CAP

- Presented as Brewer's Conjecture in 2000

- Formalized and proved in **Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**, by Lynch and Gilbert (2002)

- **Consistency**, **availability** and **partition-tolerance** cannot be achieved all at the same time in a distributed system.

- Simply, in an asynchronous network that performs as expected, where messages may be lost (partition-tolerance), it is impossible to implement a service providing correct data (consistency) and eventually responding to every request (availability) under every pattern of message loss.

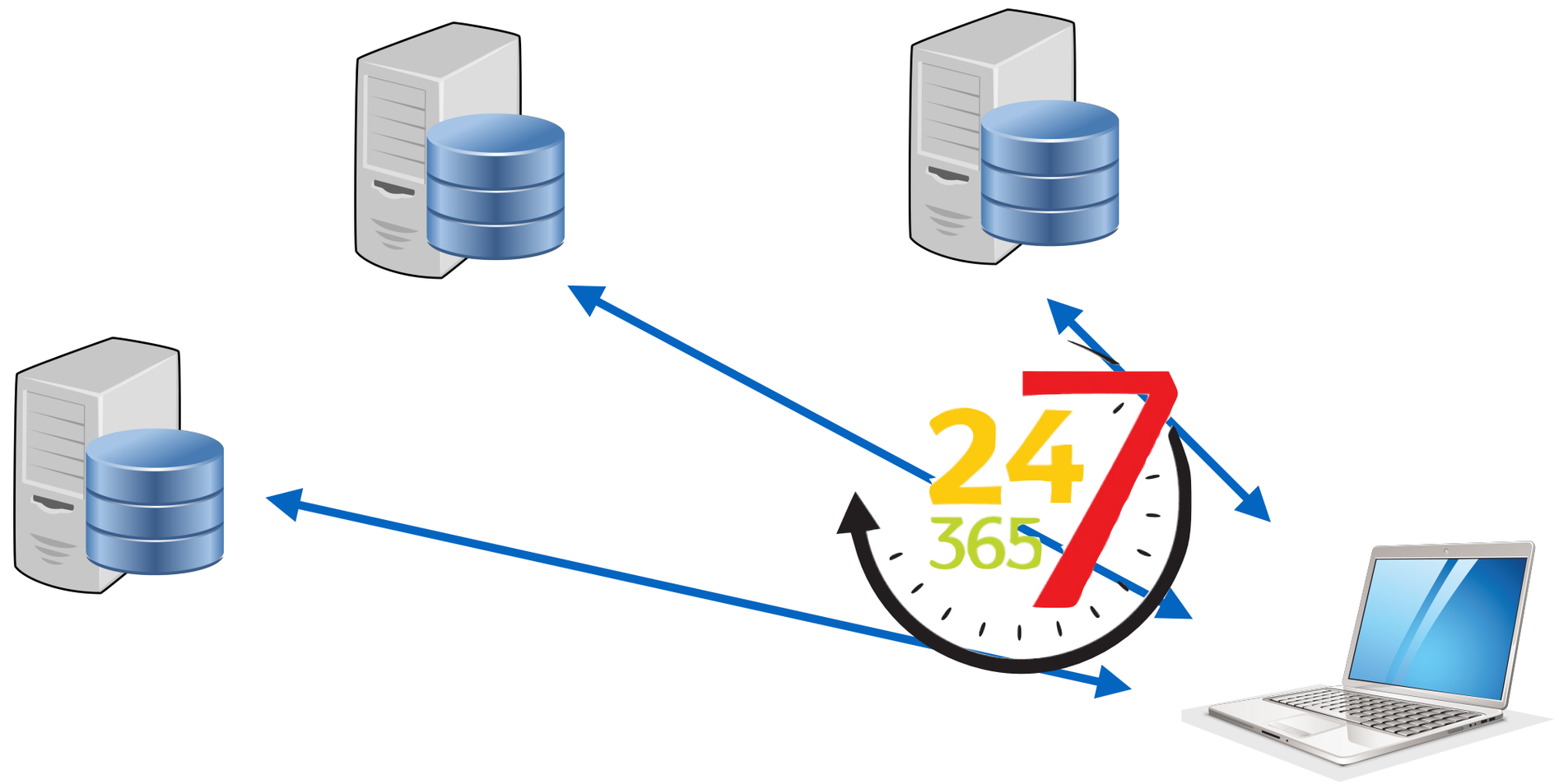
- Slides from http://cs-wwwarchiv.cs.unibas.ch/lehre/hs10/cs341/_Downloads/Workshop/Talks/2010-HS-DIS-I_Giangreco-CAP_Theorem-Talk.pdf

Consistency

- All the nodes in the system see the same state of the data
- Formally, we speak of *atomic* or *linearizable consistency*
- There exists a sequential order on all operations which is consistent with the order of invocations and responses, such that each operation looks as if it were completed at a single instant.

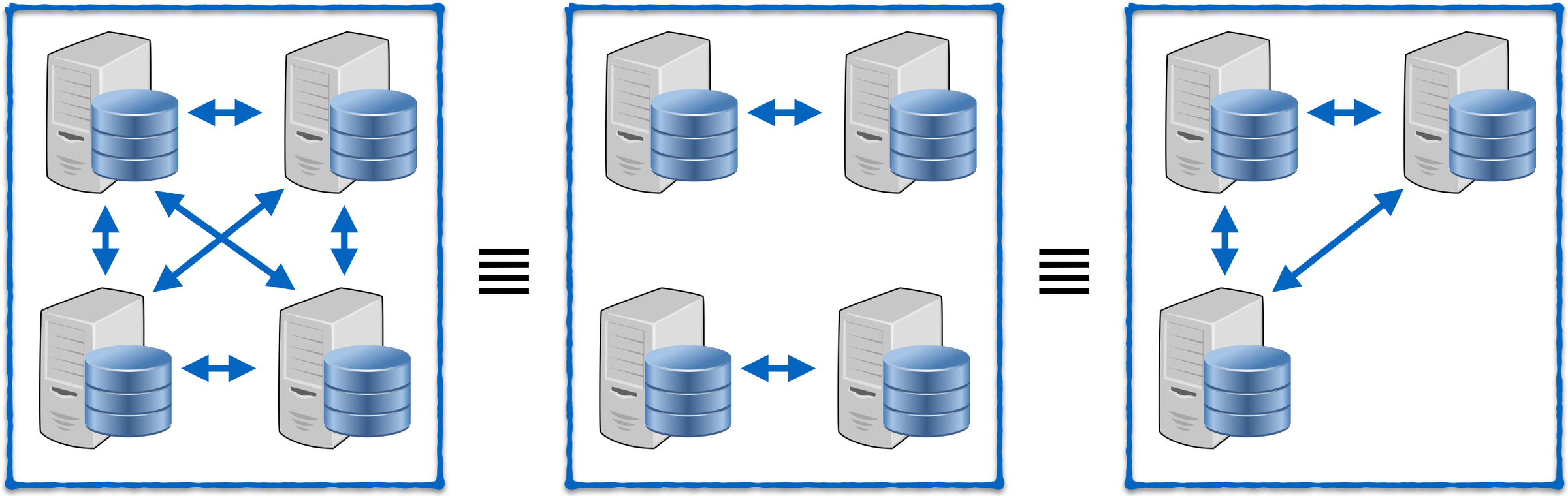
Availability

- Every request received by a non-failing node should be processed and must result in a response



Partition Tolerance

- If some nodes crash and/or some communications fail, system still performs as expected



CAP Theorem 1

It is impossible in the **asynchronous network model** to implement a read/write data object that guarantees the following properties:

- Availability
- Atomic consistency

in all fair executions (including those in which messages are lost).

