Distributed Platforms

- References:
  - http://book.mixu.net/distsys
  - http://macs.citadel.edu/rudolphg/csci604/ImpossibilityofConsensus.pdf
  - http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf
  - http://cs-wwwarchiv.cs.unibas.ch/lehre/hs10/cs341/_Downloads/Workshop/Reports/2010-HS-DIS-I_Giangreco-CAP_Theorem-Report.pdf
  - Any serious recent distributed systems book 😄

Coordination

Consensus & Agreement

- It is generally important that the processes within a distributed system have some sort of agreement
- Coordination among multiple parties involves agreement among those parties
- Agreement $\Leftrightarrow$ Consensus $\Leftrightarrow$ Consistency
- Agreement is difficult in a dynamic asynchronous system in which processes may fail or join/leave

Impossibility Theorems

- Two fundamental theorems, FLP and CAP, influences the system design choices
- FLP theorem: asynchronicity vs synchronicity

  Consensus is impossible to implement in such a way that it both a) is always correct and b) always terminates if even one machine might fail in an asynchronous system with crash fault failures

- CAP theorem: what happens when network partitions are included in the failure model

  You can’t implement consistent storage and respond to all requests if you might drop messages between processes.

FLP

- **Impossibility of Distributed Consensus with One Faulty Process**, by Fischer, Lynch and Paterson (1985)
- **Consensus Problem**: we have a set of processes, each one with a private input; the processes communicate; the processes must agree on on some process's input.

Consensus is important

- With consensus we can implement anything we can imagine:
  - leader decision
  - mutual exclusion
  - transaction commitment
  - much more…
- In some models consensus is possible, in some other models, it is not
- The goal is to learn whether, for a given model, consensus is possible or not… and prove it!

(Wrong) Consensus Protocol

- Model:
  - n > 1 processes
  - shared memory (may be accessed simultaneously by multiple processes)
  - processors can atomically *read* and *write* (not both) a shared memory location
- Protocol:
  - There is a specific memory location C
  - Initially C is in a special state $\perp$
  - Processor *1* writes its value v_1 into C, then decides on v_1
  - Processors $j \neq 1$ read C until they read something else than $\perp$ and then decide on that
- Problems with this protocol?

Consensus Properties

1. Agreement: Every correct process must agree on the same value.

2. Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.

3. Termination: All correct processes eventually reach a decision.

4. Validity: If all correct processes propose the same value V, then all correct processes decide V.

FLP System Model

- Asynchronous communication model, i.e., no upper bound on the amount of time processors may take to receive, process and respond to an incoming message

- Communication links between processors are assumed to be reliable. It is well known that given arbitrarily unreliable links no solution for consensus could be found even in a synchronous model.

- Processors are allowed to fail according to the crash fault model – this simply means that processors that fail do so by ceasing to work correctly. There are more general failure models, such as byzantine failures where processors fail by deviating arbitrarily from the algorithm they are executing.

Notation (I)

- There are $N > 1$ processors which communicate by sending messages.
- A message is a pair (p,m) where p is the processor the message is intended for, and m is the contents of the message.
- Messages are stored in an abstract data structure called the message buffer which is a multiset – simply a set where more than one of any element is allowed – which supports two operations, *send* and *receive*.
- *send(p,m)* simply places the message (p,m) in the message buffer.
- *receive(p)* either returns a (random) message for processor p (and removes it from the message buffer) or the special value ∅, which does nothing.

Notation (II)

- Configuration: the internal state of all of the processors – the current step in the algorithm that they are executing and the contents of their memory – together with the contents of the message buffer.
- Step: the system moves from one configuration to the next by a step which consists of a processor *p* performing *receive(p)* and moving to another configuration, i.e.:
    - based on *p* local state and *m*, send an arbitrary but finite number of messages
    - based on *p* local state and *m*, change *p* local state to some new state
- Event: each step is therefore uniquely defined by the message that is received (possibly ∅) and the process *p* that received it. That pair is called an *event* (equivalent to a message)
    - Configurations move from one to another through events.
    - An event *e* can be applied to a configuration *C* if either *m* is ∅ or *(p,m)* is in the message buffer
    - *C' = e(C)* means that if we apply event *e* to configuration *C* we move to configuration *C'*
- Execution: a possibly infinite sequence of events from a specific initial configuration.
    - Since the receive operation is non-deterministic, there are many different possible executions for a given initial configuration.

Notation (III)

- Schedule & Run: a particular execution *σ*, defined by a possibly infinite sequence of events from a starting configuration, is called a *schedule* and the sequence of steps taken to realize the schedule is a *run*.
  - Non-faulty processes take infinitely many steps in a run (presumably eventually just receiving ∅ once the algorithm has finished its work) – otherwise a process is considered faulty.
  - *σ* can be applied to configuration *C* if the events in σ can be applied to *C* in order
  - *C' = σ(C)* means that if we apply schedule *σ* to configuration *C* we move to configuration *C'*
- An admissible run is one where at most one process is faulty (capturing the failure requirements of the system model) and every message is eventually delivered (this means that every processor eventually gets chosen to receive infinitely many times).
- We say that a run is a deciding run provided that some process eventually decides according to the properties of consensus, and that a consensus protocol is totally correct if every admissible run is a deciding run.

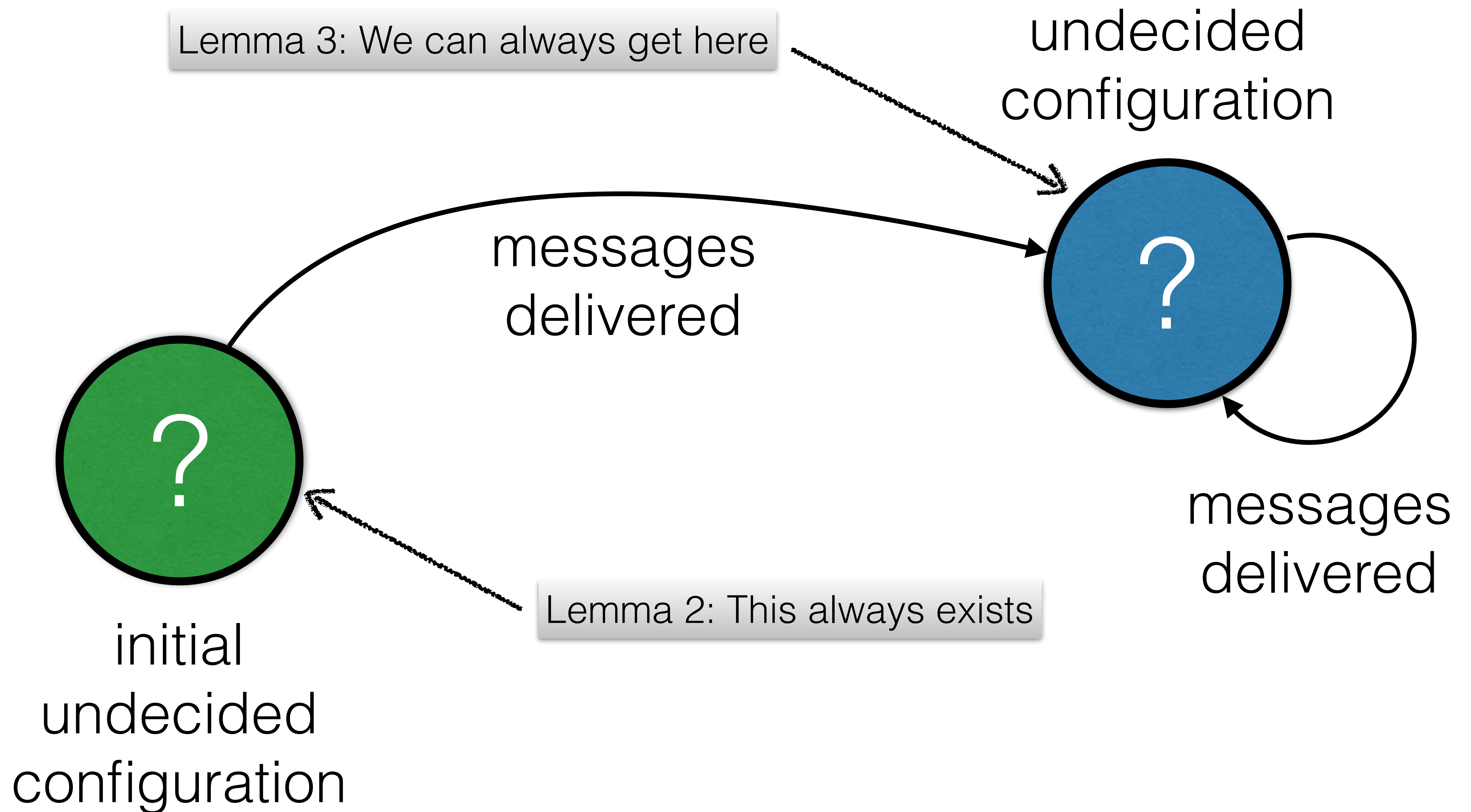
Proof Sketch

- FLP Theorem [1985]. No totally correct consensus algorithm exists (for the given system model).

- The idea behind it is to show that there is *some admissible run* – i.e., one with only one processor failure and eventual delivery of every message – *that is not a deciding run* – i.e., in which no processor eventually decides and the result is a protocol which runs for ever (because no processor decides).

