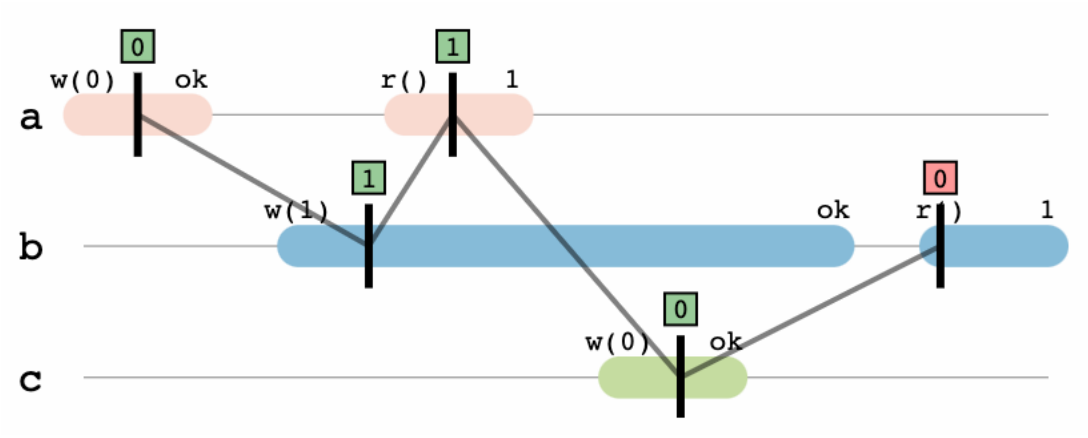


If any of the registers in our execution are shaded red, then our guess can not possibly be correct. This is bad. If all of the registers are green, then our guess could be correct. This is good.



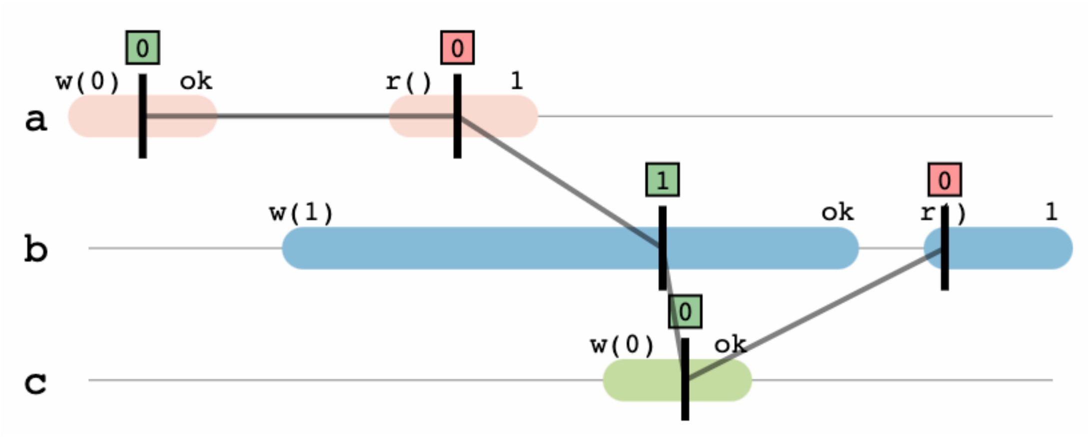
Linearizability – slow and fast delivery of messages

Can you make a valid guess?



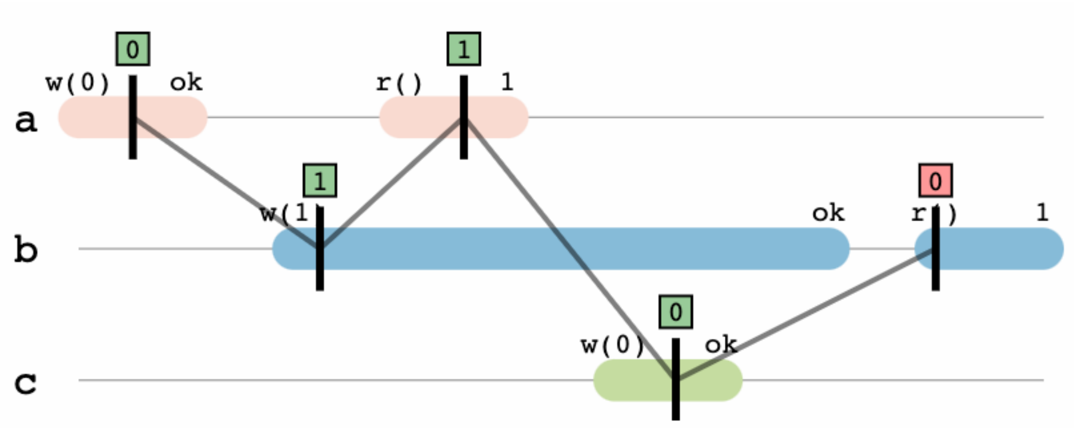
Linearizability – slow and fast delivery of messages

Can you make a valid guess?



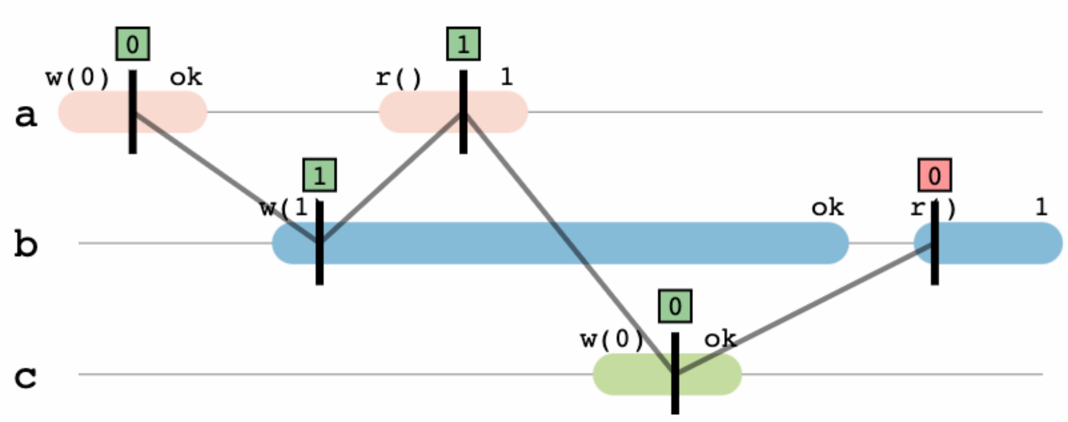
Linearizability – slow and fast delivery of messages

Can you make a valid guess?



Linearizability – slow and fast delivery of messages

Can you make a valid guess? Alas, all guesses are incorrect; there is no potentially correct guess!

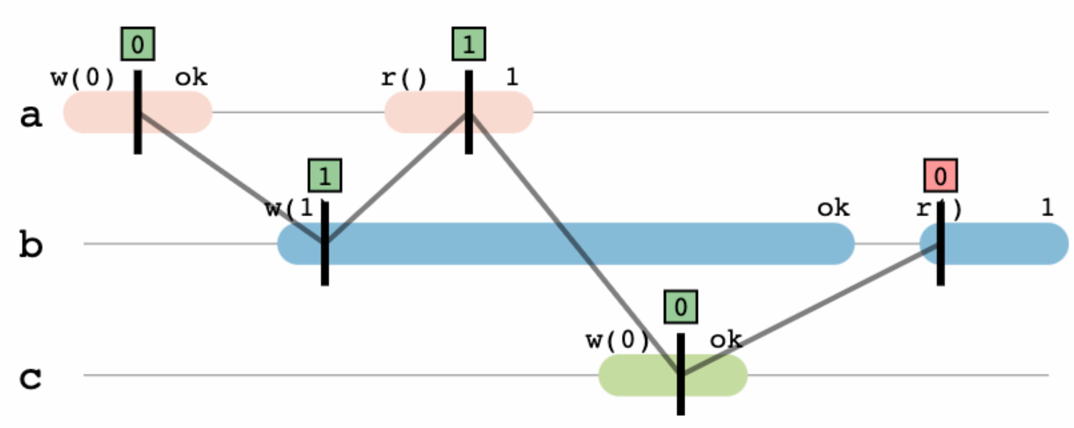


This means that there does not exist a way for us to guess the moment that each request arrives at the register such that the guess is consistent with the responses of all the read requests.

Thus, this execution could not have taken place with a register running on a single computer.

Linearizability – slow and fast delivery of messages

Can you make a valid guess? Alas, all guesses are incorrect; there is no potentially correct guess!

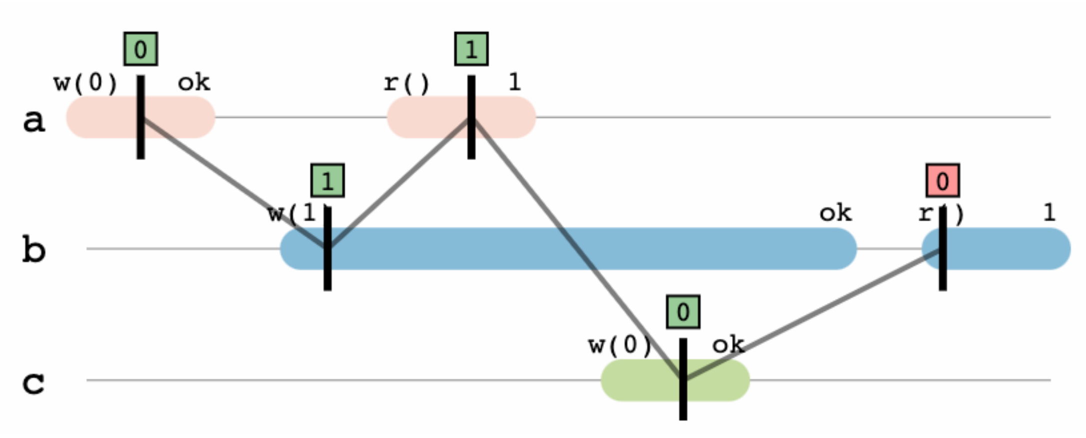


In other words, we were able to distinguish the behavior of the register from the behavior of a register running on a single computer!

Surprise, this is linearizability!

Linearizability – slow and fast delivery of messages

Can you make a valid guess? Alas, all guesses are incorrect; there is no potentially correct guess!



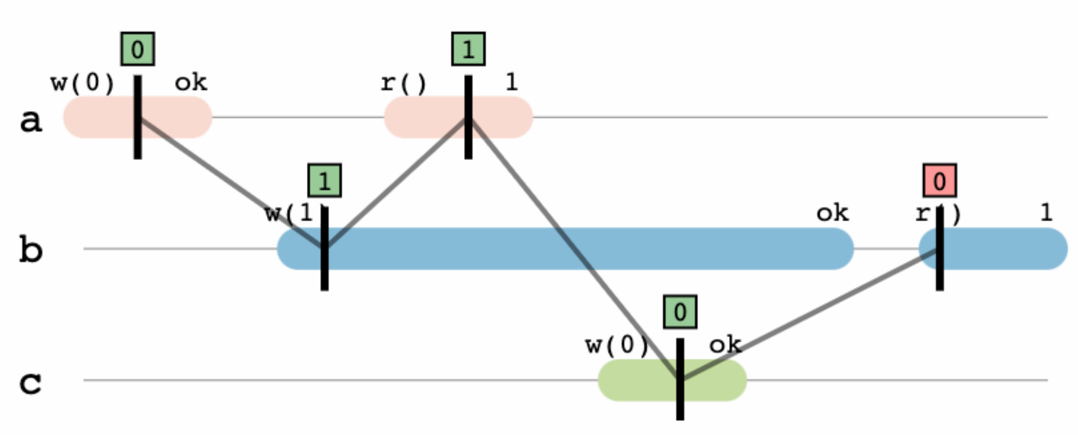
For a given execution, if there exists a potentially correct guess, then we say the execution is linearizable.

If all guesses are definitely incorrect, then we say the execution is not linearizable.

Similarly, a **linearizable register** is one that only allows linearizable executions.

Linearizability – slow and fast delivery of messages

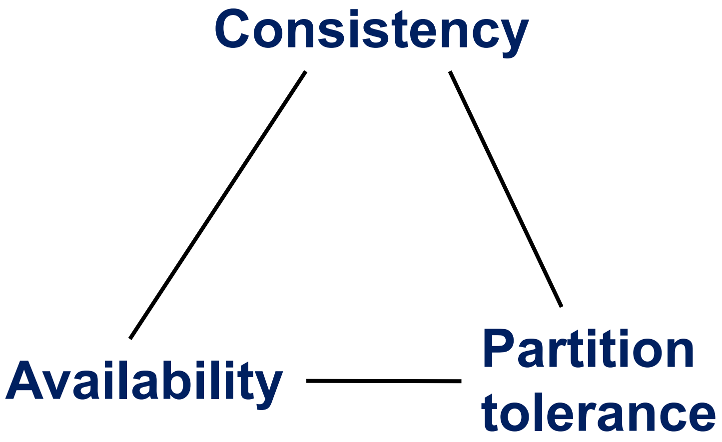
Can you make a valid guess? Alas, all guesses are incorrect; there is no potentially correct guess!



Linearizability can be extended quite naturally to deal with many other types of objects (e.g. queues, sets). This generalization of linearizability, as well as a full formalization, can be found in Herlihy and Wing's 1990 paper: *Linearizability: A Correctness Condition for Concurrent Objects*.

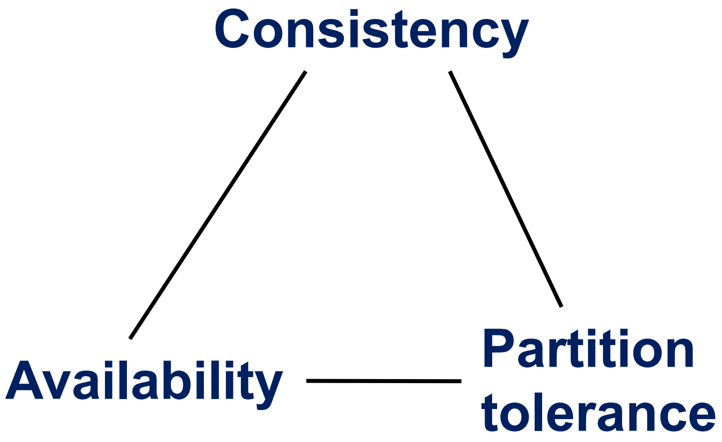
CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all writes are atomic, subsequent requests see the new value.
  - **Availability**: the distributed system is always available (and returns a value) as long a single node is running.
  - **Partition tolerance**: single node (and link) failures do not prevent the system from operation.



CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all writes are atomic, subsequent requests see the new value.
  - **Availability**: the distributed system is always available (and returns a value) as long a single node is running.
  - **Partition tolerance**: single node (and link) failures do not prevent the system from operation.

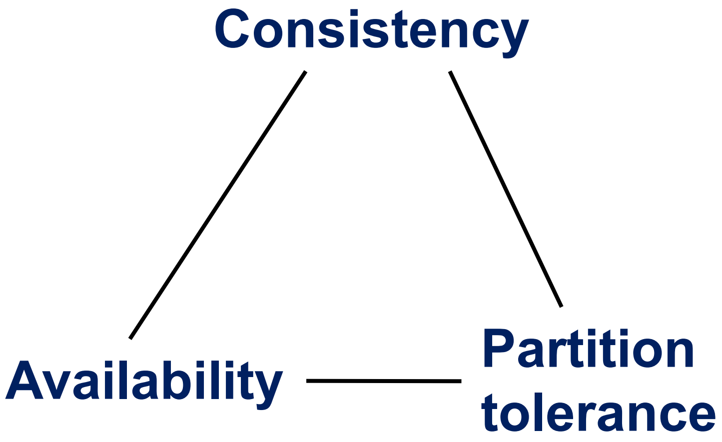


Largely a conjecture attributed to Eric Brewer (UC Berkeley), later formally proven by Gilbert/Lynch (MIT):

"A distributed system can satisfy any two of these guarantees at the same time, but not all three."