Asynchronous, i. e. there is no clock, nodes make decisions based only on the messages received and local computation.

CAP Theorem 2

It is impossible in the **partially synchronous network model** to implement a read/write data object that guarantees the following properties:

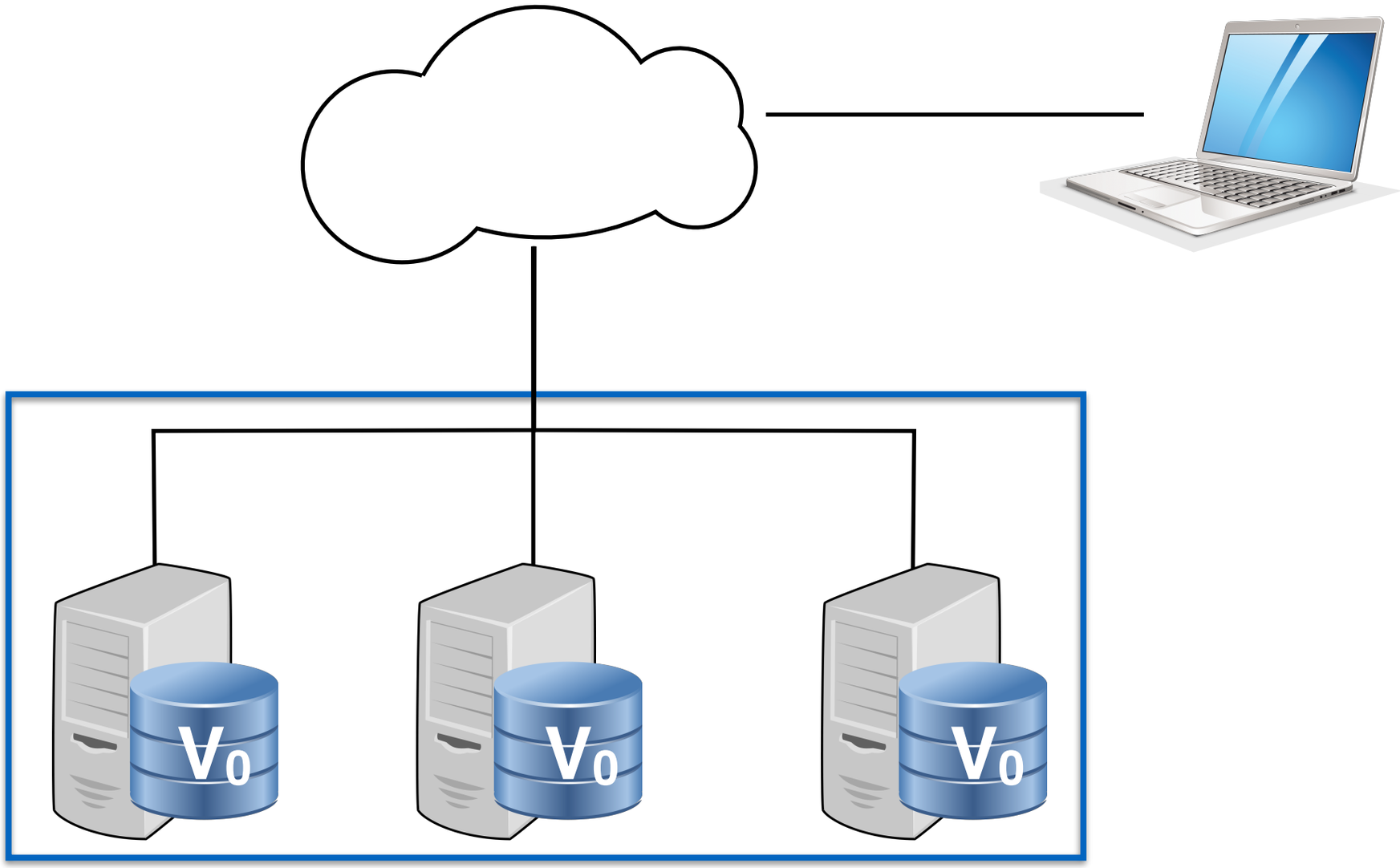
- Availability
- Atomic consistency

in all fair executions (including those in which messages are lost).

Partially synchronous, i. e. every node has a clock, and all clocks increase at the same rate. However, they are not synchronized.

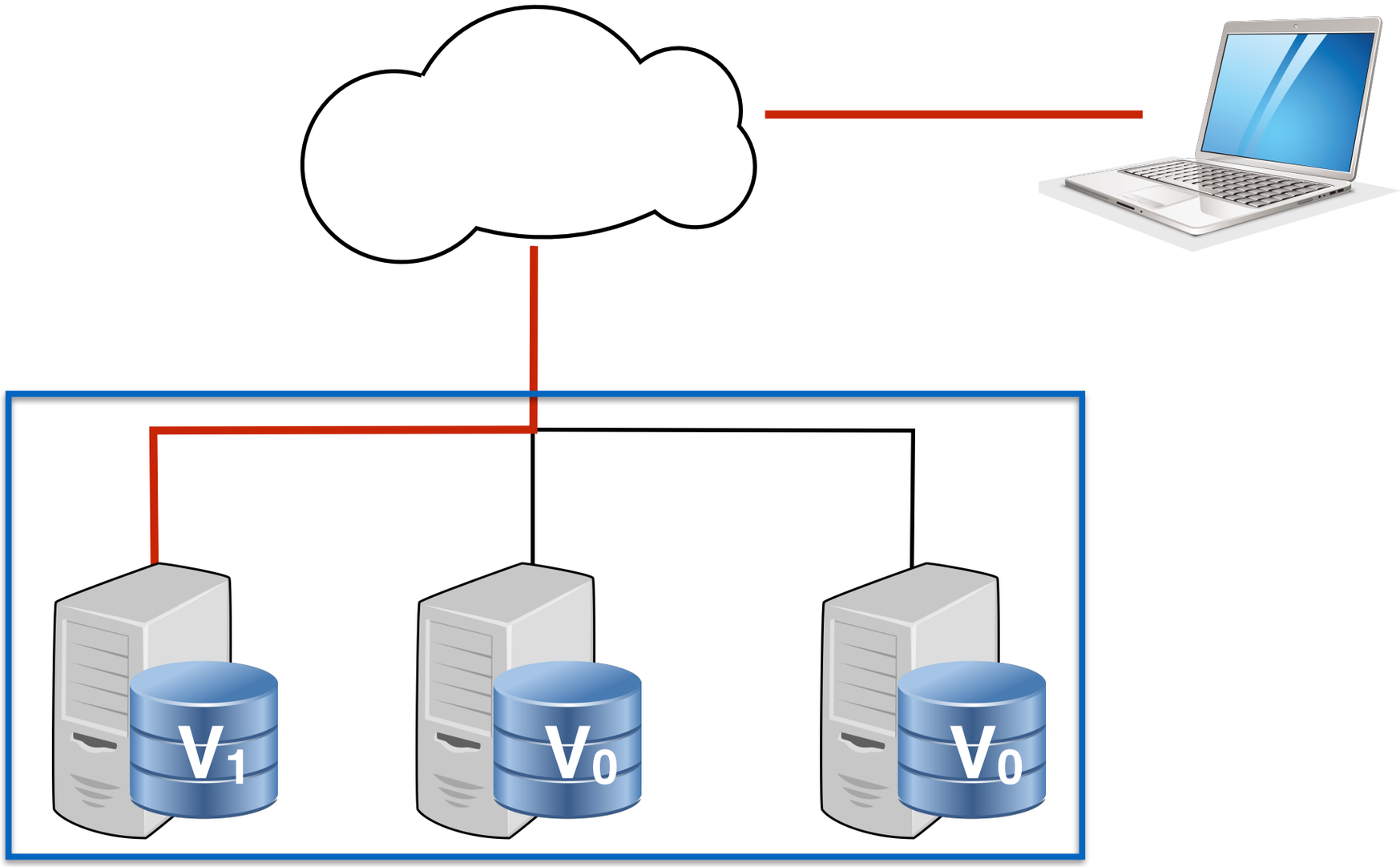
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_1.