- Two processors and binary consensus values

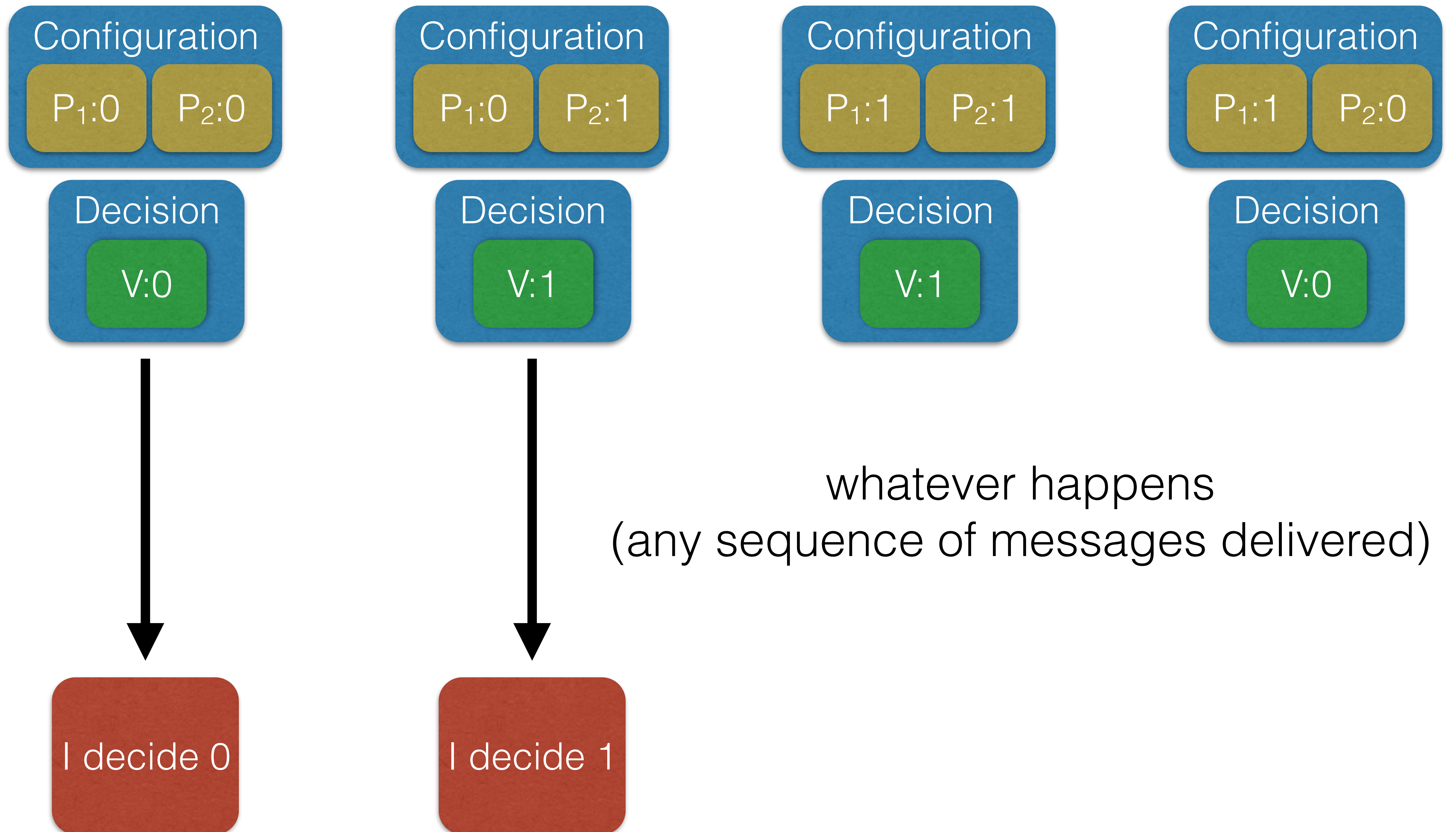
Proof Sketch



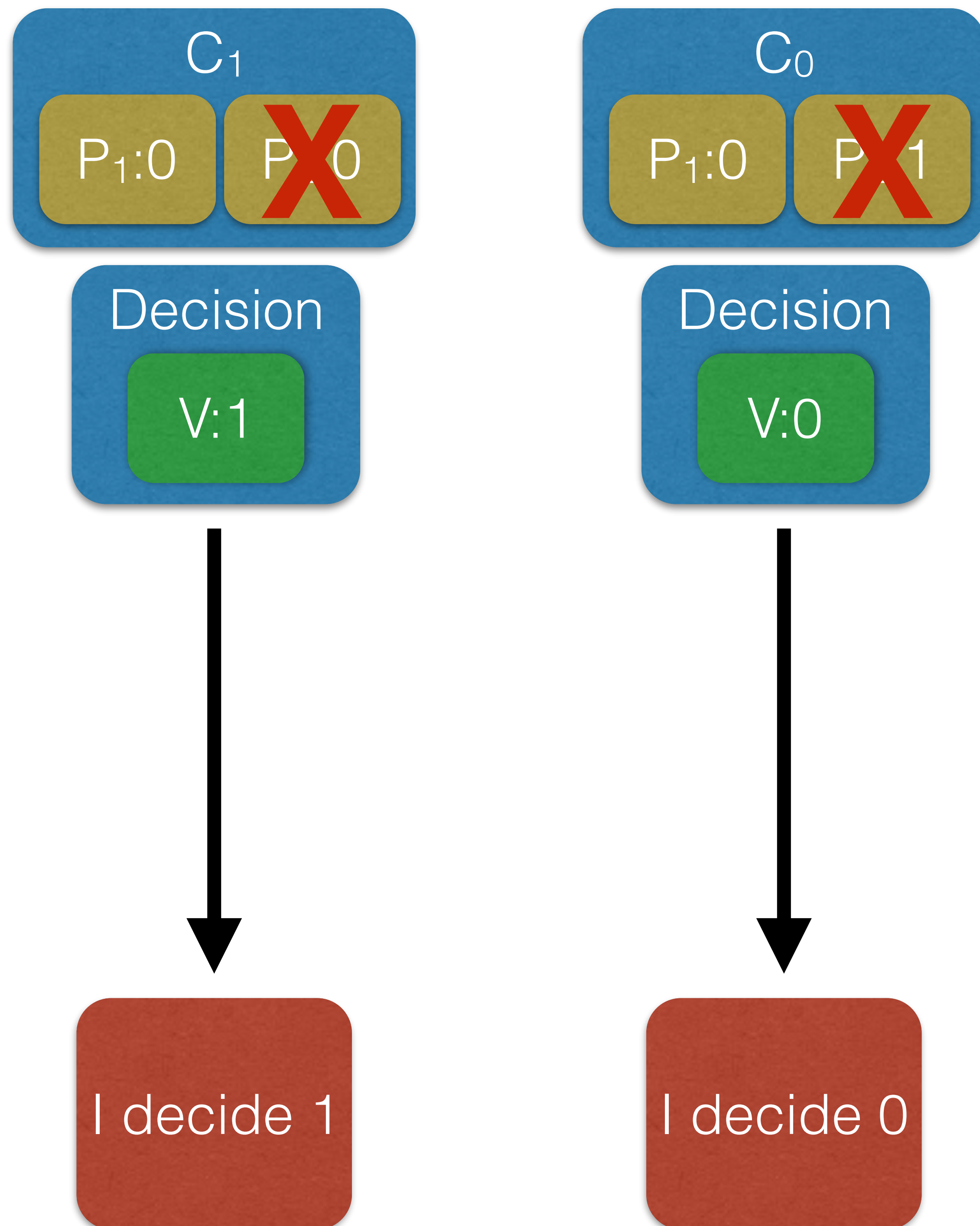
Lemma 2

- We want to show that there is some initial configuration in which the decision is not predetermined by the values of the processors, but it is a result of the messages exchanges and the occurrence of any failure
- We proceed by contradiction, using two processors and boolean decisions.
- Assume all initial configurations have predetermined executions

Two nodes system



Two nodes system



- P_2 initially fails, no messages sent or received
- P_1 can't know P_2 initial value
- There is a run from C_0 deciding 0 even if P_2 fails
- The same run can also be made by C_1
- They must decide on the same value, but they do not!

Valency

- A given configuration C is a <u>bivalent</u> if the decision if not predetermined, i.e. outcome can be 0 or 1.

- A given configuration C is <u>univalent</u> if it is not bivalent, e.g. 0-valent and 1-valent configurations.

- Undecided configuration is equivalent to bivalent

- Predetermined configuration is equivalent to univalent

Commutativity Lemma

- Let σ_1 and σ_2 be two schedules such that the set of processes executing steps in σ_1 are disjoint from the set that execute steps in σ_2. Then for any configuration C that σ_1 and σ_2 can both be applied, we have $\sigma_1(\sigma_2(C)) = \sigma_2(\sigma_1(C))$.
- Proof by induction on $k = \max(|\sigma_1|, |\sigma_2|)$

Induction Base

- We want to prove that $e_1(e_2(C)) = e_2(e_1(C))$
  - Suppose $e_1 = (p_1, m_1)$ and $e_2 = (p_2, m_2)$. Since e_1 can be applied to C, it means either m_1 is $\varnothing$ or (p_1, m_1) is in the message system. The same is for e_2. Because $p_1 \neq p_2$, e_1 can be applied to $e_2(C)$ and e_2 can be applied to $e_1(C)$.
  - Let $C_1 = e_1(e_2(C))$ and $C_2 = e_2(e_1(C))$. Then the state of the message system is the same in C_1 as in C_2. The states of all processes are the same in C_1 and C_2 as well. Thus $C_1 = C_2$.

Induction Step

- Case 1: $|\sigma_1| = k+1,\ |\sigma_2| \leq k$

  Suppose the first event in σ_1 is e and $\sigma_1 = (\sigma,e)$. Then
  $\sigma_1(\sigma_2(C)) = \sigma(e(\sigma_2(C))) = \sigma(\sigma_2(e(C))) = \sigma_2(\sigma(e(C))) = \sigma_2(\sigma_1(C))$

- Case 2: $|\sigma_1| \leq k,\ |\sigma_2| = k+1$

  Same as Case 1.

- Case 3: $|\sigma_1| = k+1,\ |\sigma_2| = k+1$

  Suppose the first event in σ_2 is e and $\sigma_2 = (\sigma,e)$. Then
  $\sigma_1(\sigma_2(C)) = \sigma_1(\sigma(e(C))) = \sigma(\sigma_1(e(C))) = \sigma(e(\sigma_1(C))) = \sigma_2(\sigma_1(C))$
  (we used Case 1 here)

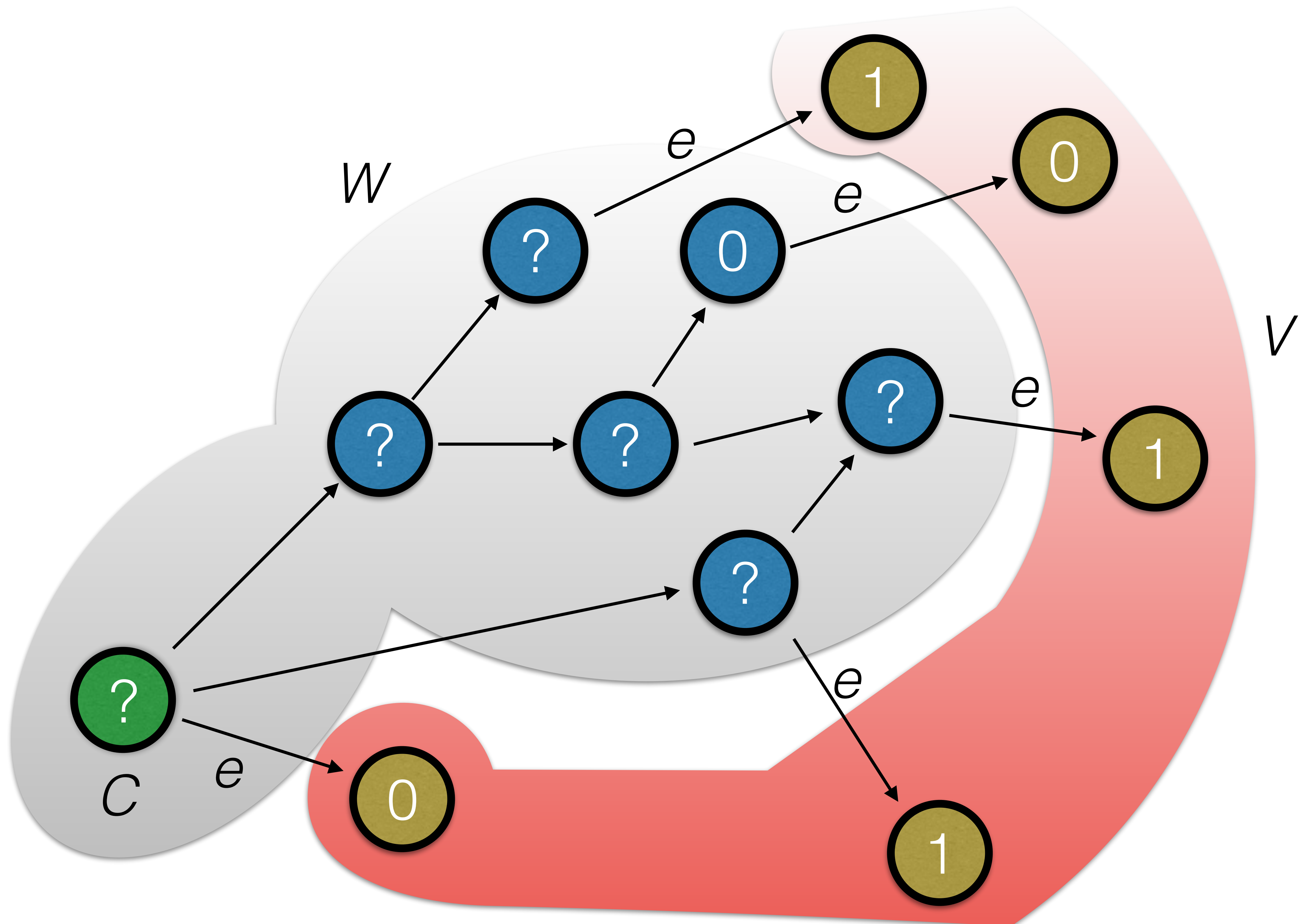
Delayed message Lemma

- Let *C* be a configuration, and *e = (p,m)* is an event that can be applied to *C*. Let *W* be the set of configurations that is reachable from *C* without applying *e*, then *e* can be applied to any state in *W*.
- Proof: trivial.

Lemma 3

- We want to show that we can keep the system in a bivalent state
- Formally, let C be a bivalent configuration, and $e=(p,m)$ any event that can be applied to C. Let W be the set of configurations that is reachable from C without applying e, and $V = e(W)$ to be the set of configurations reached by applying e to the configurations in W. Then V contains a bivalent configuration.
- We need 4 intermediate claims.
- We proceed by contradiction, assuming V contains univalent configurations only (in claims too) and reaching a contradiction.

Lemma 3

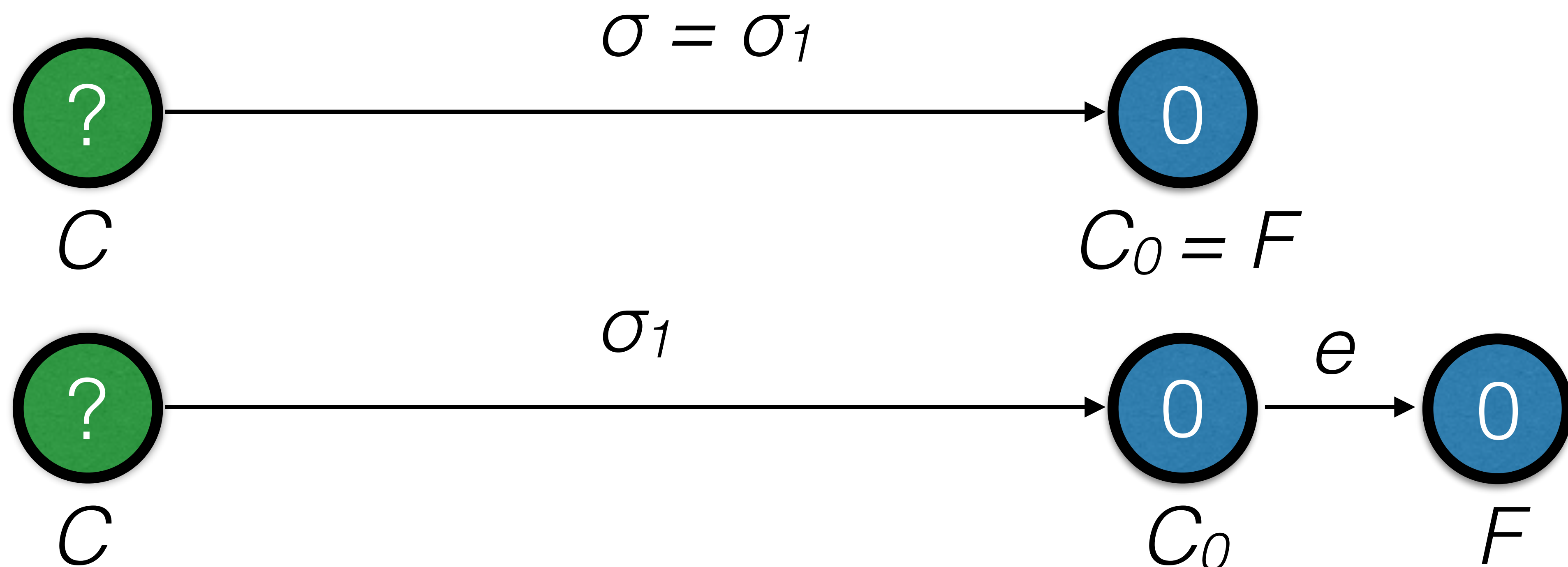


Claim 1

- There is a 0-valent configuration F such that $F = \sigma(C)$, i.e. F is reachable from C, and σ contains the event e.

  Proof: C is bivalent, so we must have a 0-valent configuration C_0 reachable from C where $C_0 = \sigma_1(C)$

  - Case 1: σ_1 contains e. Hence $F = C_0$ and $\sigma = \sigma_1$.
  - Case 2: σ_1 does not contain e. We let $F = e(C_0)$ and $\sigma = (\sigma_1, e)$. Since C_0 is 0-valent, F must be 0-valent as well.



