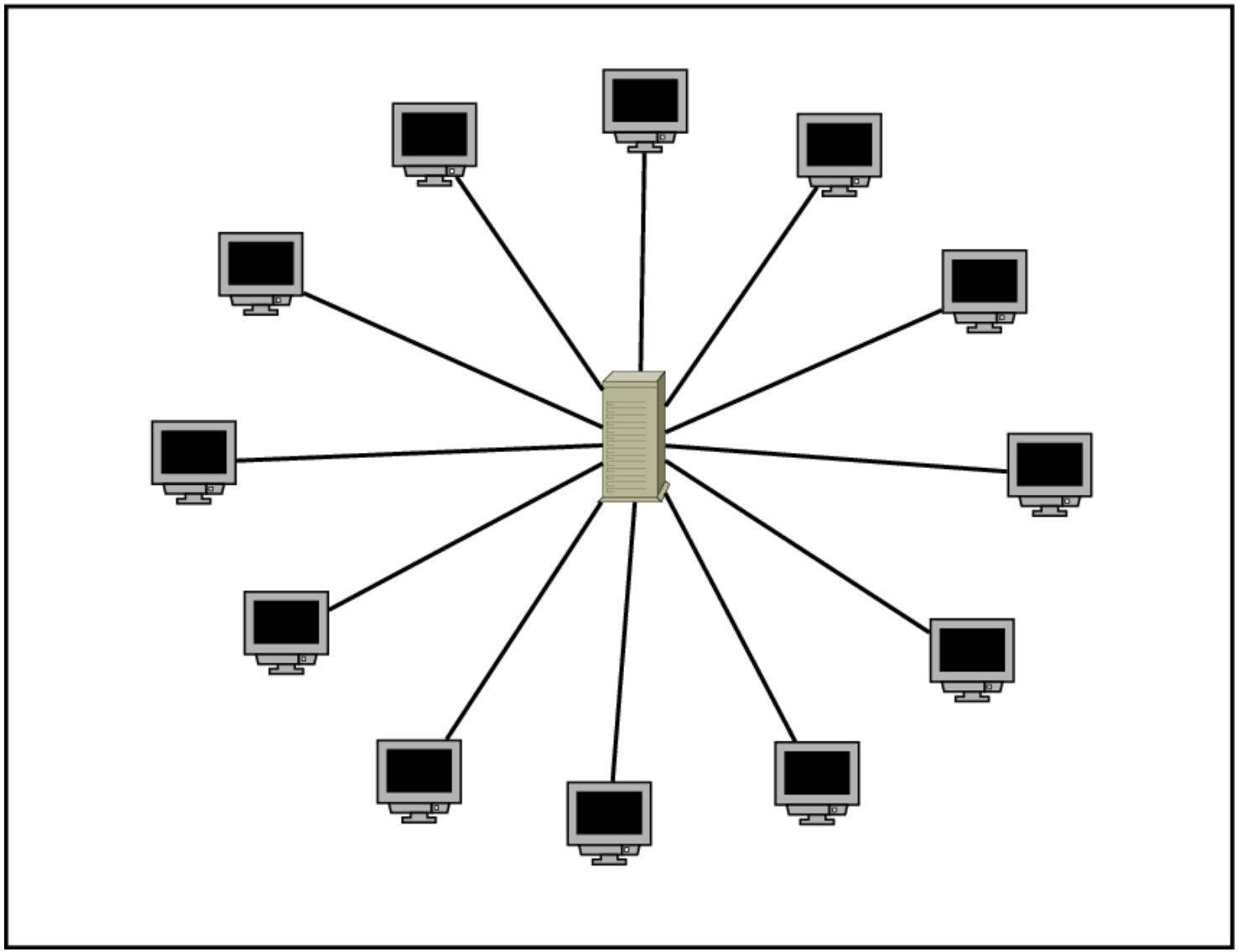
Self-Management of Large Scale Distributed Systems

Boriss Mejías

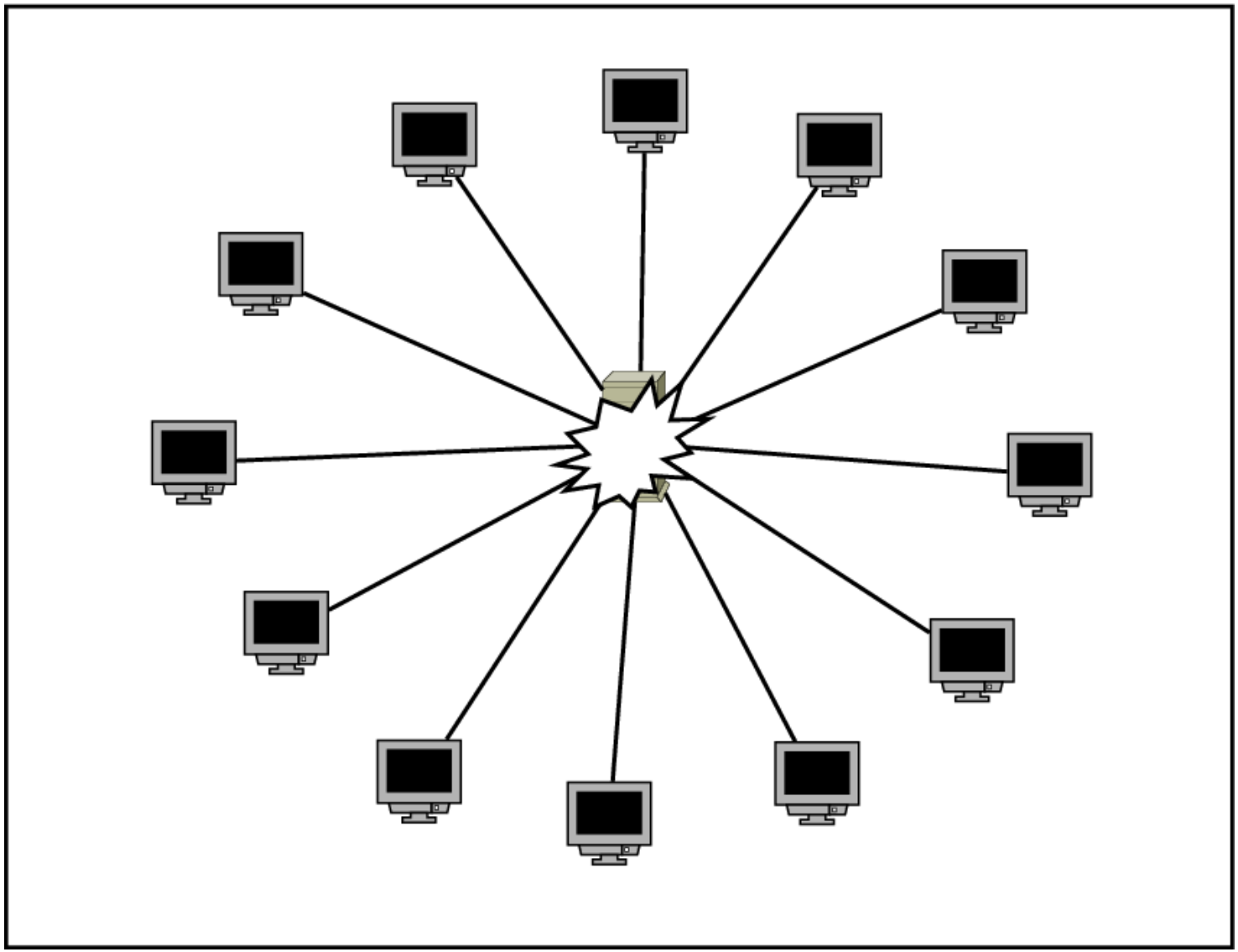
Université catholique de Louvain,
Louvain-la-Neuve, Belgium
boriss.mejias@uclouvain.be