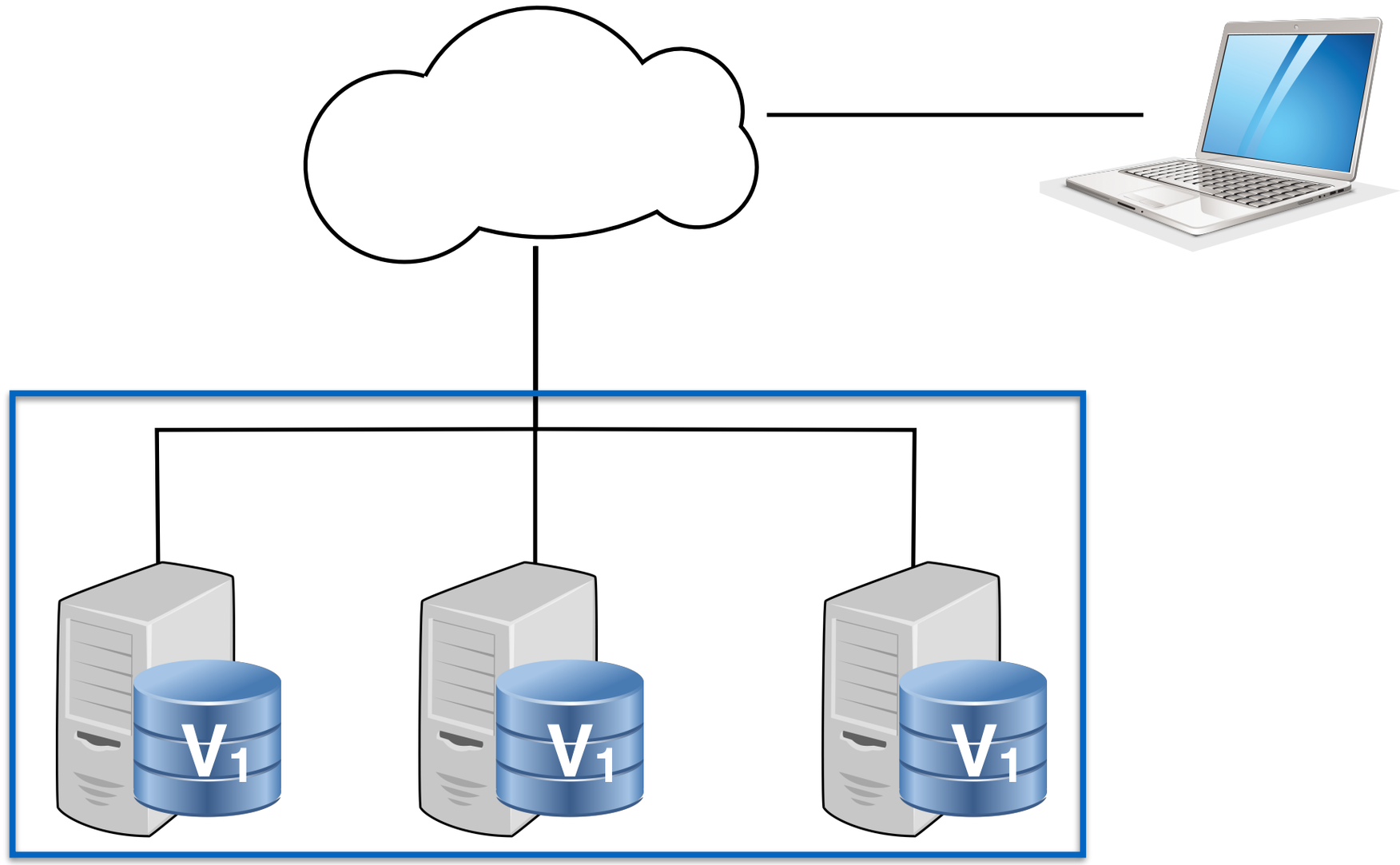
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_1.