Claim 2

- There must be a 0-valent configuration C_0 in V.

  Proof: Consider the F as defined in Claim 1, and the prefix σ' of σ whose last event is e. Let $C_0 = \sigma'(C) \in V$. Because V does not contain bivalent states and because the 0-valent state F is reachable from C_0, C_0 must be 0-valent.

Claim 3

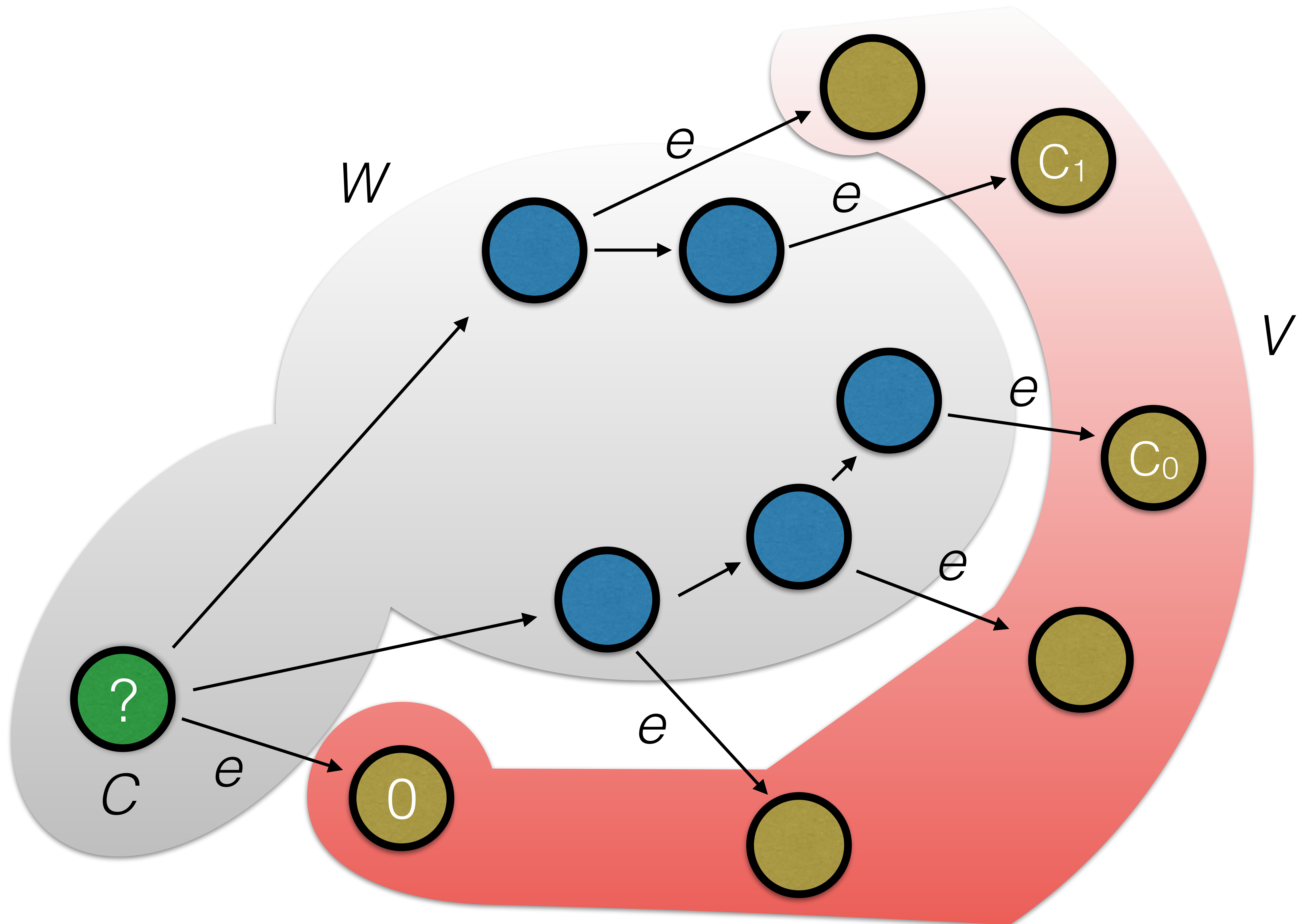
- There must a 1-valent configuration C_1 in V.

  Proof: as per Claims 1 & 2

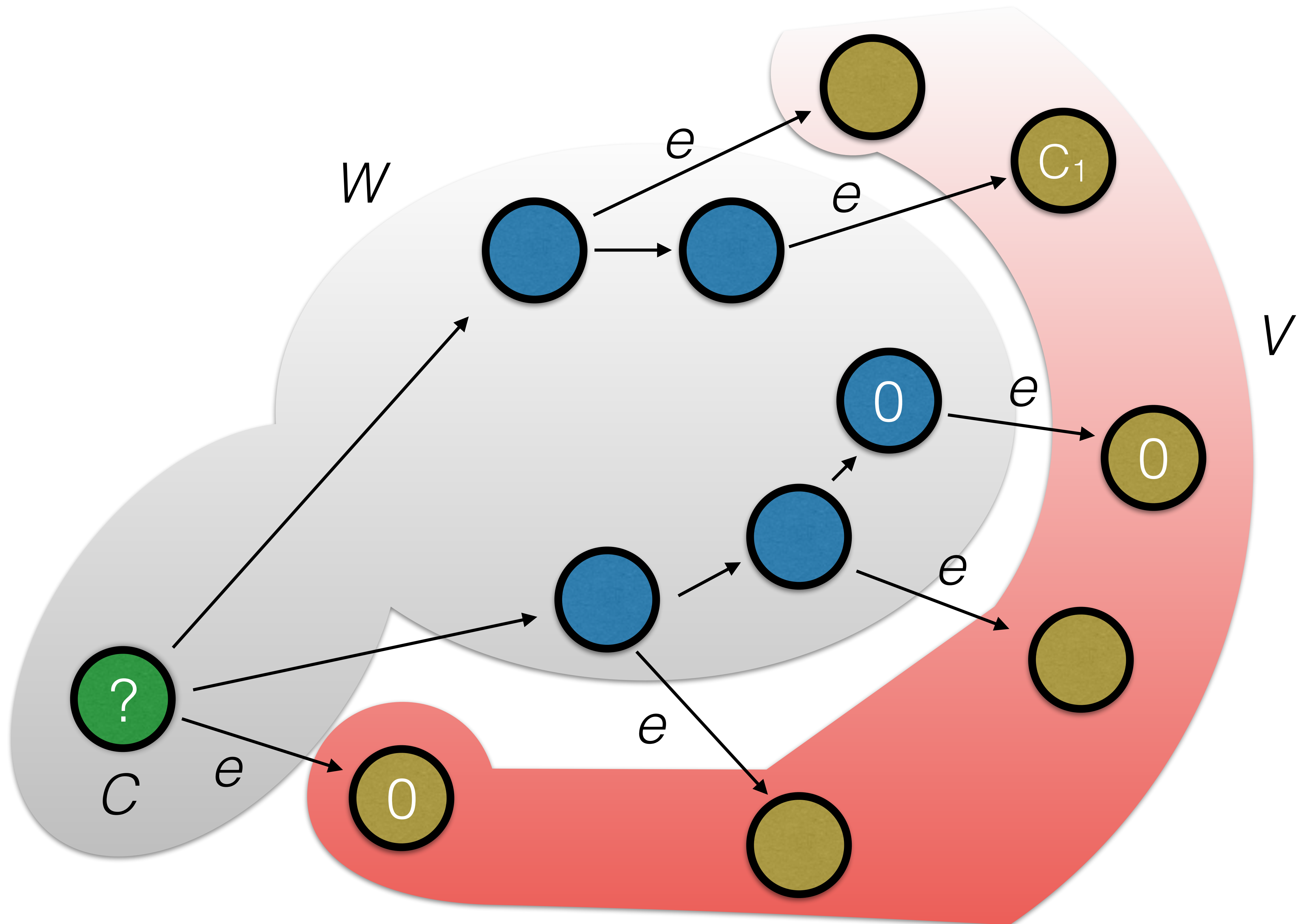
Claim 4

- There must be F_0 and F_1 in W, such that $e(F_0)$ is 0-valent, $e(F_1)$ is 1-valent, and F_0 and F_1 are neighbors, i.e., either $F_1 = d(F_0)$ or $F_0 = d(F_1)$.
- Proof: by simple induction, assuming w.l.o.g. $e(C)$ is 0-valent