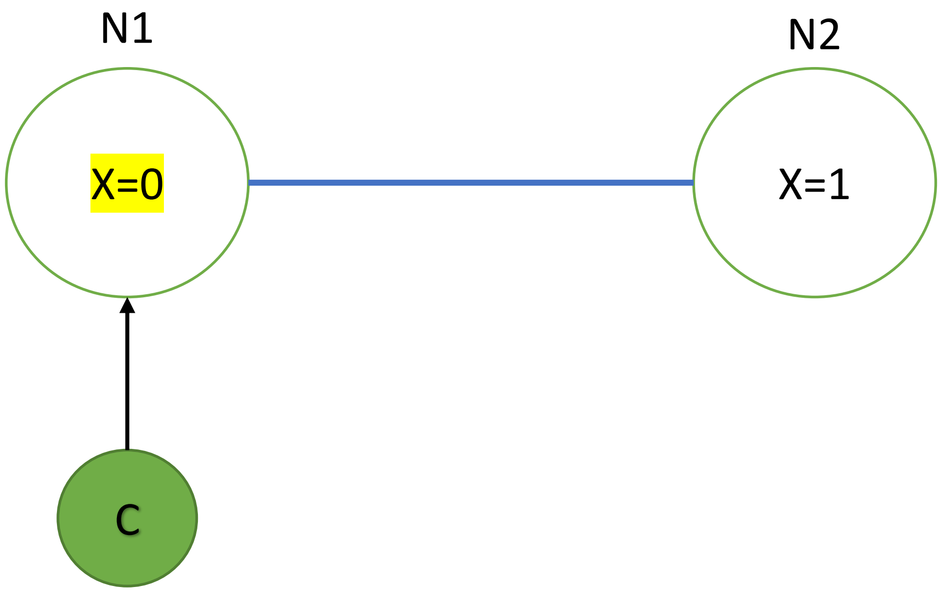
CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or return latest written value by any client

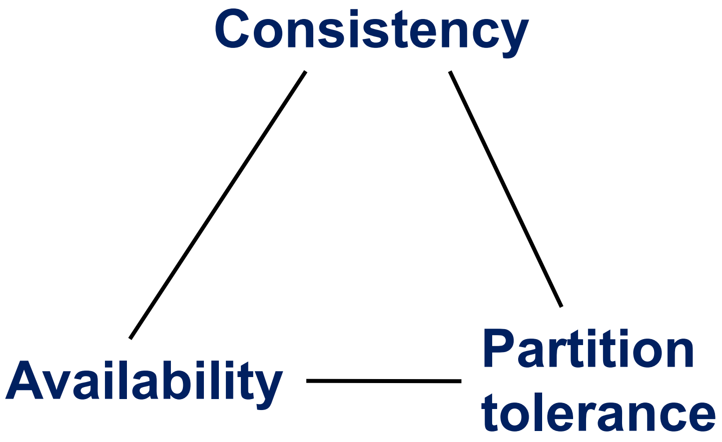


CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
    - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

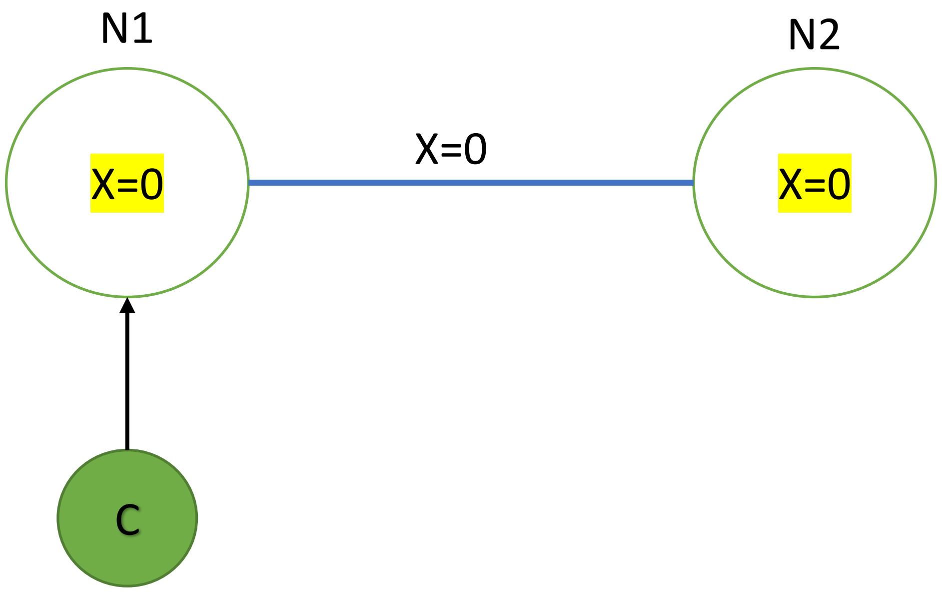


Consistent

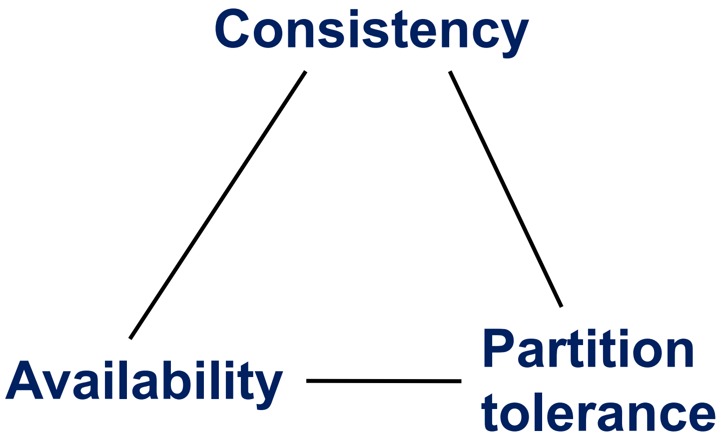


CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

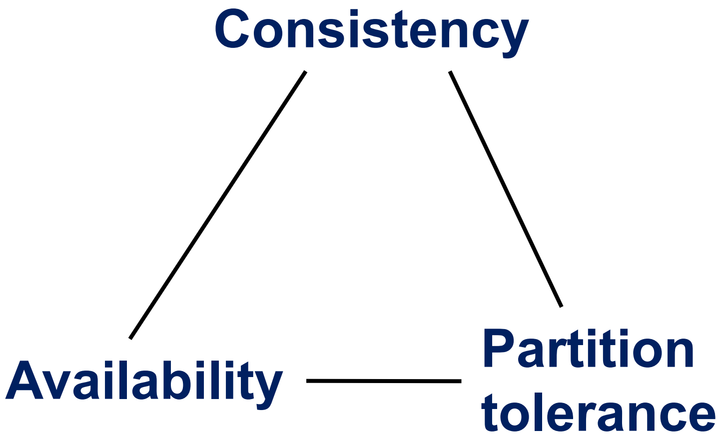


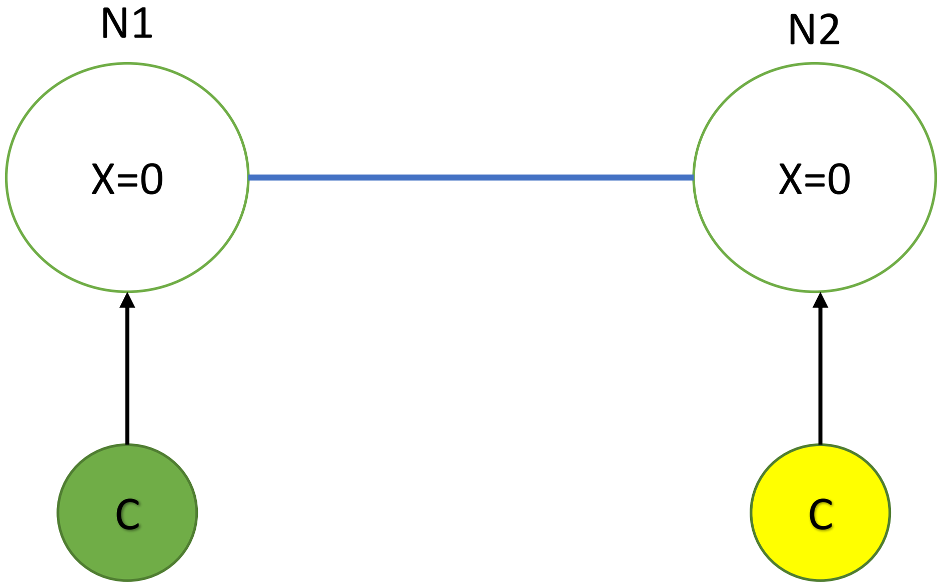
Consistent



CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client



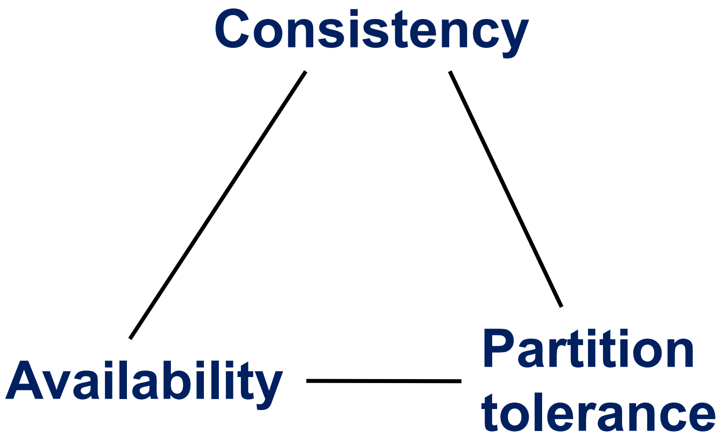


Consistent

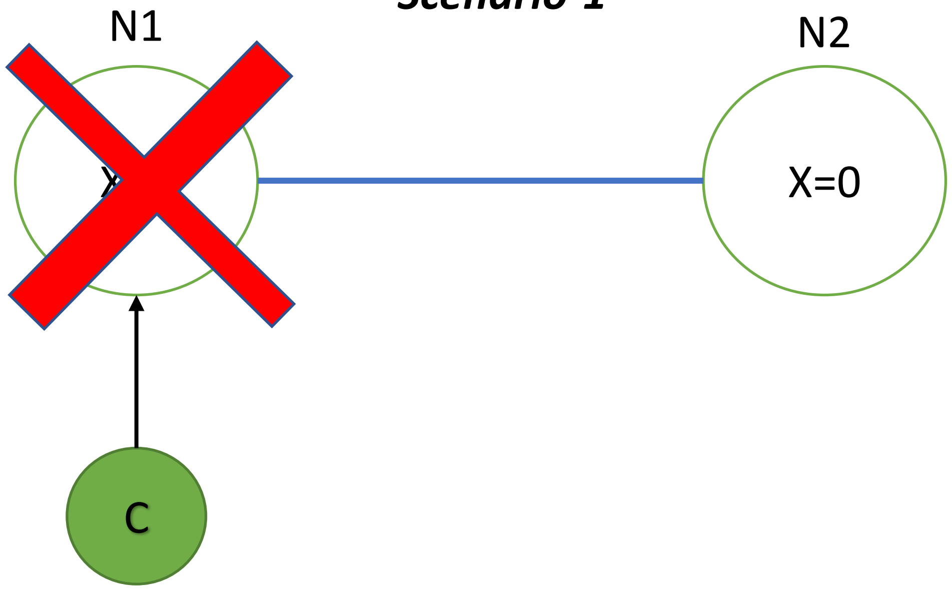
Availability: the distributed system is always available (and returns a value) as long a single node is running.

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client



Scenario-1

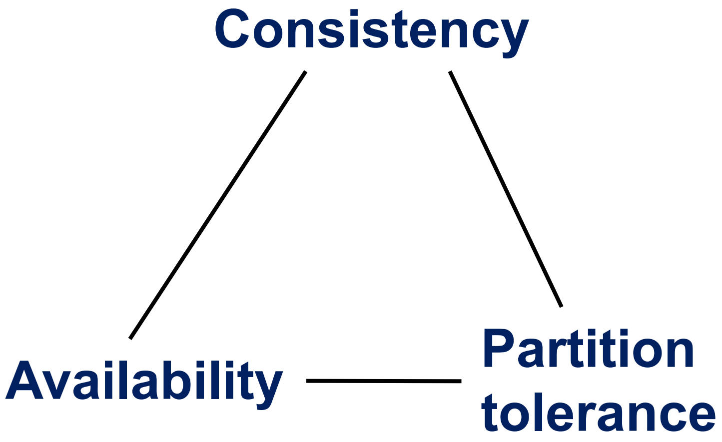


Consistent

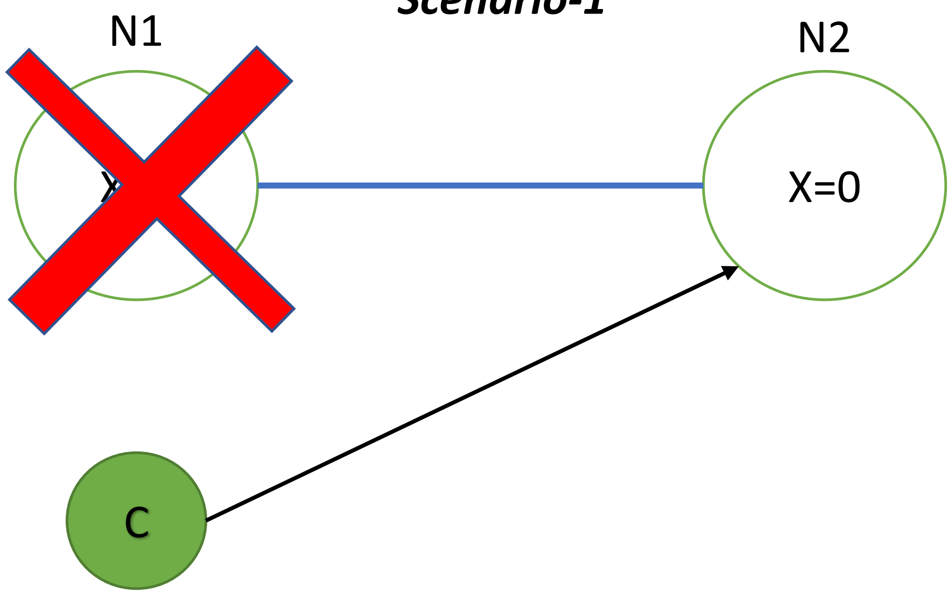
Availability: the distributed system is always available (and returns a value) as long a single node is running.

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client



Scenario-1

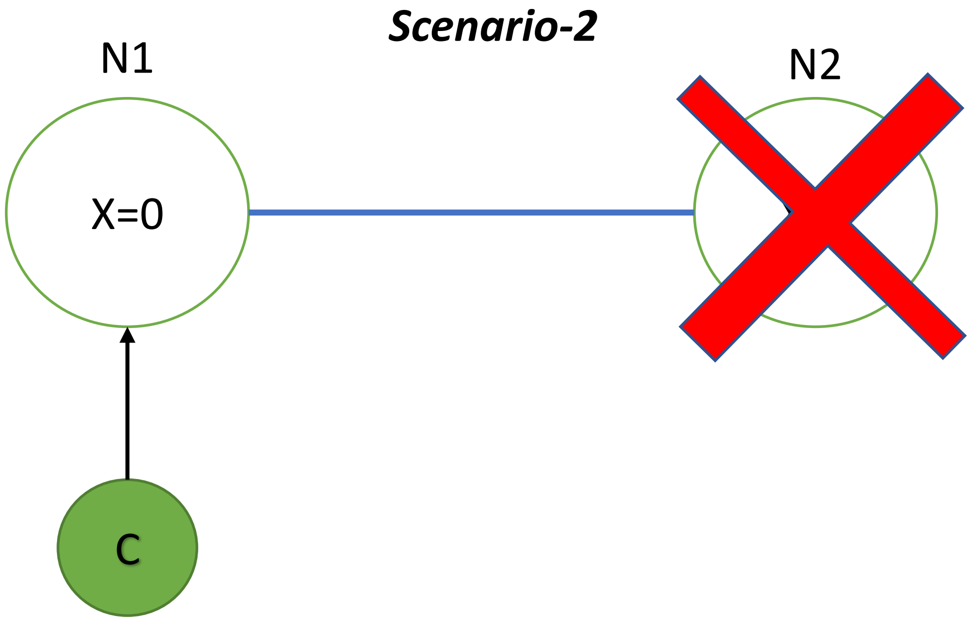


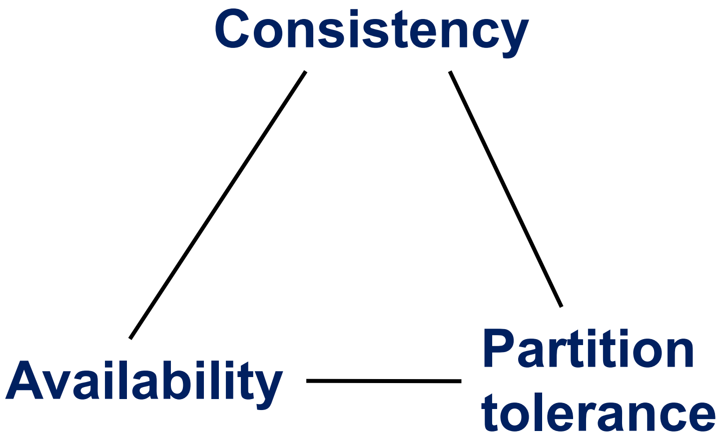
Consistent

Availability: the distributed system is always available (and returns a value) as long a single node is running.