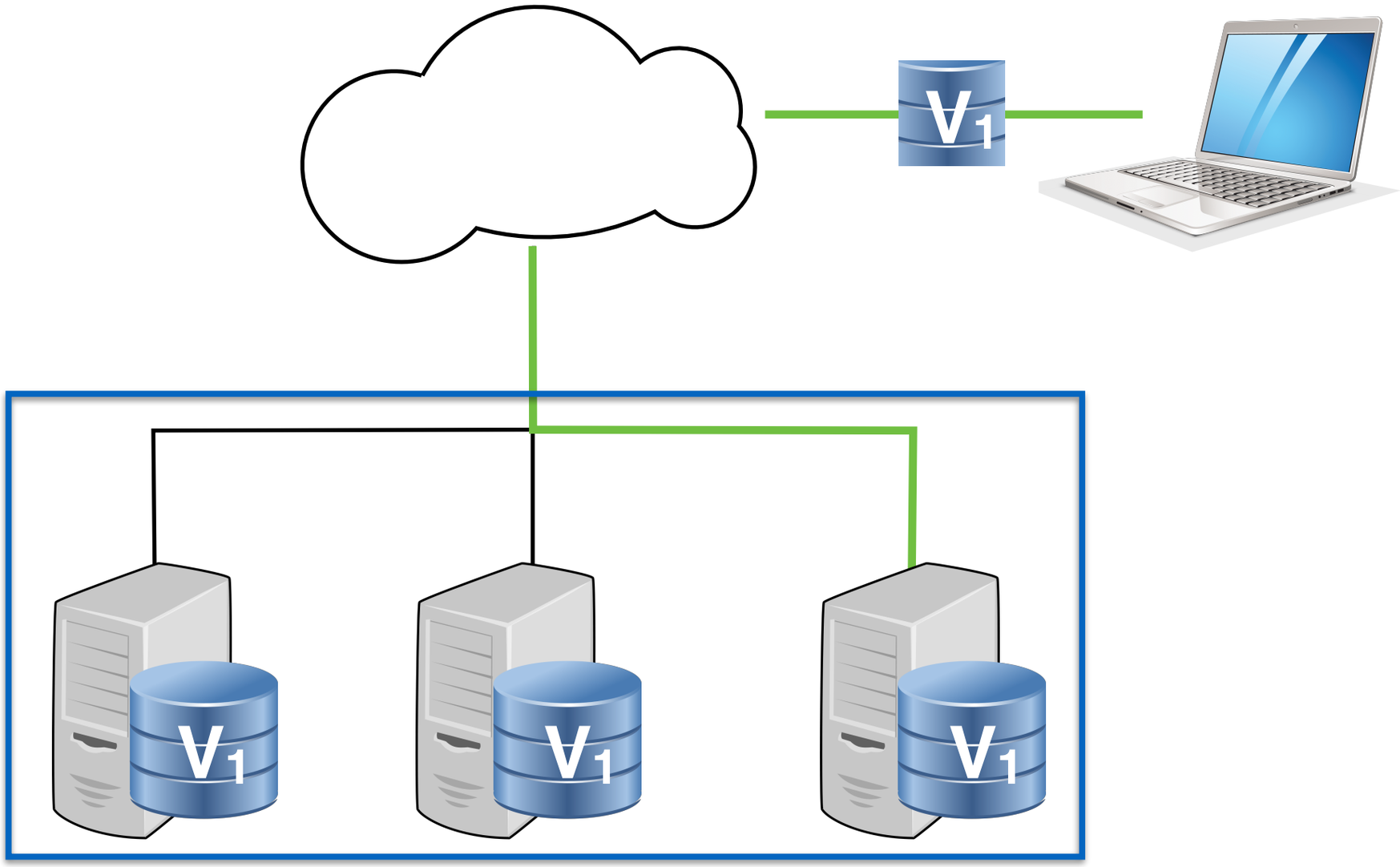
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_1.



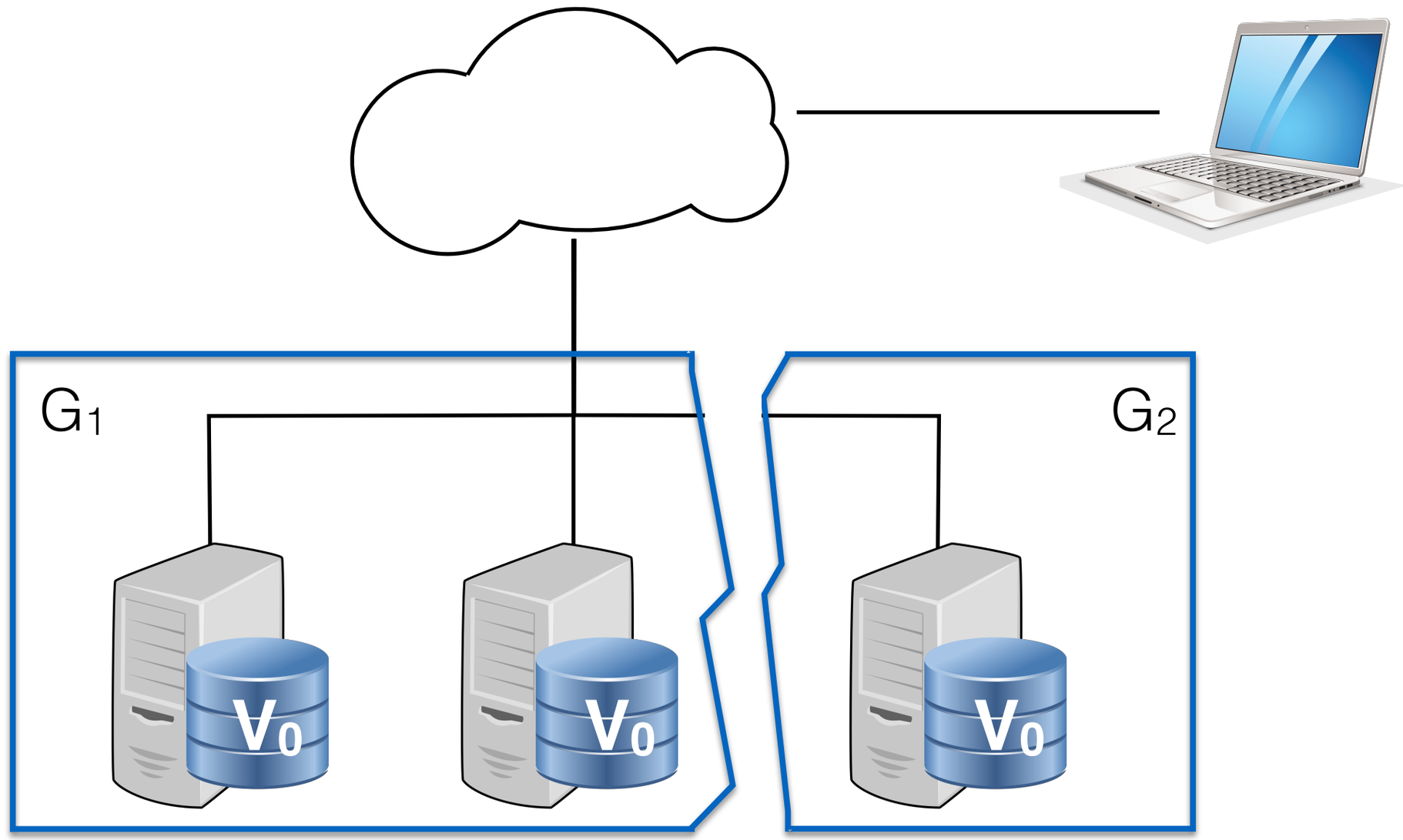
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_1.