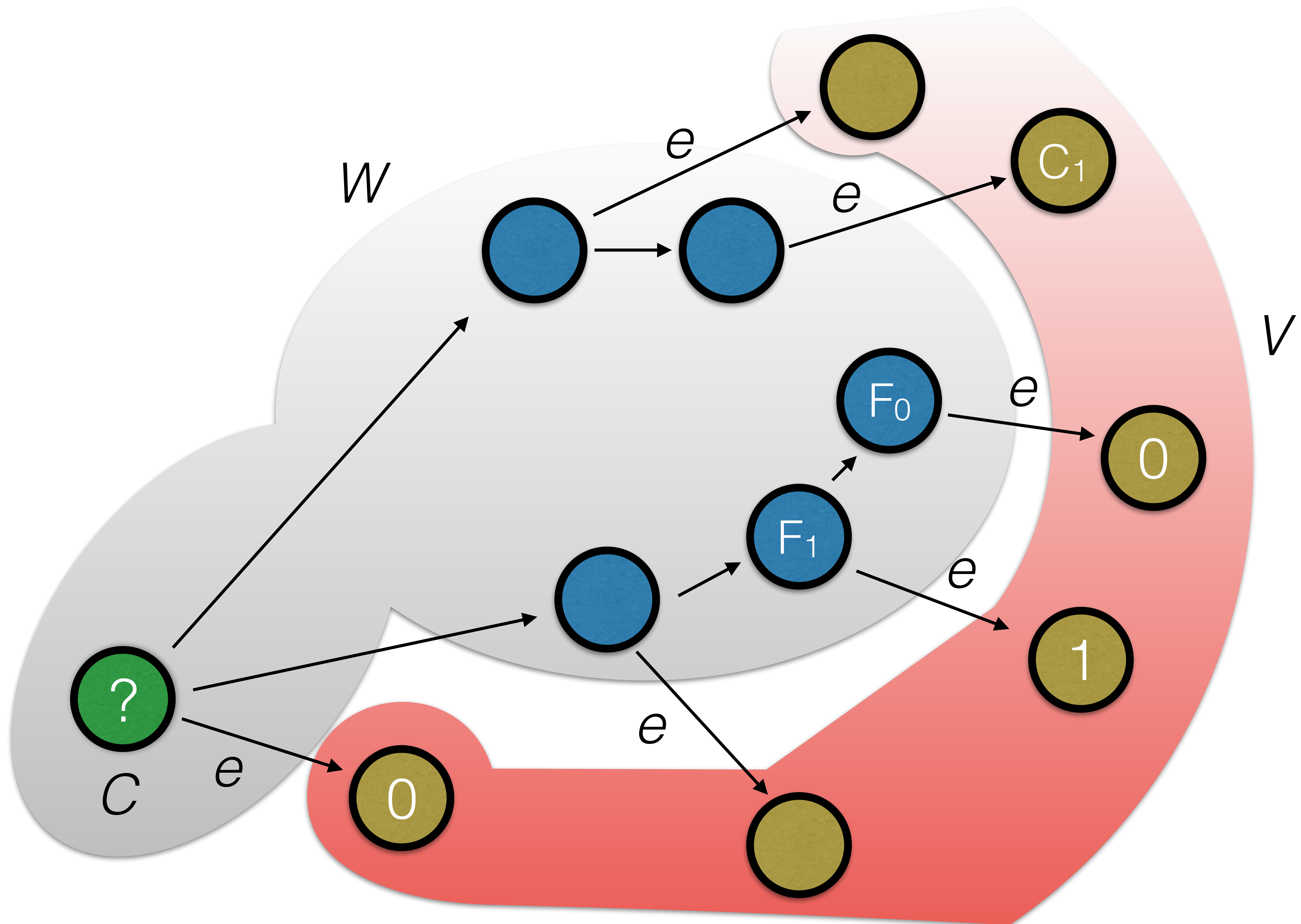
Claim 4



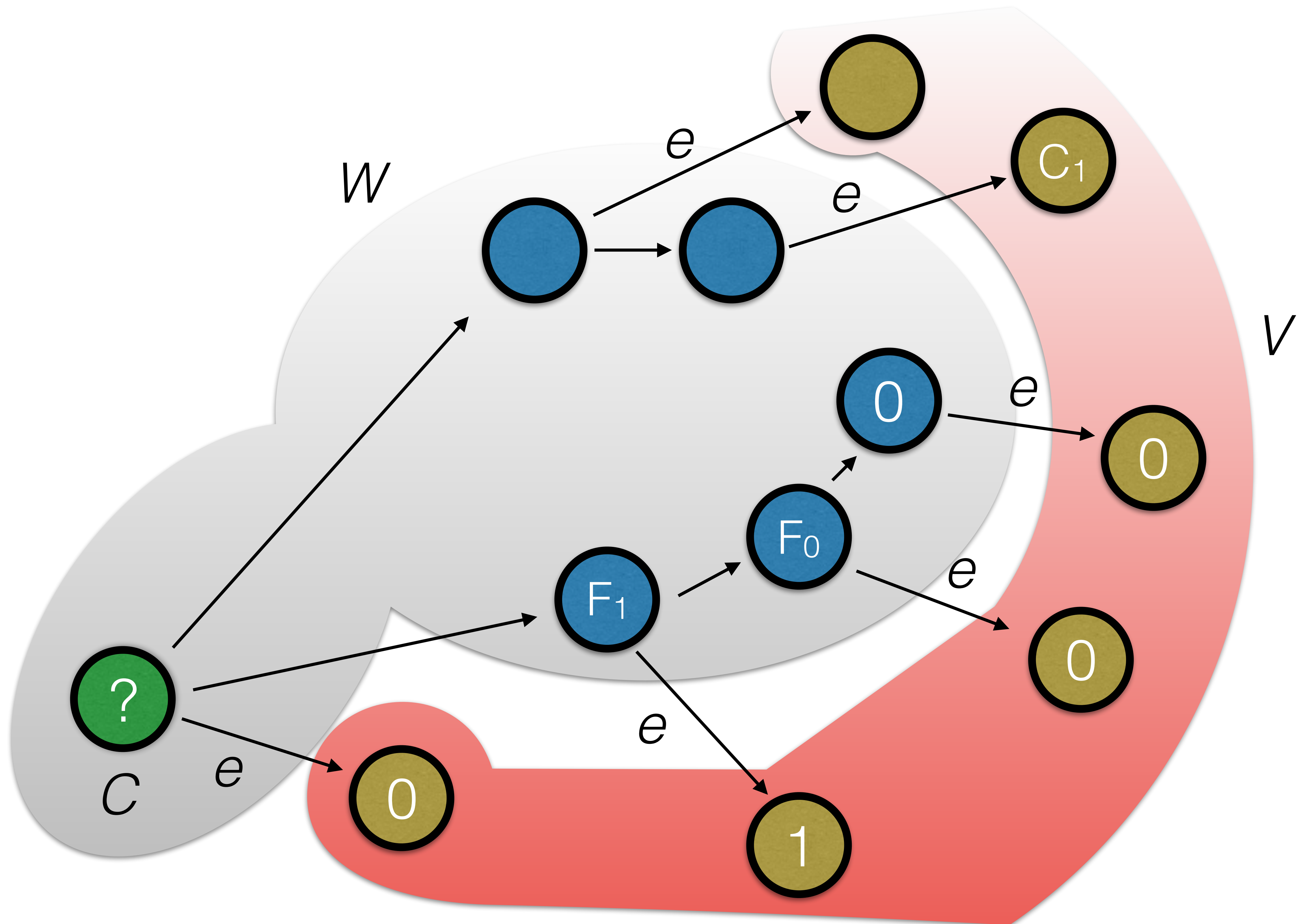
Claim 4



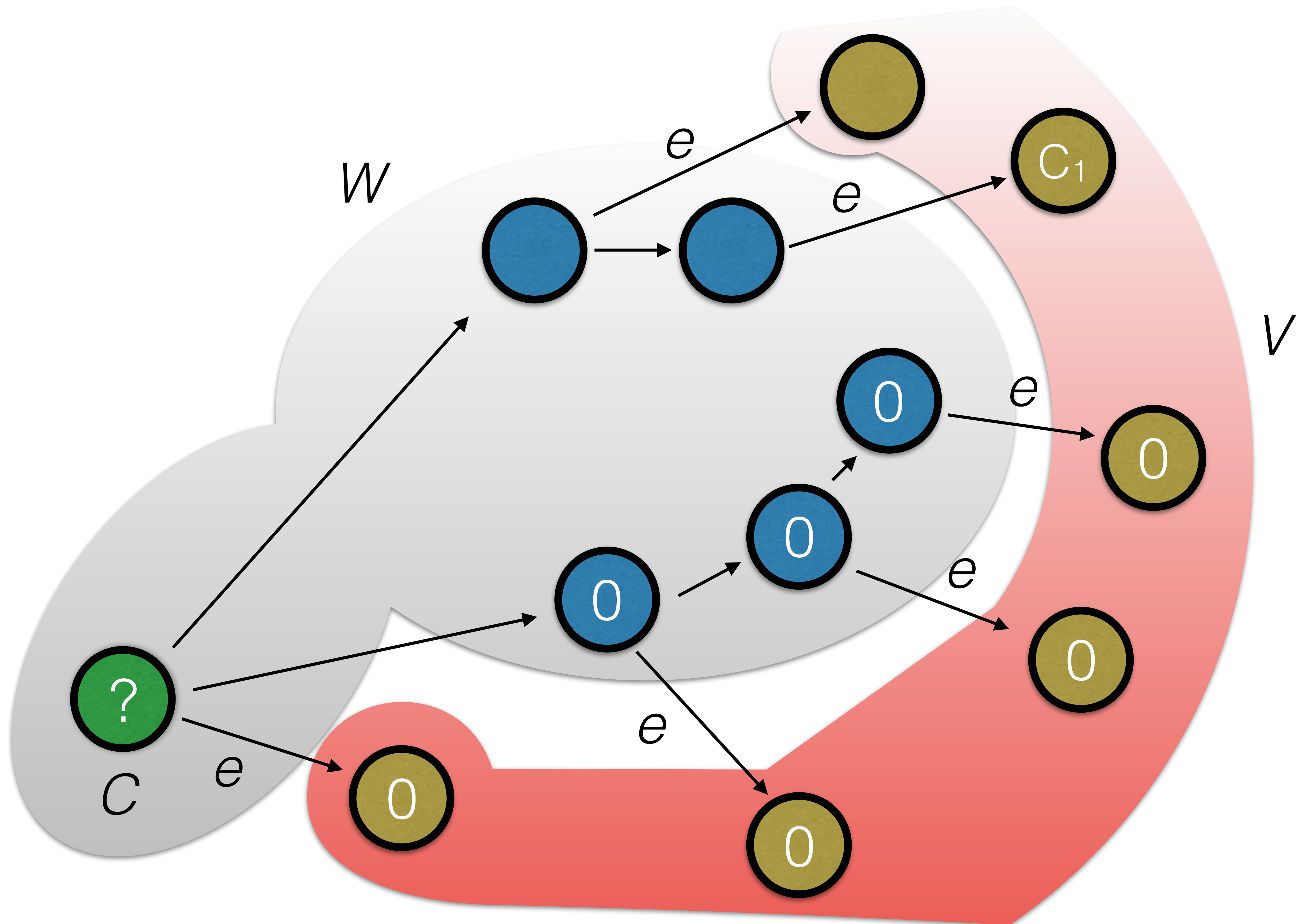
Claim 4



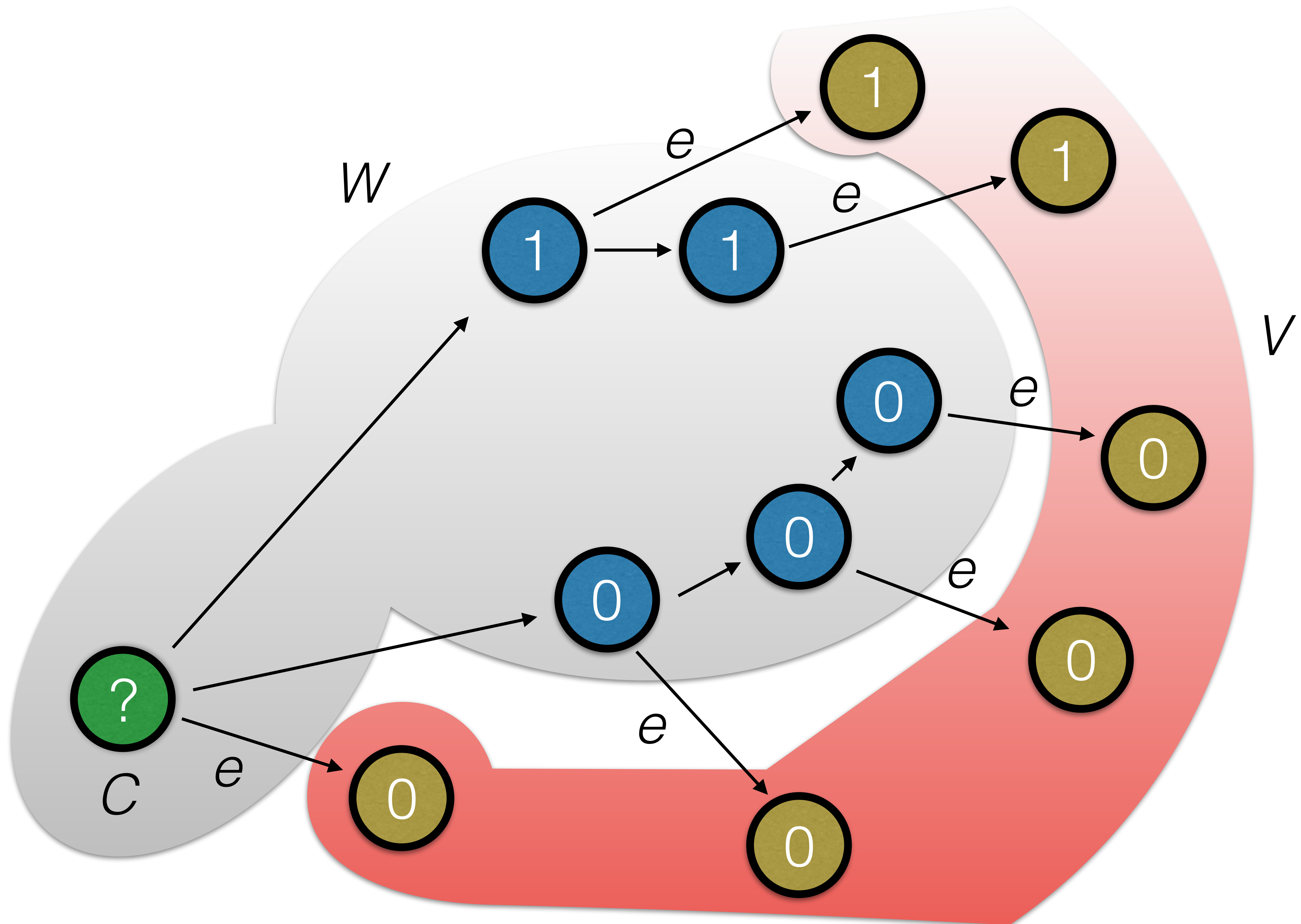
Claim 4



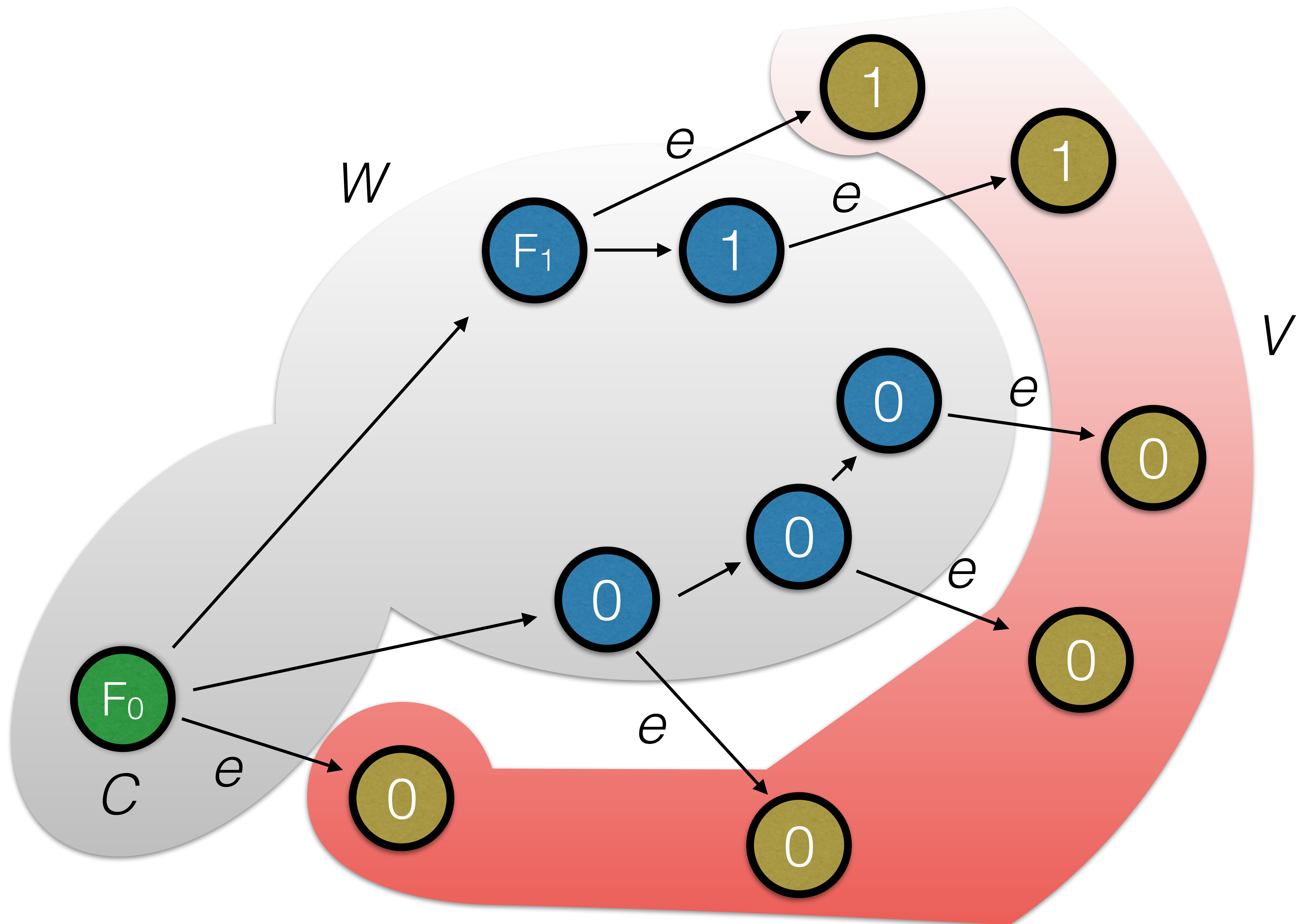
Claim 4



Claim 4



Claim 4



Lemma 3 Proof

- Consider F_0 and F_1 in W, such that $e(F_0) = C_0$ is 0-valent, $e(F_1) = C_1$ is 1-valent, and w.l.o.g. assume $F_1 = d(F_0)$ (by Claim 4)
- e and d must occur on the same process p because otherwise $C_1 = e(F_1) = e(d(F_0)) = d(C_0)$ will have a decision of 0 (by Commutativity Lemma)
- Consider all possible executions starting from configuration F_0. By termination requirement (and also to tolerate one process failure), there must be an execution where

  i) some process decides, and

  ii) process p does not execute any steps.