18^{th} March, 2009

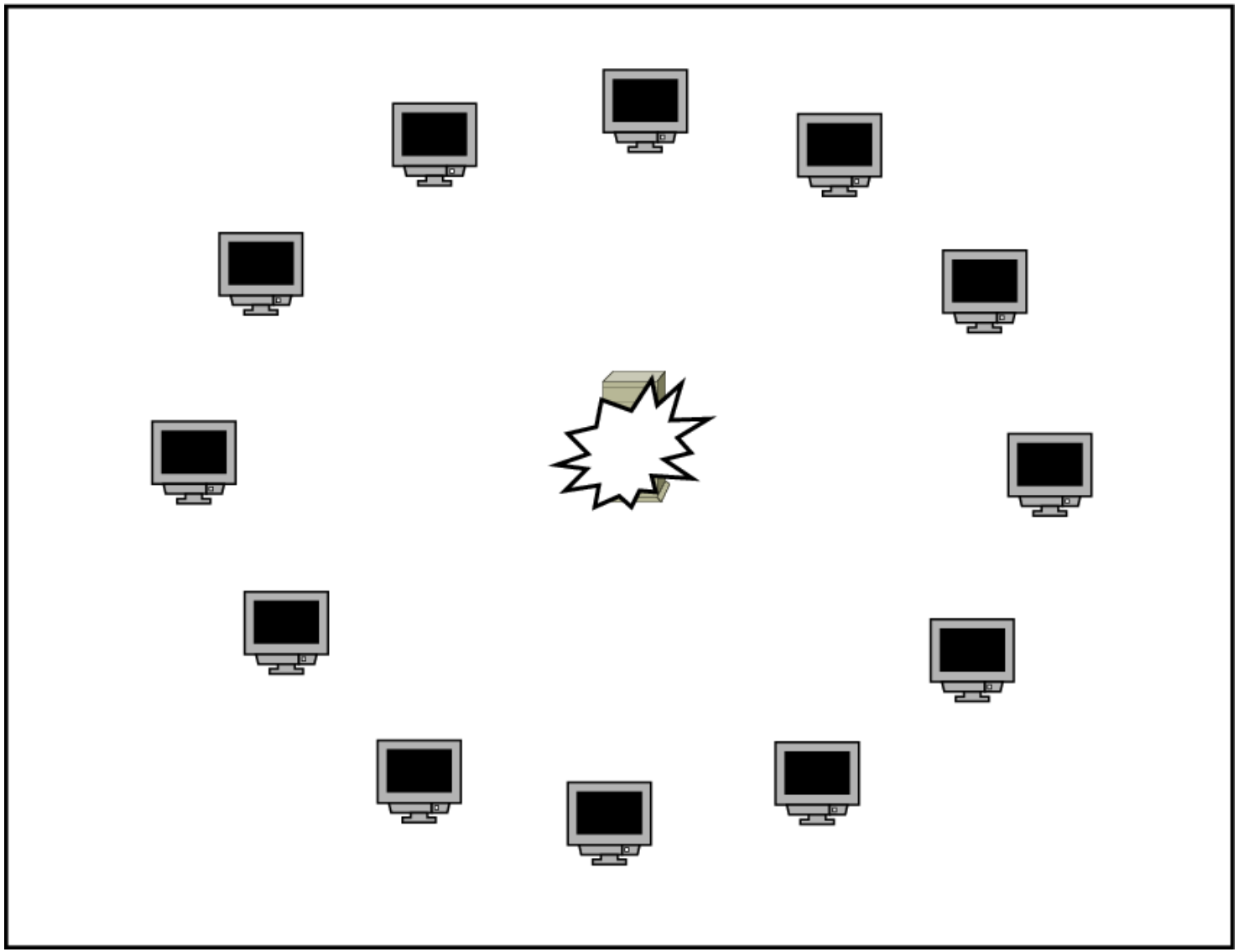
Motivation



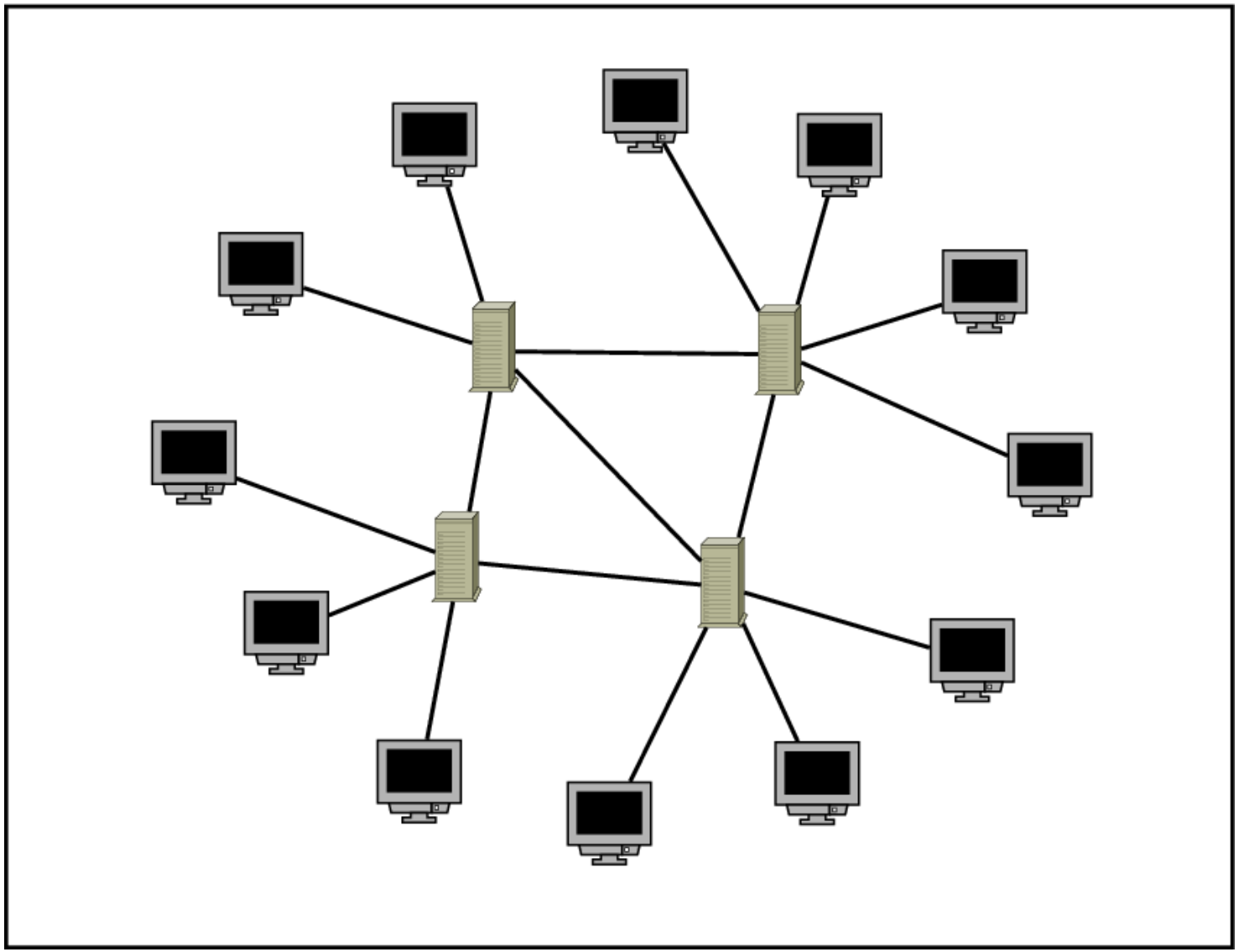
Motivation



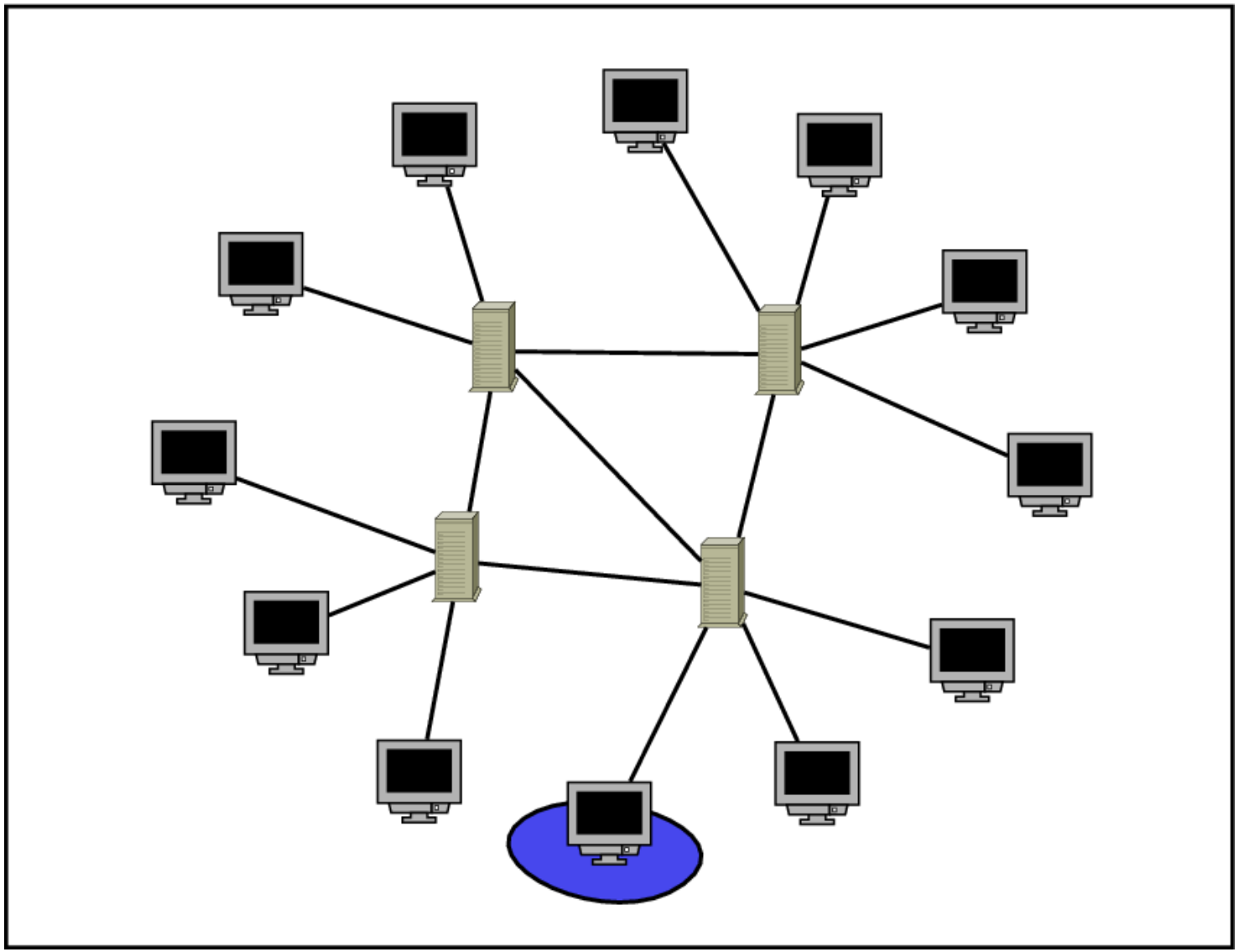
Motivation



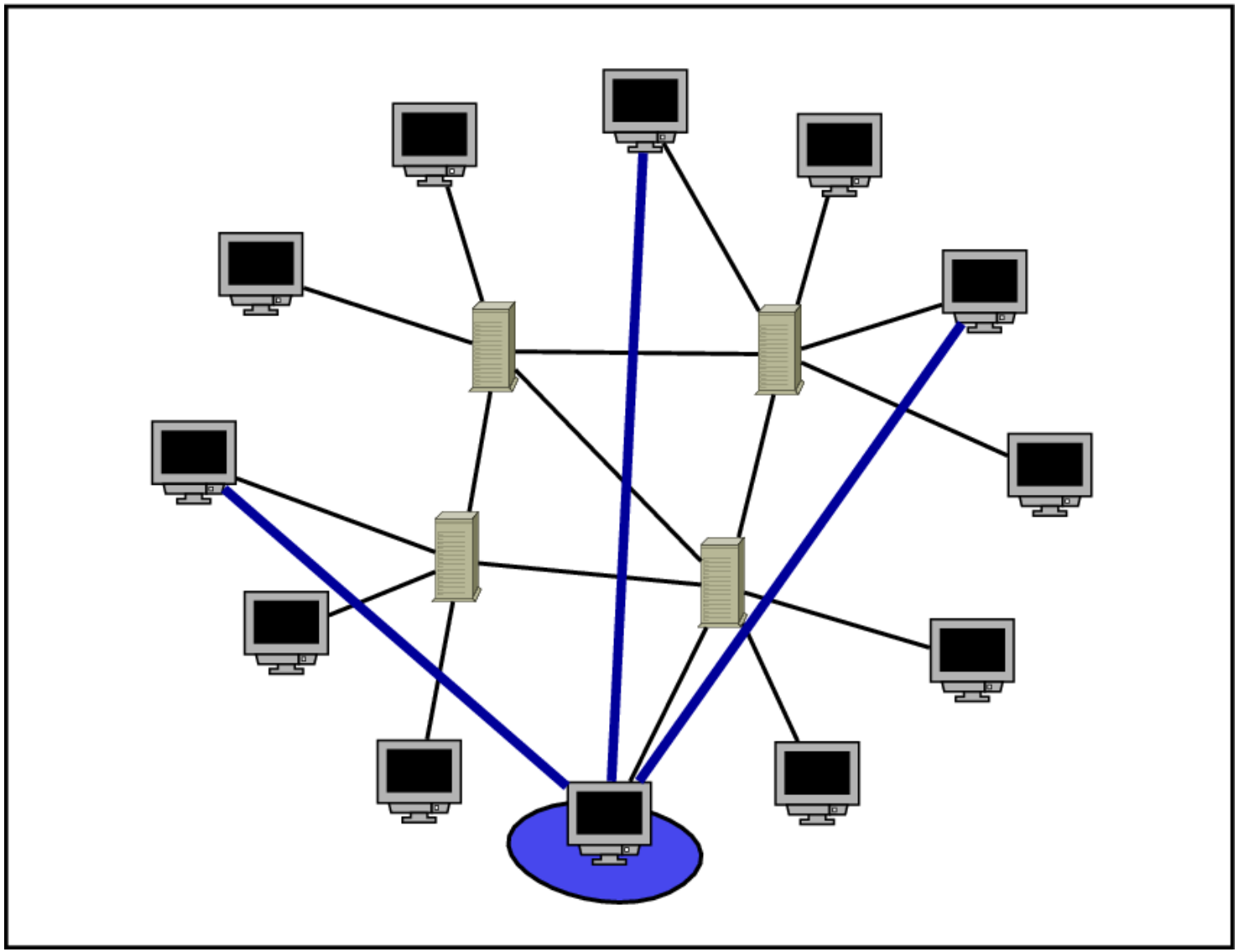
Motivation



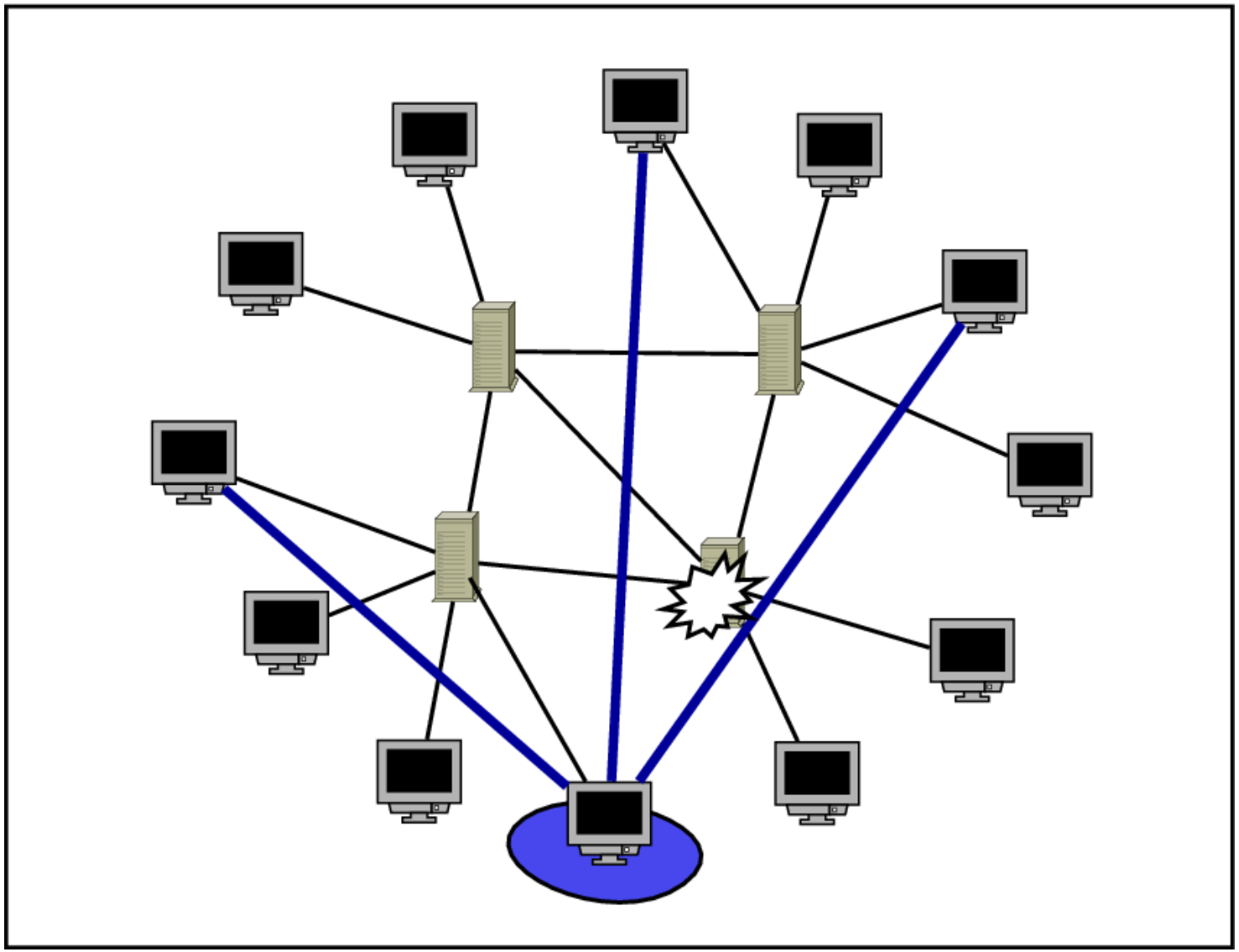
Motivation



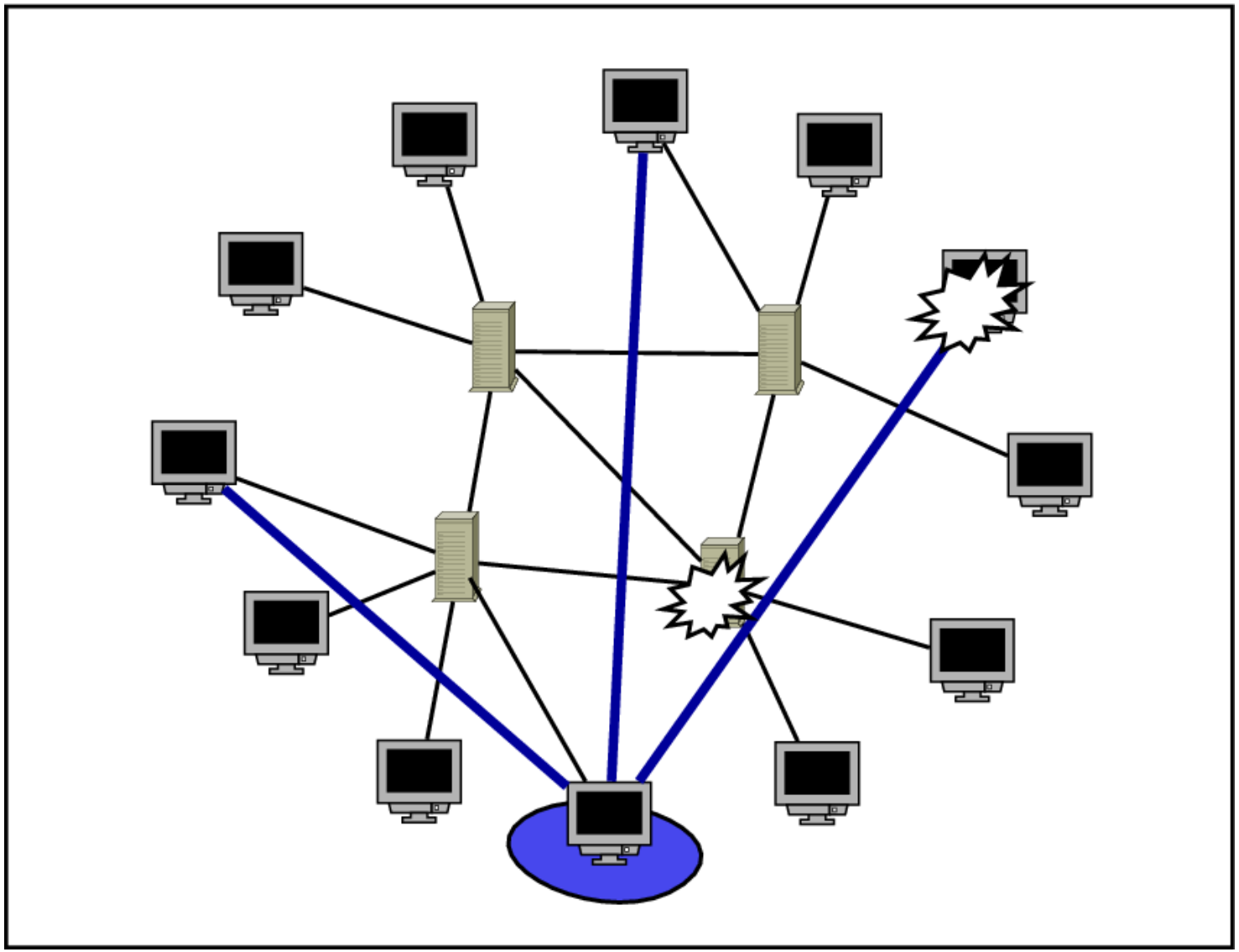
Motivation



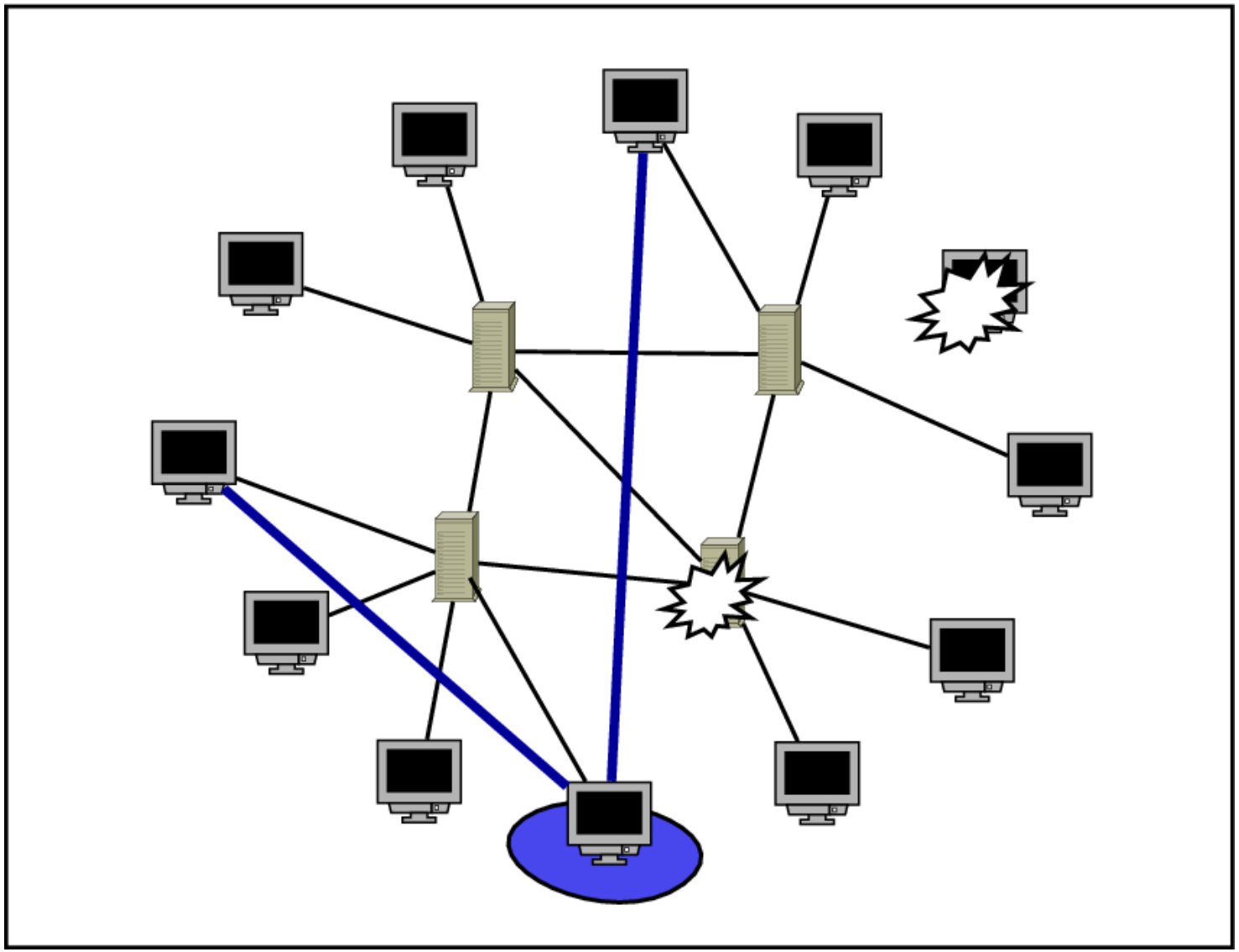
Motivation



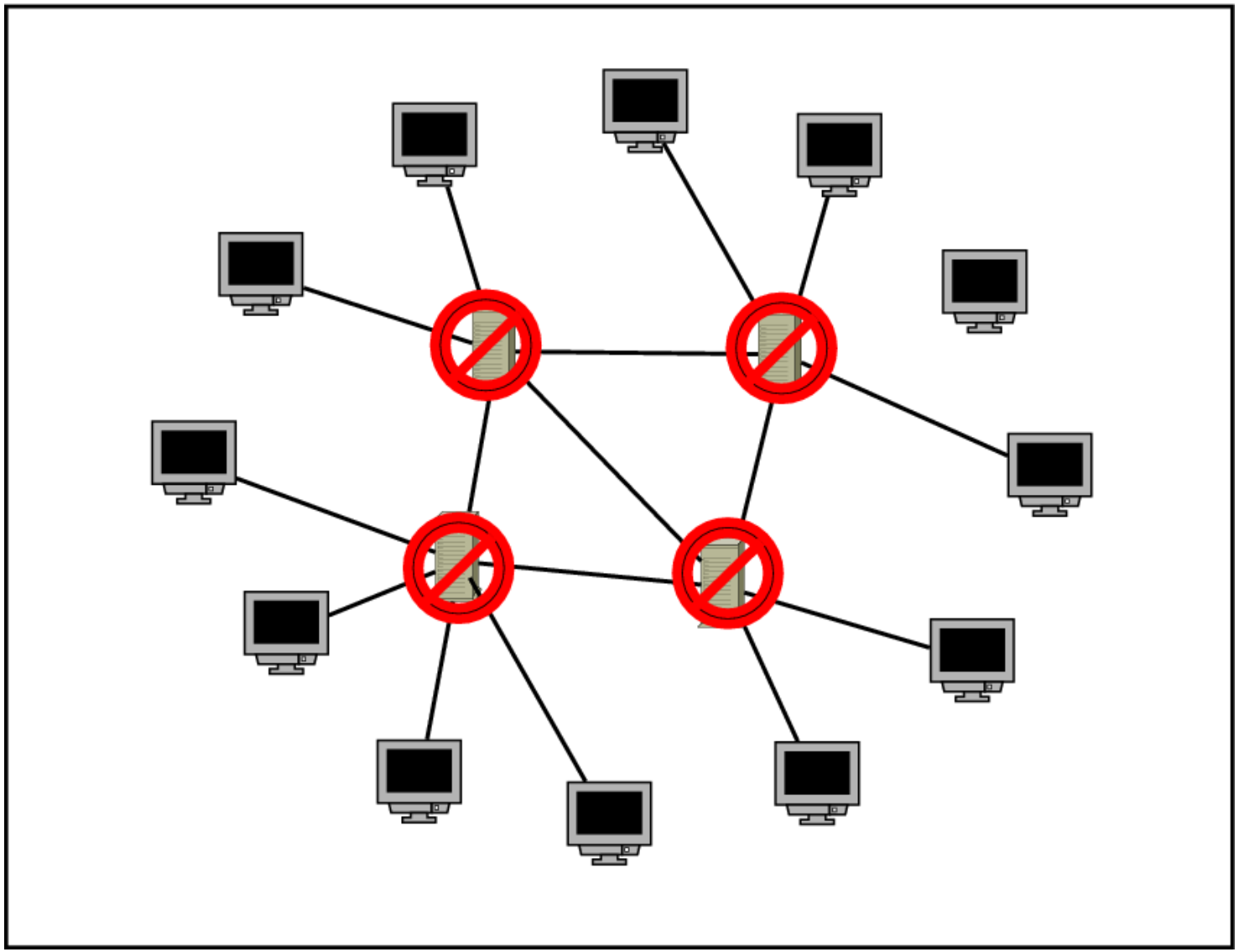
Motivation



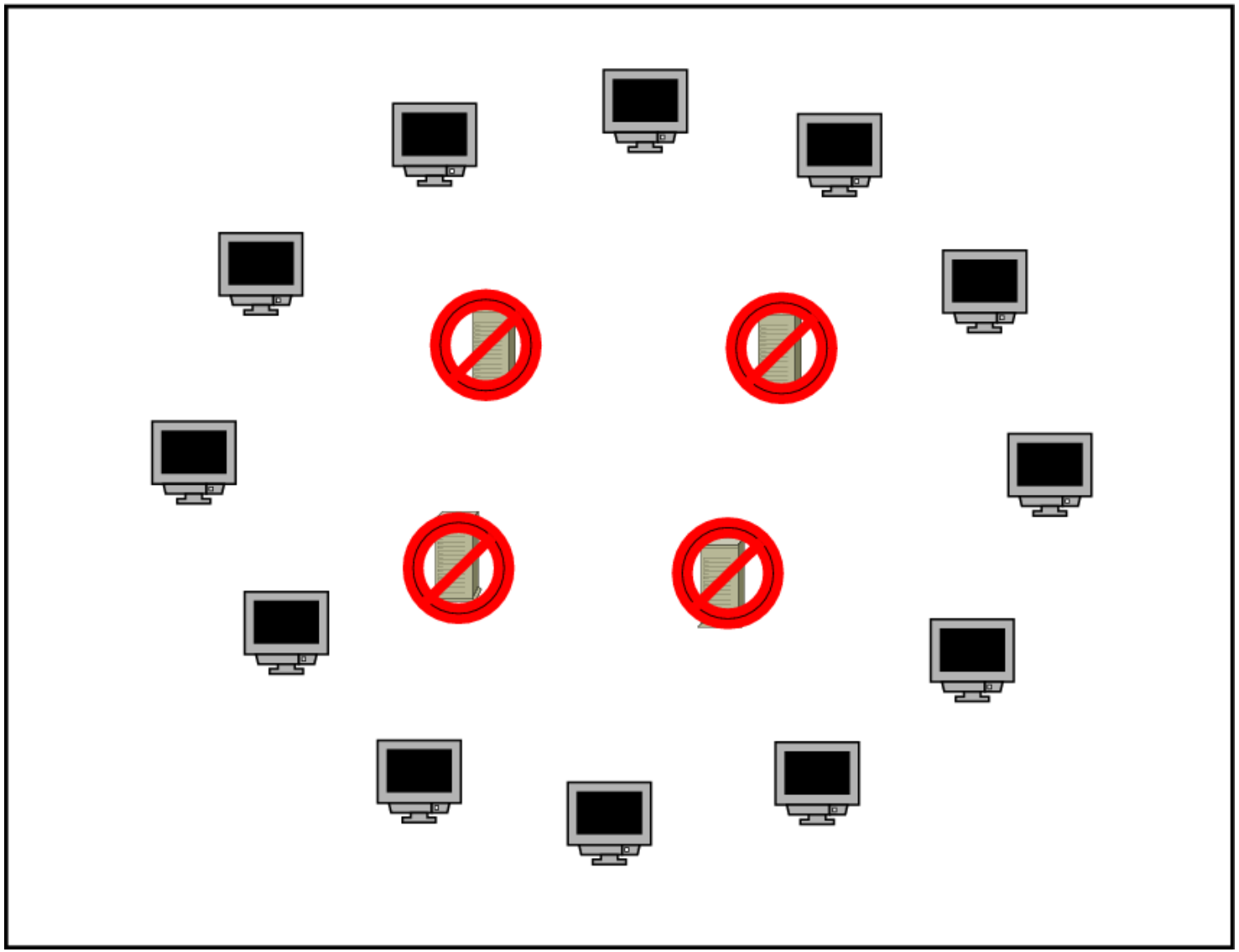
Motivation



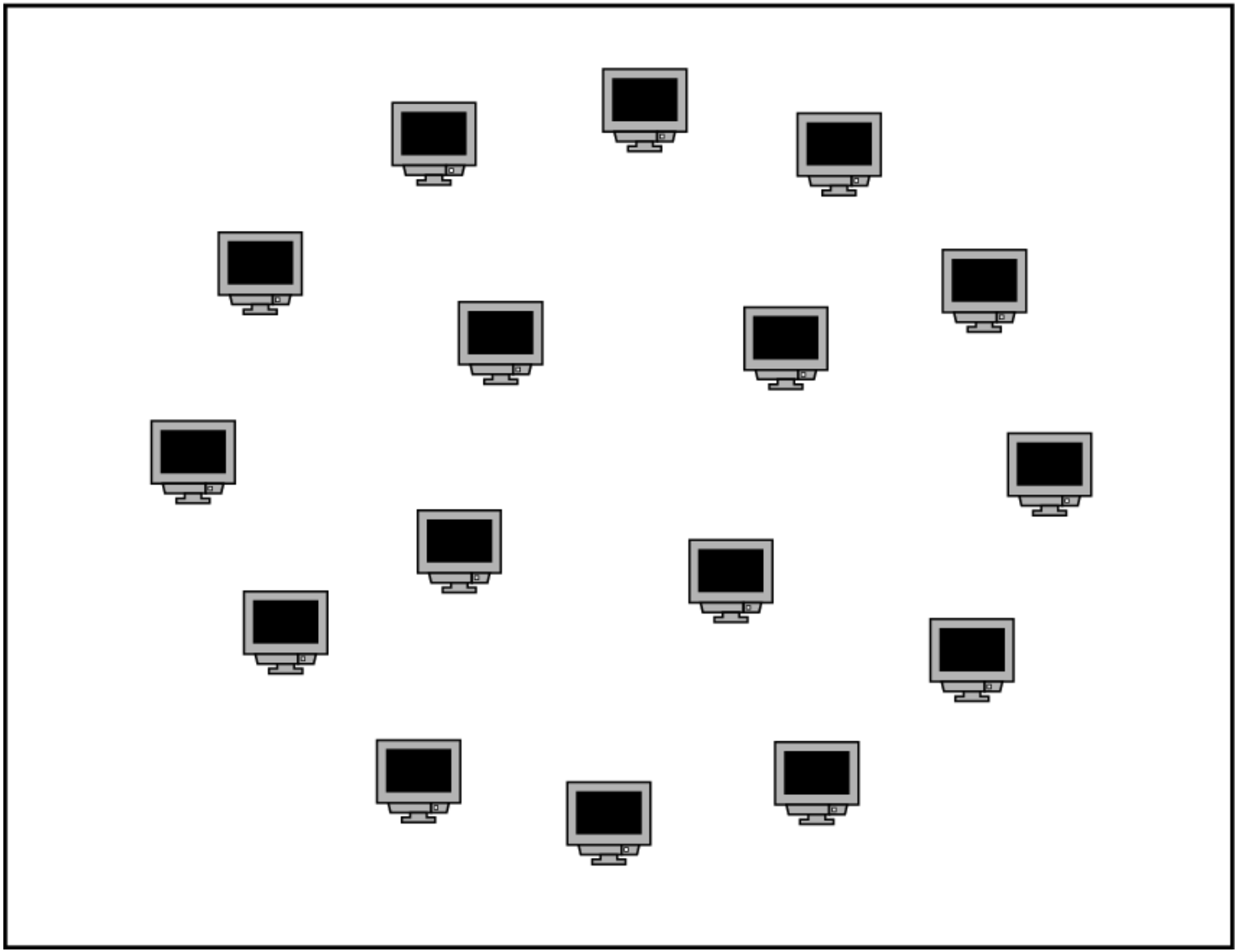
Motivation



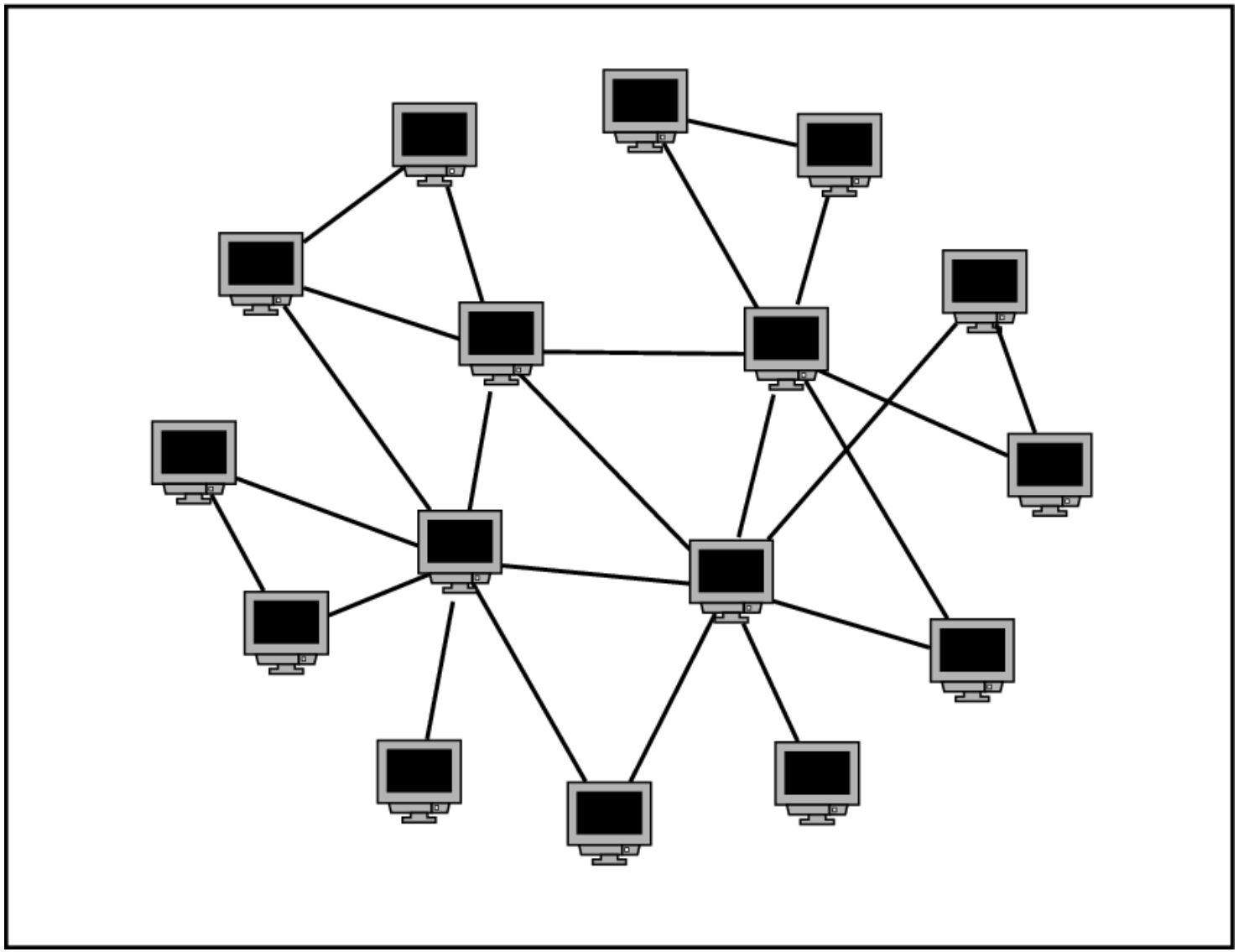
Motivation



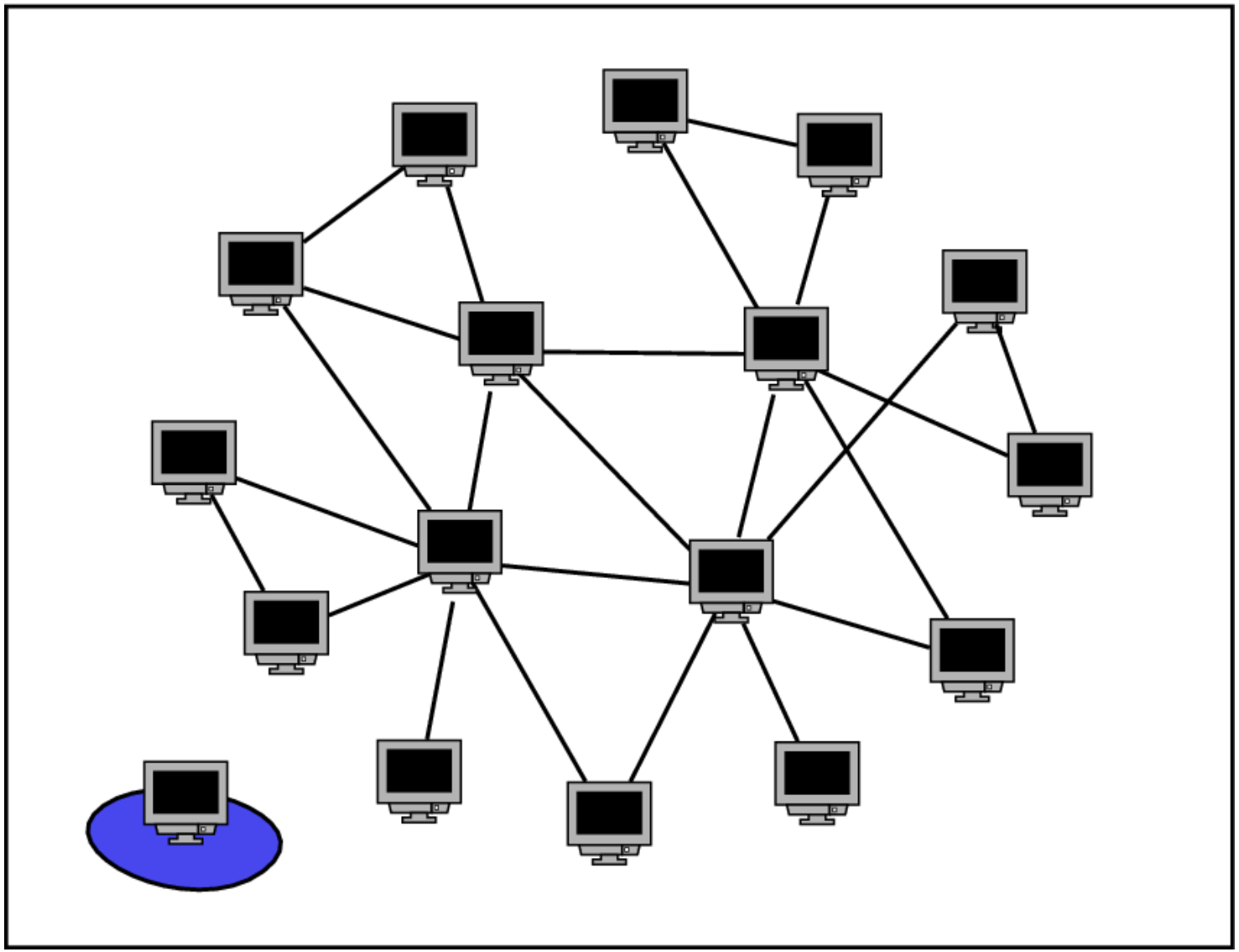
Motivation



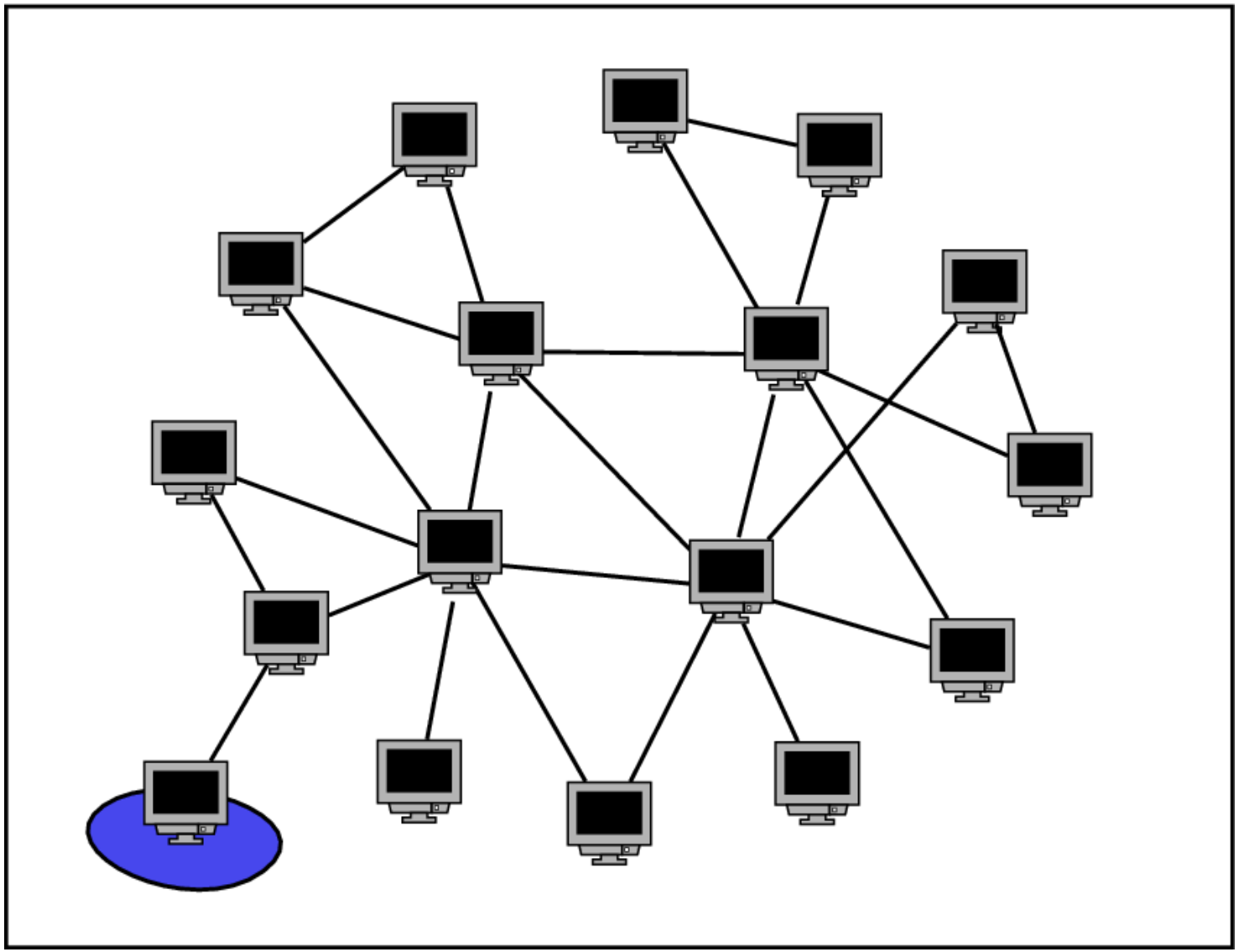
Motivation



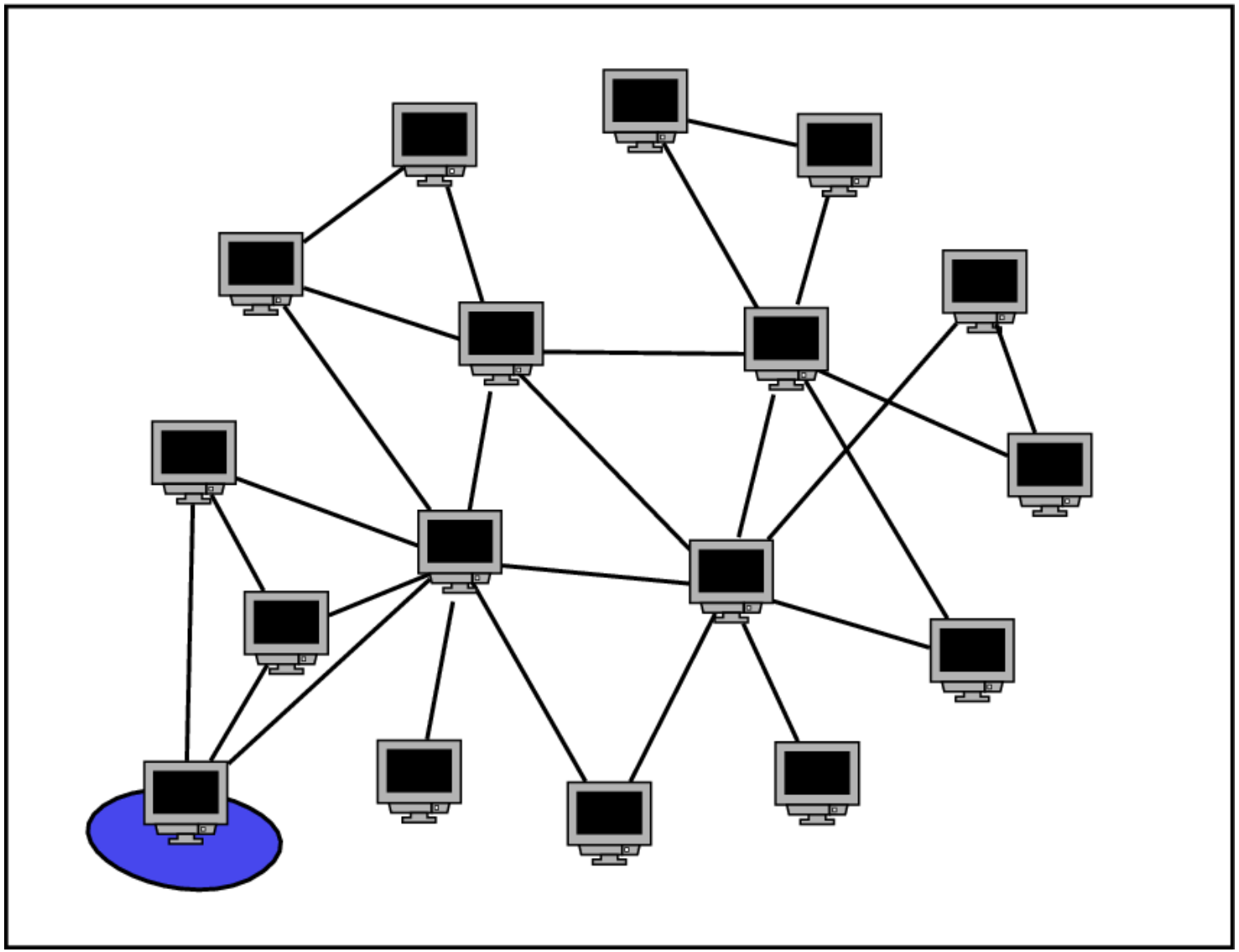
Motivation



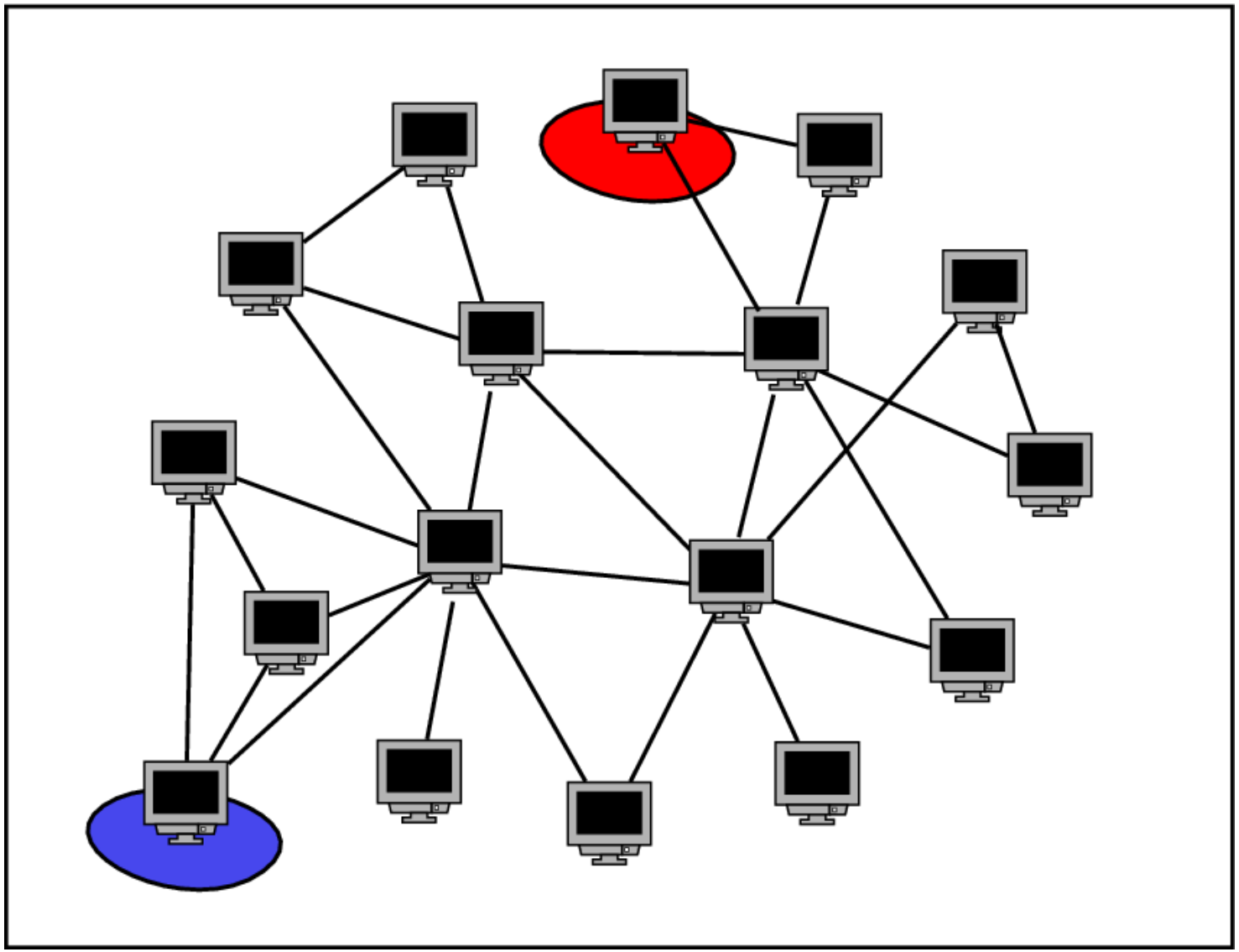
Motivation



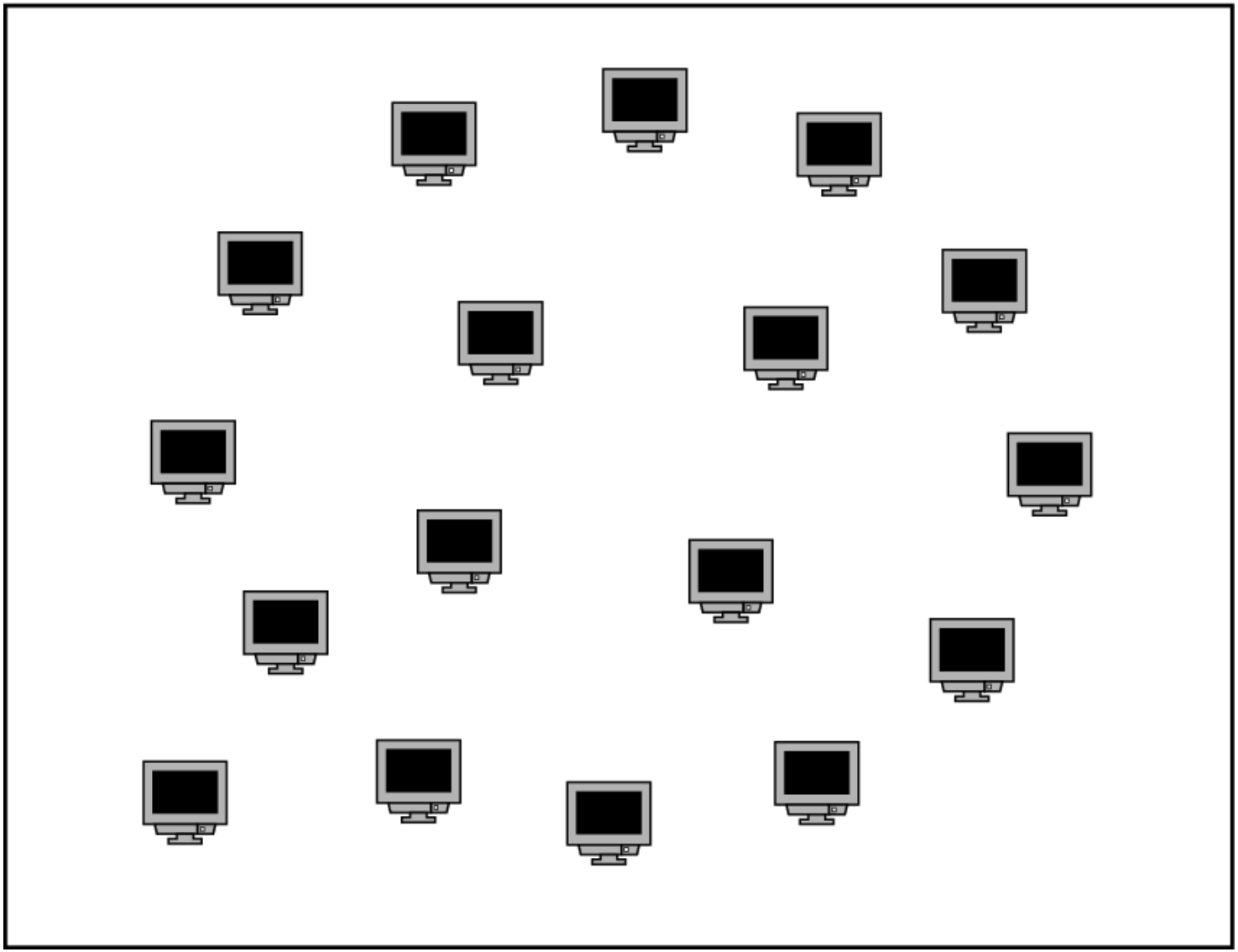
Motivation



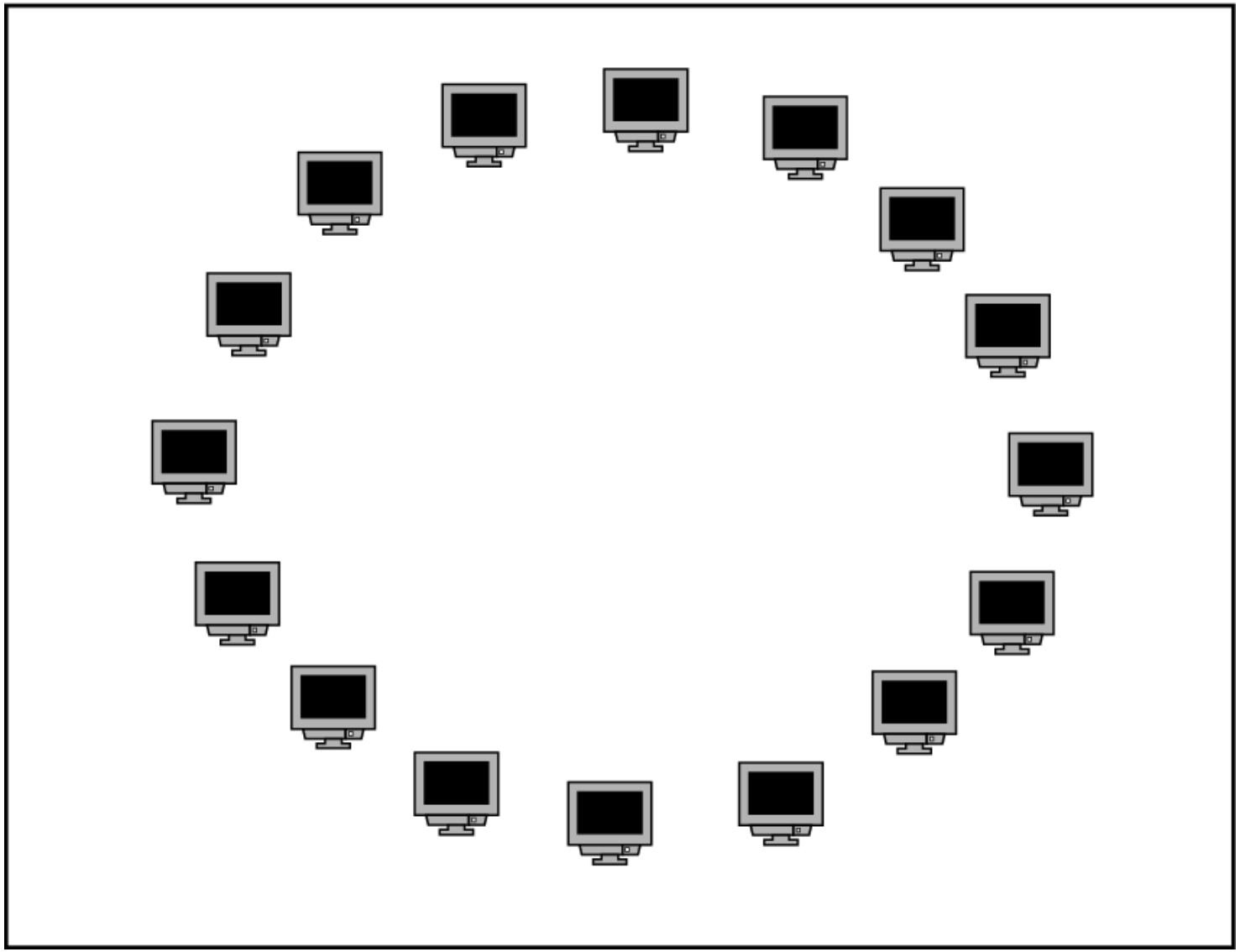
Motivation



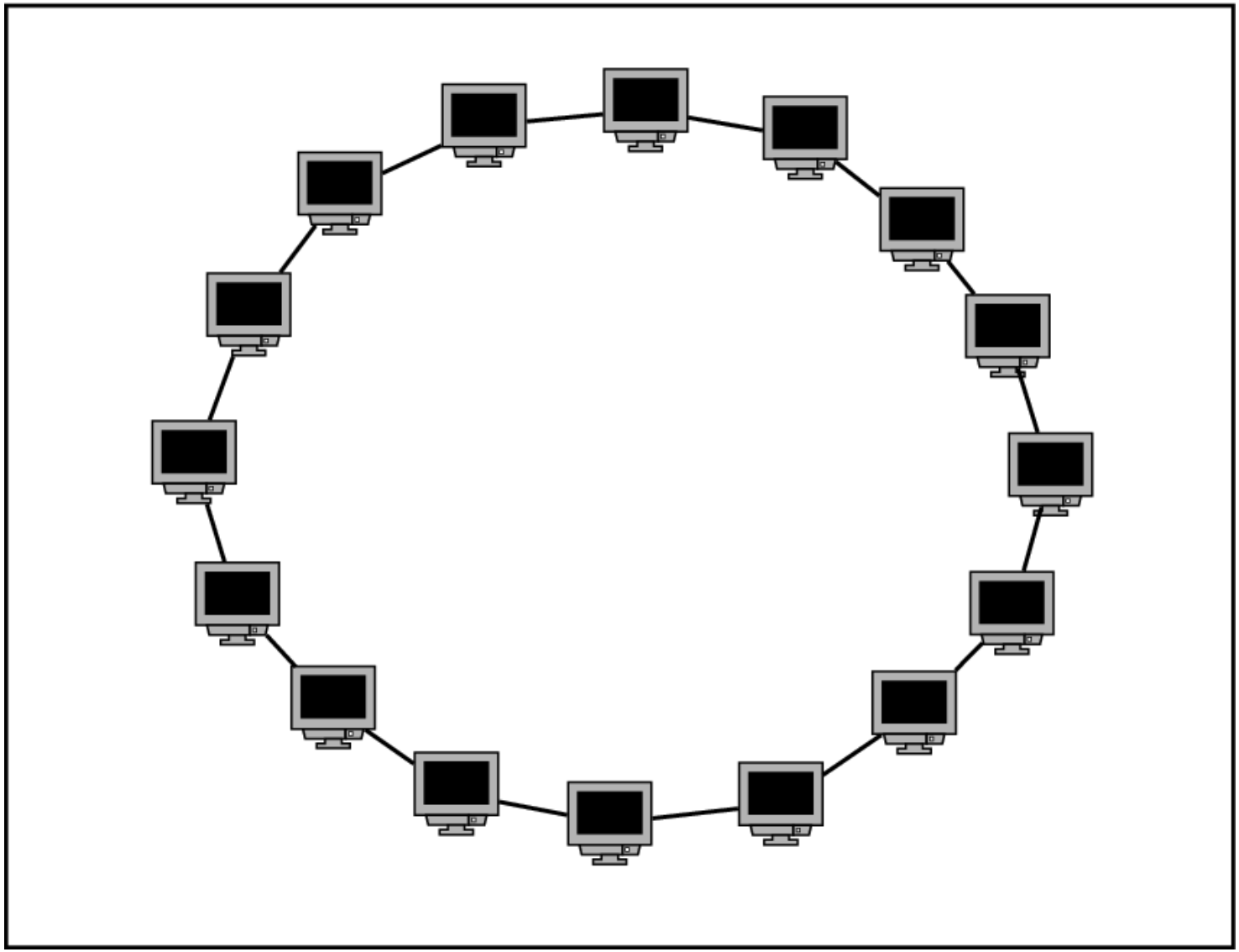
Motivation



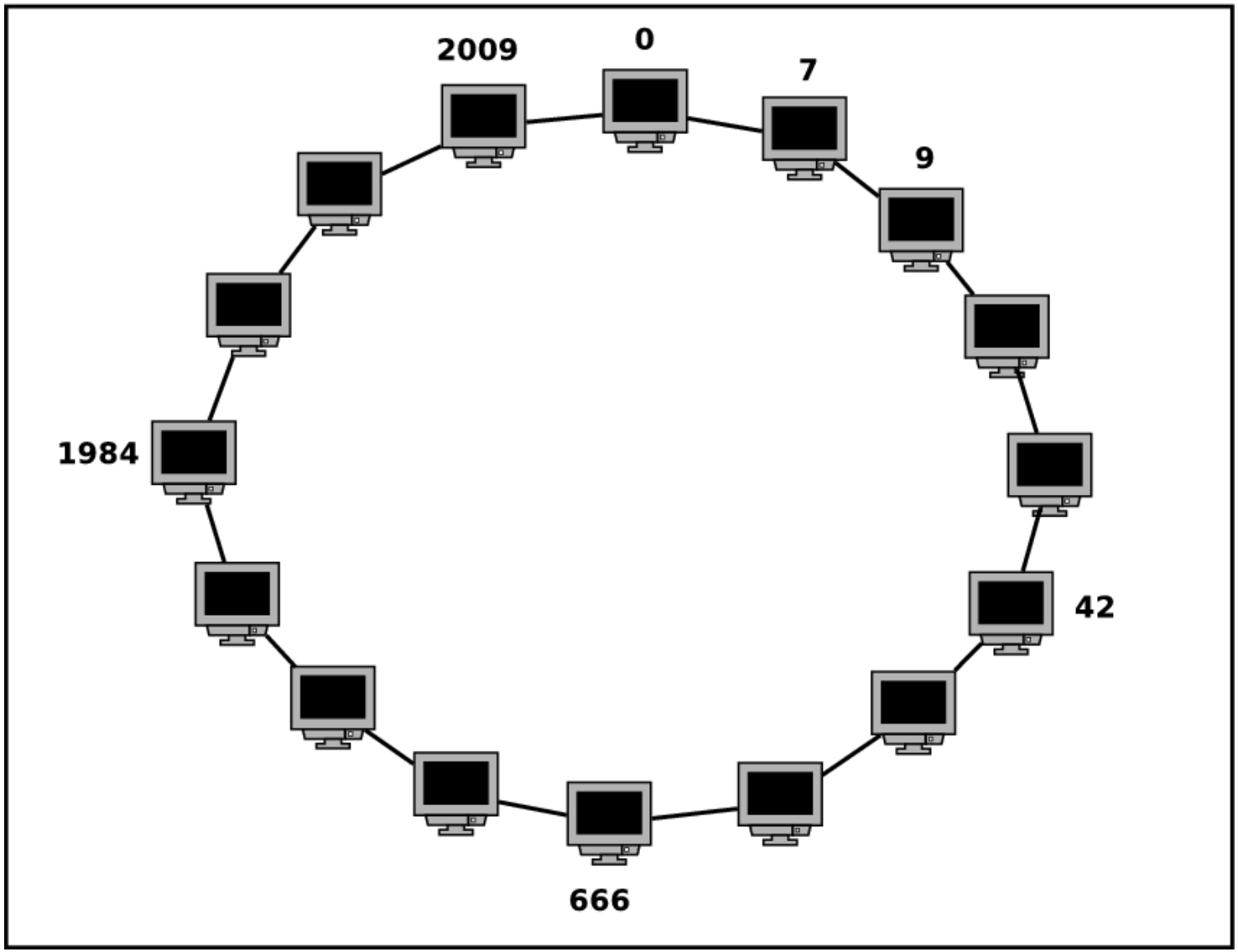
Motivation



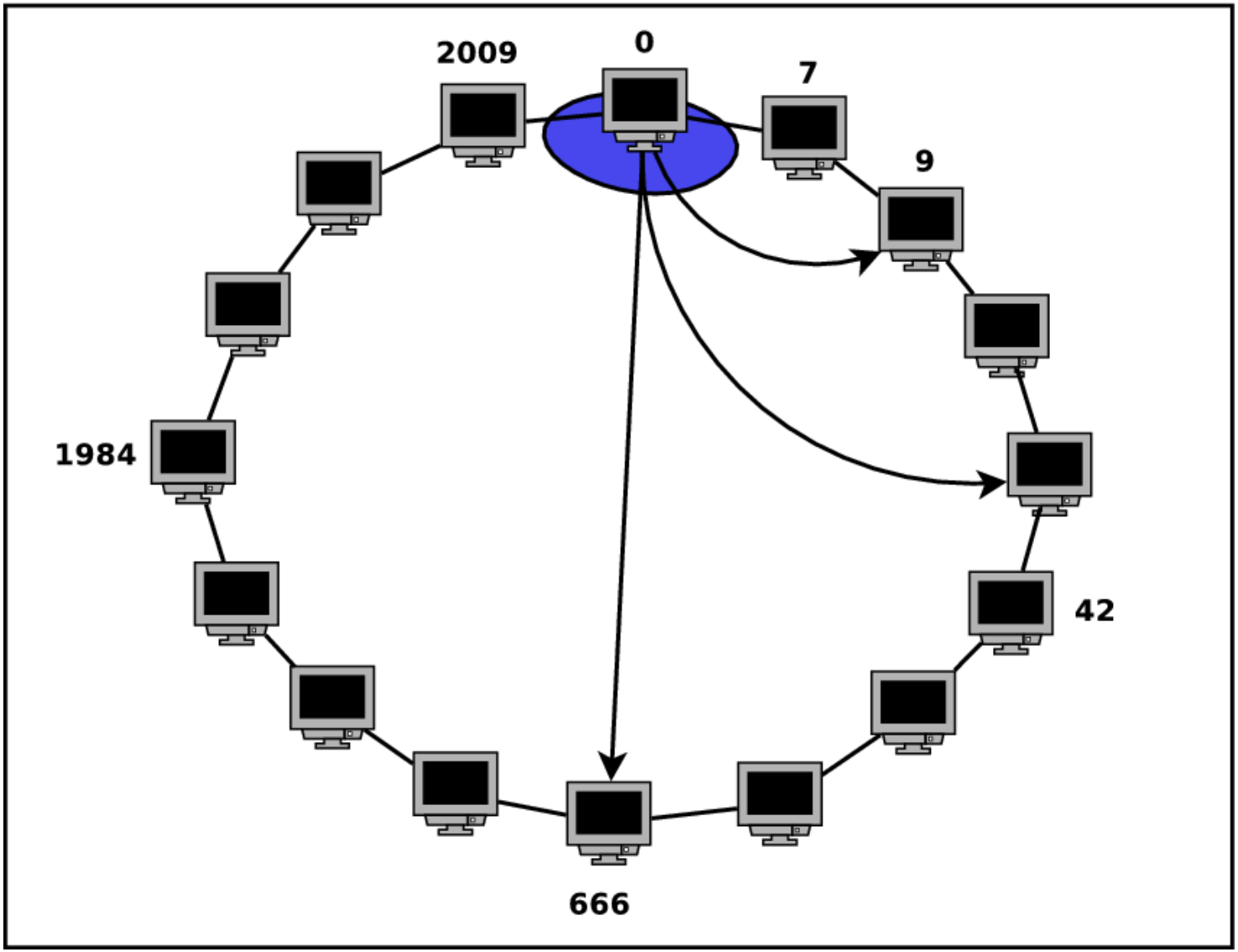
Motivation



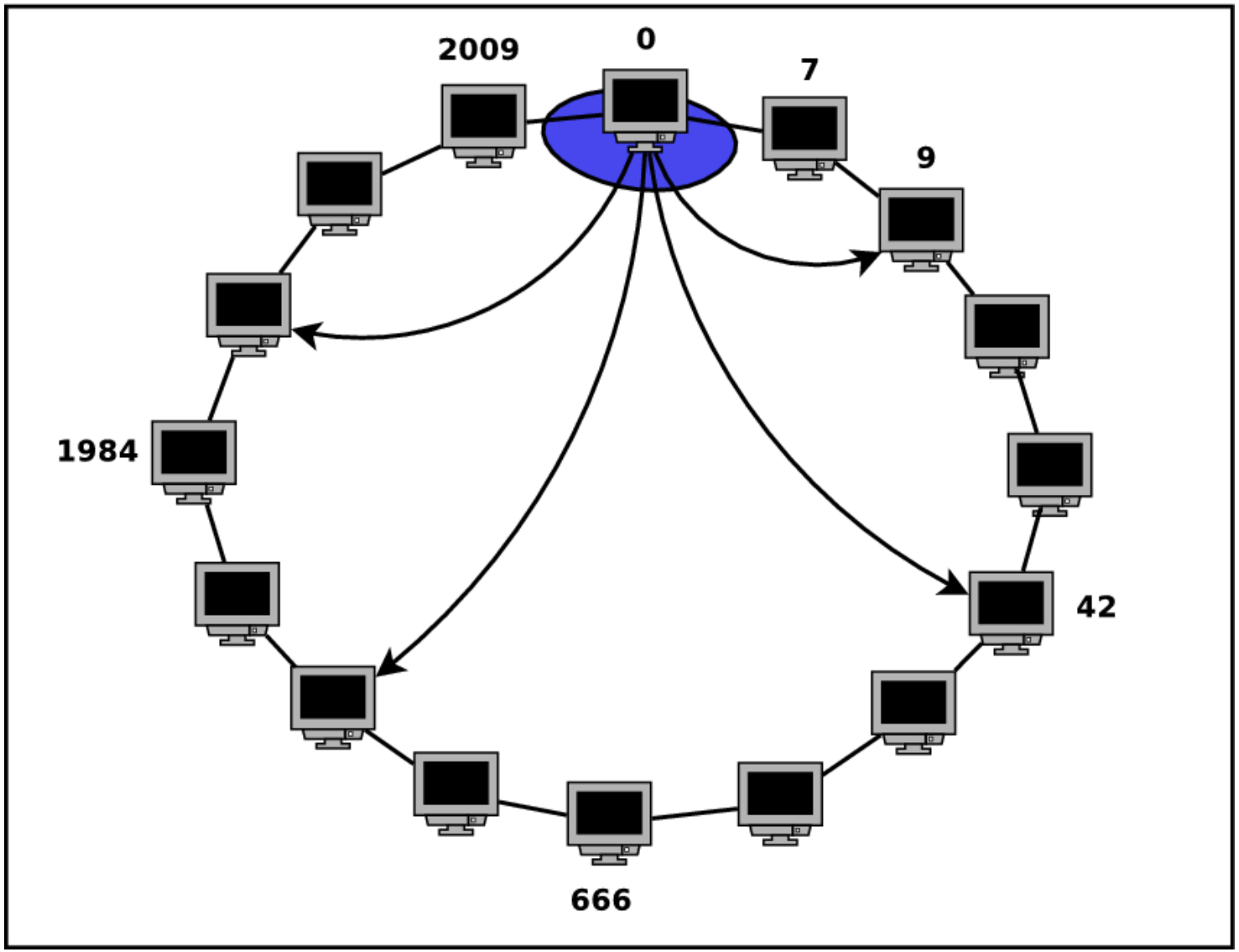
One ring to rule them all



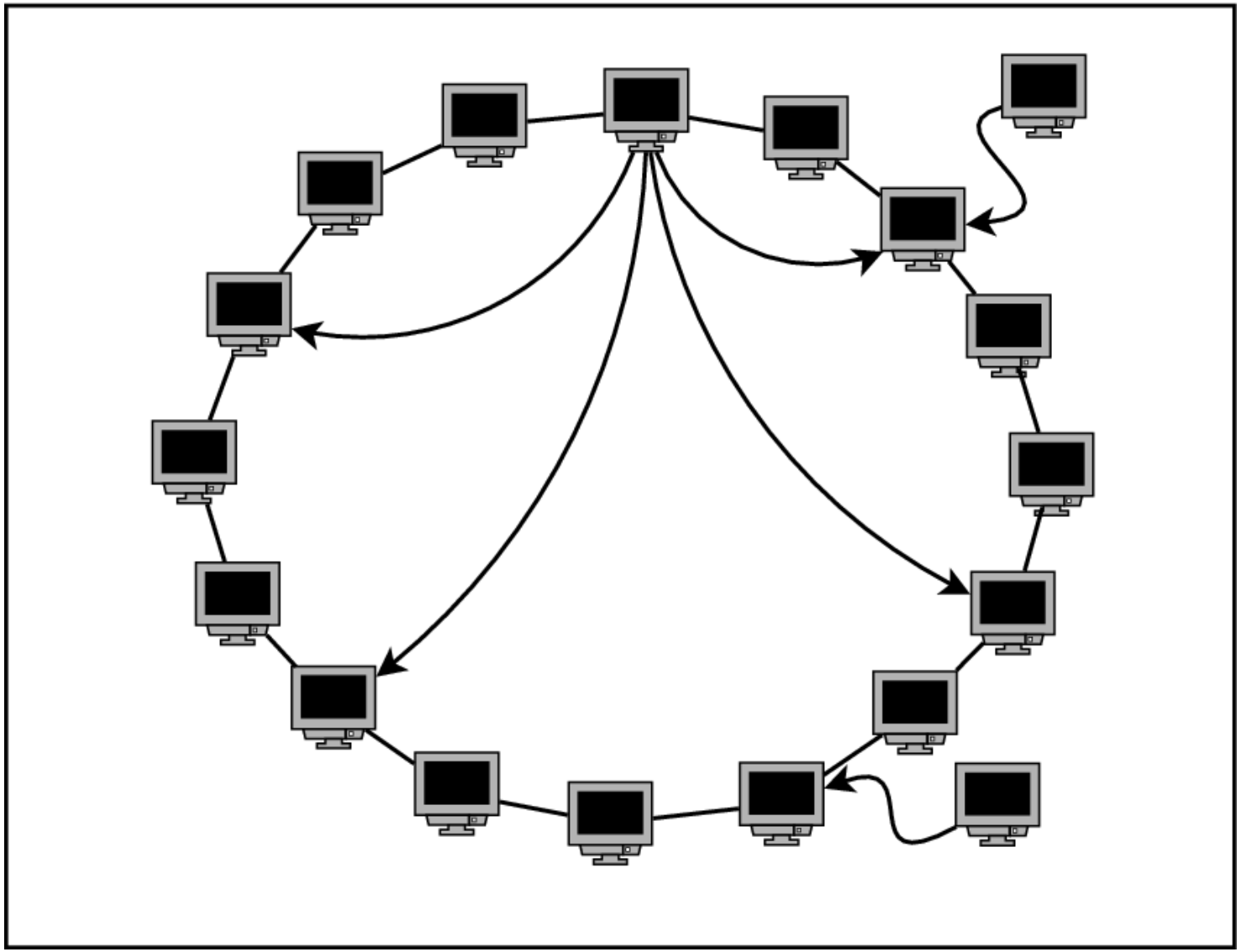
One ring to find them



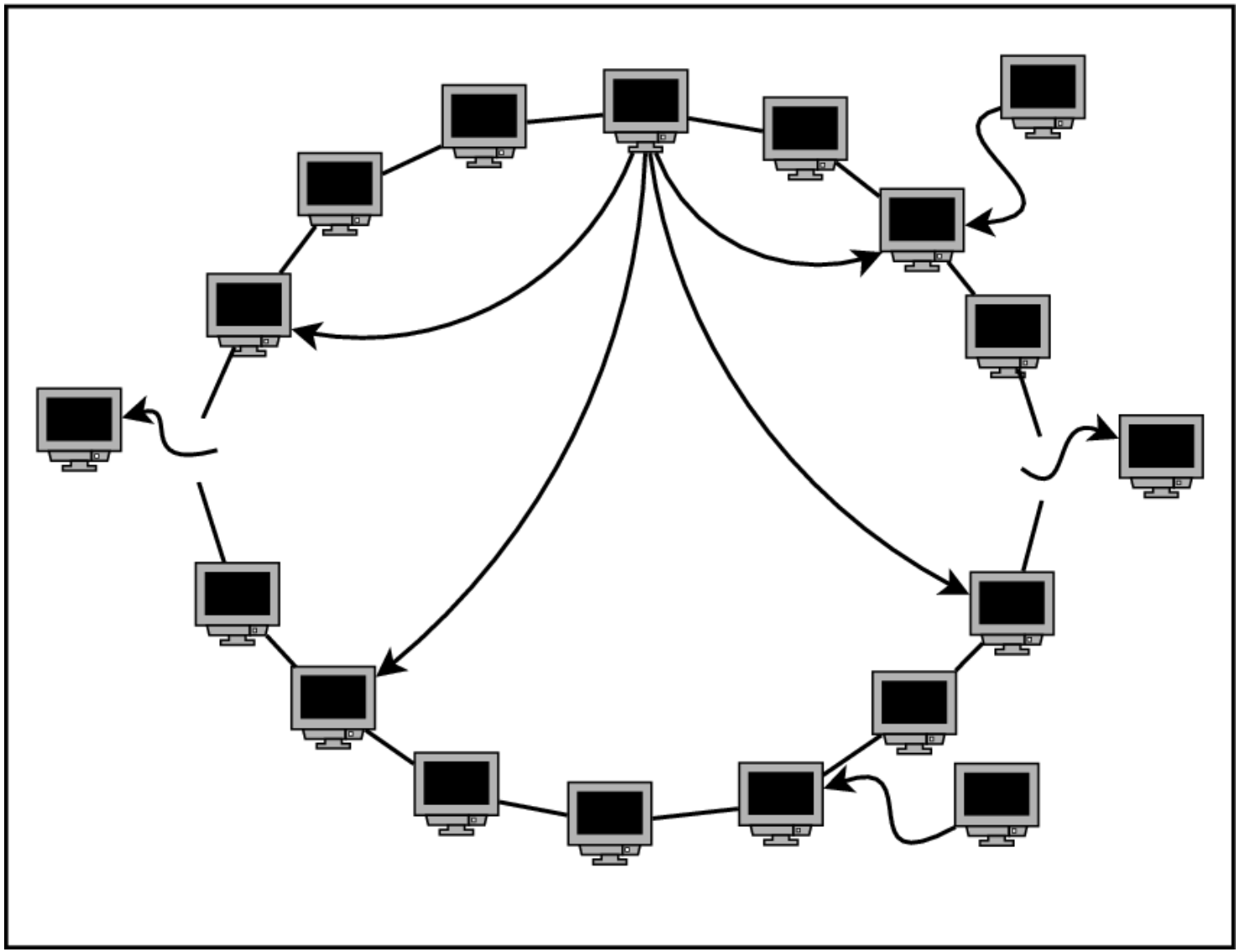
One ring to find them



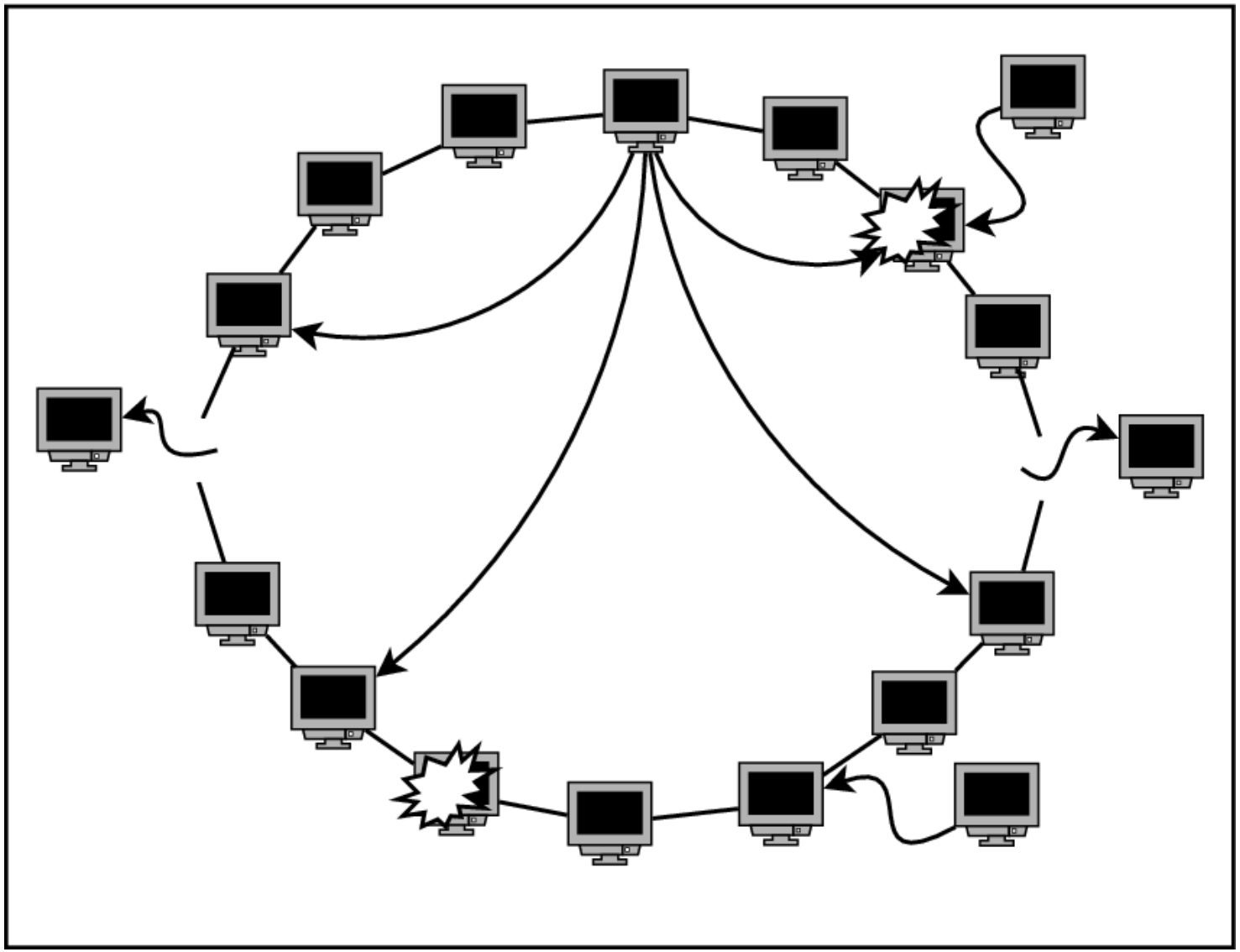
Motivation



Motivation