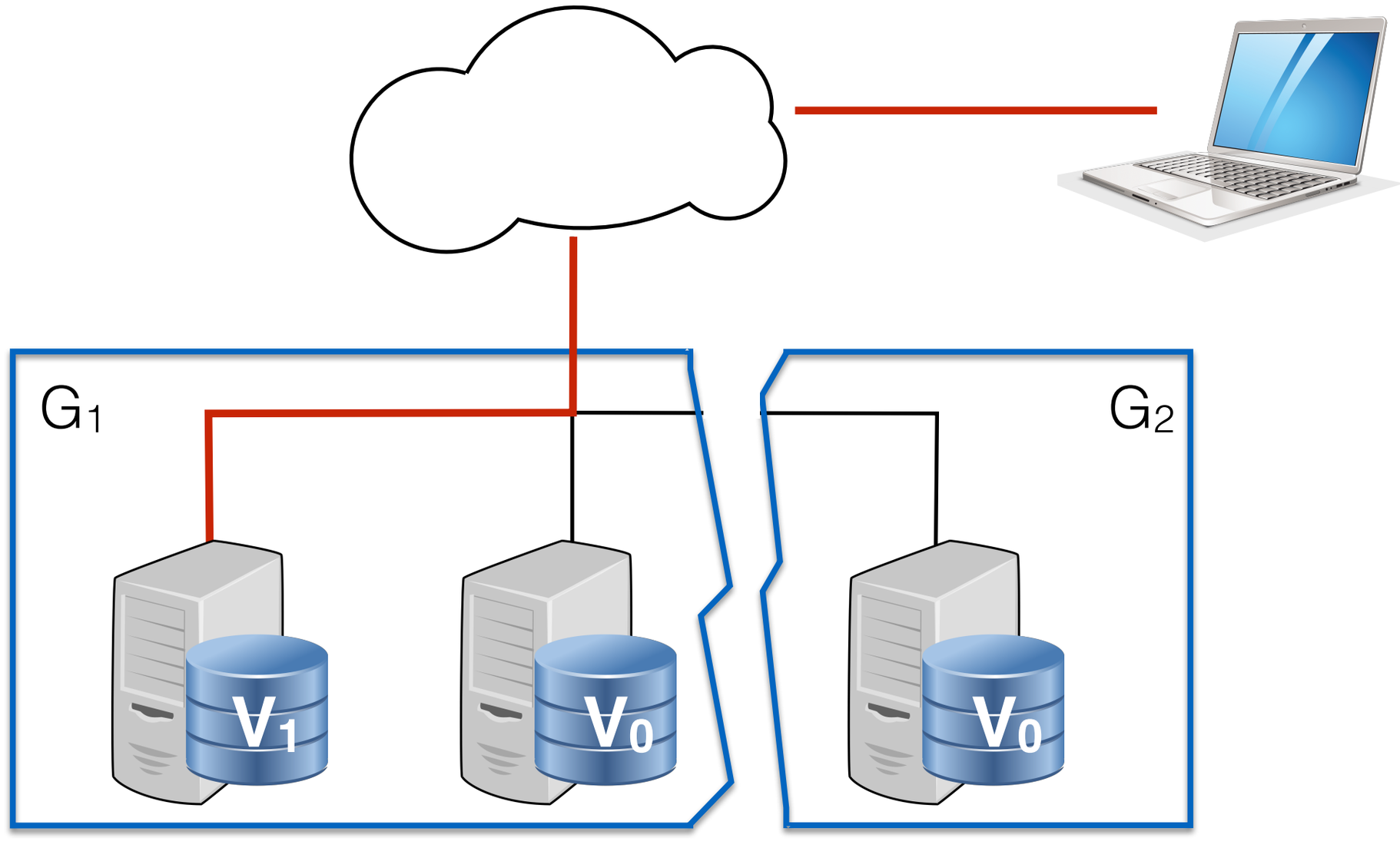
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



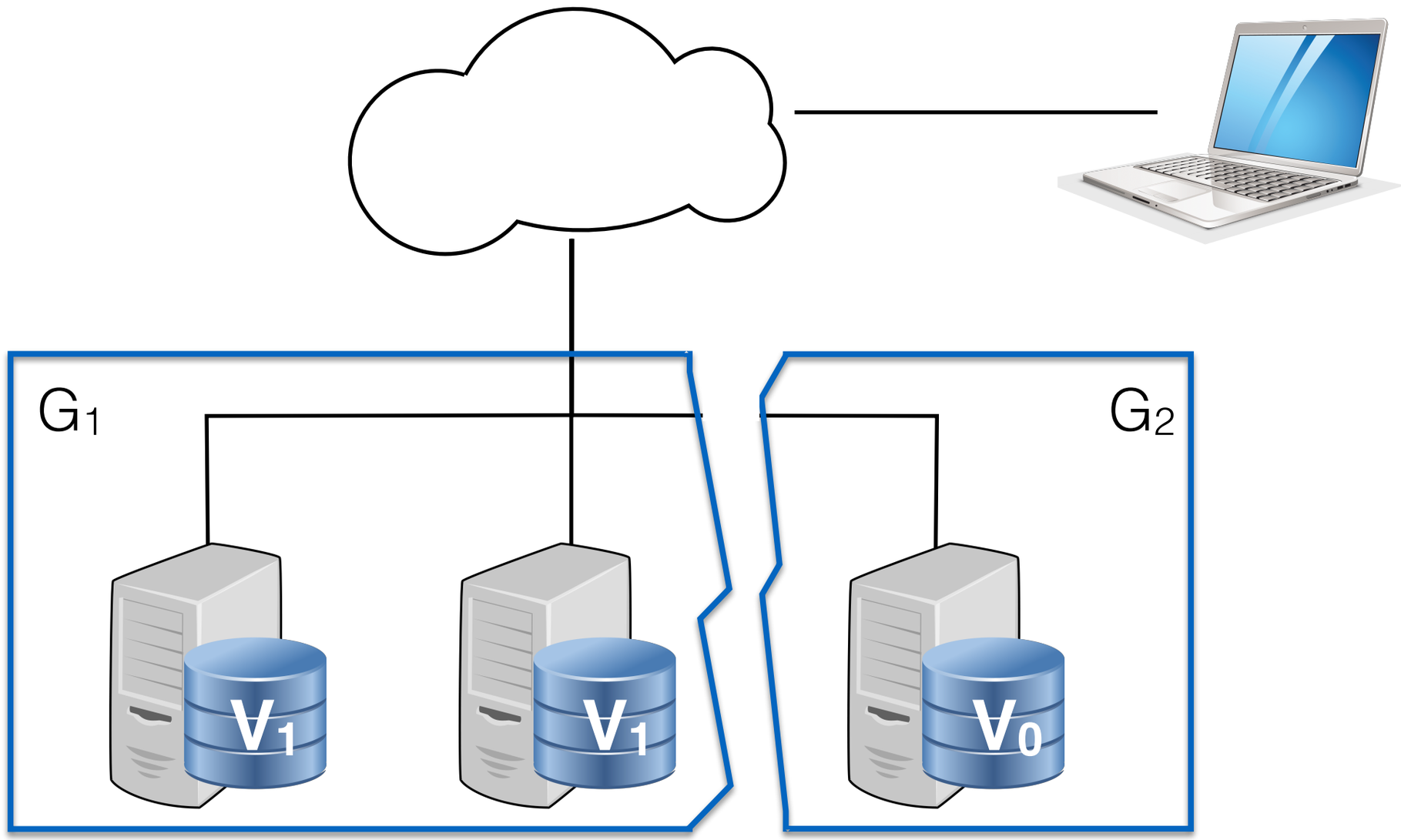
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