  Let the configuration immediately after some process decides be T where $T = \sigma(F_0)$ and σ does not contain any step by process p.
- We have $e(T) = e(\sigma(F_0)) = \sigma(e(F_0)) = \sigma(C_0)$ which is 0-valent (by Commutativity Lemma)
- We also have $e(d(T)) = e(d(\sigma(F_0))) = \sigma(e(d(F_0))) = \sigma(e(F_1)) = \sigma(C_1)$ which is 1-valent (by Commutativity Lemma)
- But some process has already decided in T. Regardless of whether the decision is 0 or 1, agreement can be violated. Contradiction.

Consequences of FLP

- There is no way to solve the consensus problem under a very minimal system model in a way that cannot be delayed forever
- Complete correctness if not possible in asynchronous models
- In practice, we may live with very low probability of disagreement (give up safety)
- In practice, we may live with very low probability of blocking (give up liveness)
- Two-phase commit or even three-phase commit can block forever
- This result is particularly relevant to people designing algorithms

CAP

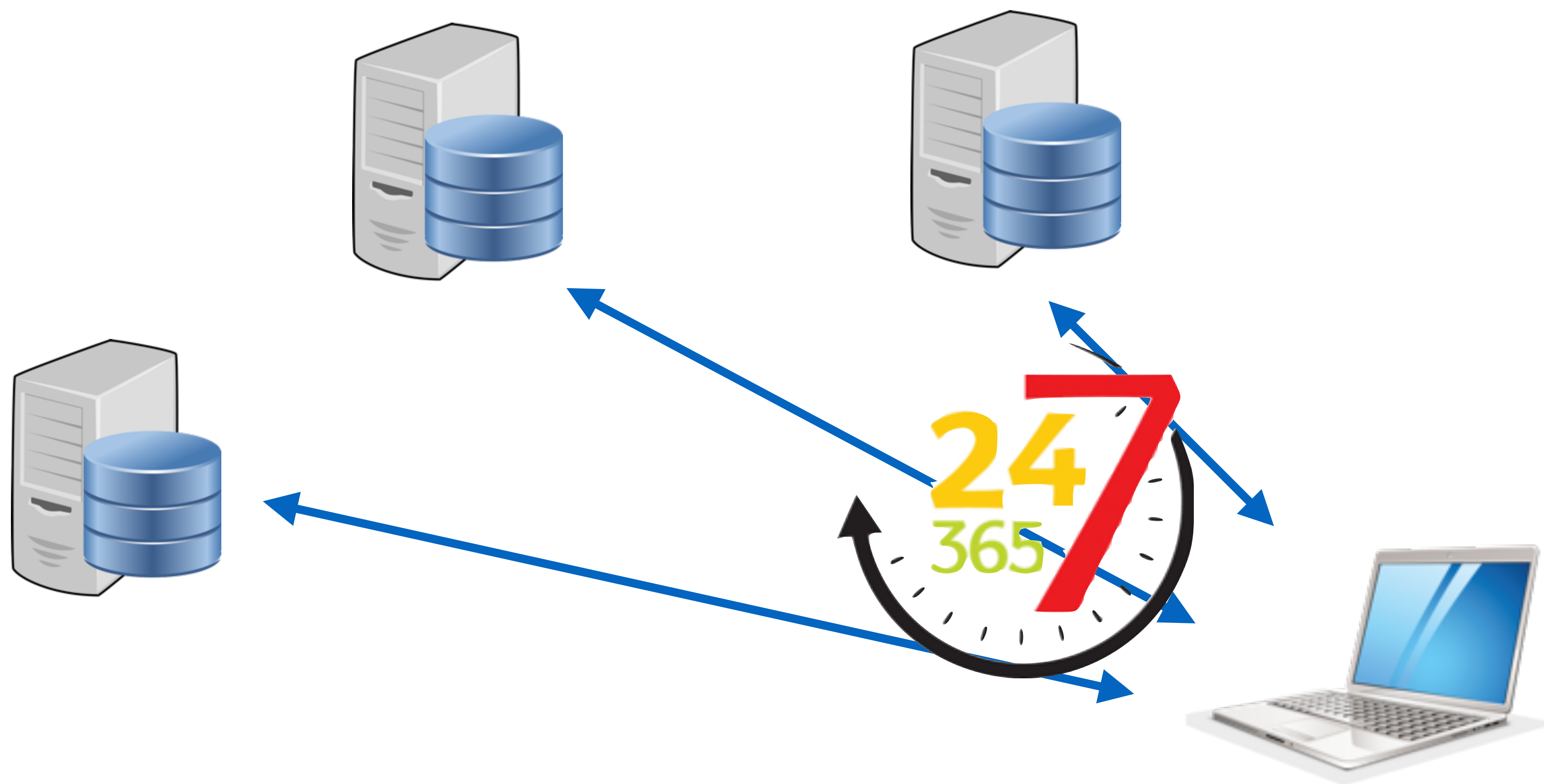
- Presented as Brewer's Conjecture in 2000
- Formalized and proved in **Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**, by Lynch and Gilbert (2002)
- **Consistency**, **availability** and **partition-tolerance** cannot be achieved all at the same time 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.

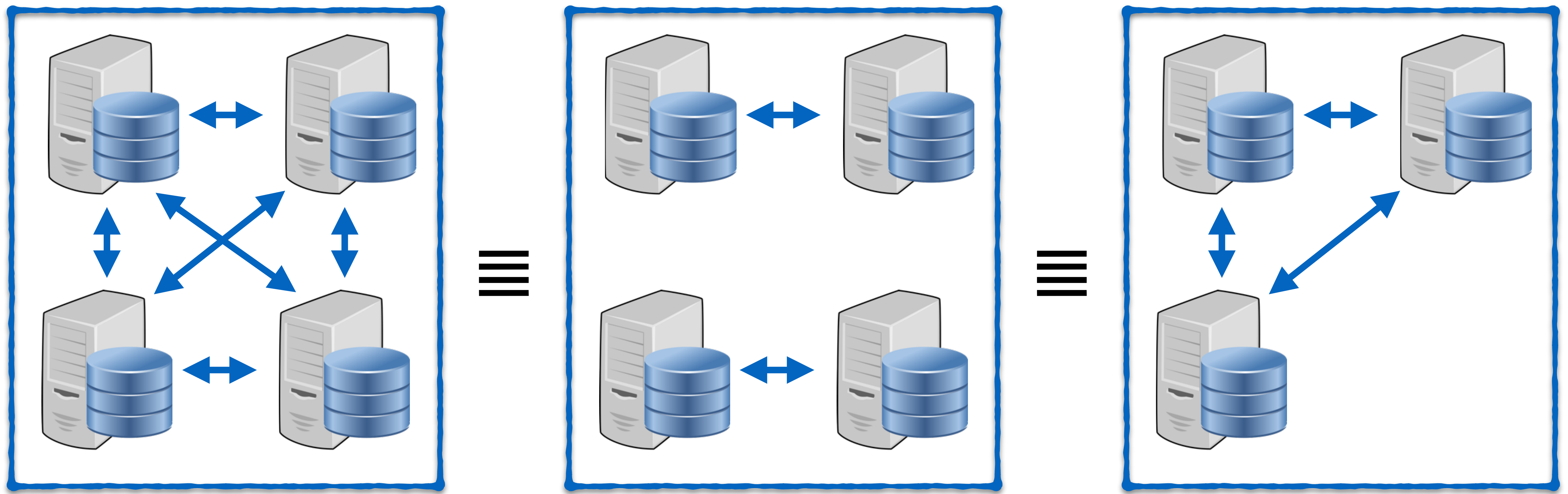
Availability

- 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

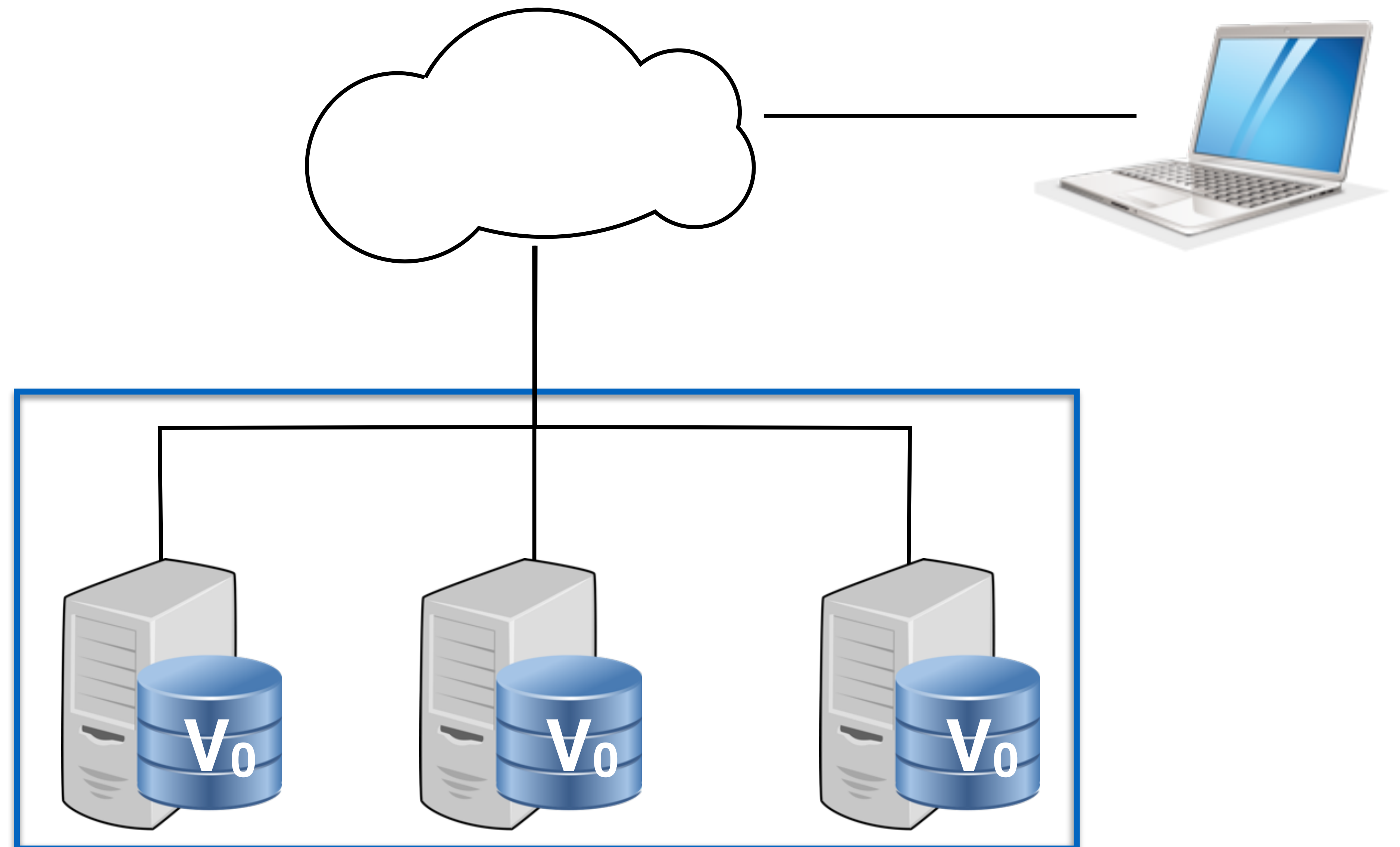
in all fair executions (including those in which messages are lost).