CAP Theorem

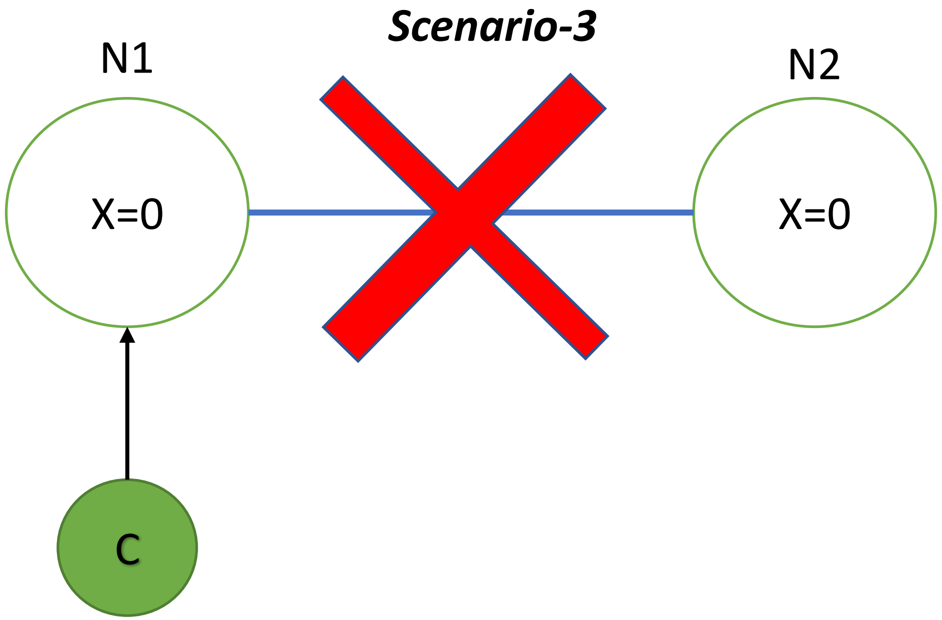
- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

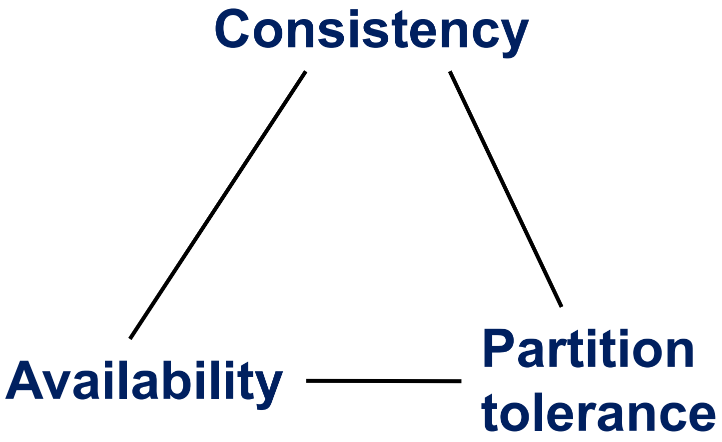




CAP Theorem

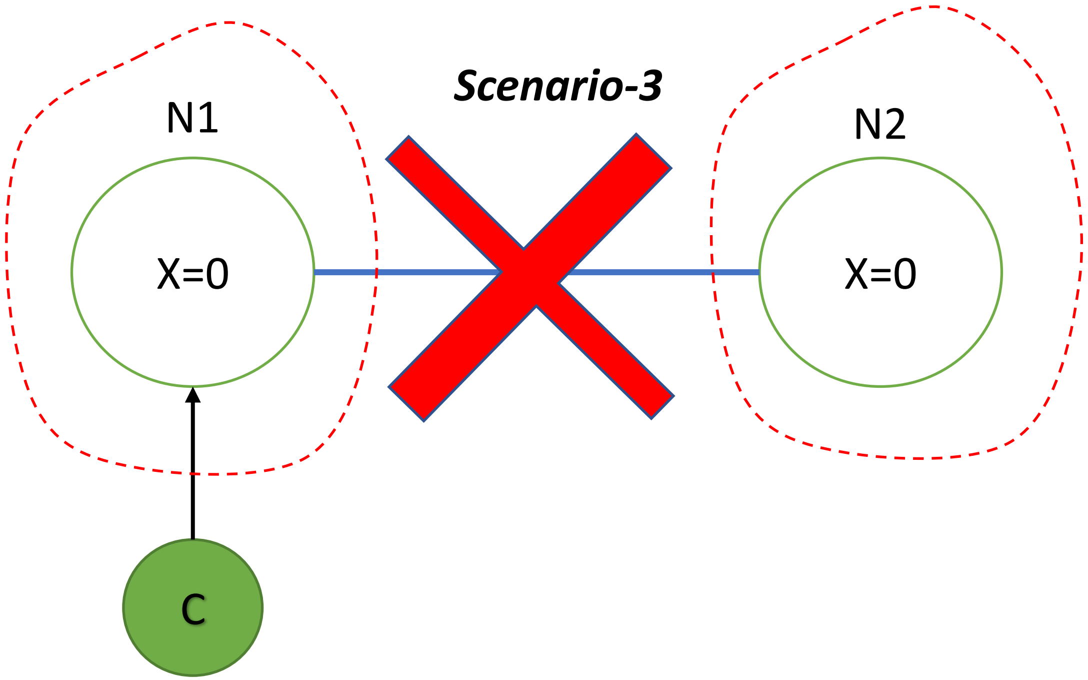
- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

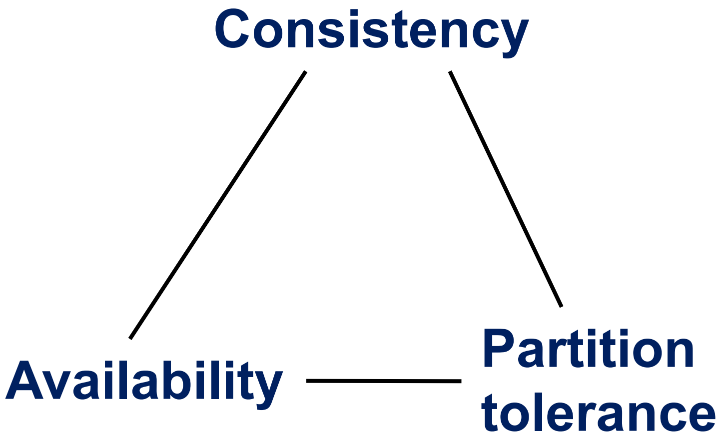




CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

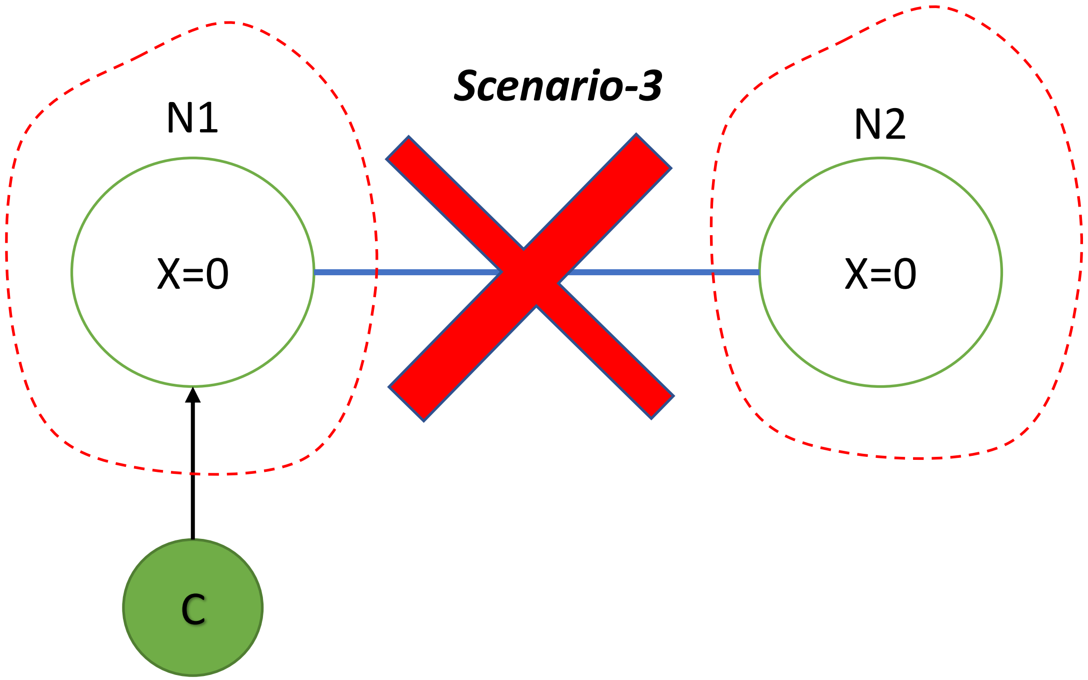


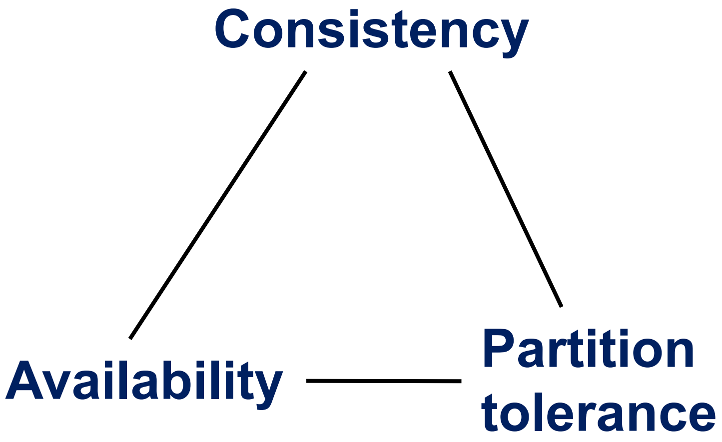


Partition tolerance: single node (and link) failures do not prevent the system from operation.

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

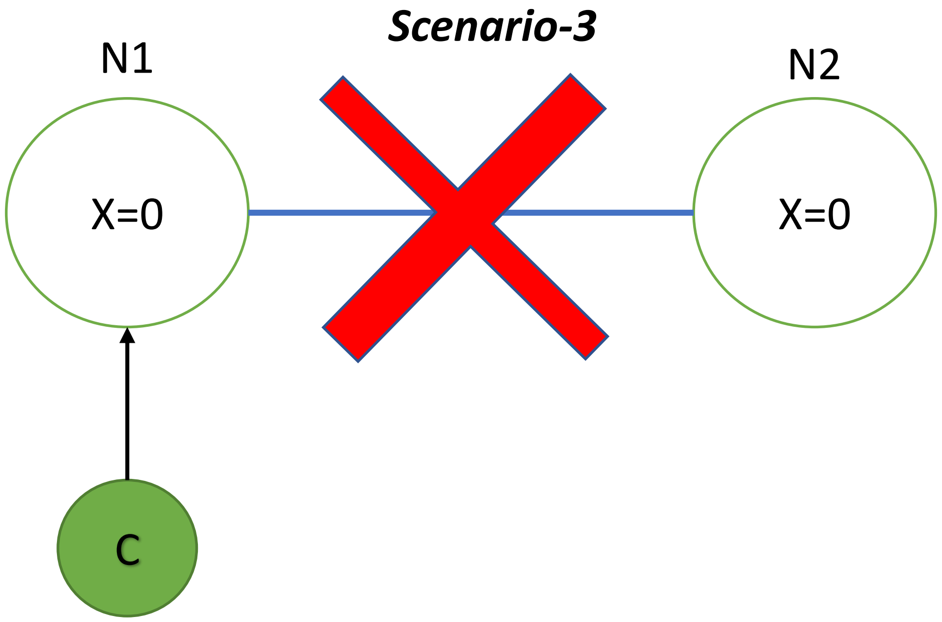


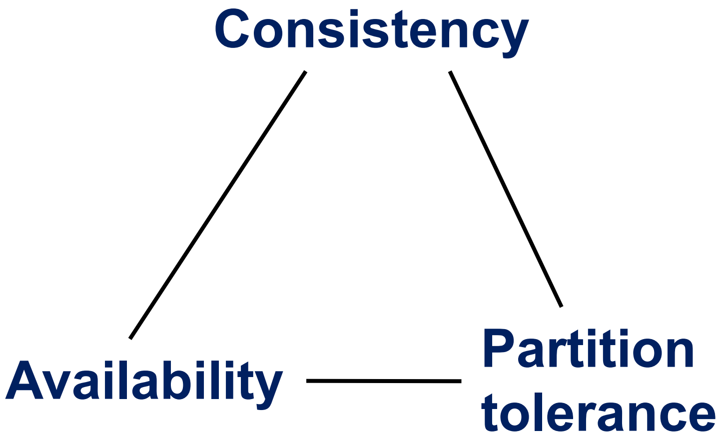


Partition tolerance: single node (and link) failures do not prevent the system from operation.

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

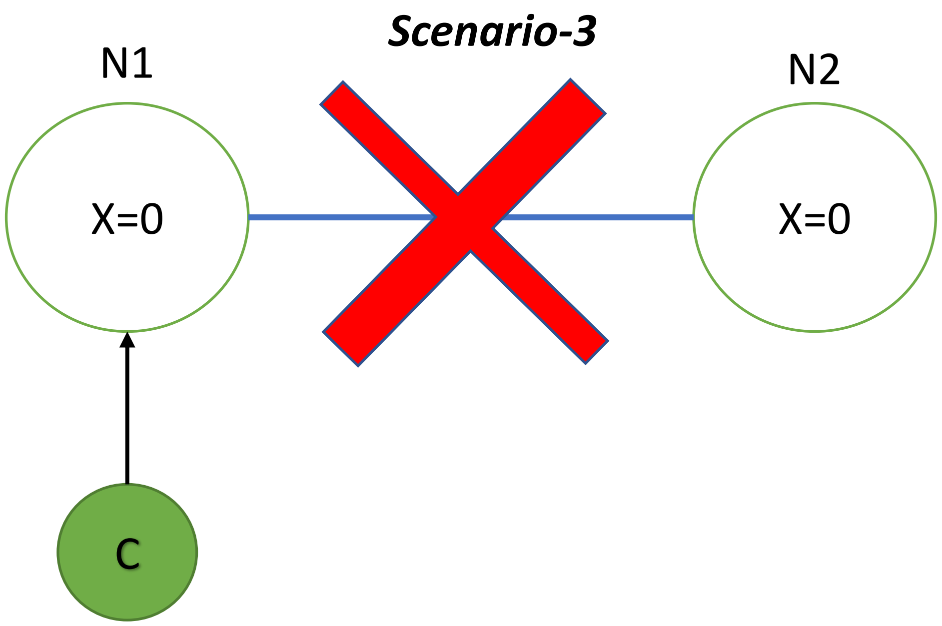




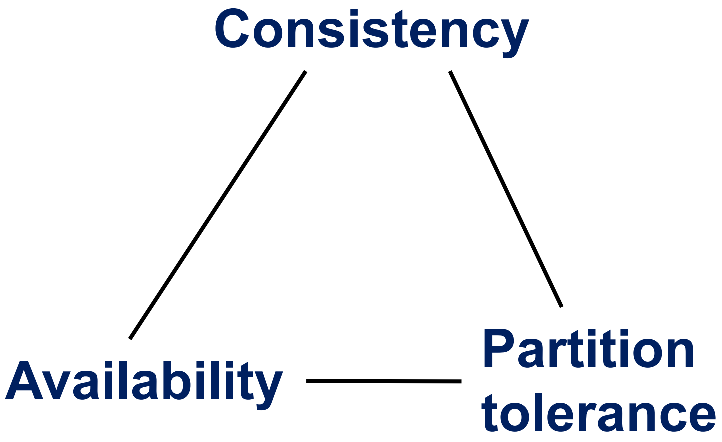
Consistent or Available??

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client



Consistent Design

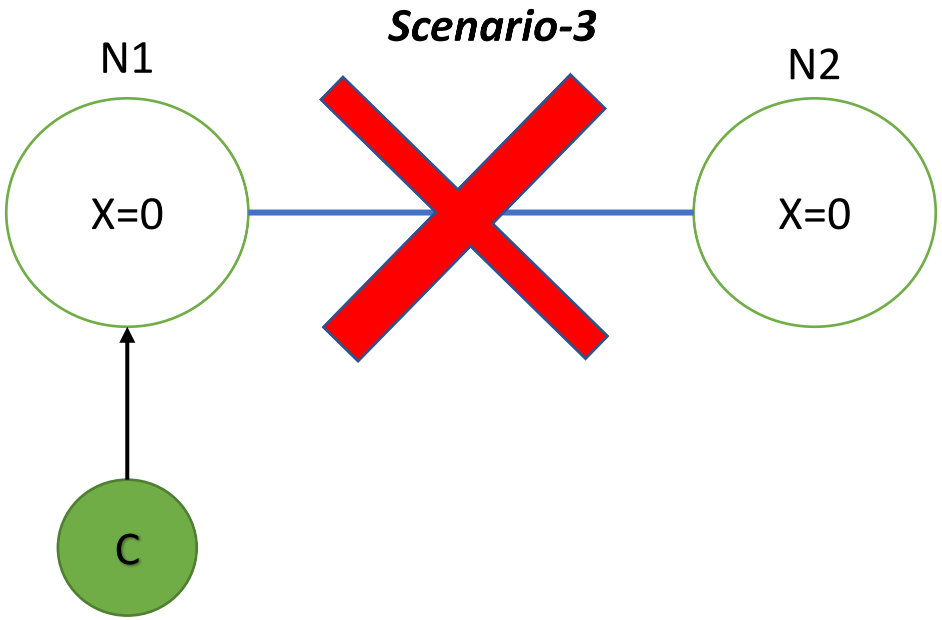


N1: “C is not allowed to read/modify the value of X now”

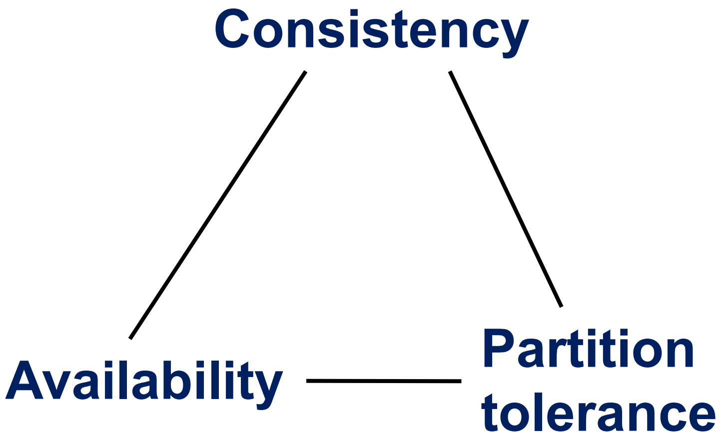
Availability: the distributed system is always available (and returns a value) as long a single node is running.

CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client



Available Design

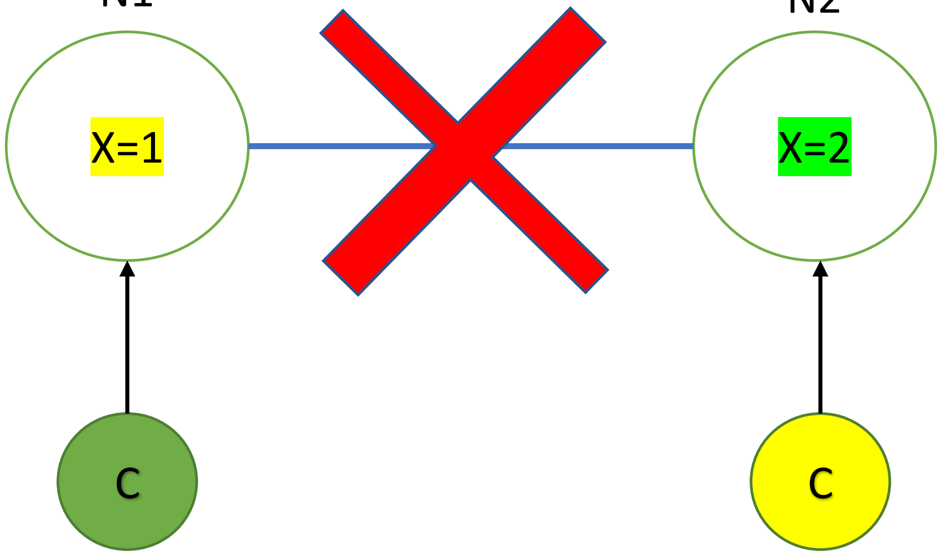


N1: “N2 is not reachable, but read/modify the value of X”

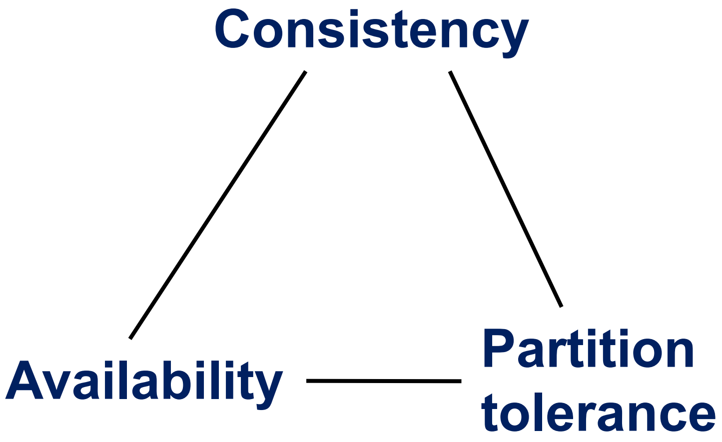
CAP Theorem

- CAP stands for Consistency, Availability, Partition tolerance.
  - **Consistency**: all nodes see same data at any time, or returns latest written value by any client

Scenario-3



Available Design

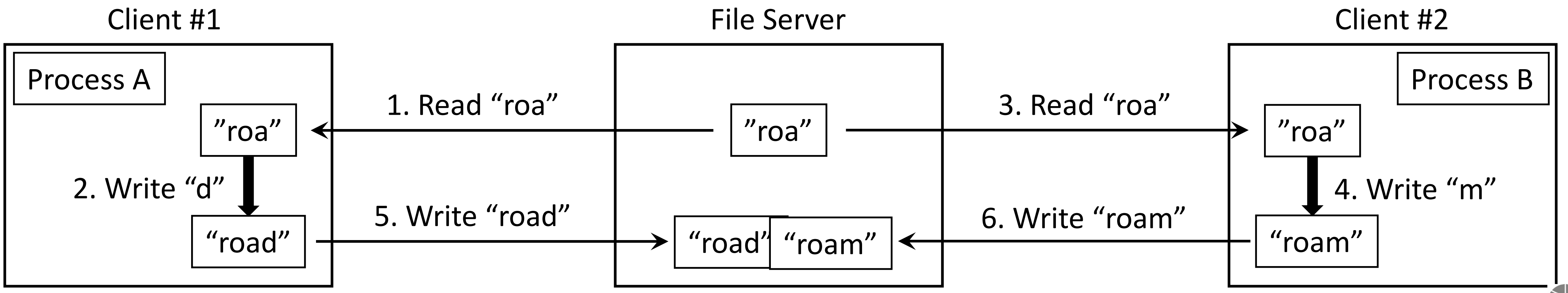


Distributed File Sharing Semantics

- Session Semantics
- Transaction Semantics

Distributed File Sharing Semantics

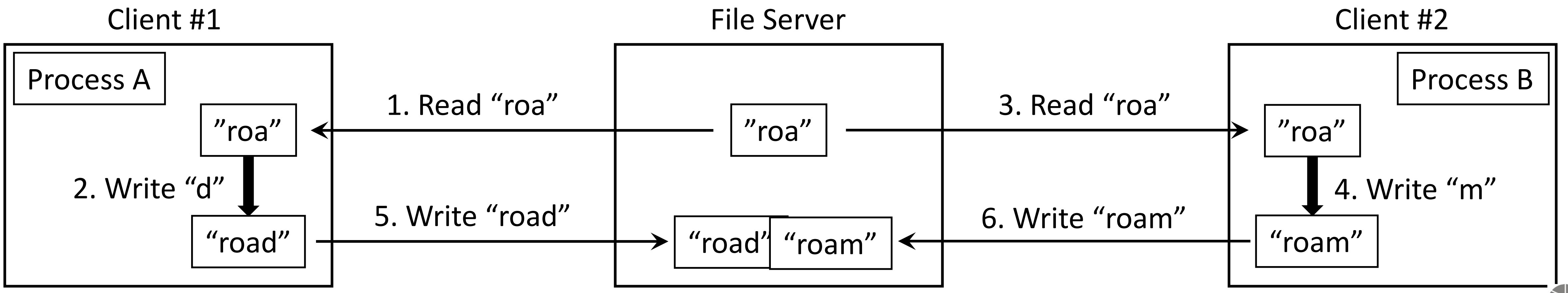
- **Session Semantics**
  - File **changes** are only **visible to the client modifying** it (e.g., in its local cache).
  - **Last process** to **modify** (i.e., close) the file **wins**
  - Simple and efficient (but not transactional)



Distributed File Sharing Semantics

- **Session Semantics**
  - File **changes** are only **visible to the client modifying** it (e.g., in its local cache).
  - **Last process** to **modify** (i.e., close) the file **wins**
  - Simple and efficient (but not transactional)

In a distributed system with caching at the client, obsolete values may be returned!



Distributed File Sharing Semantics

- **Transaction Semantics**

Distributed File Sharing Semantics

- **Transaction Semantics**

<u>Transaction-1:</u>

Debit(100 rs, A)
Credit(100 rs, B)

Distributed File Sharing Semantics

- **Transaction Semantics**

  Fully **ACID-compatible** file accesses and modifications.

  - **Atomicity**: Either all modifications inside a transaction succeed, or they all fail.
  - **Consistency**: A transaction transforms the data(base) from one consistent state to another.
  - **Isolation**: Each transaction is executed as if it were the only process accessing the data. Changes occurring in a transaction will not be visible to any other transaction until the transaction is "committed" – intermediate results are not updated!
  - **Durability**: When a transaction finishes ("commits"), the results of the transaction are persistently stored and made available to other processes.

Distributed File Sharing Semantics

- **Transaction Semantics**

  Fully **ACID-compatible** file accesses and modifications.

  - **Atomicity**: Either all modifications inside a transaction succeed, or they all fail.
  - **Consistency**: A transaction transforms the data(base) from one consistent state to another.
  - **Isolation**: Each transaction is executed as if it were the only process accessing the data. Changes occurring in a transaction will not be visible to any other transaction until the transaction is "committed" – intermediate results are not updated!
  - **Durability**: When a transaction finishes ("commits"), the results of the transaction are persistently stored and made available to other processes.

T1

$A \xrightarrow{100} B$

$A \xrightarrow{100} C$

T2

If balance of B > 50

$B \xrightarrow{50} C$

Distributed File Sharing Semantics

- **Transaction Semantics**

  Fully **ACID-compatible** file accesses and modifications.

  - **Atomicity**: Either all modifications inside a transaction succeed, or they all fail.
  - **Consistency**: A transaction transforms the data(base) from one consistent state to another.
  - **Isolation**: Each transaction is executed as if it were the only process accessing the data. Changes occurring in a transaction will not be visible to any other transaction until the transaction is "committed" – intermediate results are not updated!
  - **Durability**: When a transaction finishes ("commits"), the results of the transaction are persistently stored and made available to other processes.

T1

$A \xrightarrow{100} B$

$A \xrightarrow{100} C$

T2

If balance of B > 50

$B \xrightarrow{50} C$

Distributed File Sharing Semantics

- **Transaction Semantics**

  Fully ACID-compatible file accesses and modifications.

  - **Transaction semantics** is the **most demanding semantics** considerable for a distributed file system. It is also relevant in databases, transactional main-memory, etc.
  - Transactions require a **single master** (or **multiple synchronized masters**) that coordinate(s) the client accesses to all resources.
  - All of the **common distributed file system architectures** basically ***give up*** on implementing a clean transaction semantics!

Transaction Semantics in (Centralized) Database Systems: Locking

- Multiple clients may read from and write to the ***same table concurrently***.

Client1: |$\rightarrow$ Read table t | |$\rightarrow$ Write table t |

Client2: |$\rightarrow$ Read table t | |$\rightarrow$ Write table t |

Client3: |$\rightarrow$ Read table t | $\rightarrow$?

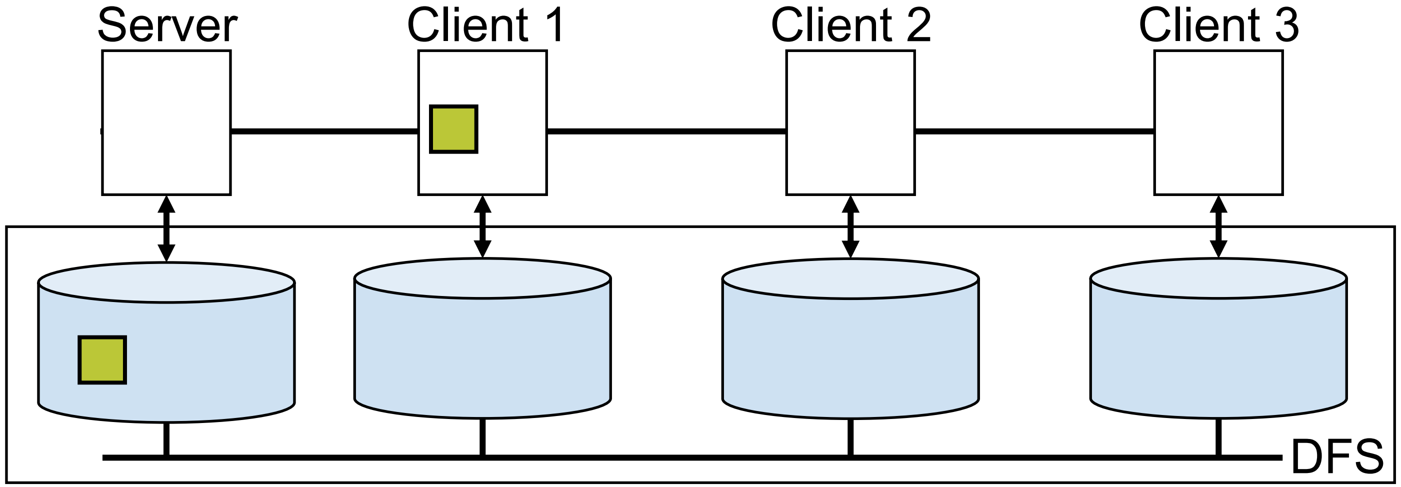
- **Transaction**: Coherent sequence of read/write/update operations of each client. (here: Client1, Client2, Client3)
- **Locking** allows for the **safe execution of transactions**.
  - Lock resource (i.e., files or tables) at first client request, keep exclusive-access lock until client finishes.
  - At the **cost of concurrency**!

Stateful or Stateless?

- **Stateful**: server maintains **client-specific state**
  - Shorter requests & better performance in processing requests.
  - **Cache coherence** is possible since the master knows who is accessing what.
  - Global **file locking** (and transactions) possible.
- **Stateless**: server maintains **no information about client accesses**
  - Each request identifies file and offsets.
  - **Server can crash and recover**: no state to lose
  - **Client can crash and recover** (as usual).
  - No server space used for state information.
  - But what if a file is deleted on master while client is working on it?
  - **Global file locking (and transactions) not** possible.

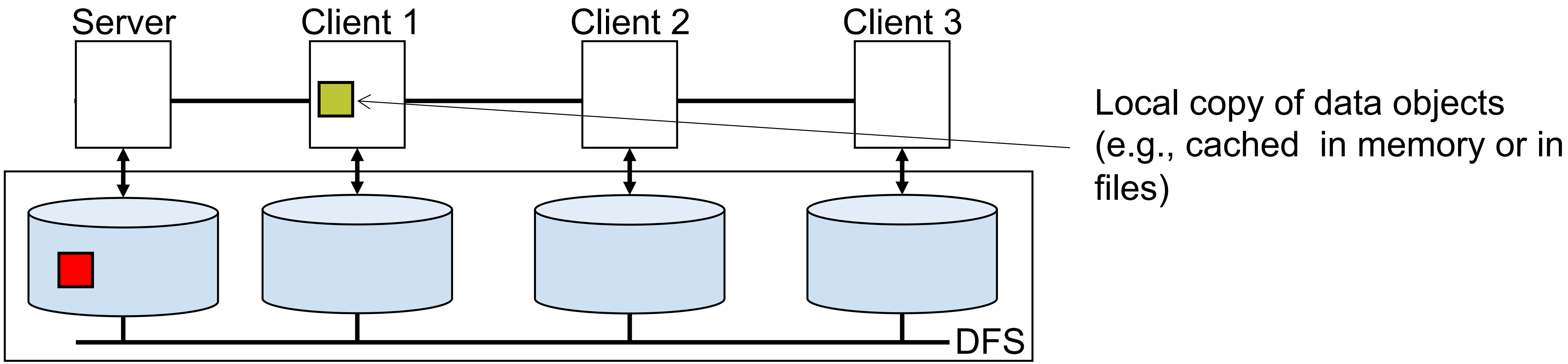
Caching

- Reduce latency to **improve performance for repeated file accesses**.
- Possible **cache locations**: servers's disk, server's memory, client's disk, client's memory.



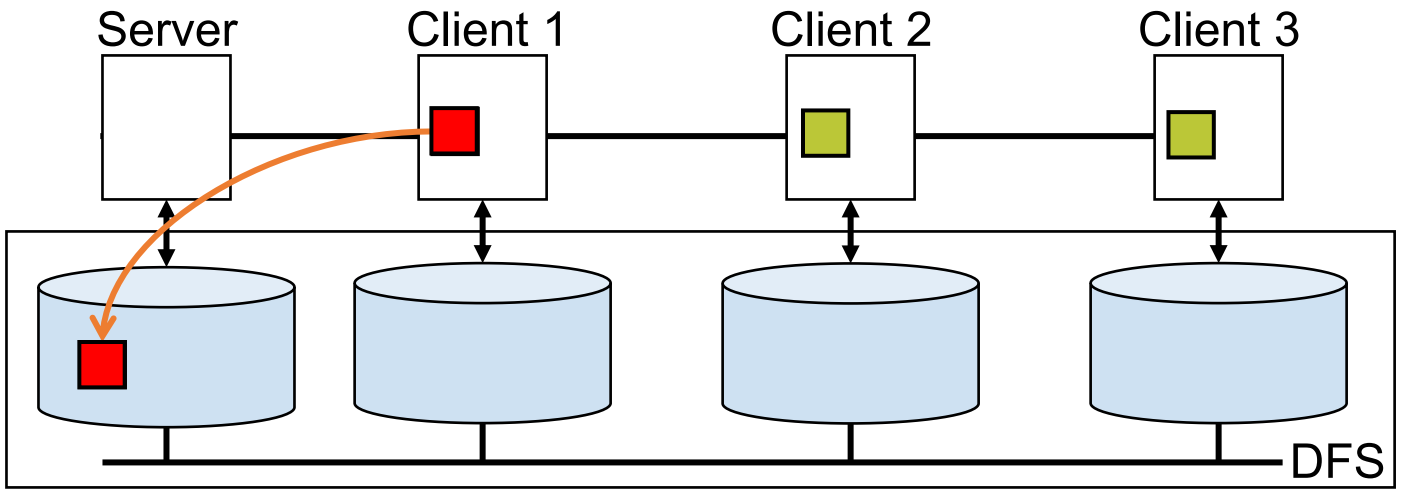
Caching

- Reduce latency to **improve performance for repeated file accesses**.
- Possible **cache locations**: servers's disk, server's memory, client's disk, client's memory.
  - The latter two may create **cache-consistency problems**. (unfortunate, since they are the **best performing options**)
  - Cache validation: time-stamps, checksums, etc.