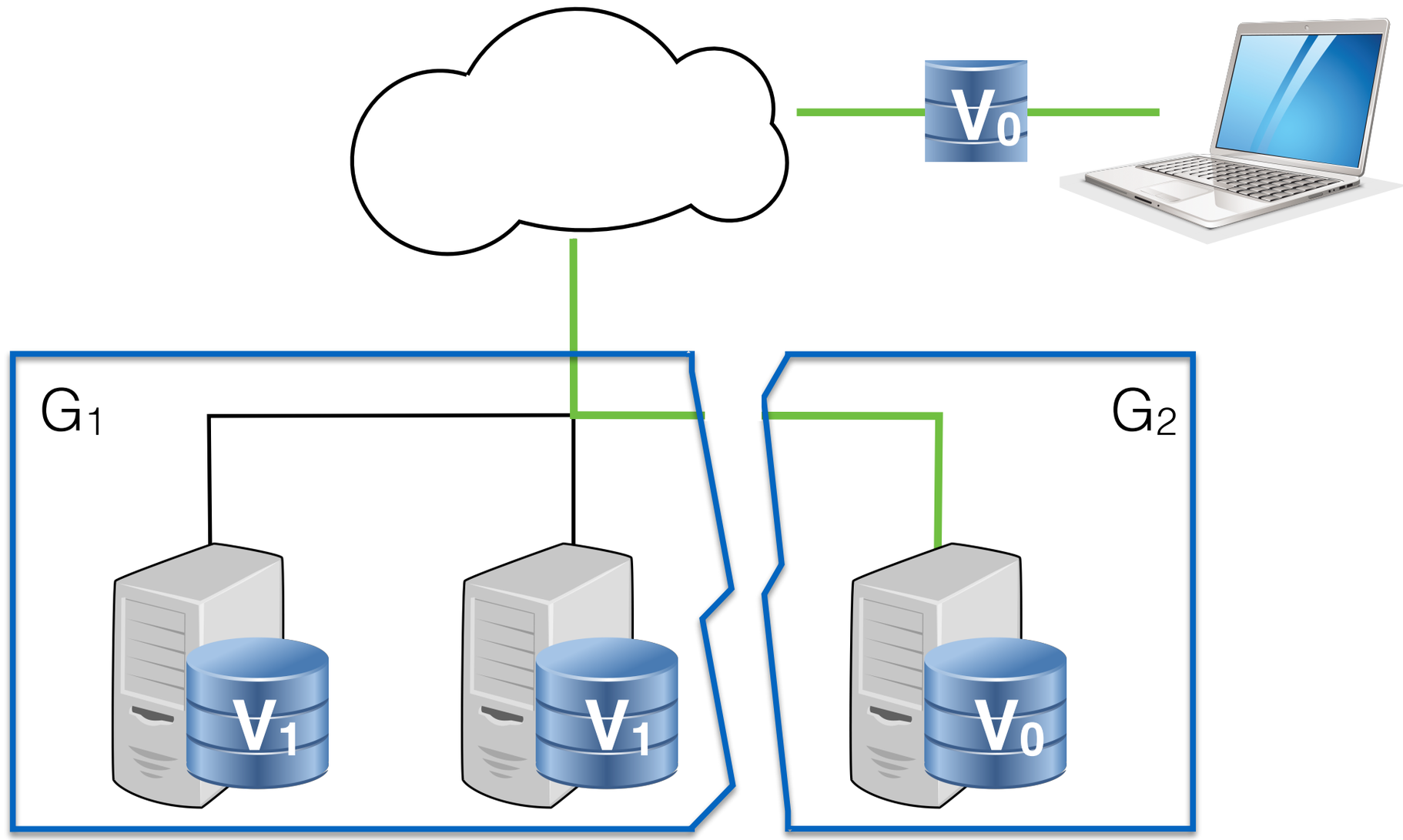
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



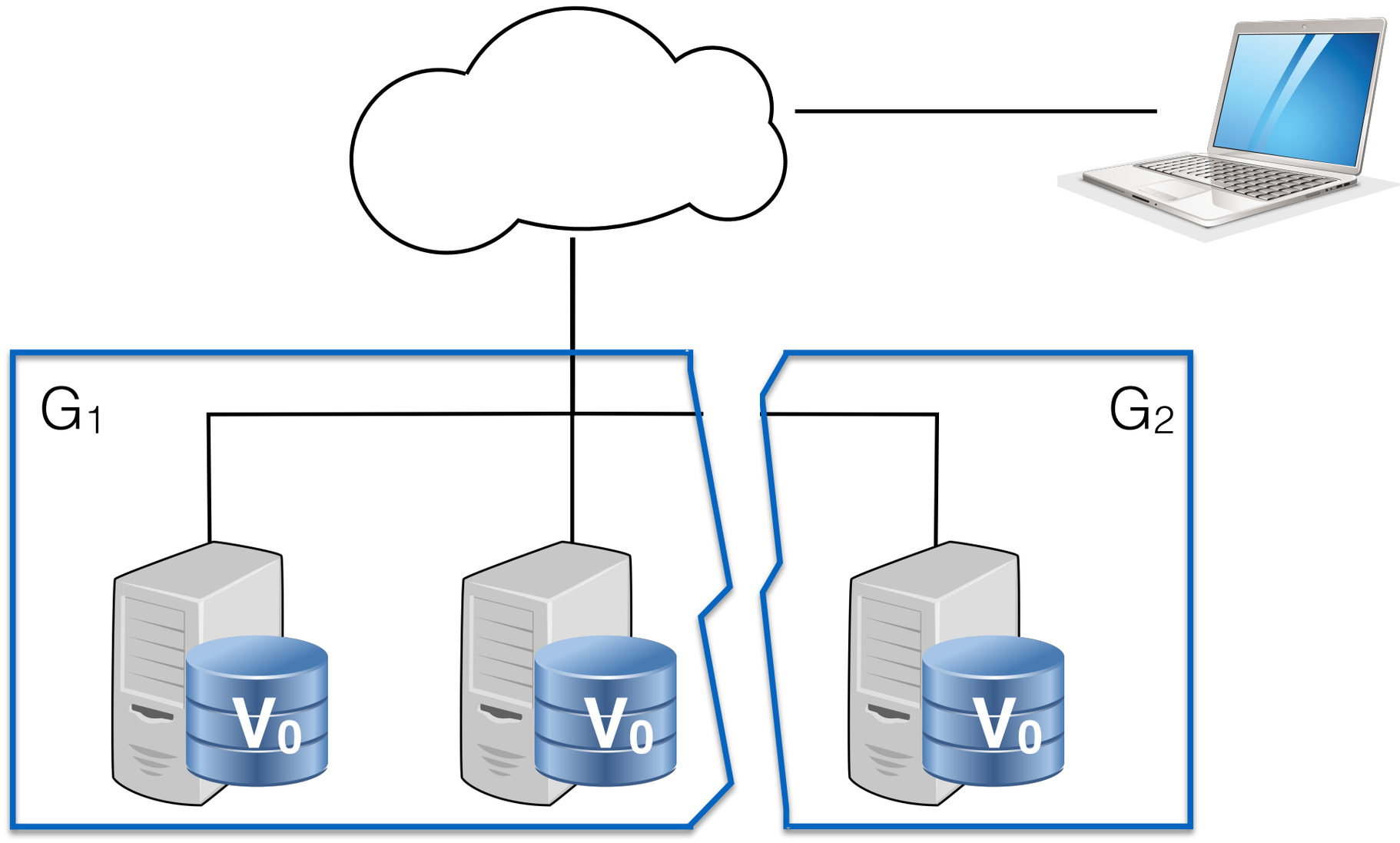
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