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

No partition

Proof 1 Sketch

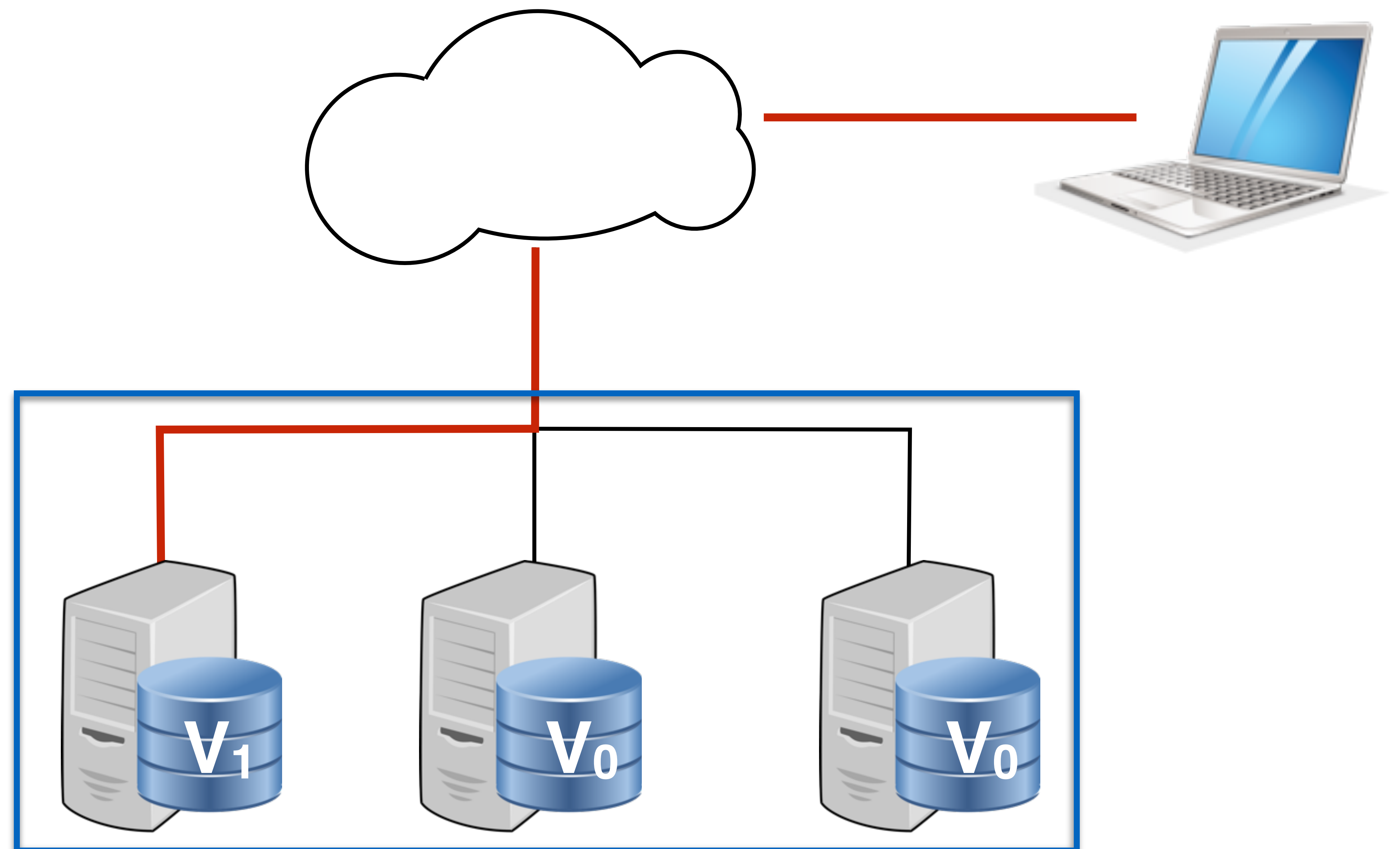
- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination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



No partition

Proof 1 Sketch

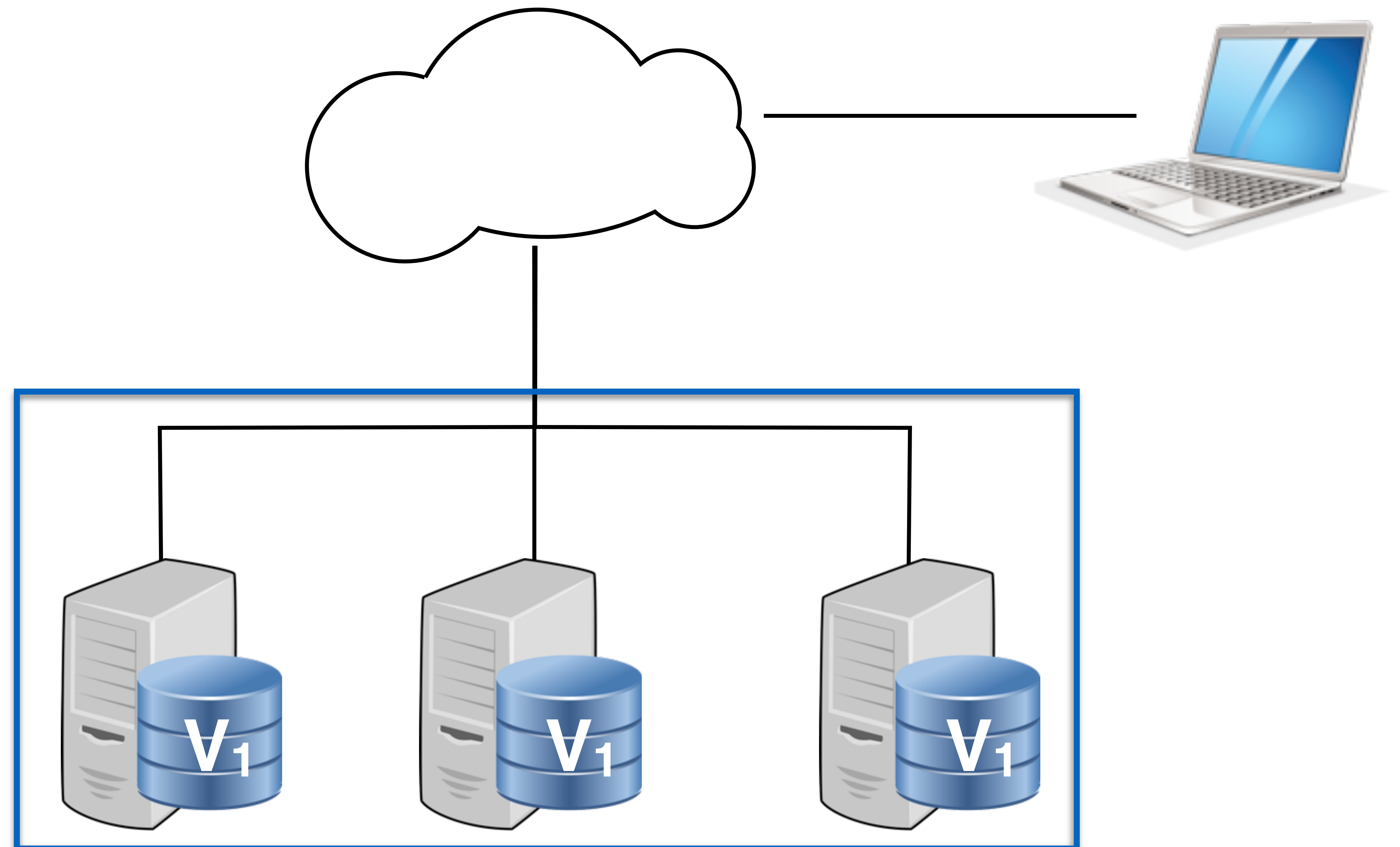
- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination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



No partition

Proof 1 Sketch

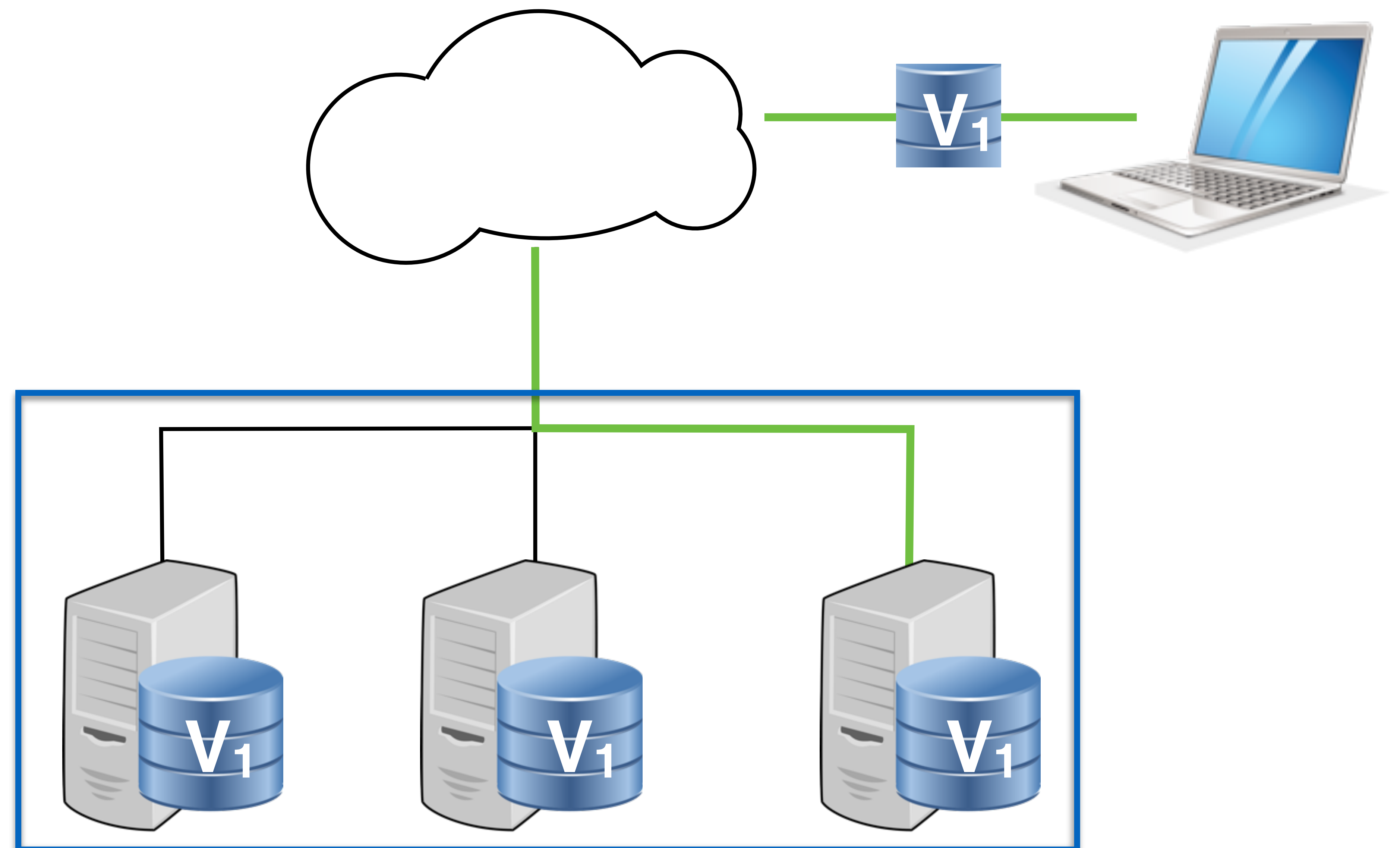
- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination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



No partition

Proof 1 Sketch

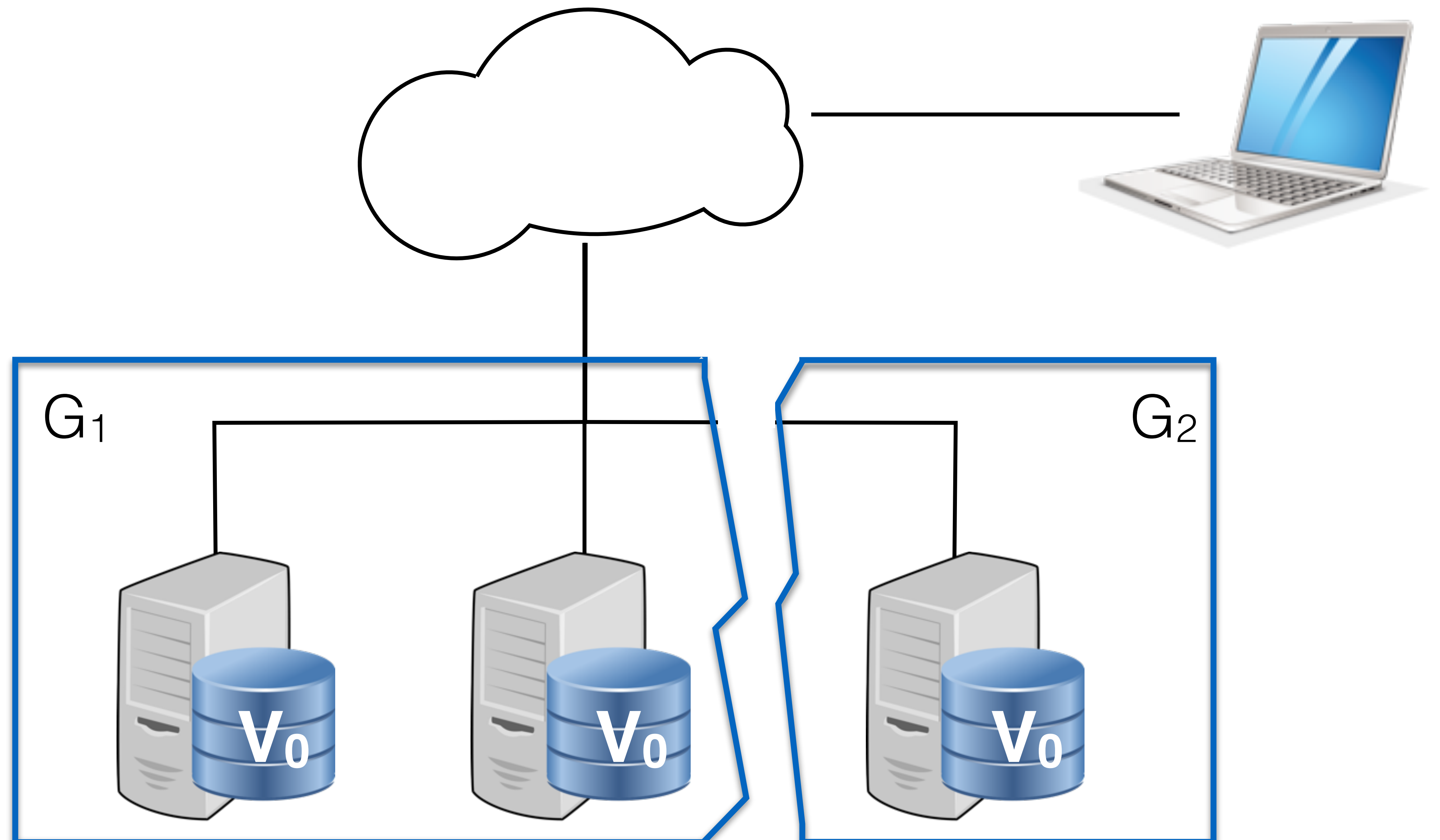
- Let v_0 be the initial value of an atomic object.
- A single *write* of a value not equal to v_0 occurs. Assume that no other client requests occur.
- We know that this write completes, by the availability requirement.
- A single *read* occurs, and no other client requests occur, ending with the termination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