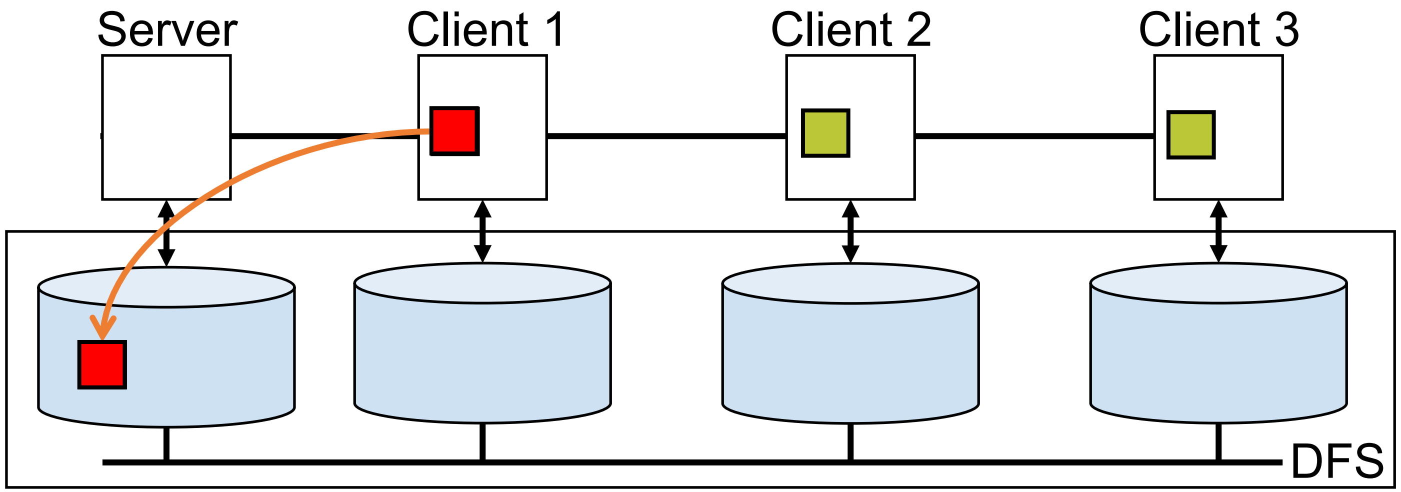
Option 1: Write-through Caching

- Every change is immediately propagated to the master-copy.



Option 1: Write-through Caching

- Every change is immediately propagated to the master-copy.
  - Improves read performance.
  - Does **not** improve on the write performance (in fact, same as before).
  - If multiple clients are writing on the same file, server maintains the state and set signals to invalidate the obsolete copies.



Option 2: Write-behind Caching

- **Delay the writes** and batch them.
  - Remote files are updated periodically.
  - One **bulk write is more efficient** than lots of little writes.
  - Problem: **ambiguous semantics** in case of conflicts!

Option 2: Write-behind Caching

- **Delay the writes** and batch them.
  - Remote files are updated periodically.
  - One **bulk write is more efficient** than lots of little writes.
  - Problem: **ambiguous semantics** in case of conflicts!

    Example:

    Client 1 reads file f → Client 2 reads and writes file f

    → Client 1 writes file f.
- Unfortunately, **not a transaction semantics nor a session semantic**s under write-behind caching!

Client 1

f1 f1’ f1’’

Client 2

Client 1

f1 f1’’ f1’

Client 2 f1’

Caching for sometime.

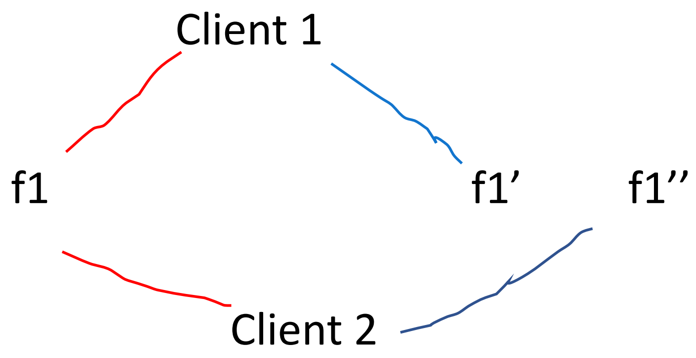
Option 2: Write-behind Caching

- **Delay the writes** and batch them.
  - Remote files are updated periodically.
  - One **bulk write is more efficient** than lots of little writes.
  - Problem: **ambiguous semantics** in case of conflicts!

    Example:

    Client 1 reads file f → Client 2 reads and writes file f

    → Client 1 writes file f.
- Unfortunately, **not a transaction semantics nor a session semantic**s under write-behind caching!



Client 1

f1 f1’’ f1’

Client 2 f1’

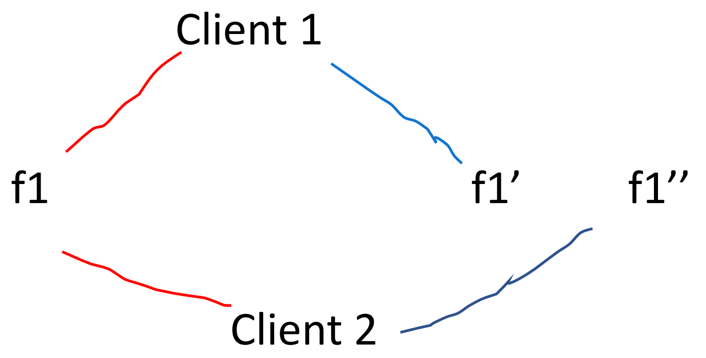
Option 2: Write-behind Caching

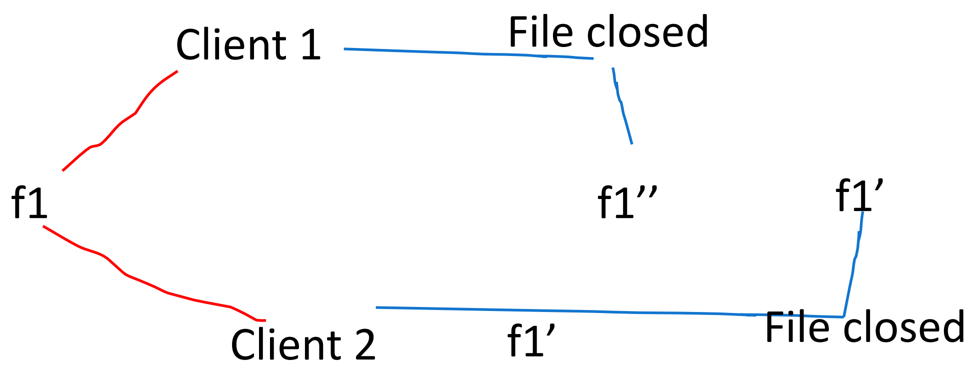
- **Delay the writes** and batch them.
  - Remote files are updated periodically.
  - One **bulk write is more efficient** than lots of little writes.
  - Problem: **ambiguous semantics** in case of conflicts!

    Example:

    Client 1 reads file f → Client 2 reads and writes file f

    → Client 1 writes file f.
- Unfortunately, **not a transaction semantics nor a session semantic**s under write-behind caching!





Option 3: Write-on-Close

- Write **file back** to server only once it has been **closed by the client**.
- Even better: wait another X seconds to see if the file will also be deleted.
- Implements **session semantics**
- But again not a transaction semantics!

Also Related: Read-ahead Caching

- Be **proactive and "prefetch" data** that is likely going to be relevant for an application in the next step.
- **Request chunks of file** (or the entire file) before it is needed and put it into the client cache.
- **Minimize latency** at the time when it actually is needed.

Caching Summary

Centralized Control	May support transactions, but poor scalability.
Write-through Caching	Improves read performance but does not improve write traffic/performance.
Write-behind Caching	Better read and write performance, but possibly ambiguous semantics. No transactions. No sessions.
Write-on Close	Matches session semantics, but not transaction semantics.

Further Reading: https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#COHDG5177

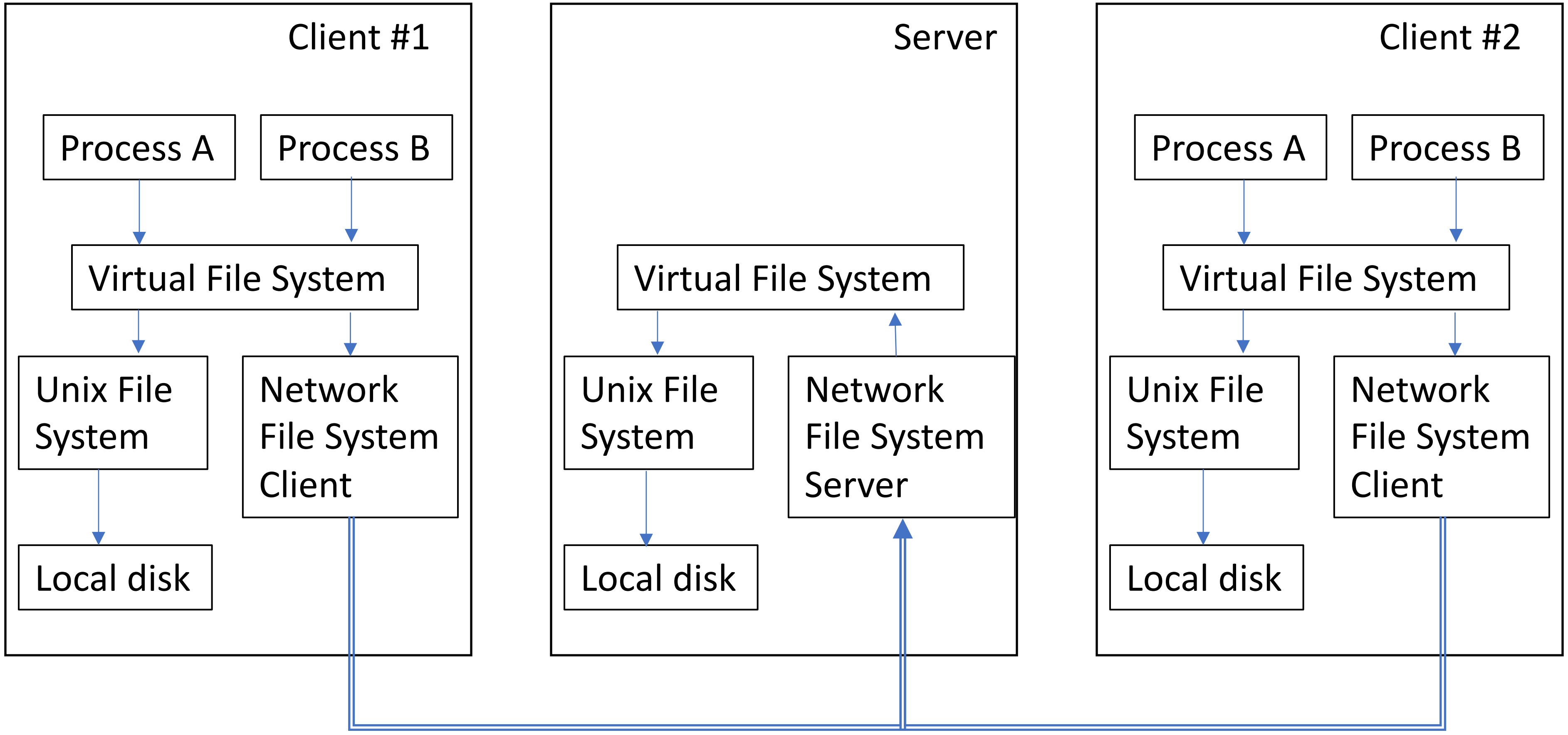
Distributed File System History

- **Network File System** (NFS)
  - Sun Microsystems (now Oracle), 1984-today
- **Andrew File System** (AFS)
  - Carnegie Mellon University & IBM, 1986 V1, 1999 V2
- **Coda File System** (not in this lecture!)
  - Carnegie Mellon University, 2003-2009
- **Google File System** (GFS)
  - Google Inc., since 2003
- **Hadoop File System** (HDFS)
  - Shipped with Apache Hadoop since 2009, major split into HDFS and YARN in late 2012 (Hadoop 2.0), currently split into HDFS, MapReduce and YARN packages (Hadoop 3.1)

NFS: Sun Microsystems, 1984

- Arguably the **first distributed file system**
- Based on **simple client-server model**
- The model underlying NFS is **Remote Access model**
- **All machines** are located within **same physical network**
- **Stateless**
- **No transaction semantics**. **Not even session semantics** (uses write-behind caching!).
- Optional locking mechanism for files.

NFS: Sun Microsystems, 1984



NFS Design Goals

- **Heterogeneity** a first-class citizen
  - Hardware and operating system are not an issue.
- **Access transparency**
  - Remotely stored files accessed as local files through standard system calls (API).
  - Separation of physical and logical storage.
- **Crash recovery**
  - No (shared) state whatsoever.
- **High performance**
  - Network I/O equal to (or faster than) disk I/O
  - Caching: read-ahead, write-behind

Further Reading: https://www.halolinux.us/red-hat-networking/nfs-disadvantages.html

AFS: Carnegie Mellon University, 1986

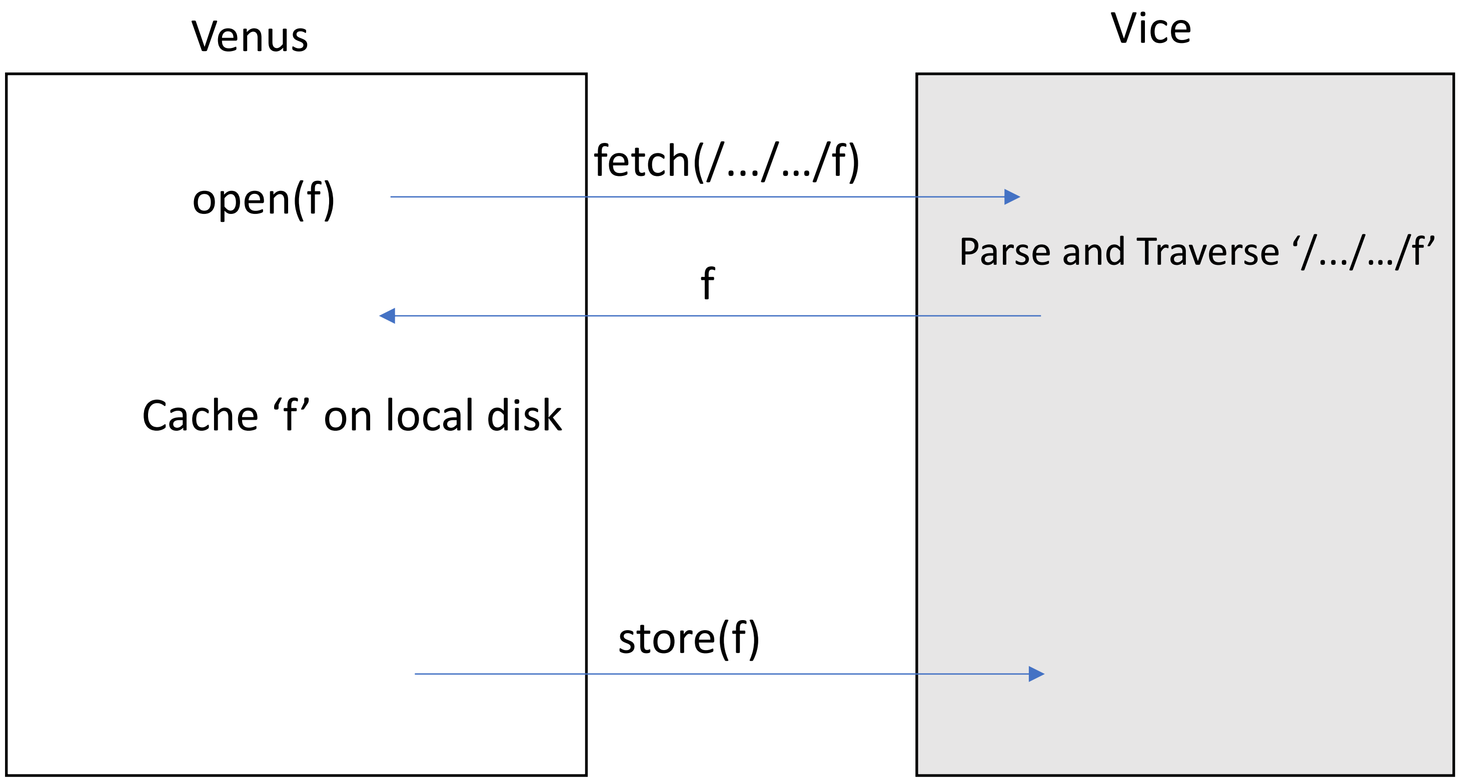
- Named after Andrew Carnegie and Andrew Mellow, the “C” and “M” in CMU
- Acquired by IBM, and subsequently open-sourced
- Still used today in some clusters (especially University Clusters)

AFS: Carnegie Mellon University, 1986

- Named after Andrew Carnegie and Andrew Mellow, the “C” and “M” in CMU
- Acquired by IBM, and subsequently open-sourced
- Still used today in some clusters (especially University Clusters)
- Two unusual design principles: **whole file sharing** & **whole file caching**. -- not in blocks!
- Based on the **assumption** that:
  - Most **files** are **small**.
  - Most files are accessed by **one user at a time**.
  - Clients could cache as large as 100MB is supportable
  - **Reads** are **more common** than **writes**.