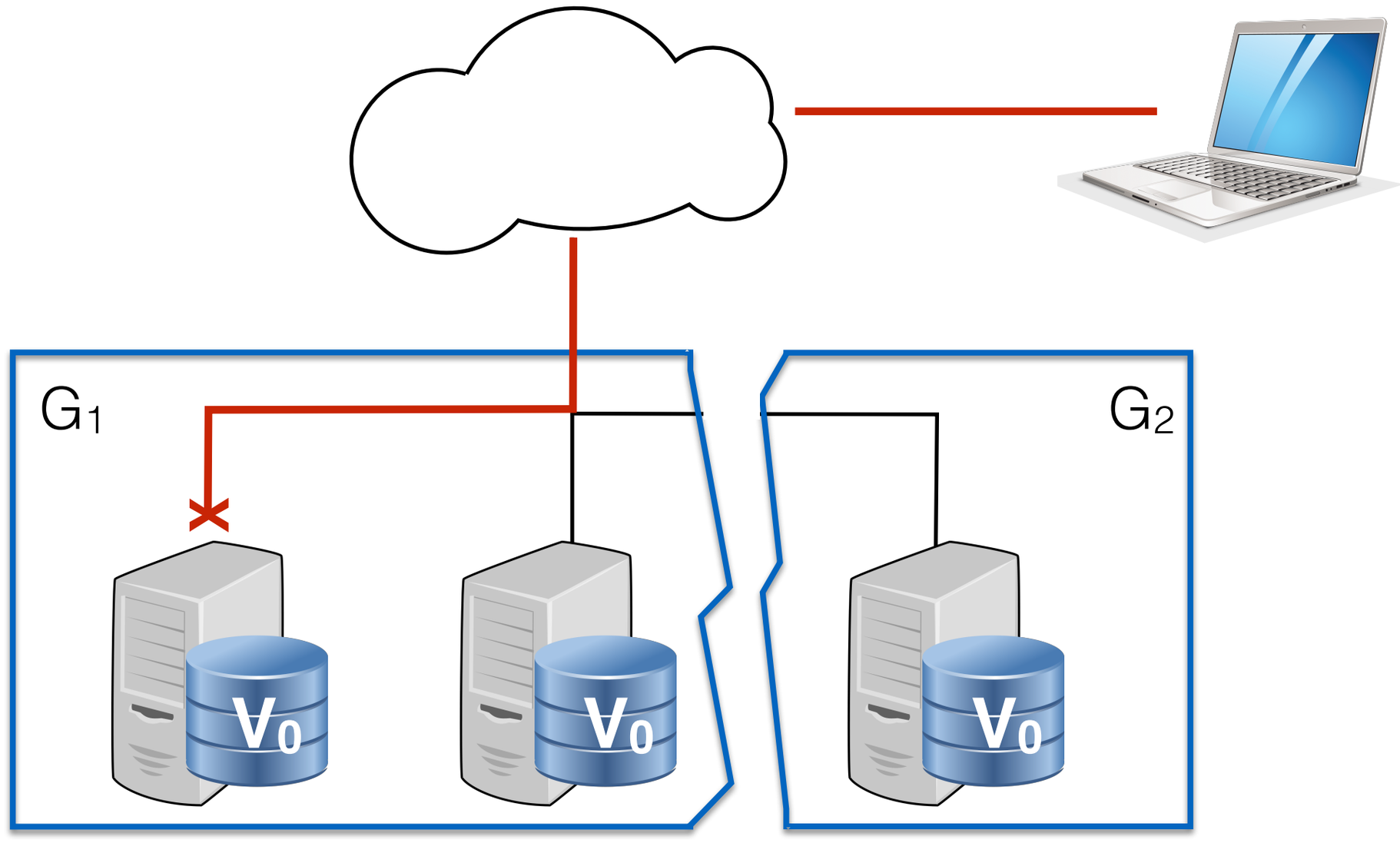
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- The write operation does not terminate. The availability requirement is violated.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