Motivation



Motivation

Decentralized systems to replace Client-Server

- Some problems of client-server architecture
  - Servers are source of congestion
  - Single point of failure: No server, no application

Motivation

Decentralized systems to replace Client-Server

- Some problems of client-server architecture
  - Servers are source of congestion
  - Single point of failure: No server, no application
- We can get rid of this architecture
  - Clients are much powerful now and they can also play the role of a server
  - Increase of Internet bandwidth and better reliability

Decentralized systems to replace Client-Server

- Some problems of client-server architecture
  - Servers are source of congestion
  - Single point of failure: No server, no application
- We can get rid of this architecture
  - Clients are much powerful now and they can also play the role of a server
  - Increase of Internet bandwidth and better reliability
- Why do we want to change?
  - Decentralized systems suit better strategies such as replication, load-balancing, failure-recovery
  - No single point of failure
  - It's more fun

New challenges

- No central point of control or synchronization.
- Good network organisation approach
- Guarantee reachability of all nodes
- Provide efficient routing
- Guarantee consistency of the distributed storage
- Need for consensus
- Partial failures cannot be avoided

New challenges

- No central point of control or synchronization.
- Good network organisation approach
- Guarantee reachability of all nodes
- Provide efficient routing
- Guarantee consistency of the distributed storage
- Need for consensus
- Partial failures cannot be avoided

To reduce complexity we need to introduce Self-Management

Self-Management

- The system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies
- Human intervention is lifted to the level of the policies

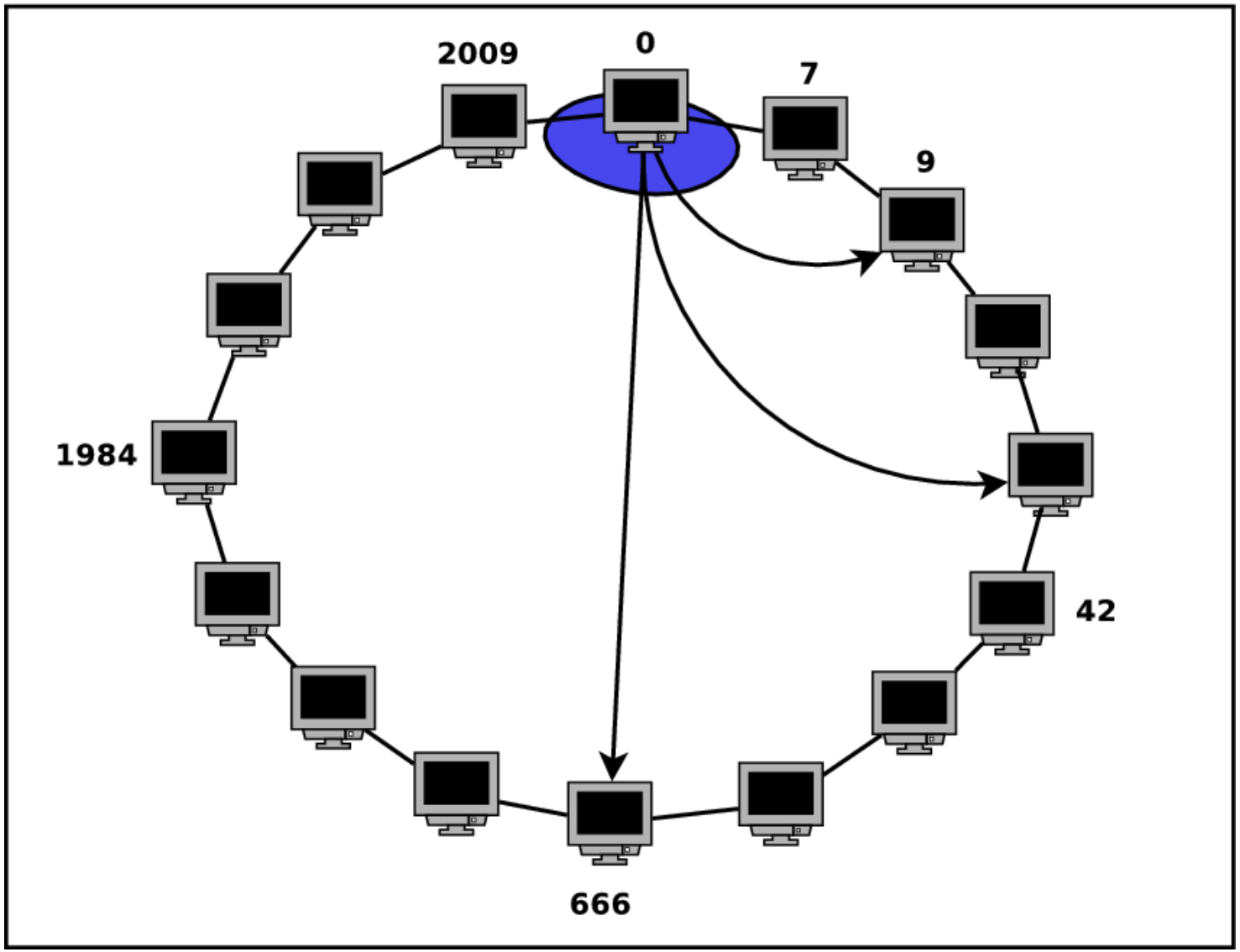
Self-Management

- The system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies
- Human intervention is lifted to the level of the policies
- Typical self-management operations include:
  - add/remove nodes
  - tune performance
  - auto-configure
  - replicate data
  - failure detection and recovery
  - intrusion detection and recovery

The Rest of the Talk

- Structured Overlay Networks
- Decentralized Transactions

The one ring



Distributed Hash Tables (DHT)

- Every peer is identified with a hash key
- Two basic operations: `put(key, value)` and `get(key)`
- Every peer is responsible for all keys contained in the range delimited by its predecessor and itself `(pred, self]`
- Operation `lookup(key)` finds the responsible for a key
- Fingers are chosen in order to provide efficient routing $log_k(N)$