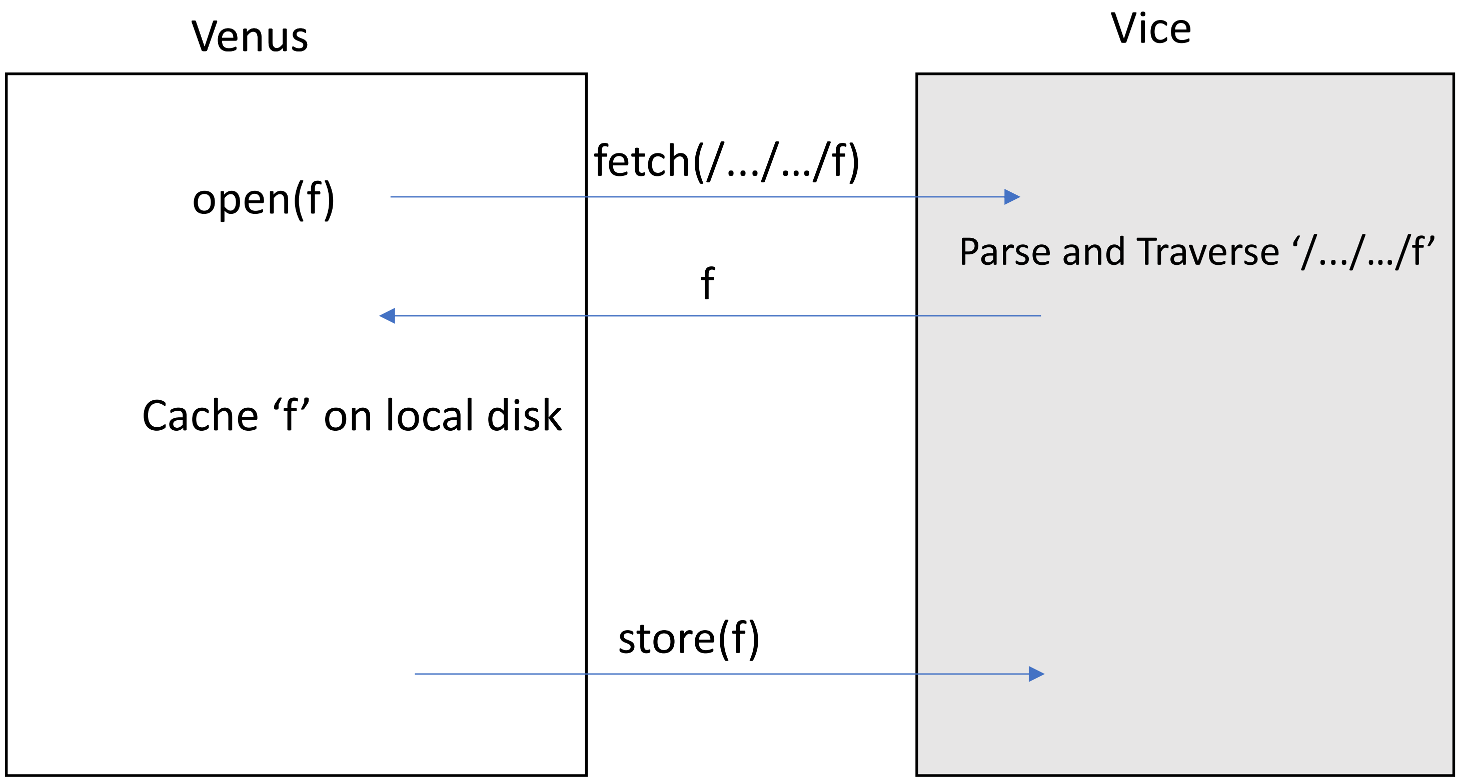
AFS: Carnegie Mellon University, 1986

- **Client system** = **Venus** service
- **Server system** = **Vice** service
- Reads and Writes are **optimistic**
  - **done on local copy** of file at Venus
  - On **closing** file, changes are **propagated to Vice**
- When a client (Venus) **opens a file**, Vice:
  - Sends the **entire file**
  - Gives client a ***callback promise***



AFS: Carnegie Mellon University, 1986

- **Client system** = **Venus** service
- **Server system** = **Vice** service
- Reads and Writes are **optimistic**
  - **done on local copy** of file at Venus
  - On **closing** file, changes are **propagated to Vice**
- When a client (Venus) **opens a file**, Vice:
  - Sends the **entire file**
  - Gives client a ***callback promise***

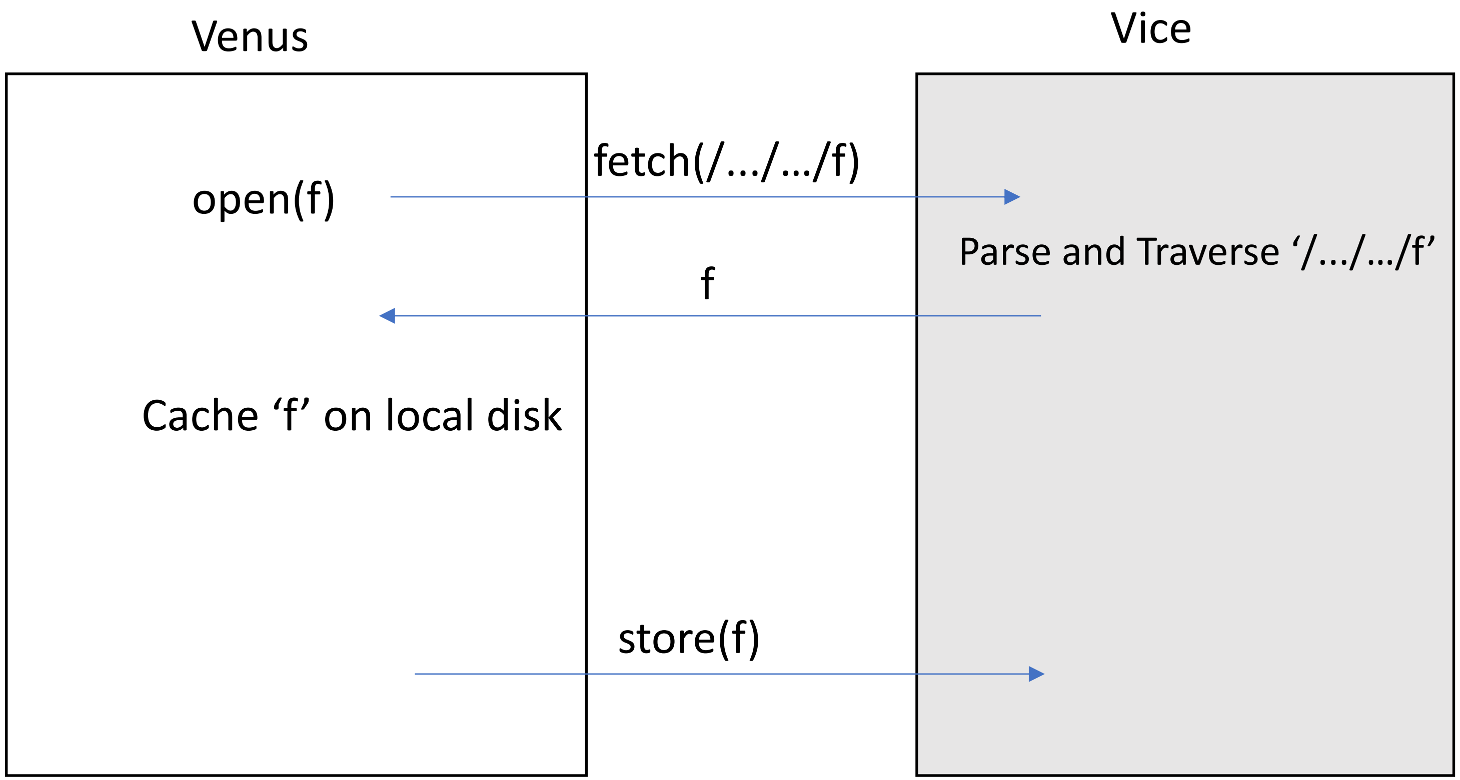


- High cost associated with path traversal
- Fetch/store request transmit the entire pathname, root to the leaf directory
- Server has to perform complete path traversal

"the path traversal costs were too high"

AFS: Carnegie Mellon University, 1986

- **Client system** = **Venus** service
- **Server system** = **Vice** service
- Reads and Writes are **optimistic**
    - **done on local copy** of file at Venus
    - On **closing** file, changes are **propagated to Vice**
- When a client (Venus) **opens a file**, Vice:
    - Sends the **entire file**
    - Gives client a ***callback promise***

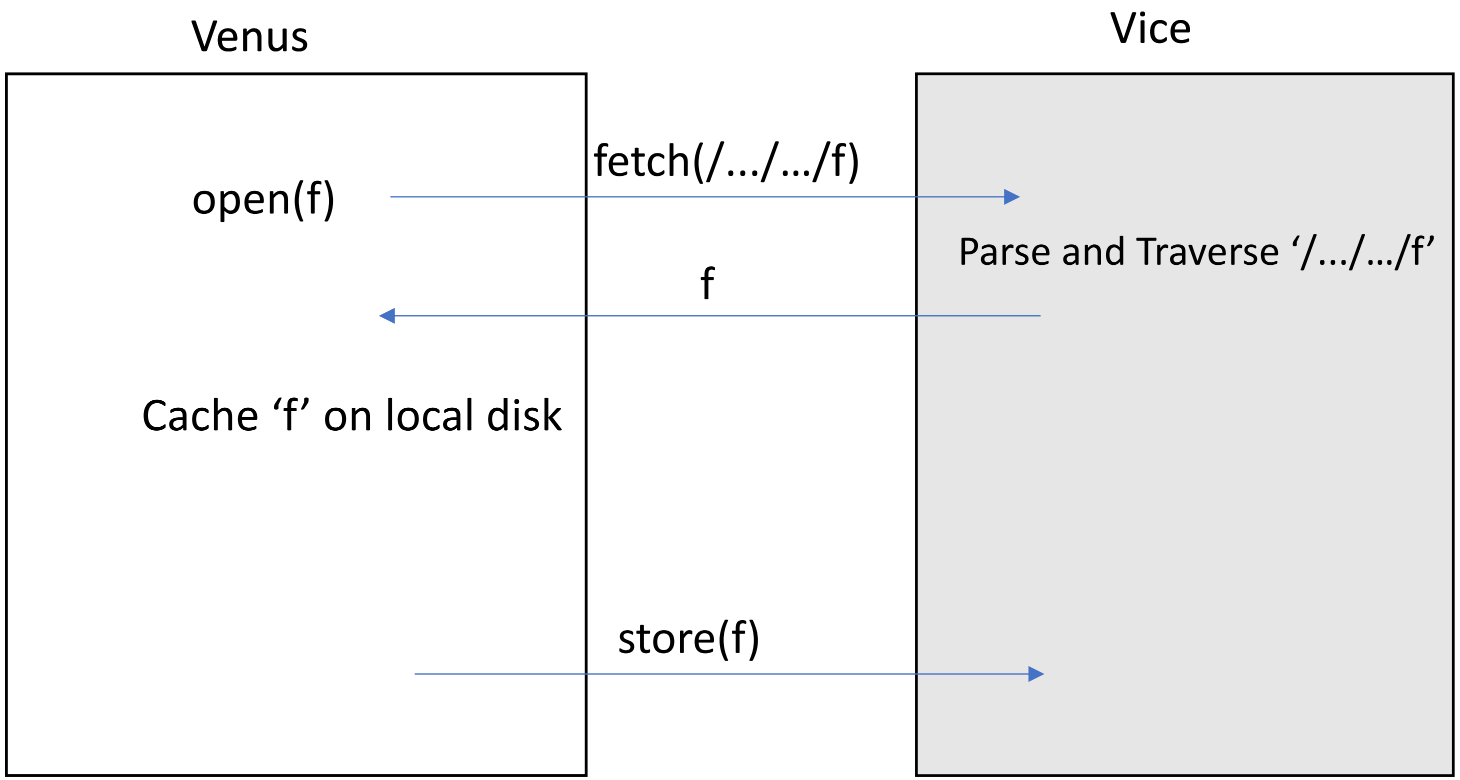


- AFS V2 introduced this notion of a file identifier

" clients issue too many testAuth protocol message"

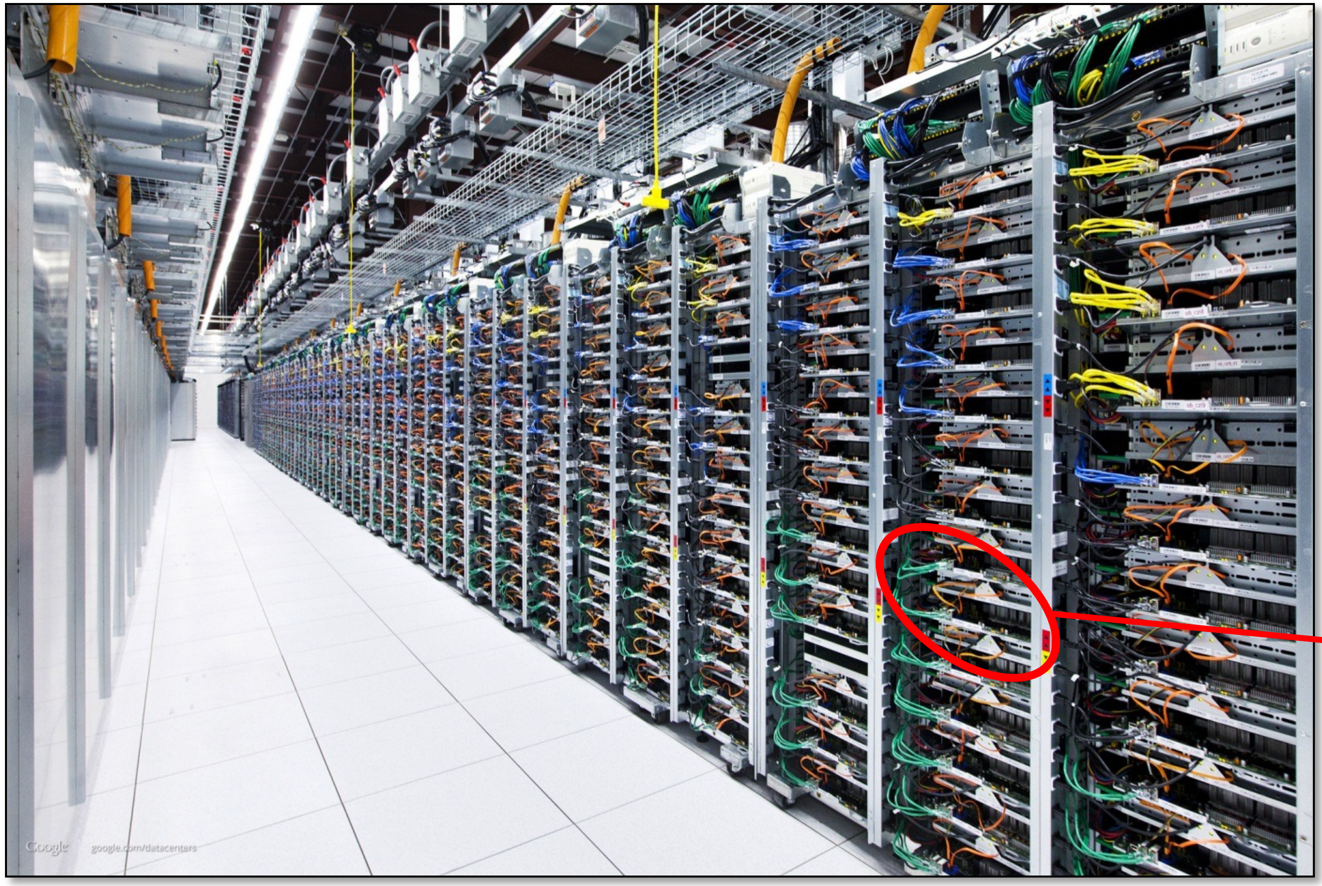
AFS: Carnegie Mellon University, 1986

- **Client system** = **Venus** service
- **Server system** = **Vice** service
- Reads and Writes are **optimistic**
  - **done on local copy** of file at Venus
  - On **closing** file, changes are **propagated to Vice**
- When a client (Venus) **opens a file**, Vice:
  - Sends the **entire file**
  - Gives client a ***callback promise***



- **Callback promise**
  - Promise that **if another client modifies and then closes a file**, a callback promise will be sent from Vice to Venus
  - Callback state at Venus is only **binary**: valid or cancelled

What are we missing so far?

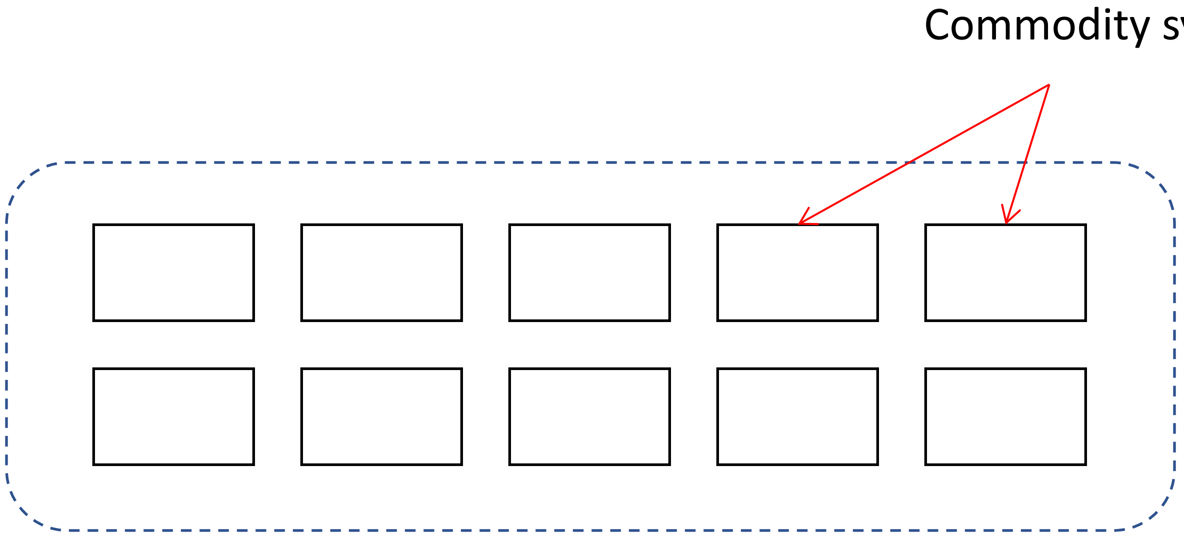


What if this guy fails?