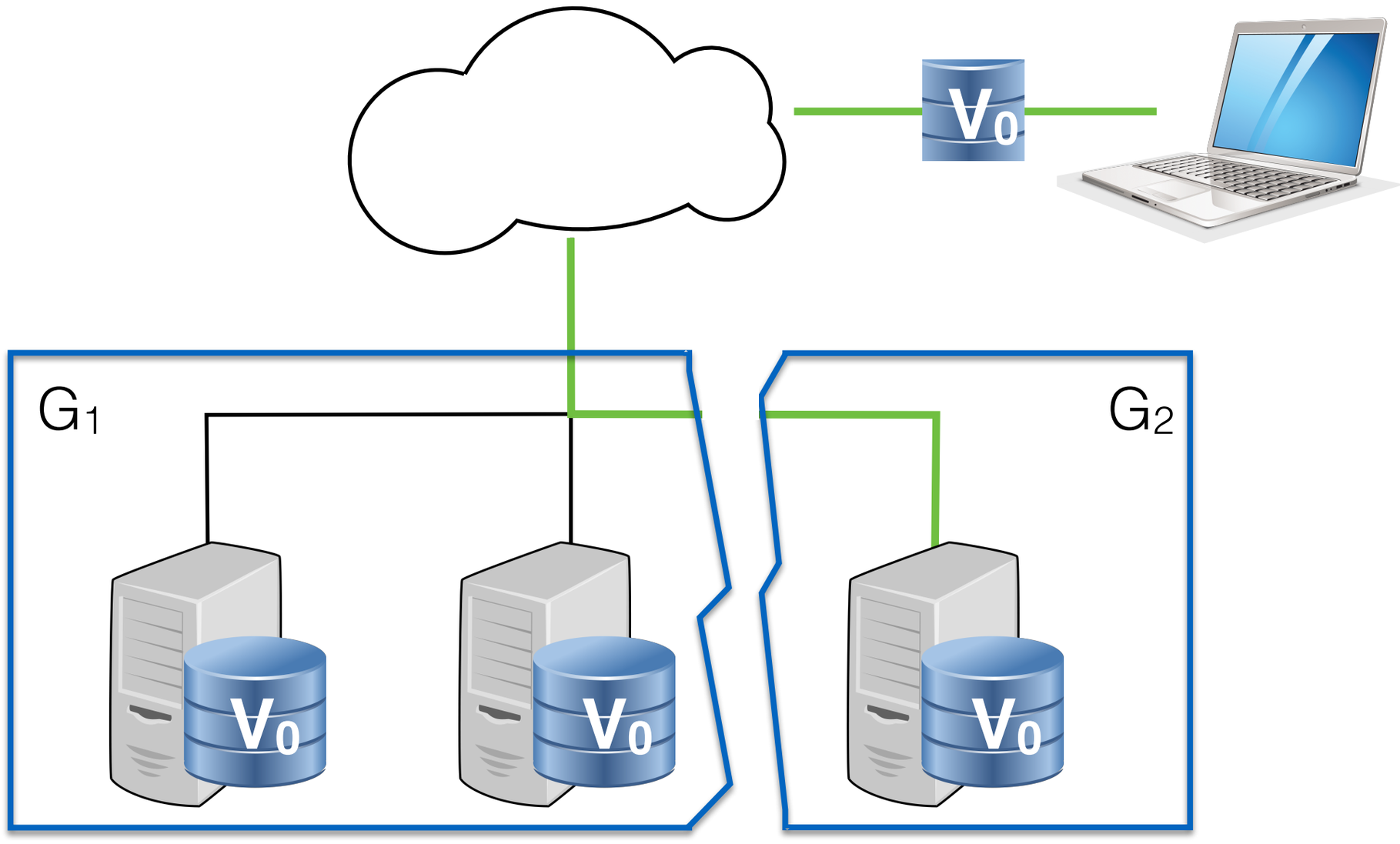
Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- The write operation does not terminate. The availability requirement is violated.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.

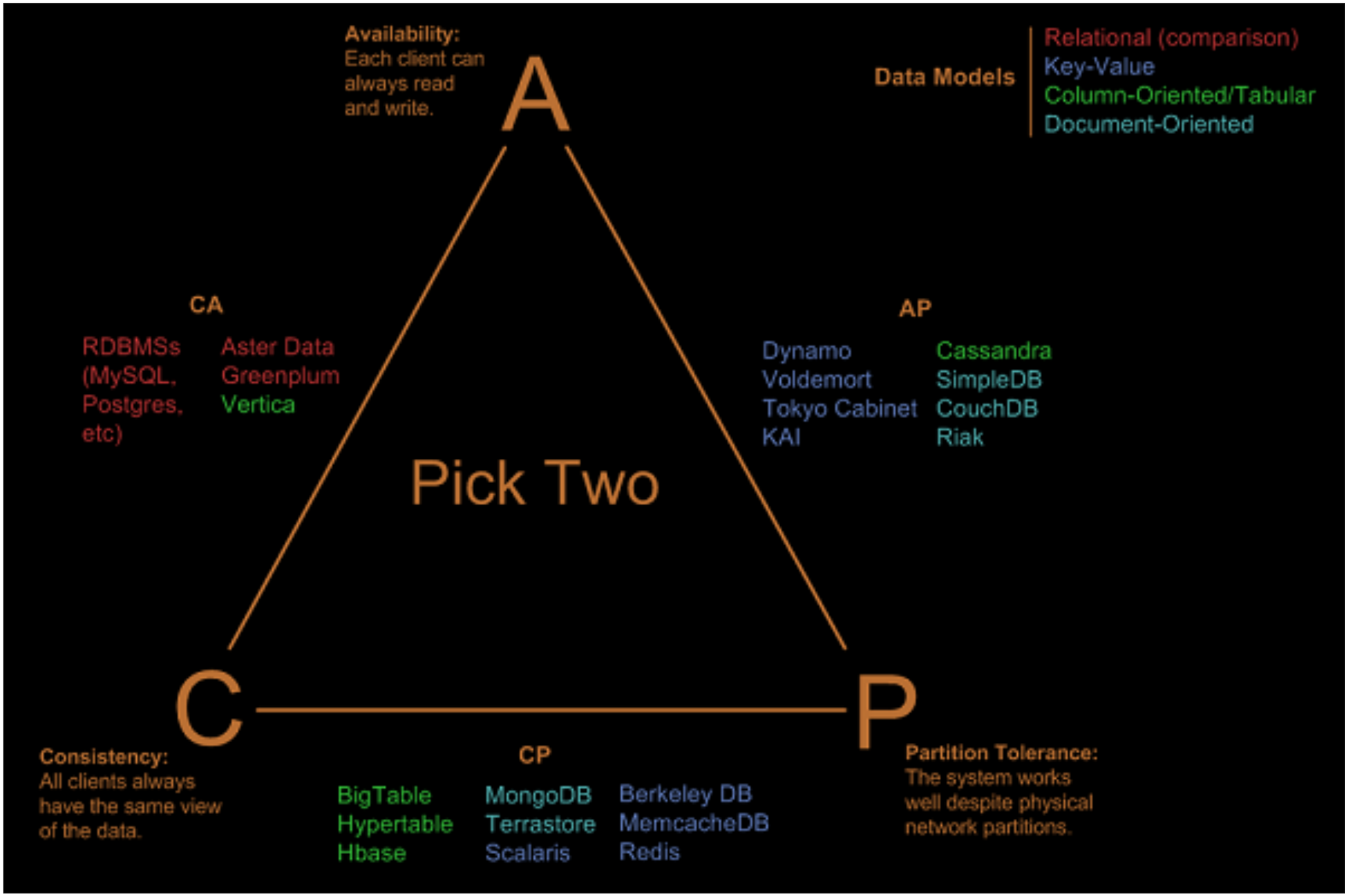


Proof 1 Sketch

- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- The write operation does not terminate. The availability requirement is violated.
- A single *read* occurs in G_2, and no other client requests occur, ending with the termination of the read operation.
- The read operation returns v_0.



Consequences of CAP



Consequences of CAP

- When partitions are rare (e.g., parallel systems), CAP should allow perfect C and A most of the time

- In distributed systems it is not possible to avoid network partitions.

- There is not a need to choose between either C or A, instead, it is more an act of balancing between the two properties.

Practical Consequences of CAP

- Many system designs used in early distributed relational database systems did not take into account partition tolerance (e.g. they were CA designs).

- There is a tension between strong consistency and high availability during network partitions. A distributed system consisting of independent nodes connected by an unpredictable network cannot behave in a way that is indistinguishable from a non-distributed system.

- There is a tension between strong consistency and performance in normal operation. Strong consistency requires that nodes communicate and agree on every operation. This results in high latency during normal operation.

- If we do not want to give up availability during a network partition, then we need to explore whether consistency models other than strong consistency are workable for our purposes.