- Lookup consistency implies that there will be only one responsible for any key at any time, or the responsible is temporary unavailable.

Chord peer-to-peer network

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$

Chord peer-to-peer network

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$
- Lookup inconsistencies (due to churn)
- Expensive maintenance (periodic stabilization)

Chord peer-to-peer network

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$
- Lookup inconsistencies (due to churn)
- Expensive maintenance (periodic stabilization)

One solution: atomic join/leave

- Locks on successor, predecessor and the node

Chord peer-to-peer network

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$
- Lookup inconsistencies (due to churn)
- Expensive maintenance (periodic stabilization)

One solution: atomic join/leave

- Locks on successor, predecessor and the node
- Only two locks: the node and its successor (A. Ghodsi)

Chord peer-to-peer network

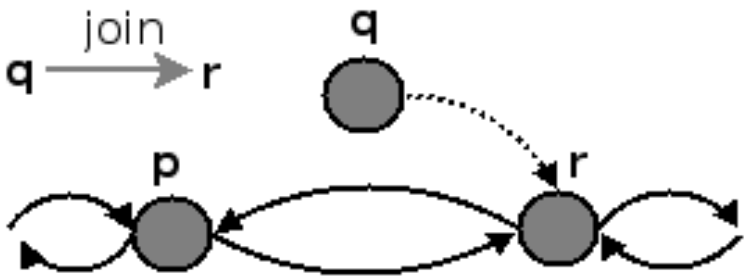
- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$
- Lookup inconsistencies (due to churn)
- Expensive maintenance (periodic stabilization)

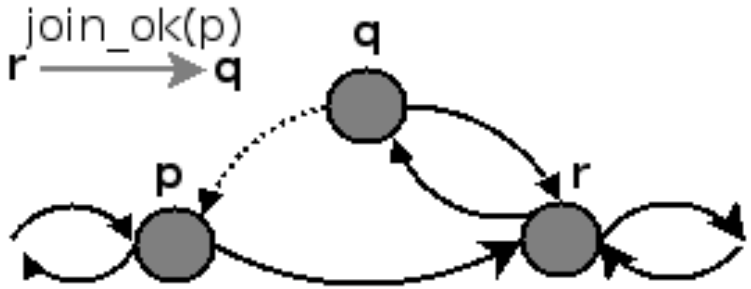
One solution: atomic join/leave

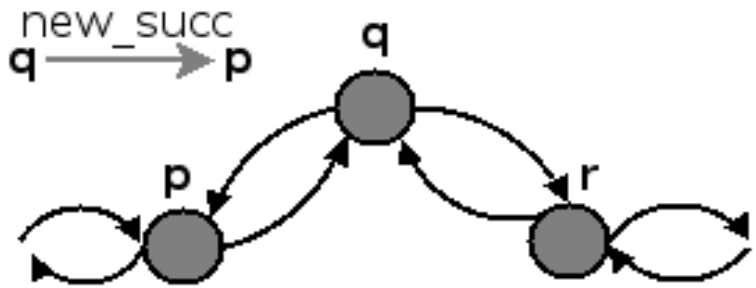
- Locks on successor, predecessor and the node
- Only two locks: the node and its successor (A. Ghodsi)
- and the failures?