Google File System (GFS) & Hadoop File System (HDFS)

Google File System (GFS)

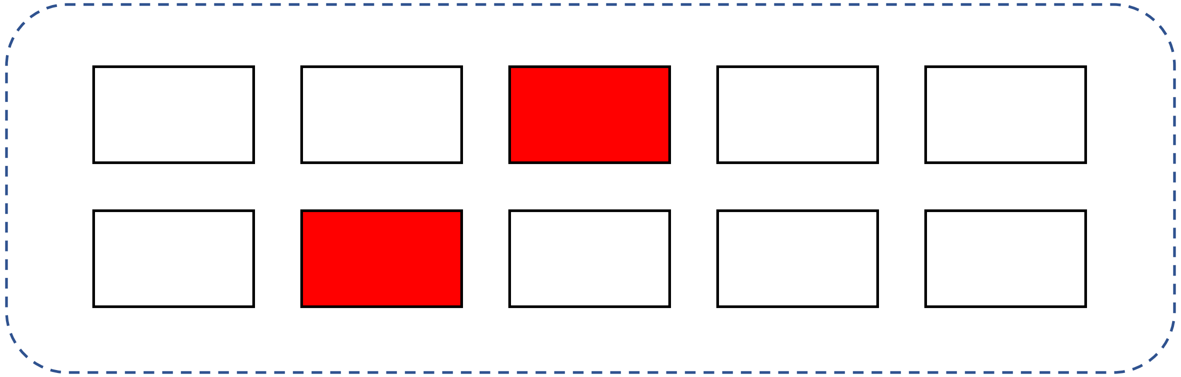
- the Google File System (GFS) was designed and implemented to meet the Google's data processing needs
- GFS is a Distributed File Storage utilizing **commodity systems**
- GFS is used as a **basis to design Hadoop**
  - Corresponding to GFS, the component in hadoop is called **HDFS**



Cluster consisting of 100s of commodity systems

Google File System (GFS)

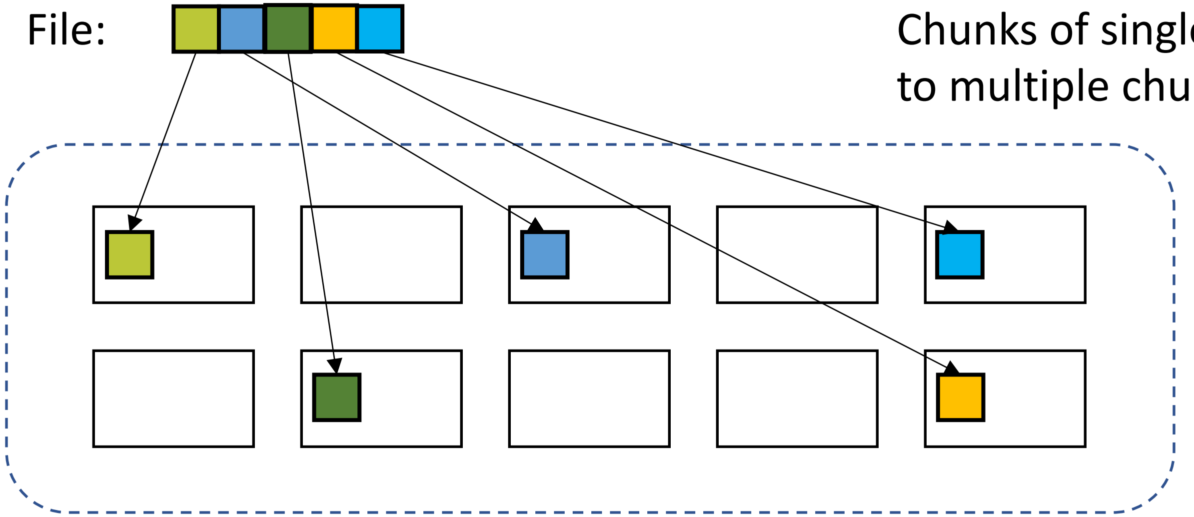
- **Design considerations-1:**
  - **Use commodity hardware** than buying expensive servers
  - **Scale-out with right software**
  - But: **Commodity hardware fail all the time**: disk failure, network issues or even OS bugs
  - Design challenge: perform **reliably in a fault-tolerant manner** in the phase of constant failures



Commodity hardware often fail!

Google File System (GFS)

- **Design consideration-2:**
  - Optimized to **store and read large files**: 100MB to multi-GB files
  - Files are split into **chunks**
    - Each chunk is of 64MB; identified by 64 bit ID; Stored in Chunk-Servers

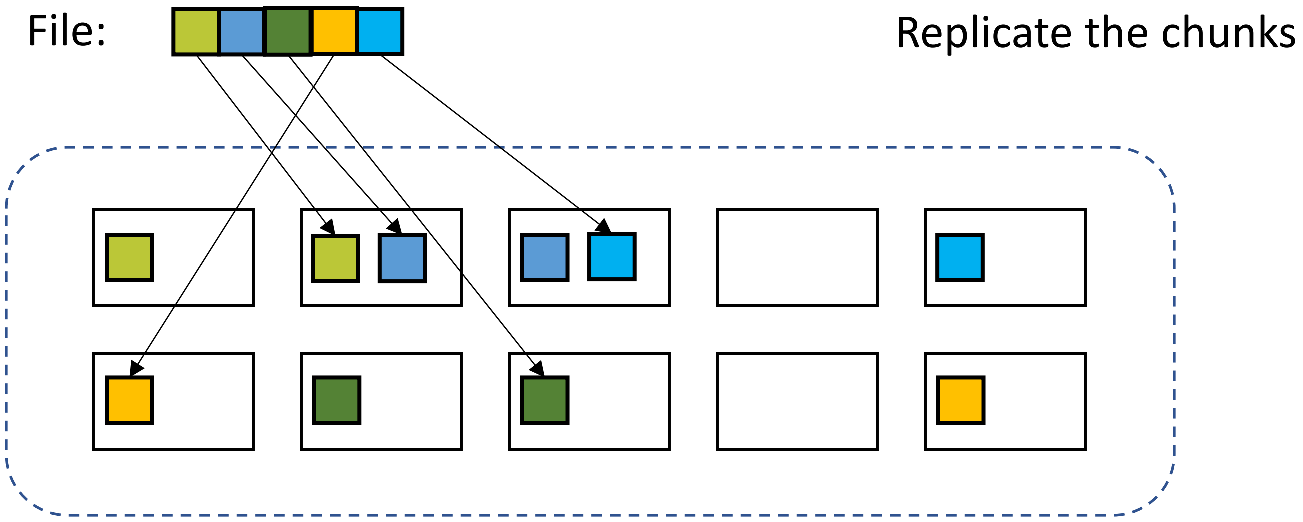


Chunks of single file are distributed to multiple chunk-servers

Each commodity hardware acts as a Chunk-Server

Google File System (GFS)

- **Design consideration-2**:
  - Optimized to **store and read large files**: 100MB to multi-GB files
  - Files are split into **chunks**
    - Each chunk is of 64MB; identified by 64 bit ID; Stored in Chunk-Servers
- Replicate the chunks since chunk-servers fail at anytime

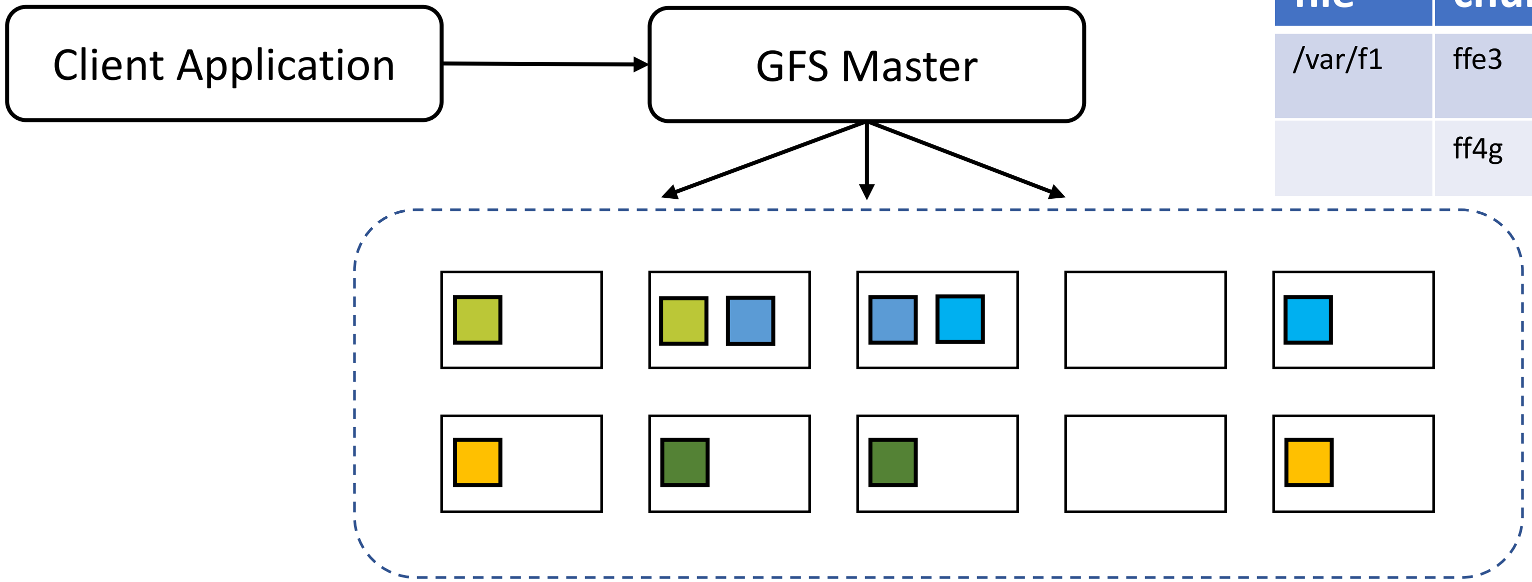


Each commodity hardware acts as a Chunk-Server

Google File System (GFS)

- By default 3 replicas of chunks
- How do we know **which chunk of a file resides in which chunk-server**?
  - **GFS Master**: stores the entire **metadata of the cluster**
  - Metadata: **File names-> (chunk-id, location)***; access control details

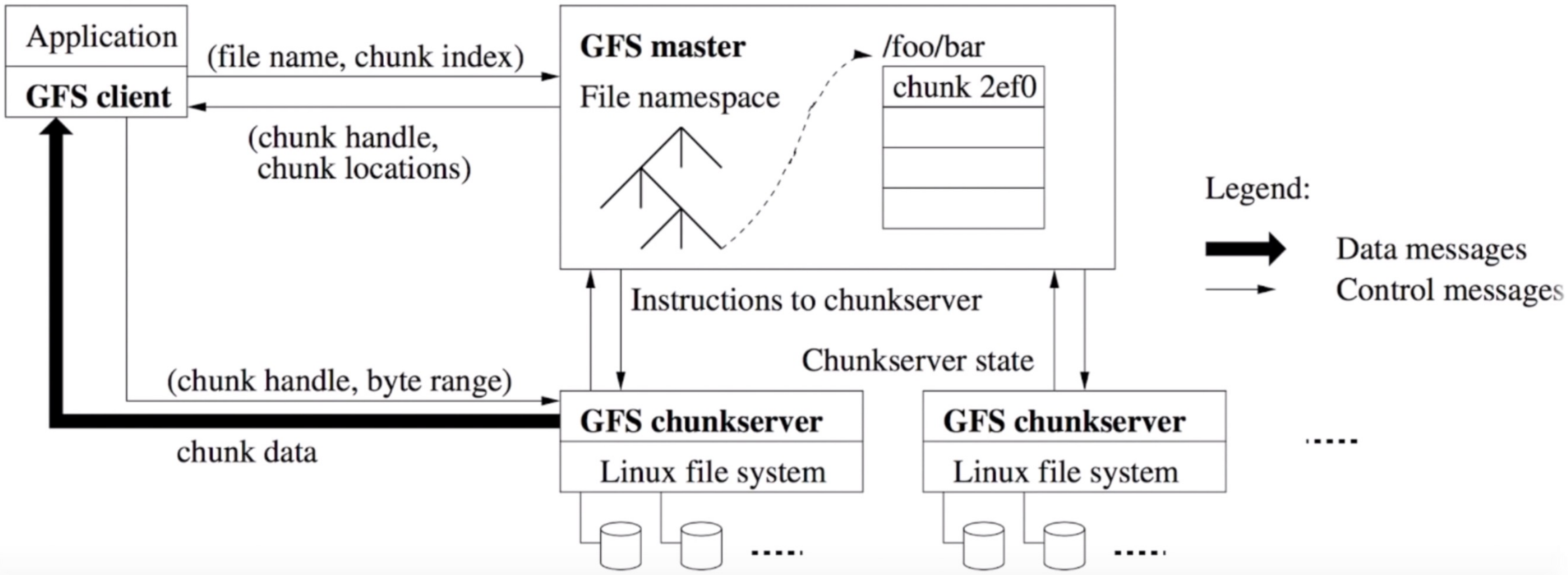
file	chunks	location
/var/f1	ffe3	Server1 (replica 1) Server2 (replica 2)
	ff4g	Server4 (replica 1)



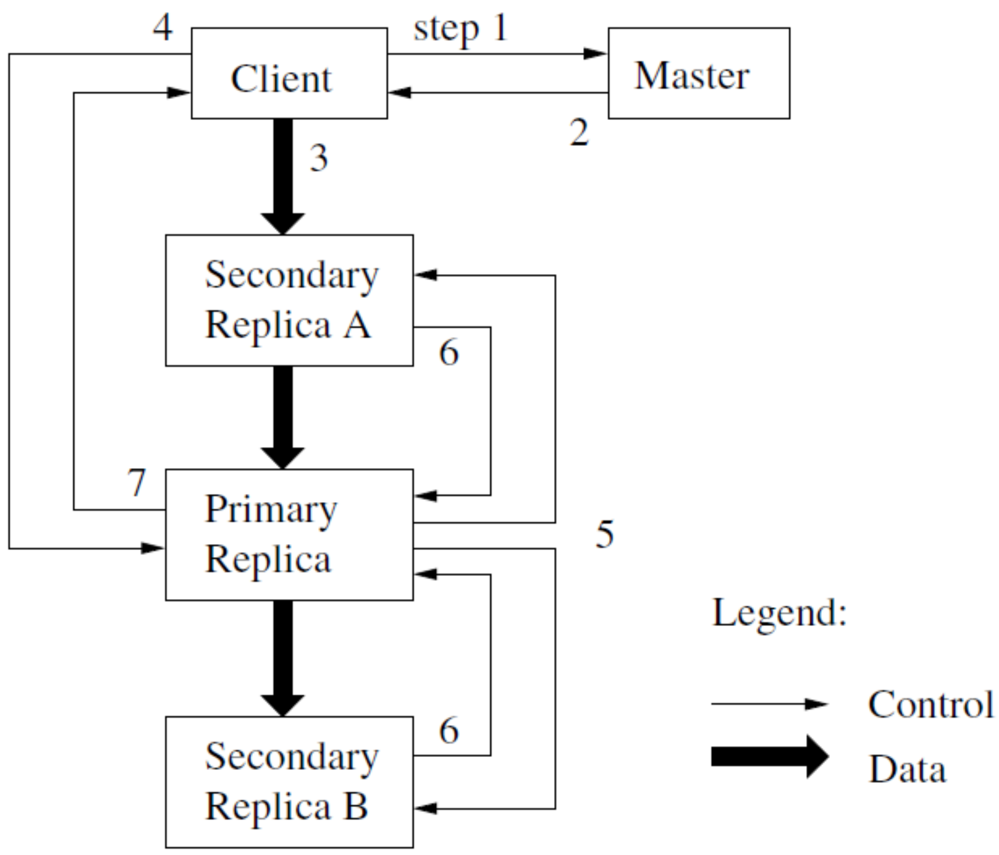
Each commodity hardware acts as a Chunk-Server

Google File System (GFS)

- **Reads**
  - GFS client request to access the file using **filename** and **chunk-index**
  - GFS master gives the **id of the chunk** and all the **ip-addresses** of chunk-servers having its replica
  - Client uses the ip-address and **read the chunk directly** from the chunk-server – Client does not read the data via GFS master!
  - Client would **caches** the chunk handle and chunk location



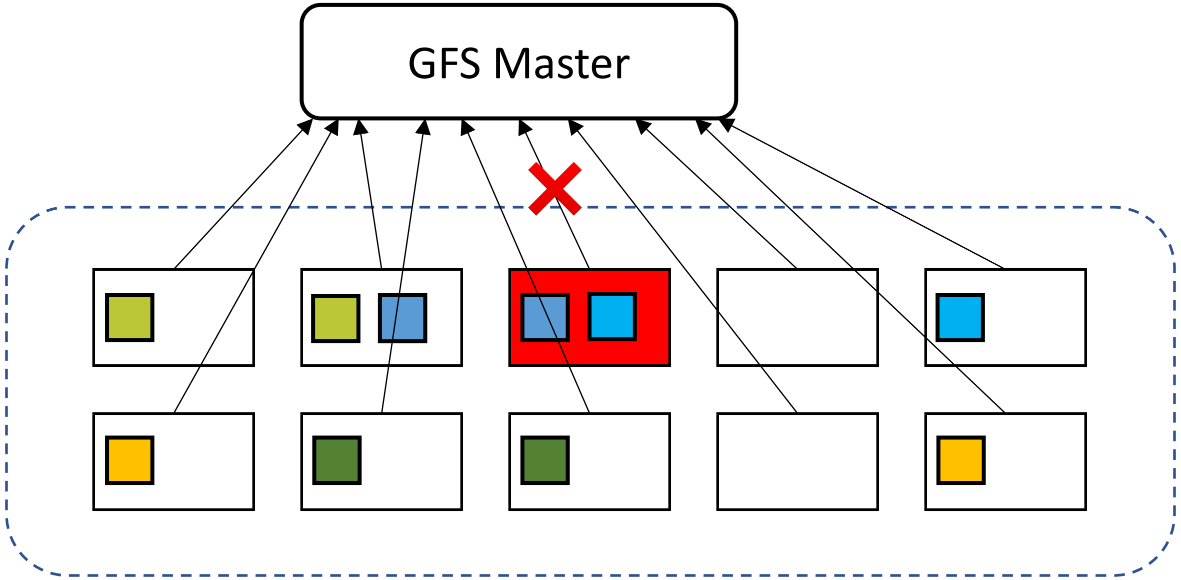
Google File System (GFS)