No consistency

Proof 1 Sketch

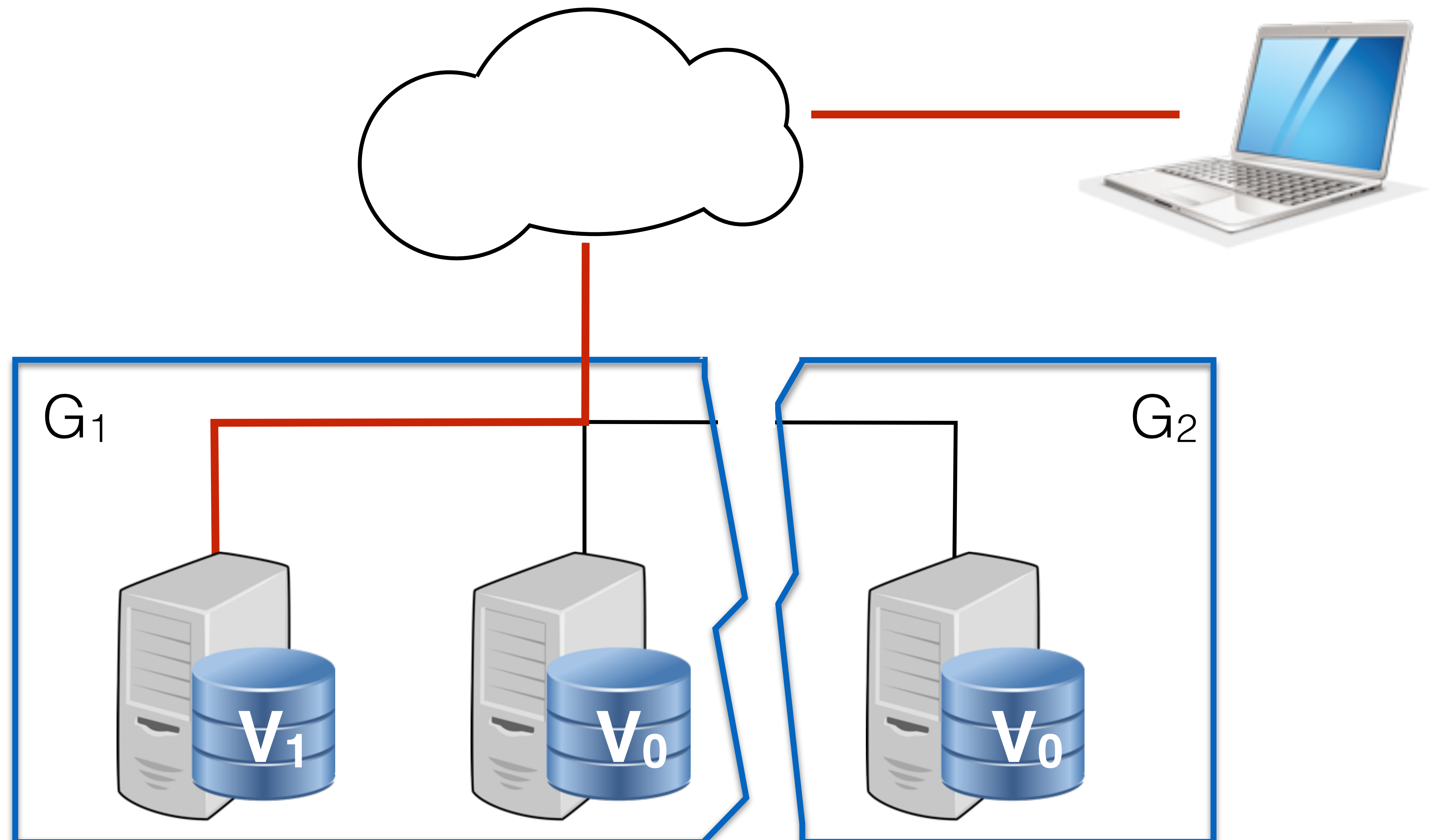
- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write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



No consistency

Proof 1 Sketch

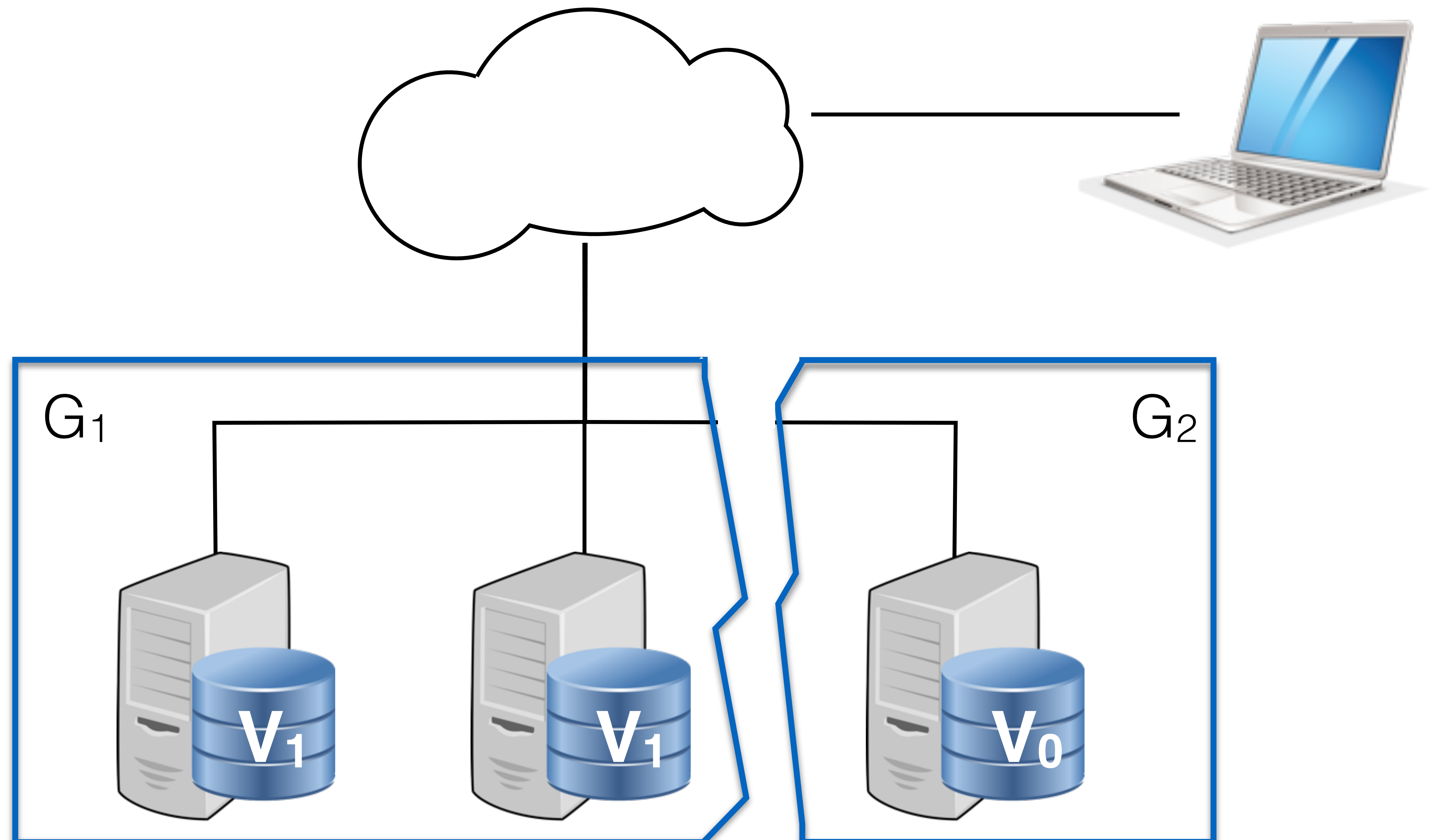
- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write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



No consistency

Proof 1 Sketch

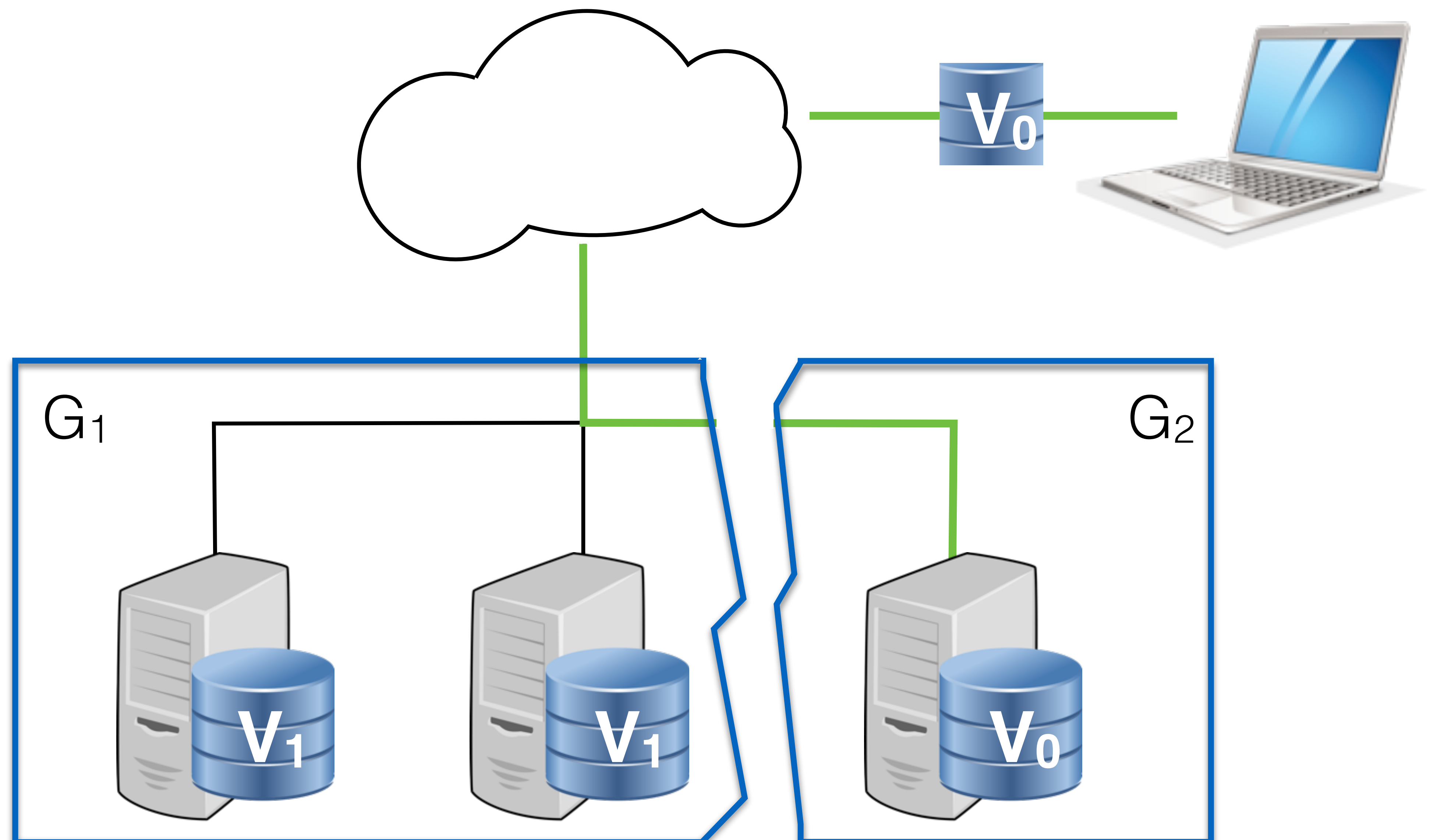
- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write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



No consistency

Proof 1 Sketch

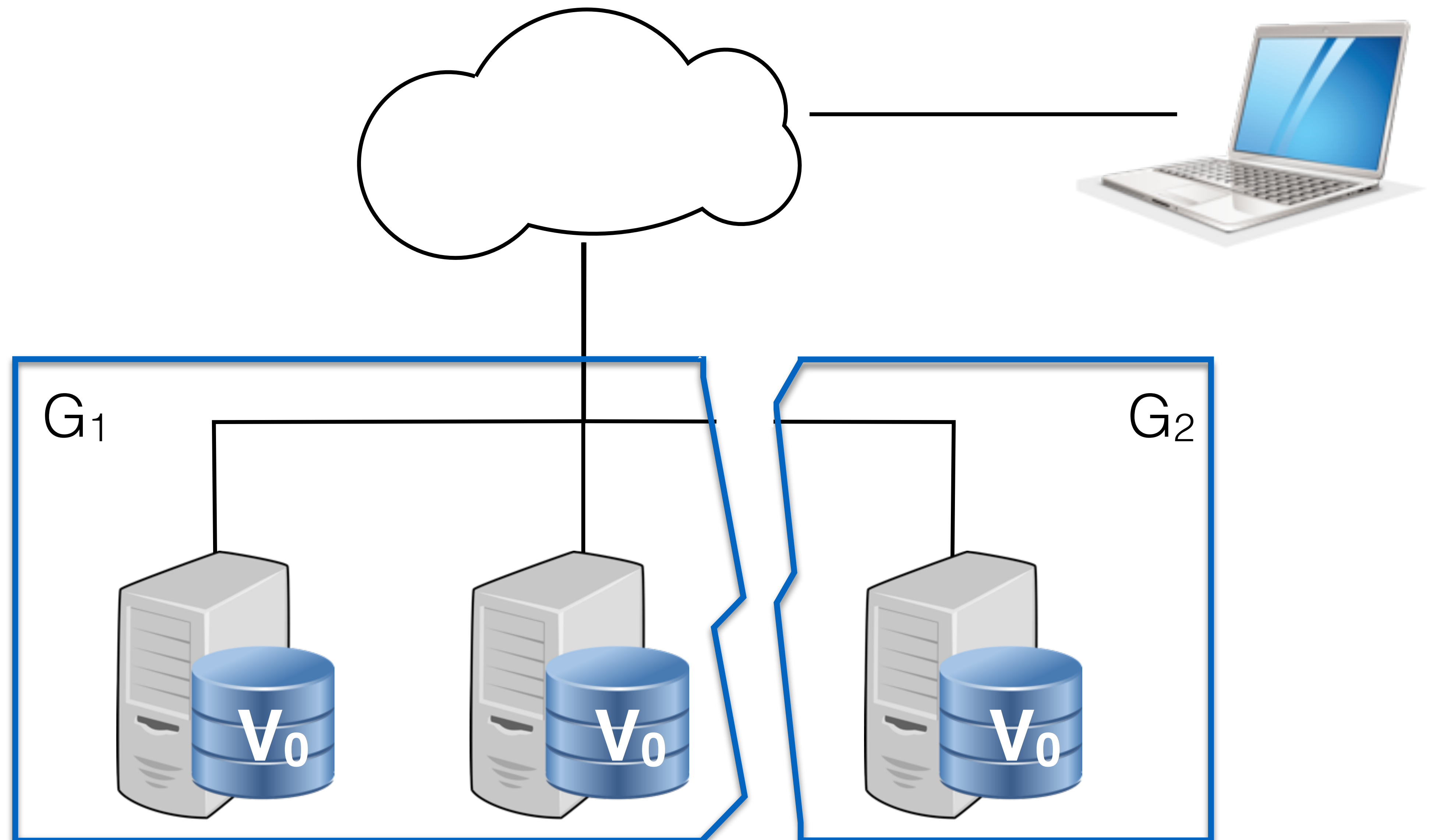
- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1, G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- We know that this write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