The Relaxed-Ring

- joining peer *q* requests a lookup for its key
- *q* sends the *join* message to its successor candidate *r*
- *r* accepts new pred and sends reference *p* to *q*
- *q* contacts *p* to inform that it is its new succ

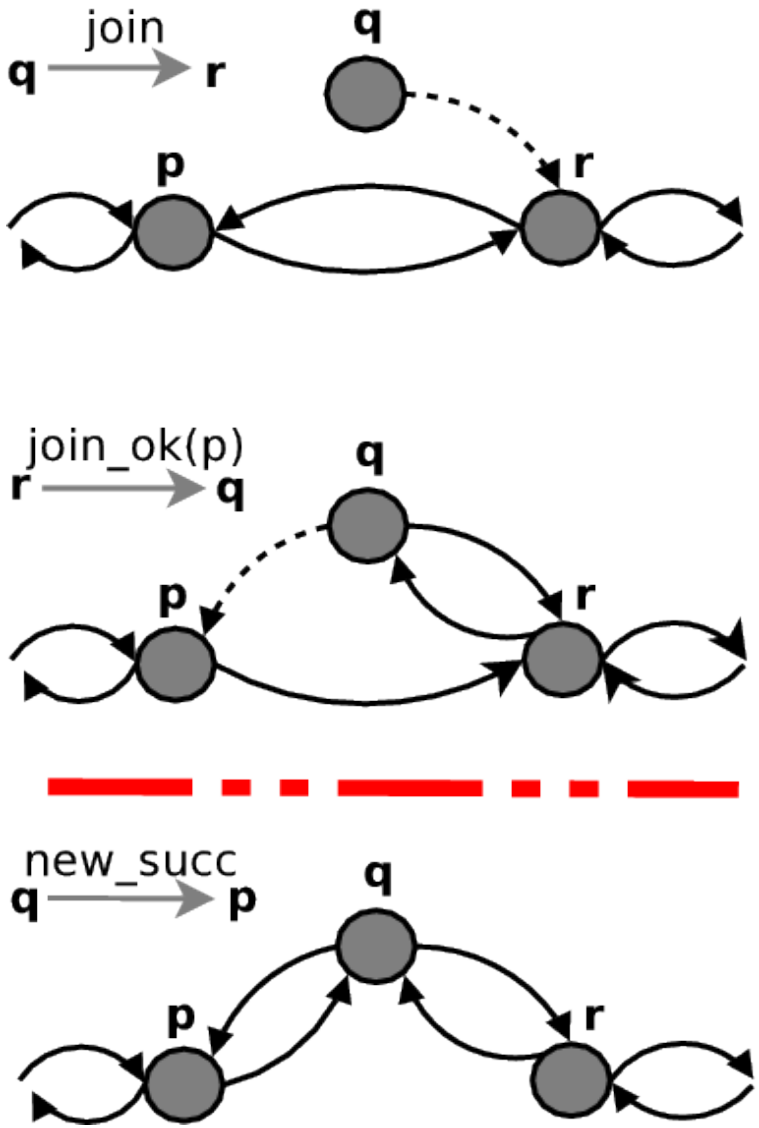






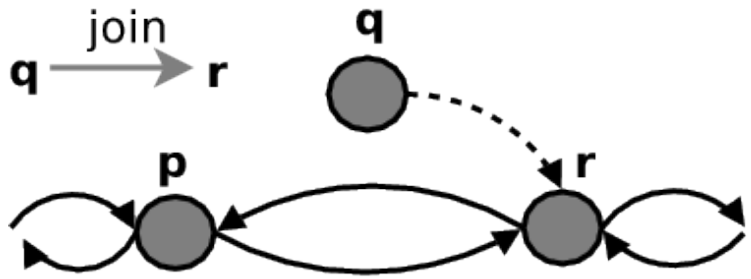
The Relaxed-Ring

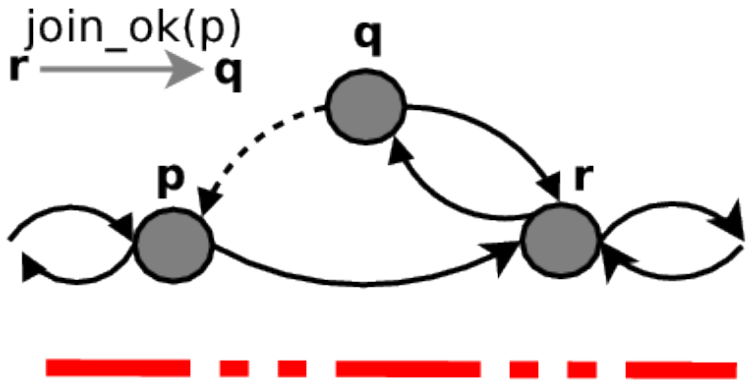
- joining peer *q* requests a lookup for its key
- *q* sends the *join* message to its successor candidate *r*
- *r* accepts new pred and sends reference *p* to *q*
- *q* contacts *p* to inform that it is its new succ

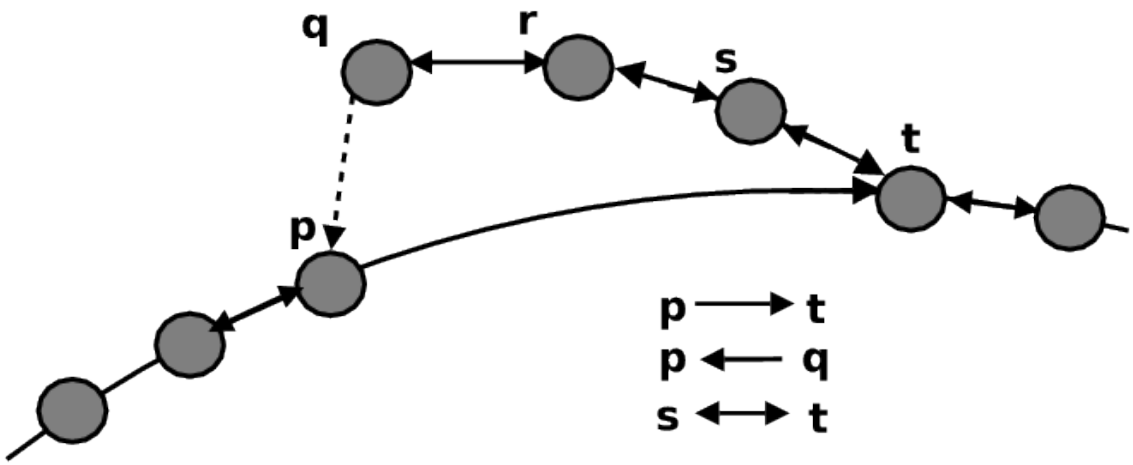


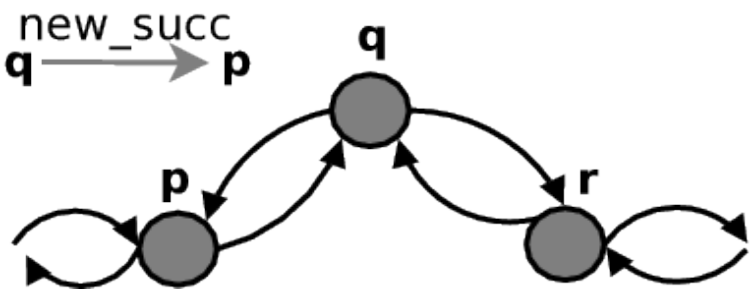
The Relaxed-Ring

- joining peer *q* requests a lookup for its key
- *q* sends the *join* message to its successor candidate *r*
- *r* accepts new pred and sends reference *p* to *q*
- *q* contacts *p* to inform that it is its new succ

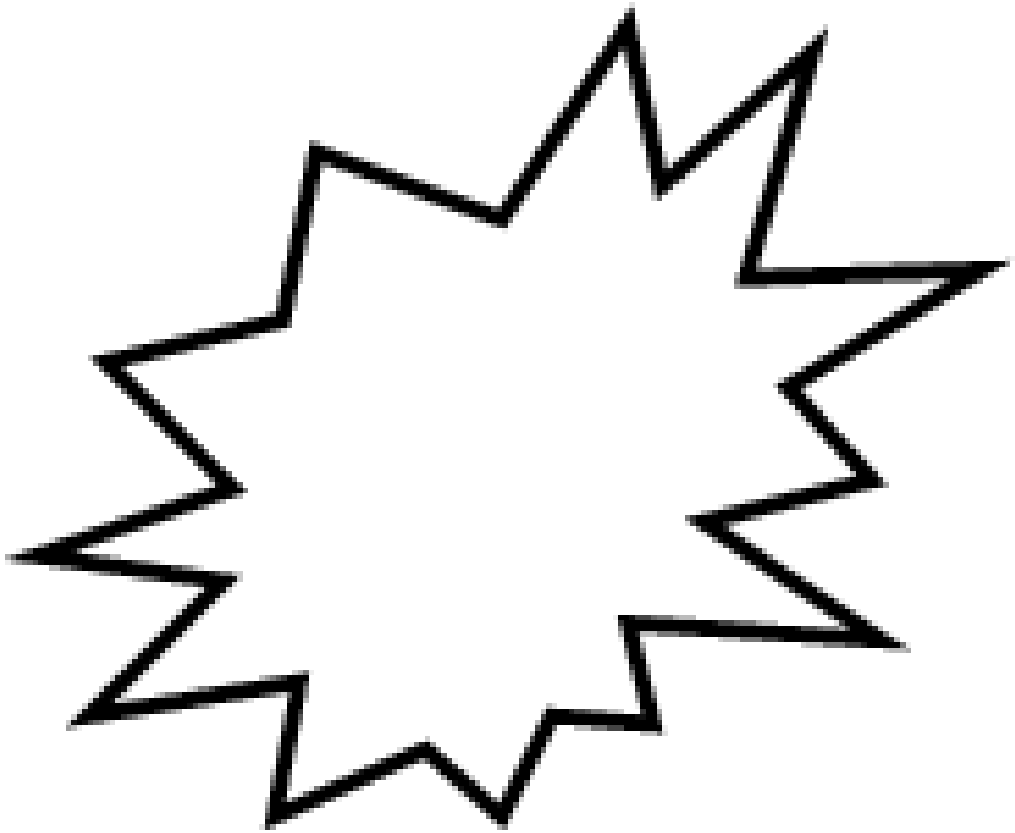








Failures

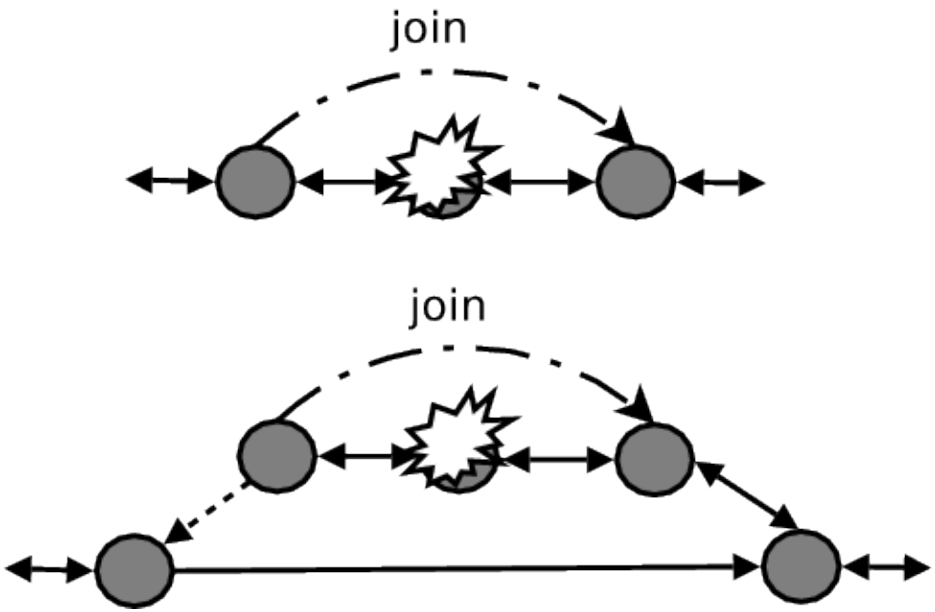


Failure Recovery

- We cannot assume perfect failure detection
- We assume imperfect failure detection with the following properties (Internet-style):
  - Strongly complete: all failed nodes are detected
  - Eventually accurate: false suspicions will be corrected
- Broken links cannot be ignored
- We do not assume symmetric links
- Limitations:
  - It survives network partitioning but lookup consistency is broken (Brewer's conjecture with respect to consistent available partition-tolerant services)
  - Resilience is given by the size of successors list

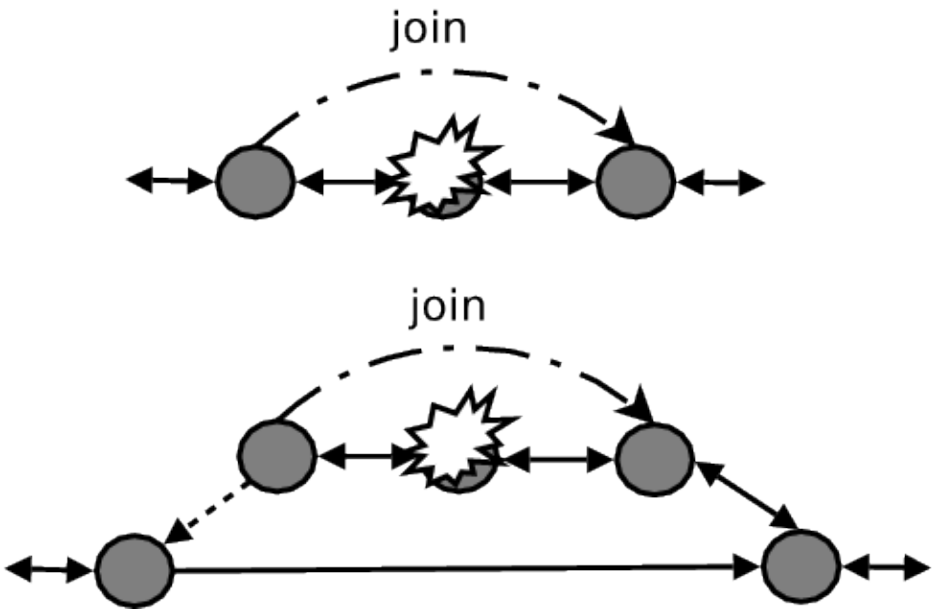
Failure Recovery

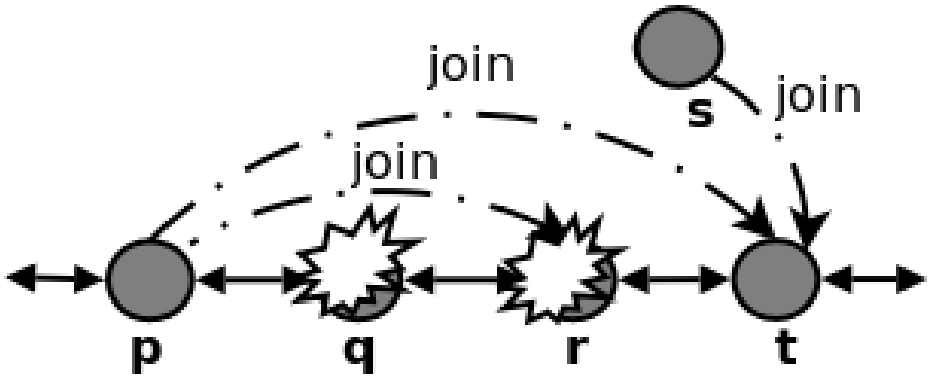
- When a peer detects that its successor has crashed, it contacts the first peer in its successor list for recovery



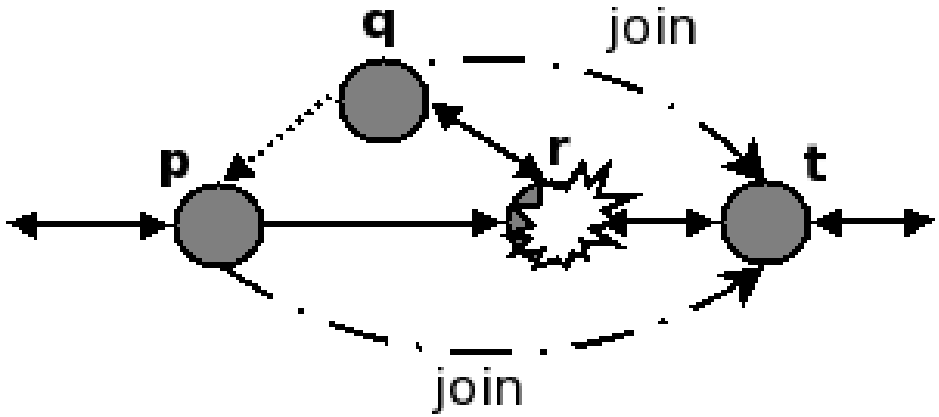
Failure Recovery

- When a peer detects that its successor has crashed, it contacts the first peer in its successor list for recovery



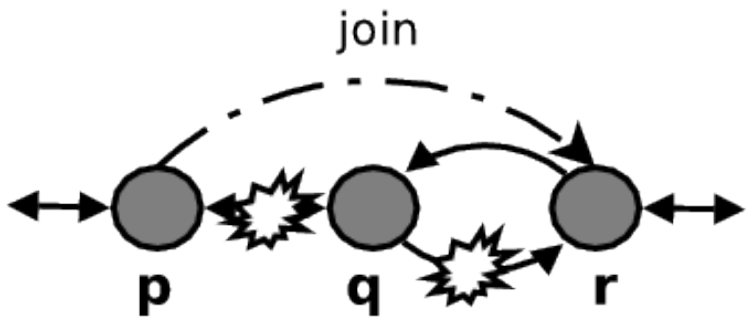


Limitations - Crash of the root of a branch



- If *r* the root of a branch crashes, it may introduce temporary inconsistent lookup if *p* contacts *t* for recovery before *q*. The inconsistency will involve the range $(p, q]$, and it will be corrected as soon as *q* contacts *t* for recovery
- Peer *q* does not have a valid successor during the inconsistency

Limitations - False suspicions with asymmetric links



- Broken link between peers p and $q \Rightarrow$ *false suspicion*
- Peer r cannot hear $q \Rightarrow$ *false suspicion* with *asymmetric link*
- Peer q considers r as valid successor $\Rightarrow$ inconsistency in the range $(p, q]$

Relaxed-Ring

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Fault tolerant (Self-healing)

Relaxed-Ring

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Fault tolerant (Self-healing)
- Relies only on point-to-point link
- No transitivity: if $a \rightarrow b$ and $b \rightarrow c$ does not imply $a \rightarrow c$
- join/fail algorithms requires the agreement of only two nodes (2 steps with 2 nodes, instead of 1 step with three)

Relaxed-Ring

- Scalable
- Provides a Distributed Hash Table (DHT)
- Fully decentralized
- Self-organized
- Fault tolerant (Self-healing)
- Relies only on point-to-point link
- No transitivity: if $a \rightarrow b$ and $b \rightarrow c$ does not imply $a \rightarrow c$
- join/fail algorithms requires the agreement of only two nodes (2 steps with 2 nodes, instead of 1 step with three)
- Almost no lookup-inconsistency
- Cost-efficient ring maintenance (no periodic stabilization)
- Efficient routing $O(\log(N) + b)$

Intermezzo

This talk is brought to you by

Decentralized Transactions

- Provide a decentralized transactional distributed database with strong data consistency
- DHT is the underlying infrastructure
- Use replication of data to achieve higher availability and fault tolerance
- Required properties:
  - Concurrency control
  - Atomic commit protocol

Consistency of Replicated Data

- Main ideas
  - All operations on data include a majority of replicas
  - Use version numbers to determine the current version
- Each operation maintains the invariant that a majority of replicas contains the latest version of an item
  - Write has to update at least a majority
  - Read includes a majority
  - Maintenance of replication degree (handling of node failure): Restore replication degree by reading from a majority
  - Node join/leave will not violate the invariant

Distributed Transactions

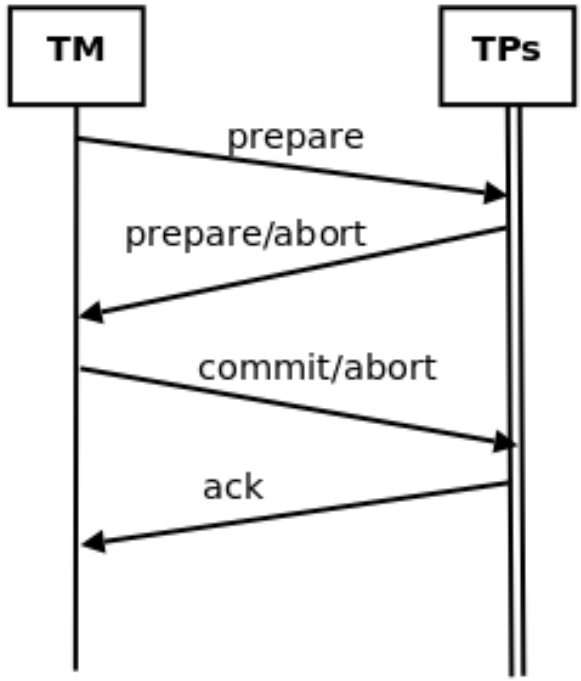
- A transaction consists of a set of operations on data
- Main issues of transactions:
  - Isolate concurrent transactions from each other (Serializability)
  - Either all operations take place or none of them (Atomicity)
- Data involved in the transaction is distributed over several nodes

Distributed Transactions

- A transaction consists of a set of operations on data
- Main issues of transactions:
  - Isolate concurrent transactions from each other (Serializability)
  - Either all operations take place or none of them (Atomicity)
- Data involved in the transaction is distributed over several nodes
- One node acts as the transaction manager (TM)
- Nodes that are responsible for data involved in the transaction act as transaction participants (TP)
- A transaction is processed on three steps:
  - **Read**: Collection of operations which are part of the transaction
  - **Validation**: Each TP checks whether it can execute the operation
  - **Write**: If the validation is successful for all TPs, each TP has to make changes permanent

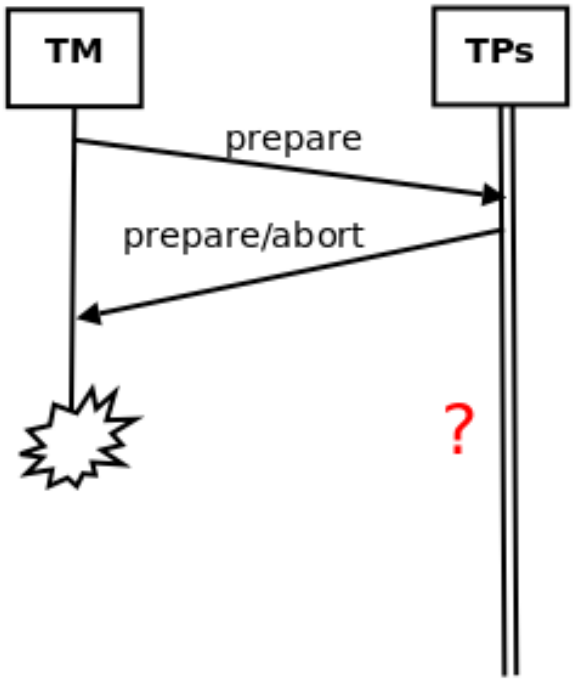
Atomic Commit in Distributed Systems

- The outcome of the protocol is:
  - **Commit**, if all TPs can successfully validate the operations
  - **Abort**, if there exists at least one TP that cannot validate an operation
- Most common commit protocol: 2-Phase-Commit