- **Writes**
  - GFS client **asks** the master for the **location to write**
  - GFS master gives 3 **free chunk locations** (in 3 diff. chunk-servers): Replica A, B & C
  - Client **writes** data to **closest replica**
  - Subsequently the data is **propagated** to the **other 2 replicas**
  - Client sends **request** to the **primary replica to commit** the data to the disk
  - **Primary replica coordinate** with the other replica and **send confirmation** to the client

GFS Fault-Tolerance Model

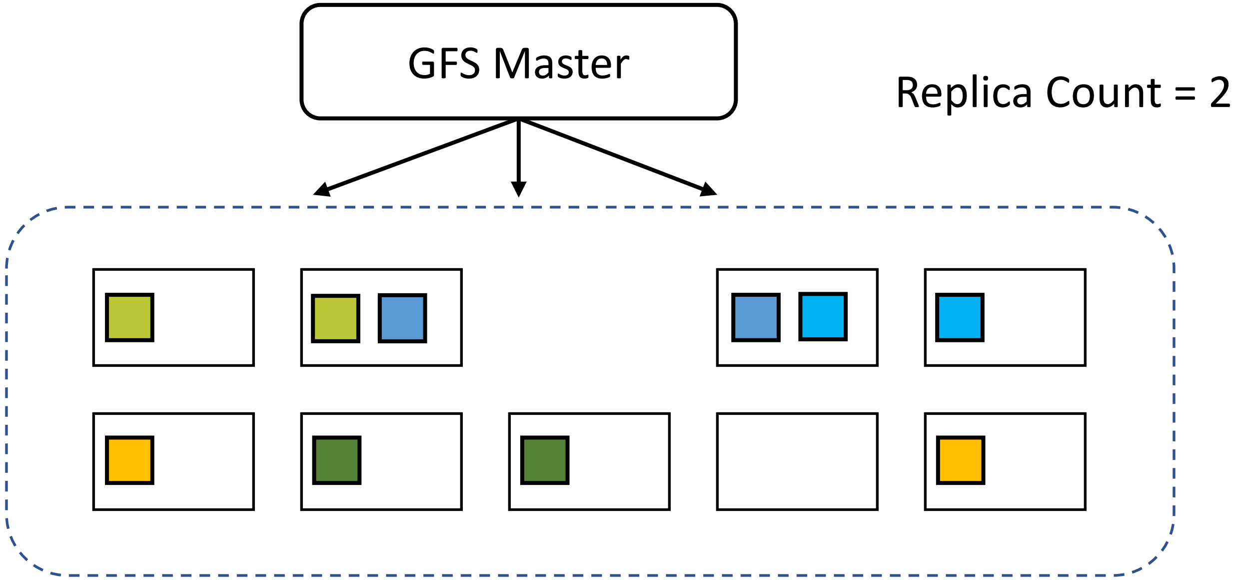
- **Chunk-servers** send regular "**heartbeat**" messages to master to inform that they are alive
- If chunk-server is down, master ensures **all chunks that were on it are copied on other servers**
- Ensures **replica** counts **remains same**



Each commodity hardware acts as a Chunk-Server

GFS Fault-Tolerance Model

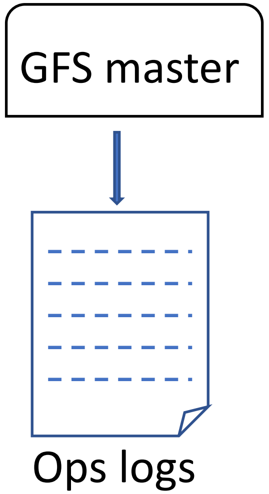
- **Chunk-servers** send regular "**heartbeat**" messages to master to inform that they are alive
- If chunk-server is down, master ensures **all chunks that were on it are copied on other servers**
- Ensures **replica** counts **remains same**



Each commodity hardware acts as a Chunk-Server

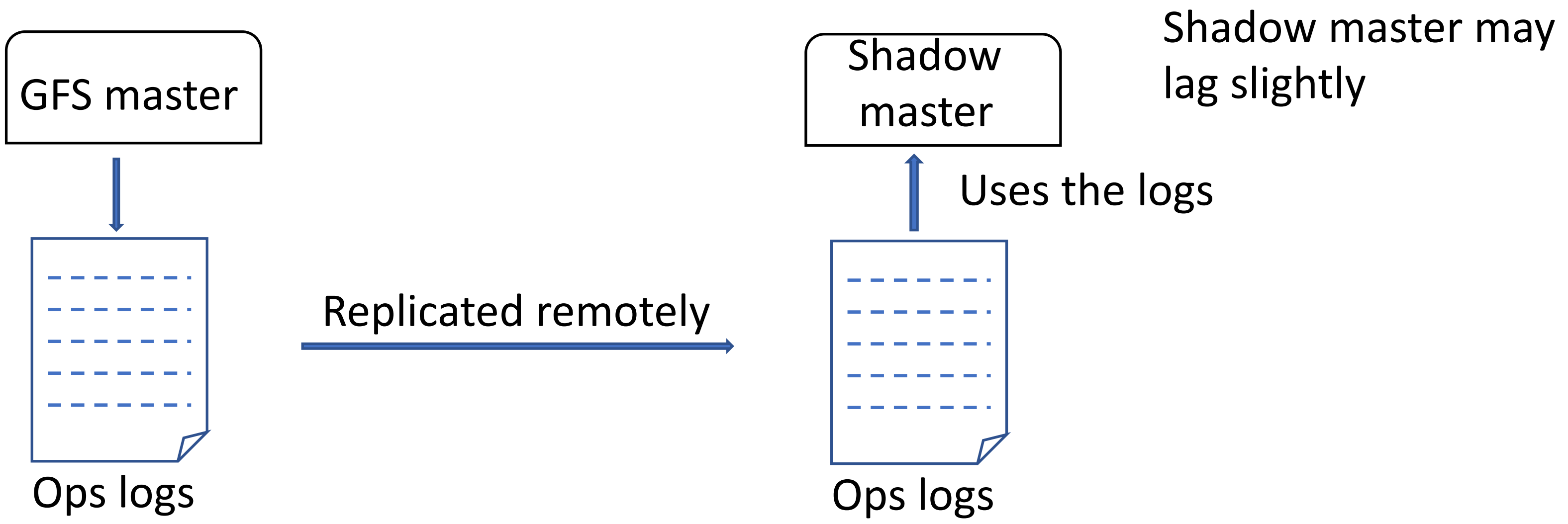
GFS Fault-Tolerance Model

- **What if the single master fails?**
  - Create a **new master** and read the **operations-log**
  - **Operation-log:** append only log
    - Stores all the file operations, timestamp, user details
    - Directly written to the disk and replicated remotely
  - New master **generates the entire file system namespace and chunk-ids**, from the operations-log
  - **Shadow master**



GFS Fault-Tolerance Model

- **What if the single master fails?**
  - Create a **new master** and read the **operations-log**
  - **Operation-log:** append only log
    - Stores all the file operations, timestamp, user details
    - Directly written to the disk and replicated remotely
  - New master **generates the entire file system namespace and chunk-ids**, from the operations-log
  - **Shadow master**



Hadoop File System (HDFS)

- Part of Apache's **Hadoop framework** (which is essentially an open-source Java implementation of Google's MapReduce → see next chapter!).
- Follows the **GFS architecture**.
- Basis for many other open-source Apache toolkits:
  - Yahoo's PIG/PNuts (file-based data storage and scripting language)
  - Apache HIVE (distributed data warehouse)
  - Apache HBase (distributed database)
  - Apache Cassandra (fault-tolerant distributed database)
  - …

HDFS Design Goals

- Support for **very large files** (>1 TB), even Petabytes of overall data.

  (See "Scaling Hadoop to 4000 nodes at Yahoo!", Yahoo! 2008 blog entry)

- **Streaming data access** (write-once, read-many-times pattern)
- **Commodity hardware** (many distributed machines, frequent failures)

HDFS Design Goals

- Support for **very large files** (>1 TB), even Petabytes of overall data.

  (See "Scaling Hadoop to 4000 nodes at Yahoo!", Yahoo! 2008 blog entry)
- **Streaming data access** (write-once, read-many-times pattern)
- **Commodity hardware** (many distributed machines, frequent failures)
- But:
  - **No low-latency file access** (below tens of milliseconds)
    - HDFS is designed for delivering a high throughput of data
  - **Not too many small files** (millions of files are OK, but not billions!)
    - No of files in HDFS is limited to the amount of memory in the master node
  - **Single writer to a file at any time, append-mode only**.

    (in principle allows for session and even transaction semantics, but no concurrency for mixed read/write or write/write operations at all!)

HDFS Design Goals

- **Blocks** (same as "chunks")
  - **Large files are broken down into blocks of 64-128 MB**.

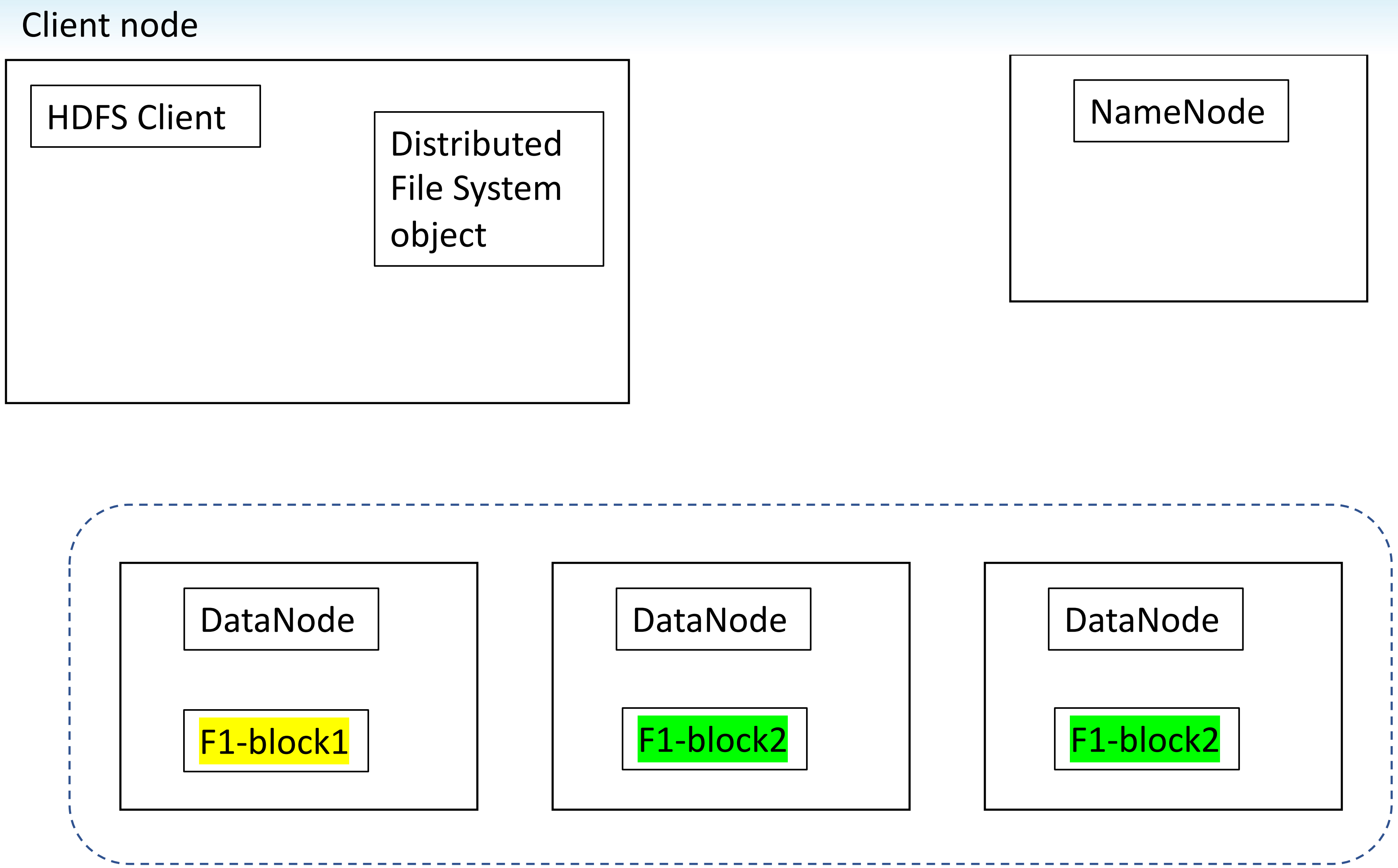
    (again, compare to 64 MB GFS blocks and 16-32 KB DBMS blocks)
  - **Minimize the amortized cost of seek operations**: large blocks guarantee consecutive layout of data on disk, disk seek times (~3-5ms) occur only across block boundaries.
- Namenode = GFS master
  - manages entire namespace of files and blocks.
- Datanodes = Chunk-server

HDFS Interfaces

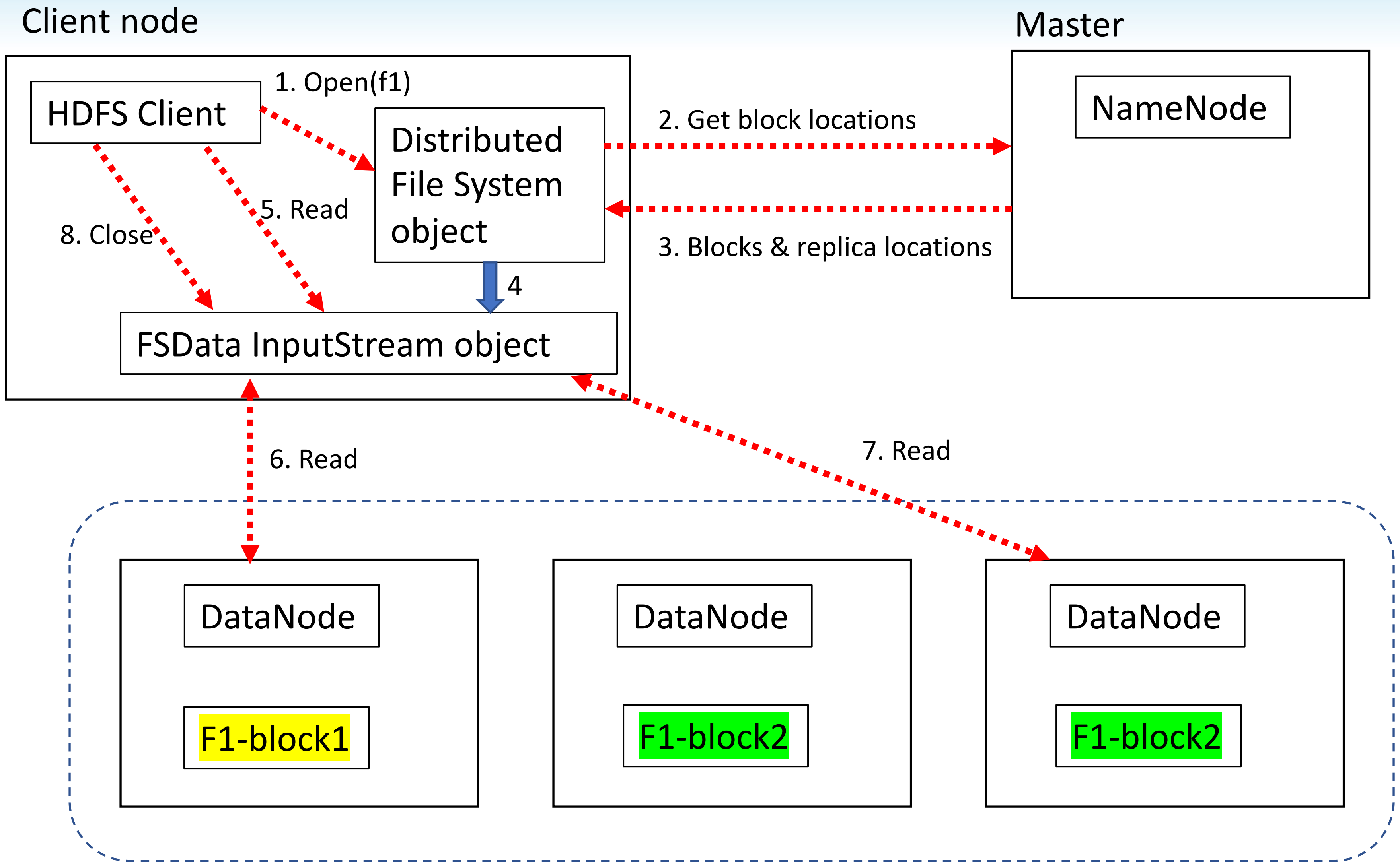
- Hadoop **provides many interfaces** to allow users to access and manipulate its file system.
  - Command-line (from a Linux/UNIX shell)
  - Java API
  - Thrift proxy (supporting C++, Perl, PHP, Python, Ruby)
  - C via a Java Native Interface (JNI)
  - WebDAV for mounting file systems via HTTP
  - HTTP (read-only), FTP (read&write)

  and many more…

HDFS Architecture



HDFS Architecture (Read)



HDFS Architecture (Write)