No availability

Proof 1 Sketch

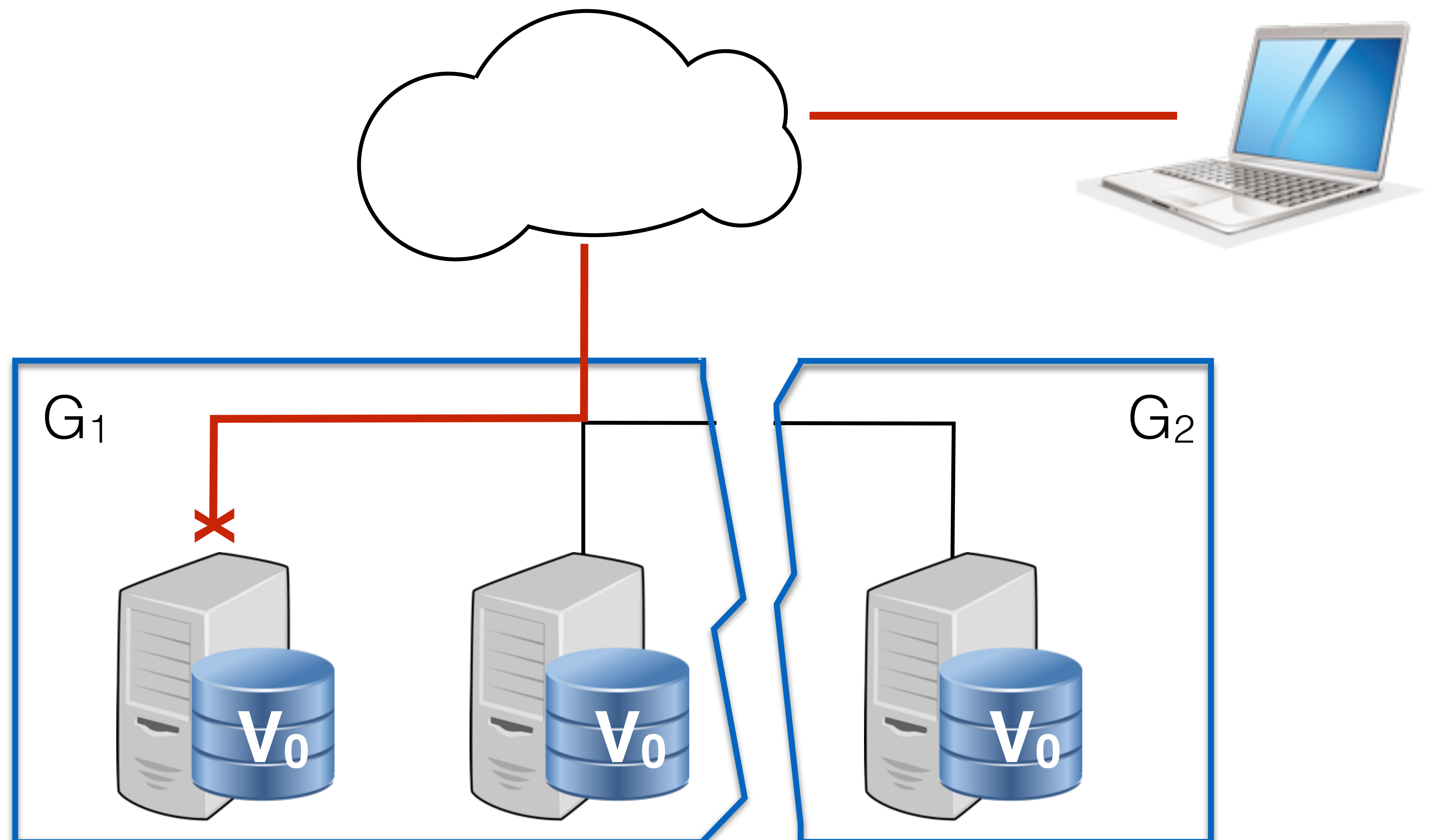
- Let v_0 be the initial value of an atomic object.
- Assume the network is divided into two disjoint sets $\{G_1,G_2\}$ and that all messages between G_1 and G_2 are lost.
- A single *write* of a value not equal to v_0 occurs in G_1. Assume that no other client requests occur.
- The write operation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



No availability

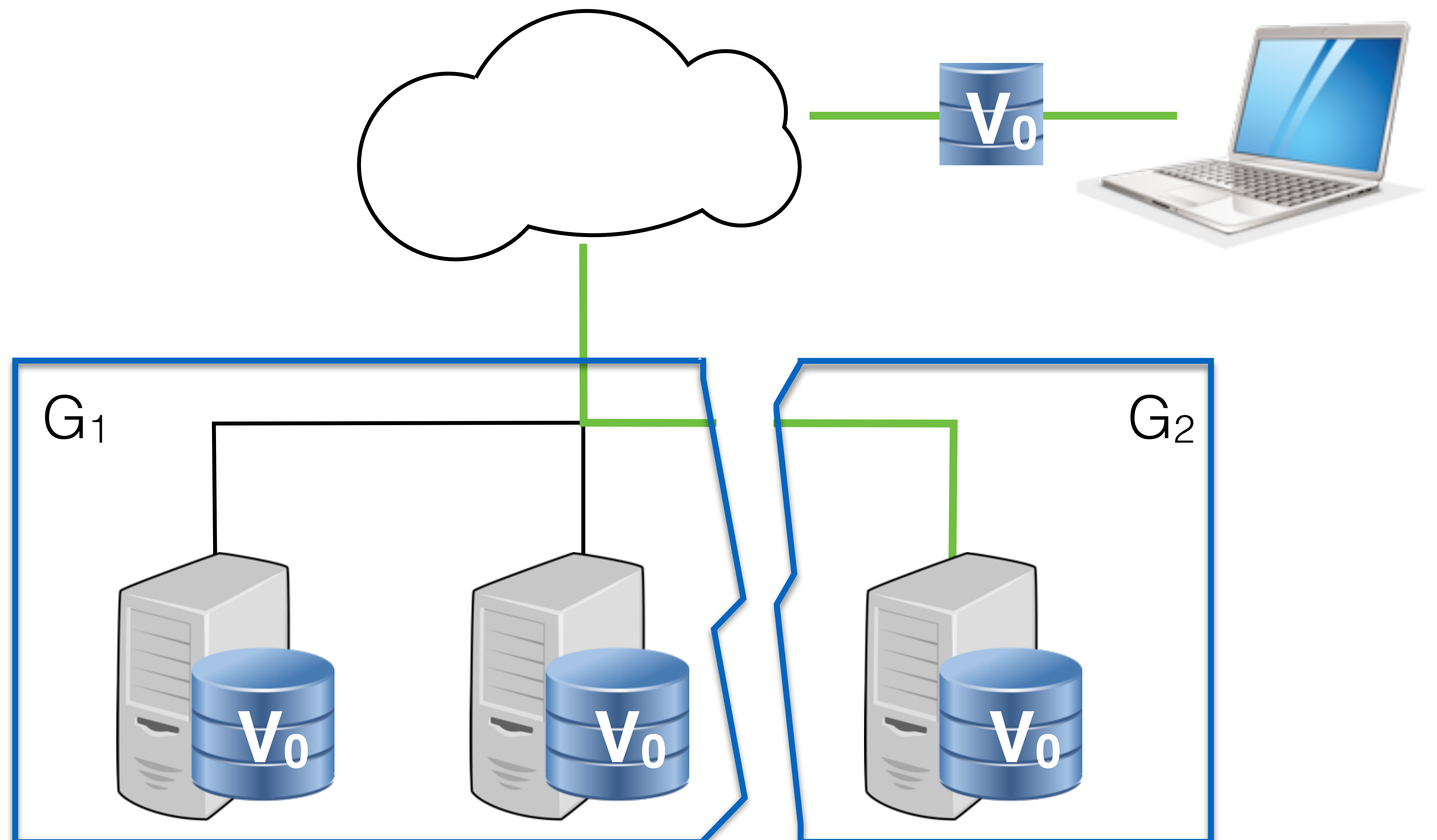
Proof 1 Sketch

- 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.

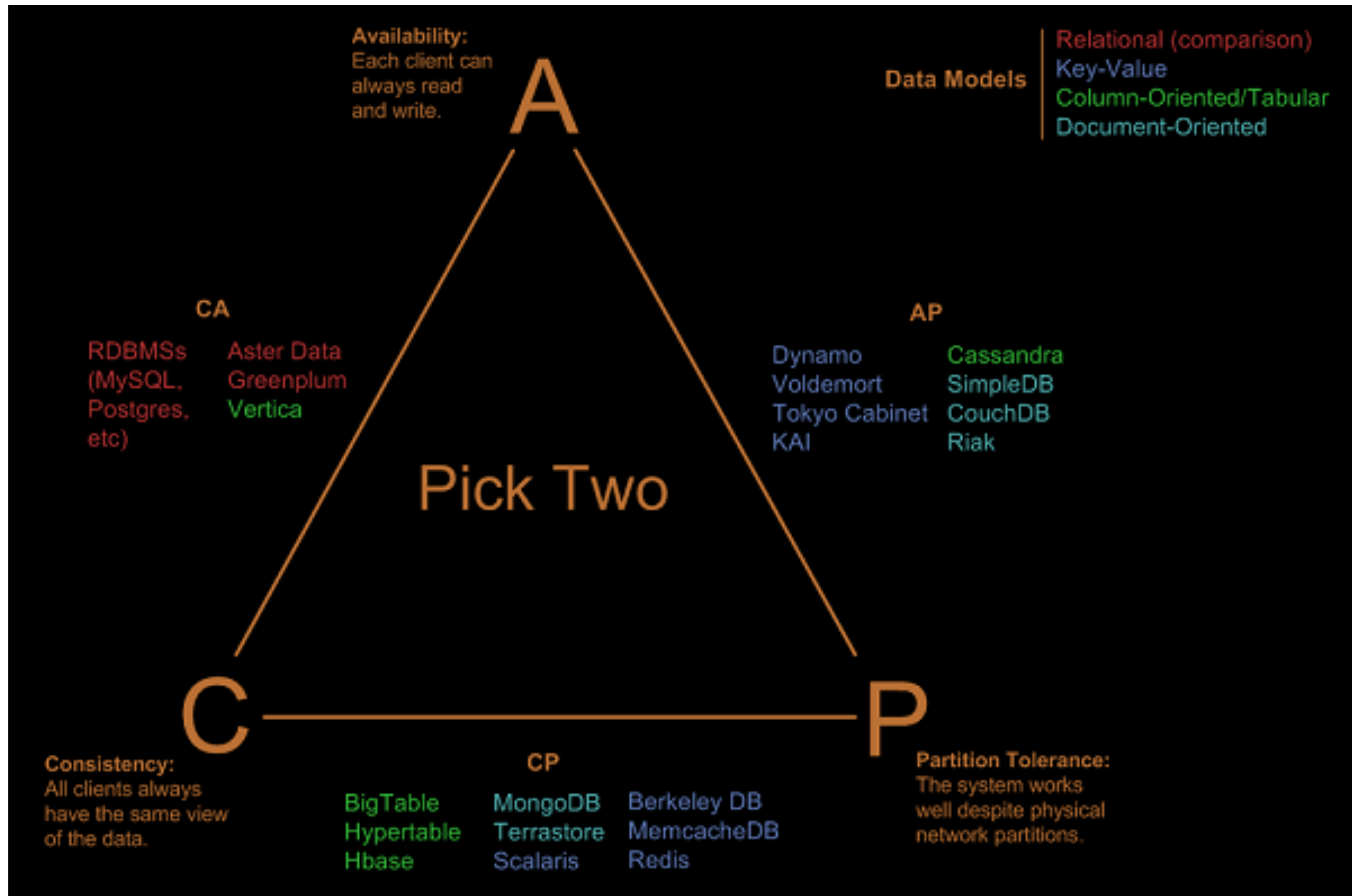


Proof 1 Sketch

- 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.