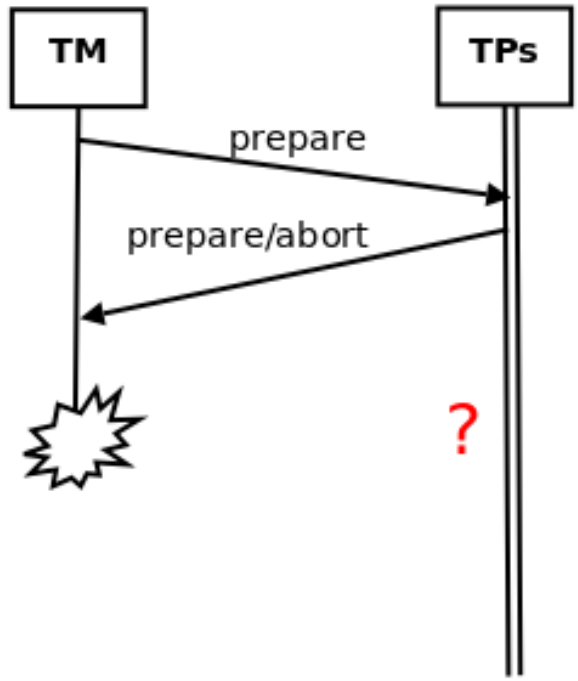
Atomic Commit in Distributed Systems

- The outcome of the protocol is:
  - **Commit**, if all TPs can successfully validate the operations
  - **Abort**, if there exists at least one TP that cannot validate an operation
- Most common commit protocol: 2-Phase-Commit

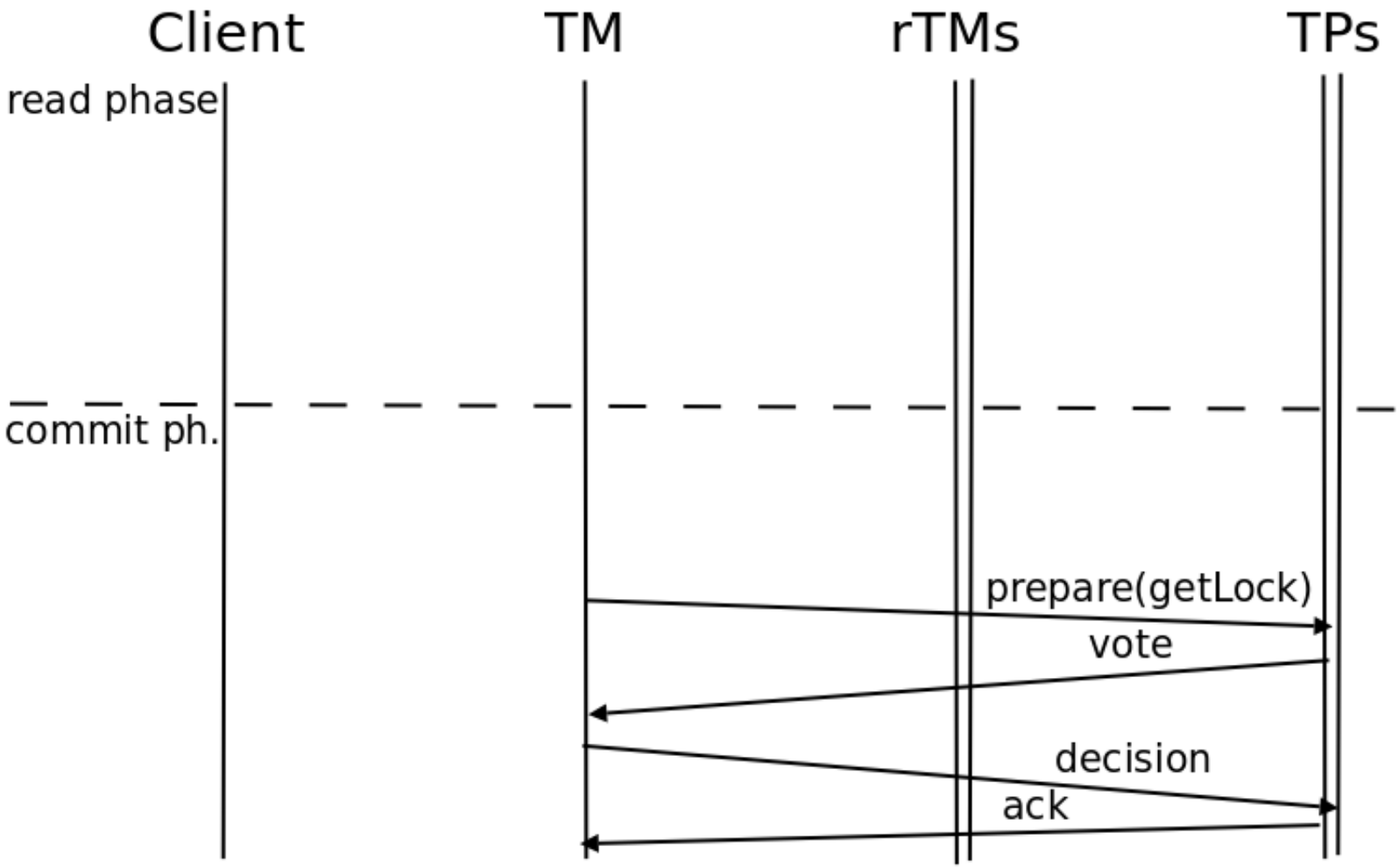


Atomic Commit in Distributed Systems

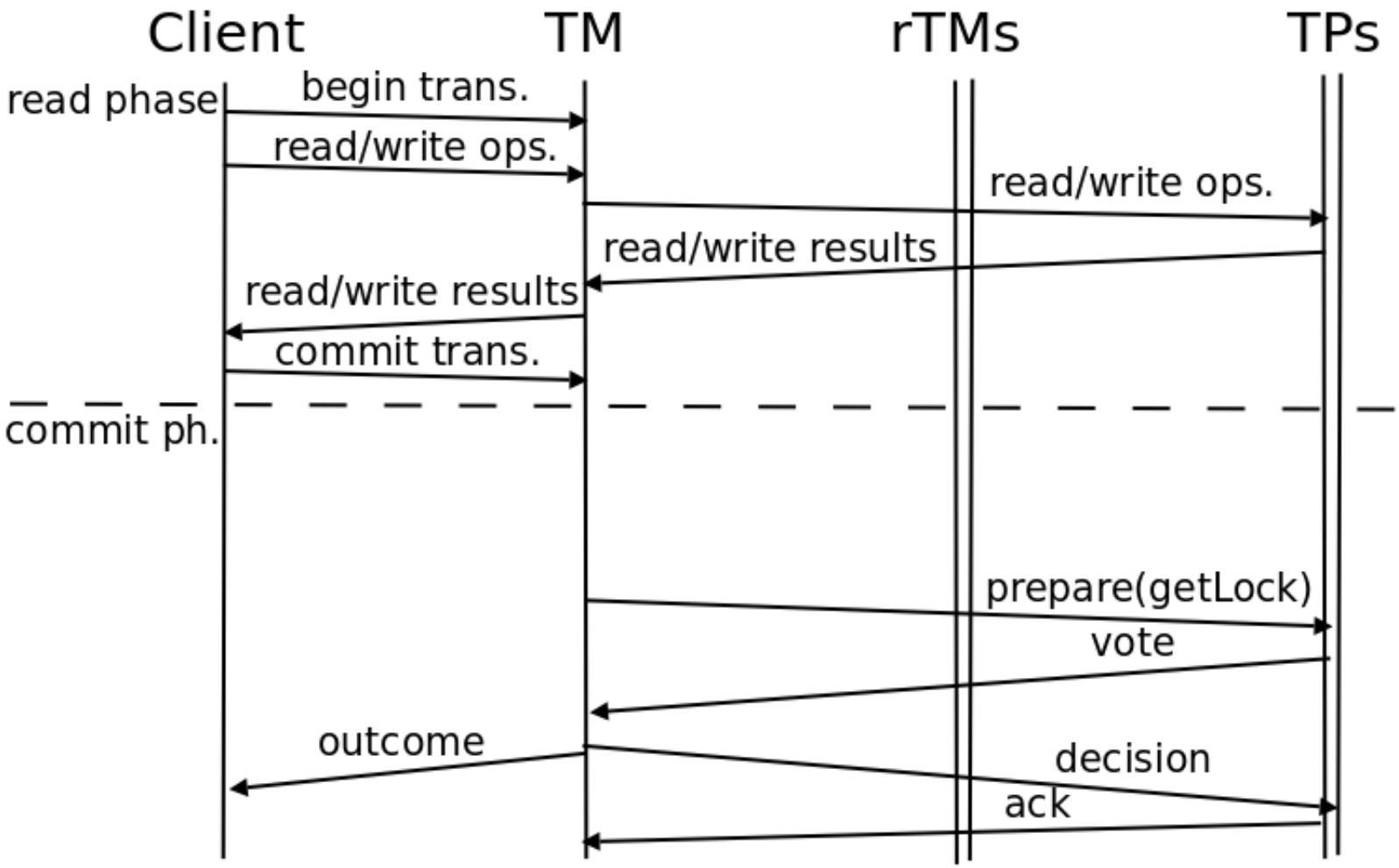
- The outcome of the protocol is:
  - **Commit**, if all TPs can successfully validate the operations
  - **Abort**, if there exists at least one TP that cannot validate an operation
- Most common commit protocol: 2-Phase-Commit
- It blocks the participants!
- Designed for LAN
- It uses reliable nodes
- 3-Phase-Commit?



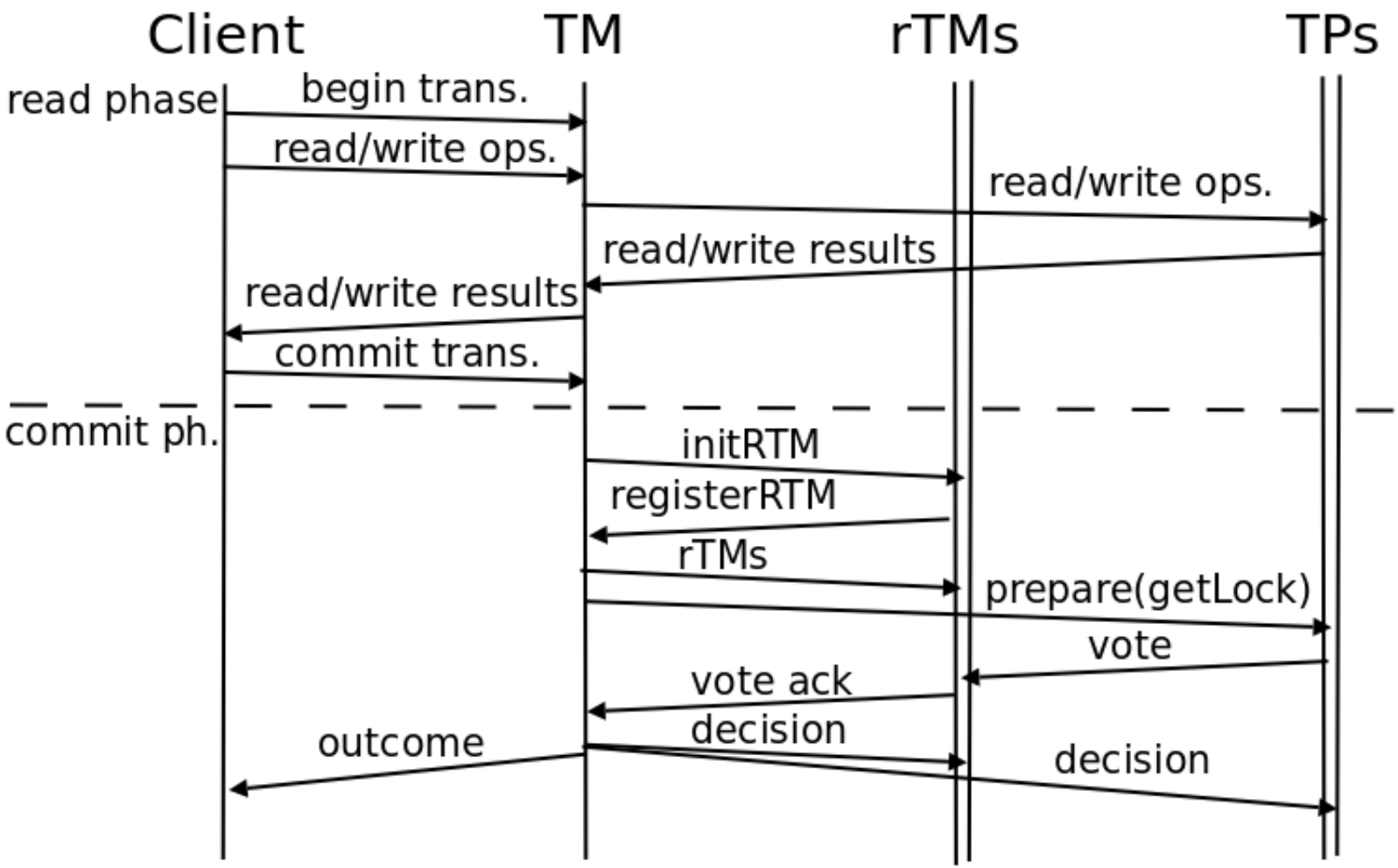
Atomic Paxos Commit



Atomic Paxos Commit



Atomic Paxos Commit

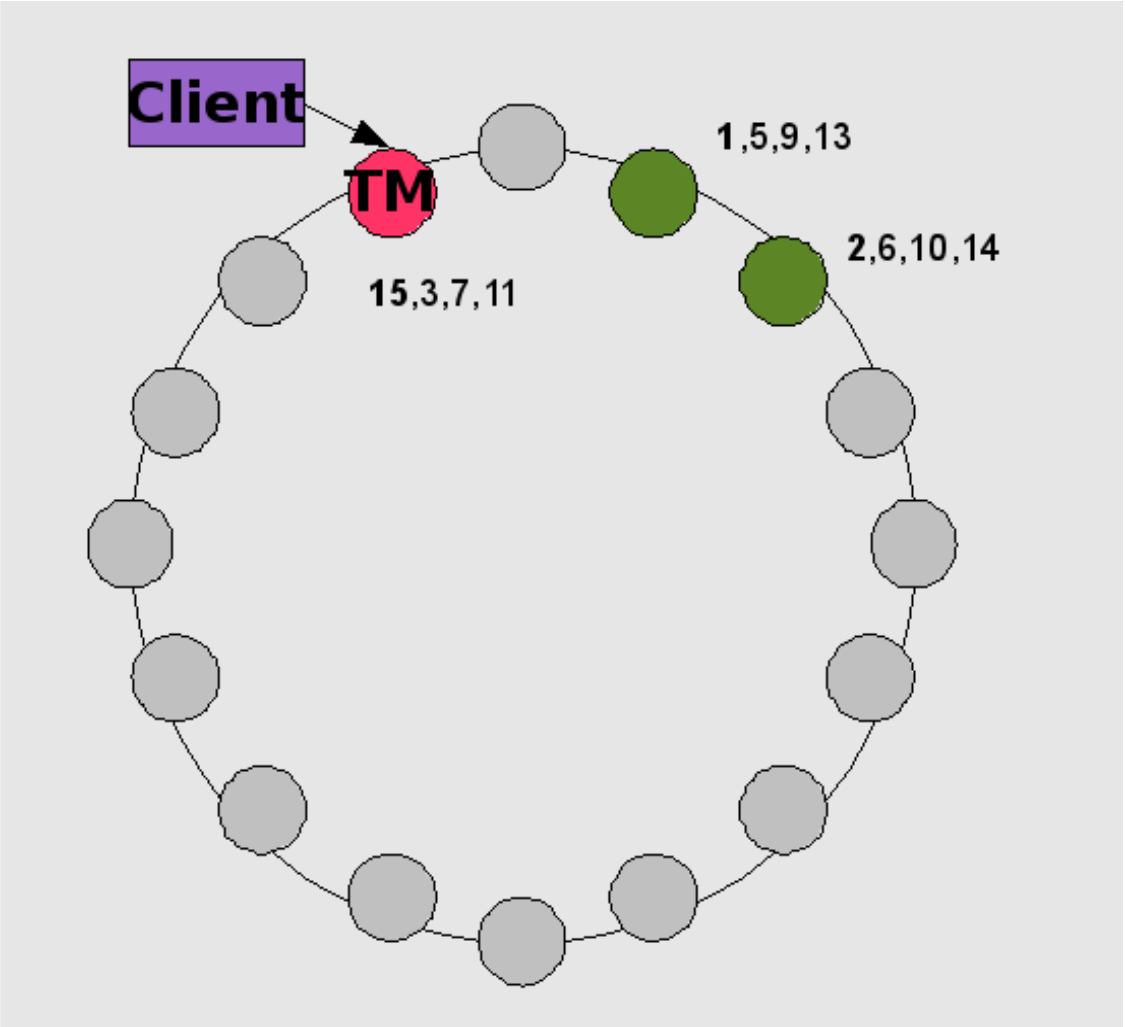


Atomic Paxos Commit

- Participants and Roles:
  - Transaction Manager (*TM*) + replicated *TMs* (*rTMs*) which act as acceptors
  - Transaction Participants (*TPs*) act as proposers
- Exploit the symmetric replication existing in the DKS DHT to determine the set of *rTMs*
- Collect the votes of the *TPs* per item
  - We need the majority of participants per item
  - We need all items to vote for commit

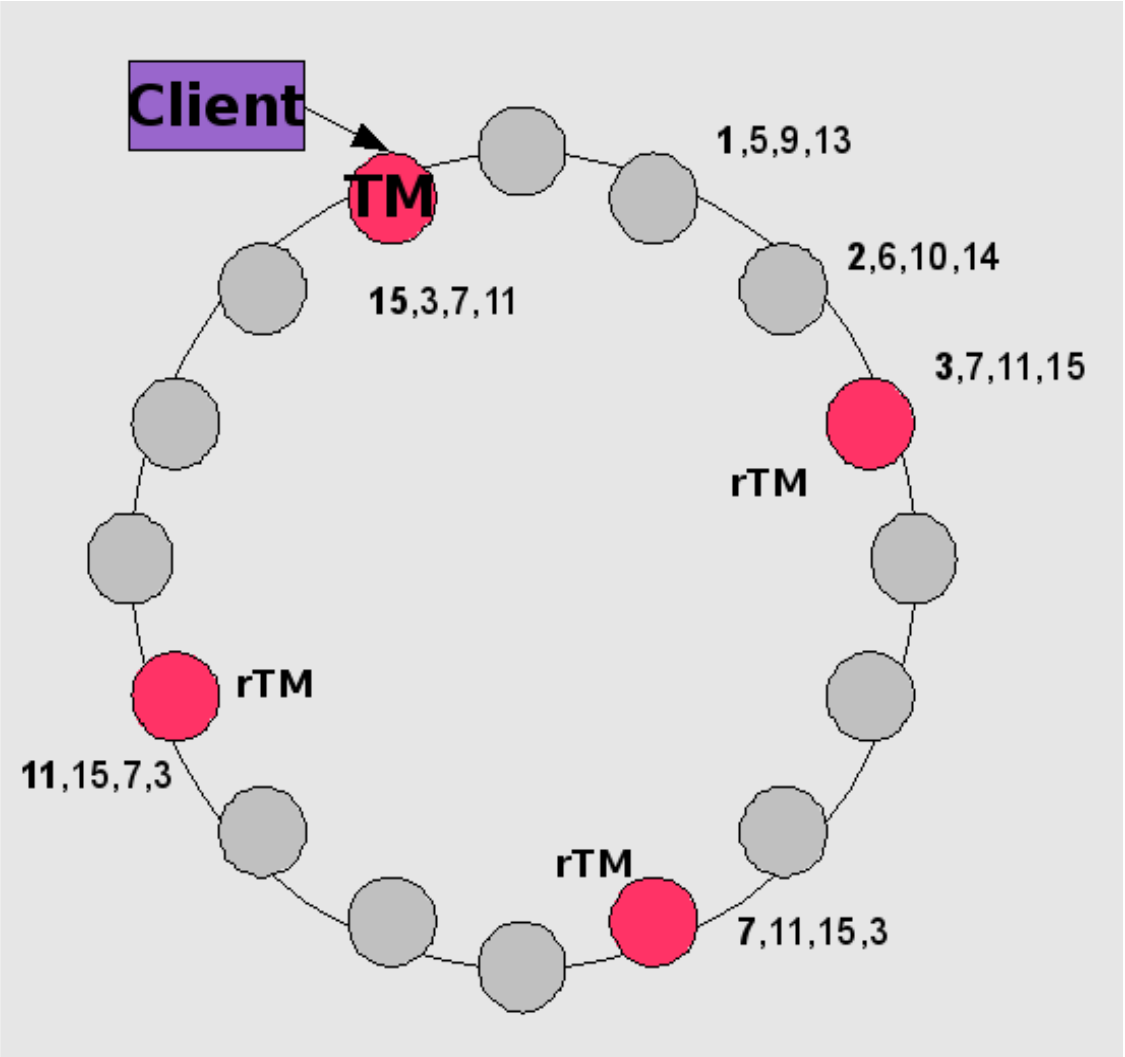
Atomic Commit in a DHT

- Client:
  ```
  BOT
  write item(1)
  write item(2)
  EOT
  ```
- Node 15 becomes the Transaction Manager (*TM*)
- *TM* creates a transaction item with a key for which it is responsible for (*key* = 15)



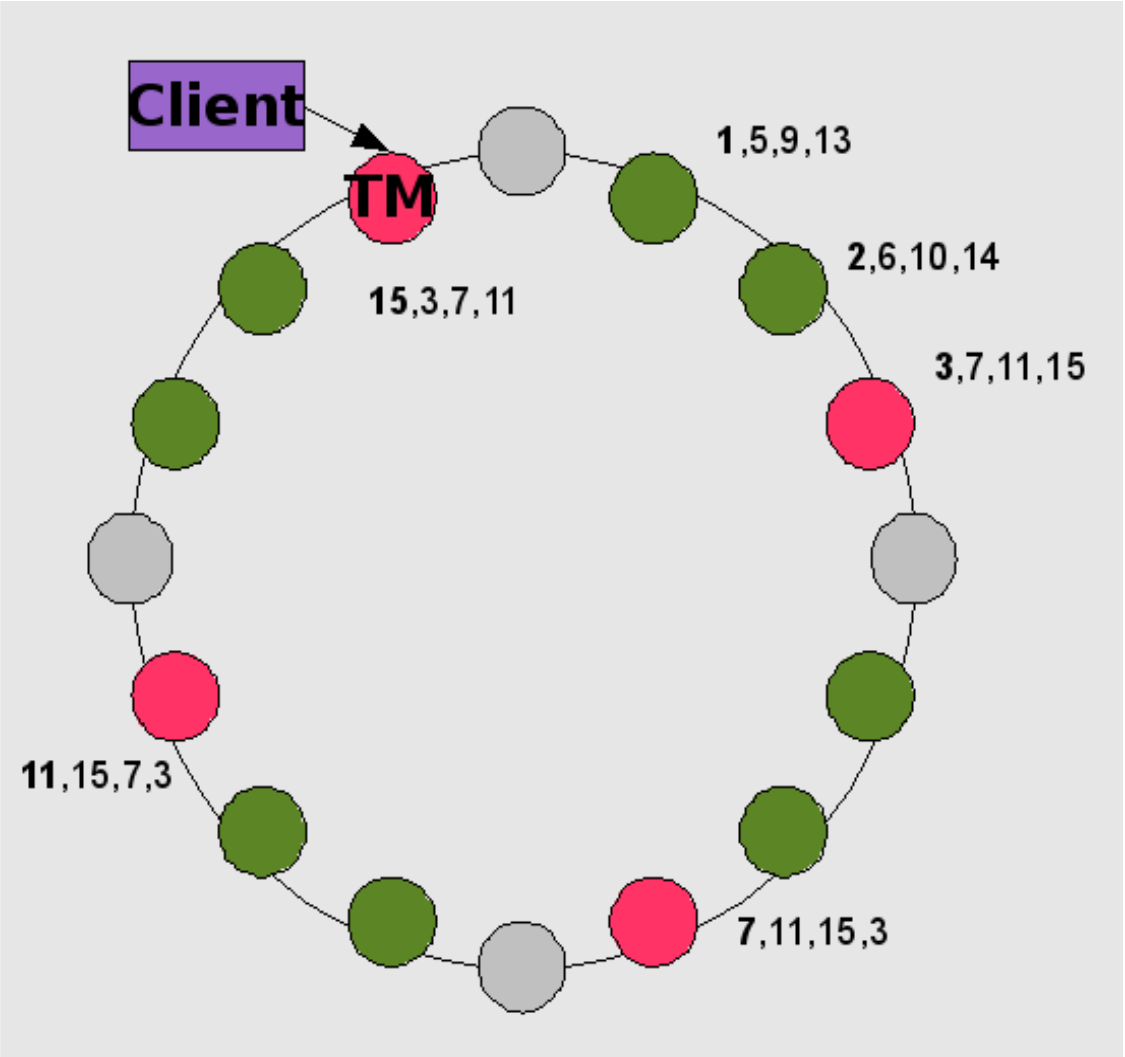
Atomic Commit in a DHT

- Node 15 becomes the Transaction Manager (*TM*)
- Nodes 3, 7, 11 become replicated Transaction Managers (*rTM*), according to the replication of the transaction item



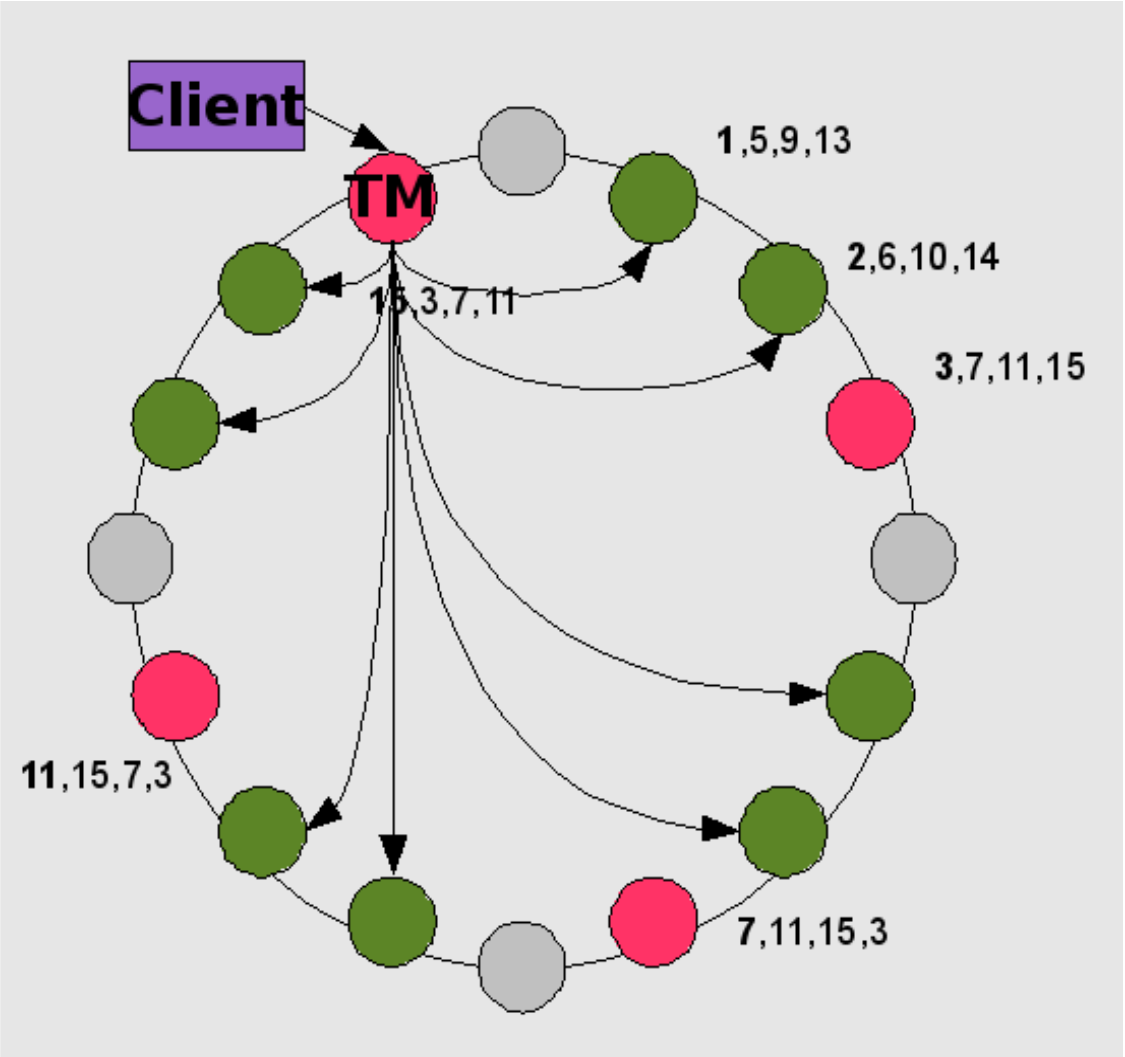
Atomic Commit in a DHT

- Nodes 1,2,5,6,9,10,13,14 become Transaction Participants (*TP*)



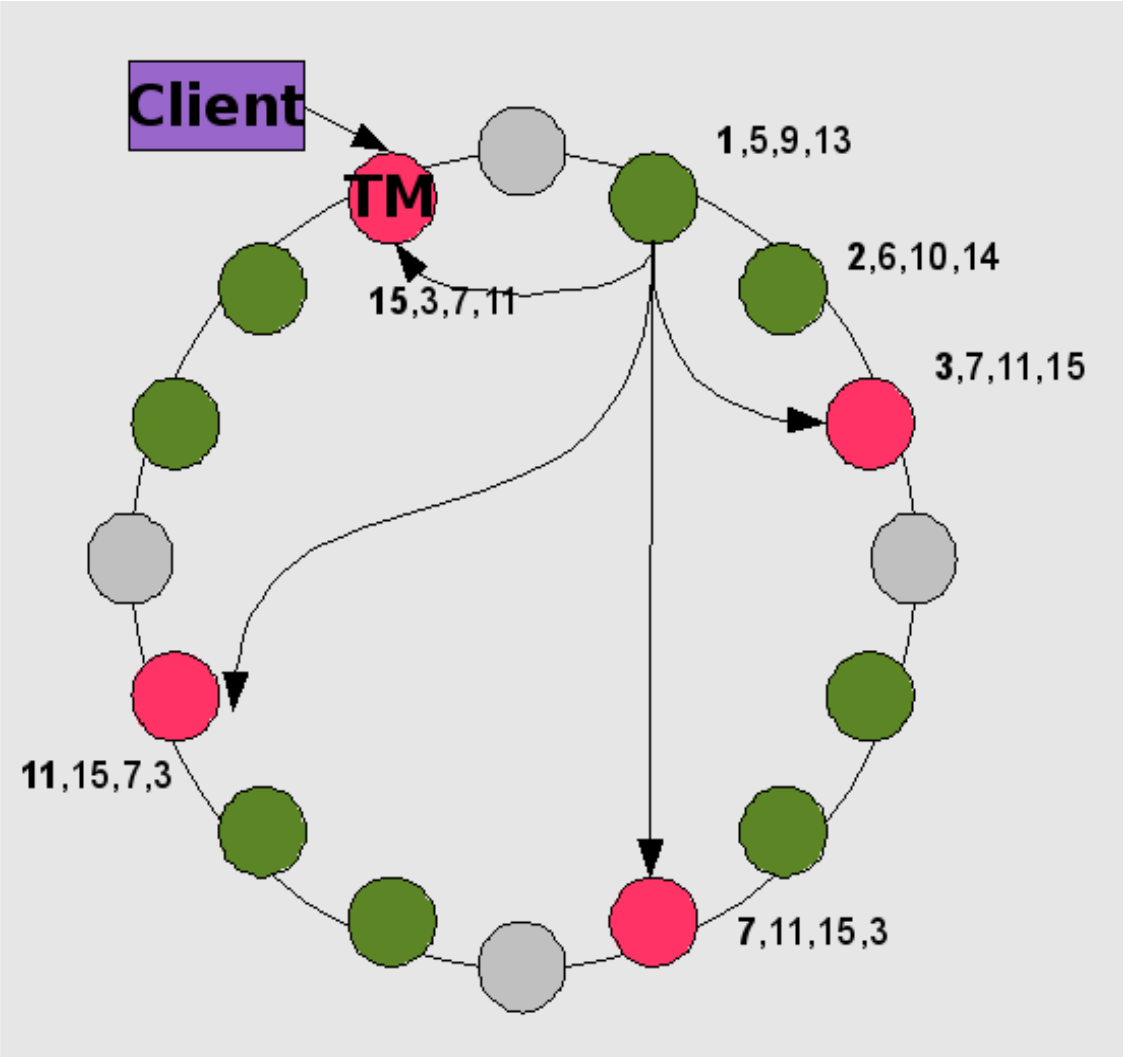
Atomic Commit in a DHT

- *TM* sends `prepare` together with the information needed for validation to all *TPs*



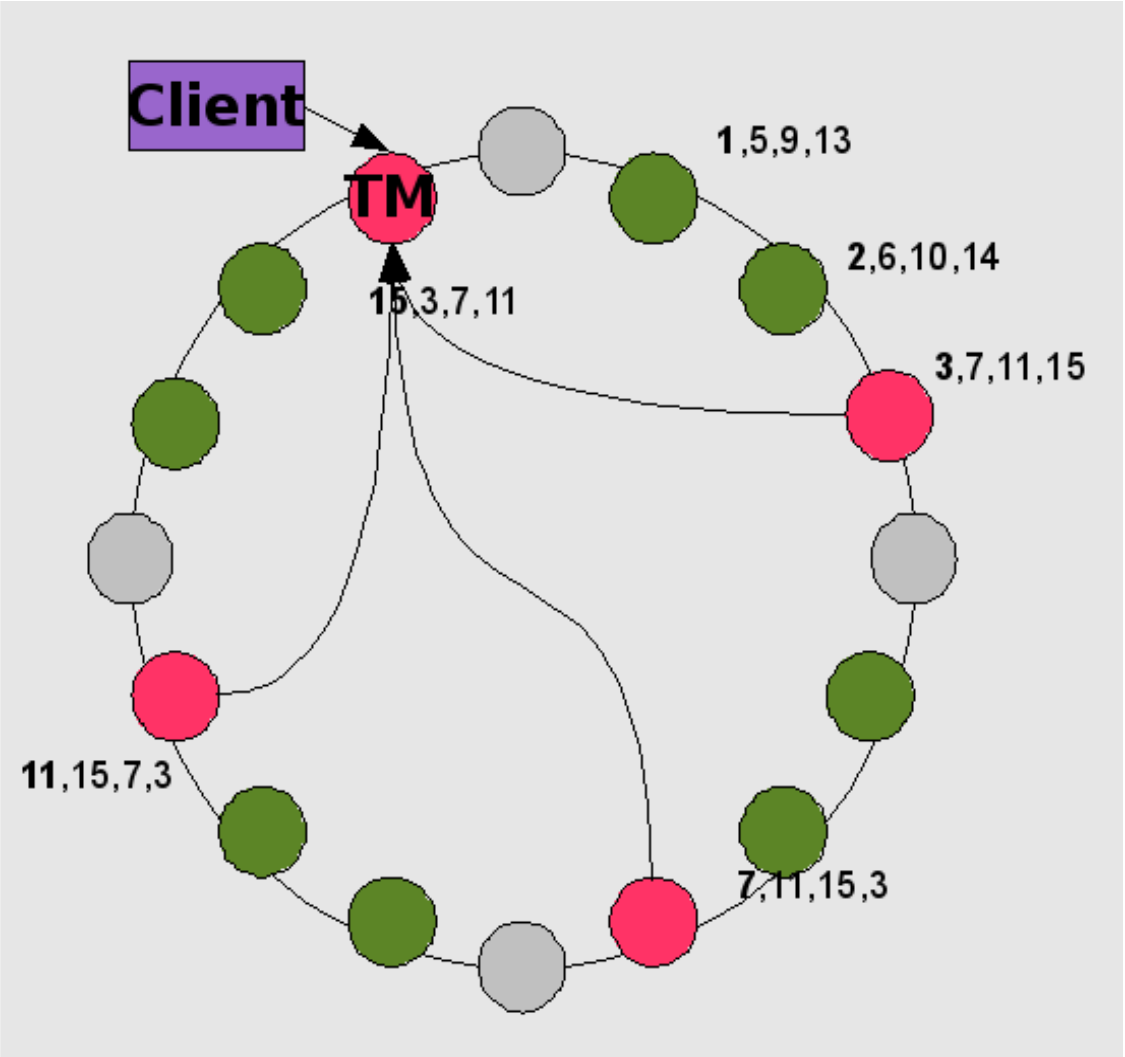
Atomic Commit in a DHT

- After having received `repare` from the *TM*, each *TP* sends it's `prepare` message to all *rTMs*, if it can successfully validate the operation on its item, otherwise it sends `abort`



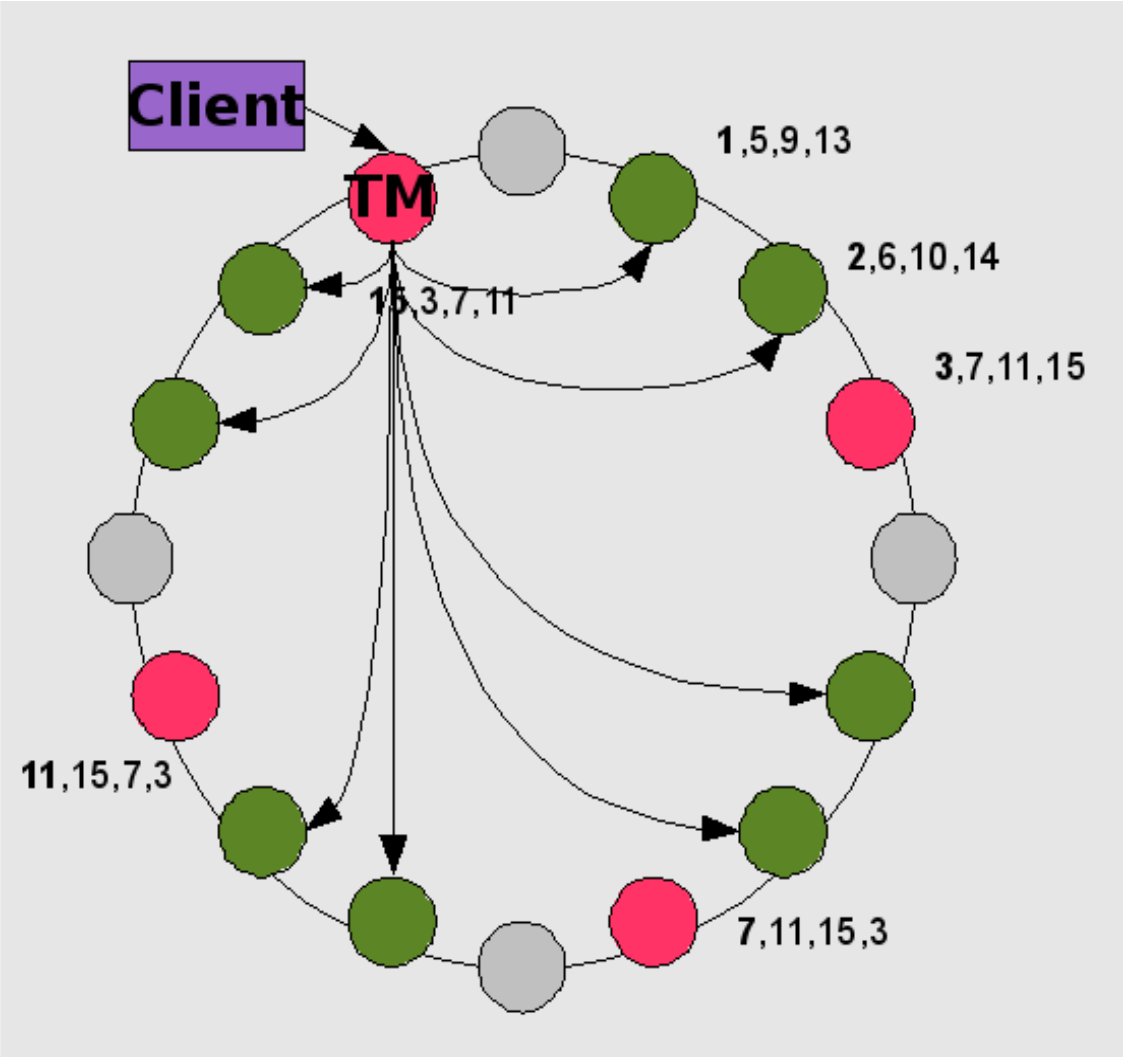
Atomic Commit in a DHT

- The *rTMs* collect the votes from a majority of *TPs* per item and locally decides on `abort` or `commit`
- Each *rTM* sends the outcome to the leading *TM*



Atomic Commit in a DHT

- The *TM* has to collect the outcome from at least a majority of *rTMs*
- After having collected a majority, the *TM* sends the decision to all *TPs*



Beernet

- Mozart-Oz implementation of the relaxed-ring (P2PS's successor)
- Component-based architecture
- Event-driven algorithms with asynchronous message passing and no shared state concurrency
- DHT + Symmetric replication
- Transaction layer with Paxos consensus algorithm (M. Moser)
- Survives network partition
- Ring merge based on gossip algorithm (T. Mahmood)

Conclusion

- The Relaxed-Ring
  - Consistent lookup
  - Realistic failure detection
  - Self-organisation: It handles joins/leaves of peers re-organising the network in a autonomous fashion.
  - Self-healing: It recovers from failures and survives network partitioning.

Conclusion and Future Work

- The Relaxed-Ring
  - Consistent lookup
  - Realistic failure detection
  - Self-organisation: It handles joins/leaves of peers re-organising the network in a autonomous fashion.
  - Self-healing: It recovers from failures and survives network partitioning.
- Distributed Transactions
  - Non-blocking atomic commit protocol for DHTs
  - Decentralized algorithm with replicated transaction manager
  - Fault tolerance and availability based on symmetric replication
- Beernet: implementation of relaxed-ring with transactional layer

Conclusion and Future Work

- The Relaxed-Ring
  - Consistent lookup
  - Realistic failure detection
  - Self-organisation: It handles joins/leaves of peers re-organising the network in a autonomous fashion.
  - Self-healing: It recovers from failures and survives network partitioning.
- Distributed Transactions
  - Non-blocking atomic commit protocol for DHTs
  - Decentralized algorithm with replicated transaction manager
  - Fault tolerance and availability based on symmetric replication
- Beernet: implementation of relaxed-ring with transactional layer
- **Future Work**
  - Implement an application with Beernet to conquer the world
  - Work on reversible phase transitions

Last slide where I should ask for *questions*
or simply put a big *question mark*