The Relaxed Approach for the Greater Good of Functionality and Elasticity

Boriss Mejías
Université catholique de Louvain, Belgium

March 14, 2011

Motivation

- Build **scalable** distributed systems with **self-managing** behaviour and **transactional robust** storage

Motivation

- Build **scalable** distributed systems with **self-managing** behaviour and **transactional robust** storage
- Build **elastic** systems for **cloud computing**

Motivation

- Build **scalable** distributed systems with **self-managing** behaviour and **transactional robust** storage
- Build **elastic** systems for **cloud computing**
- Decentralized architecture
  - Scalable: no central point of congestion
  - Fault-tolerant: no single point of failure

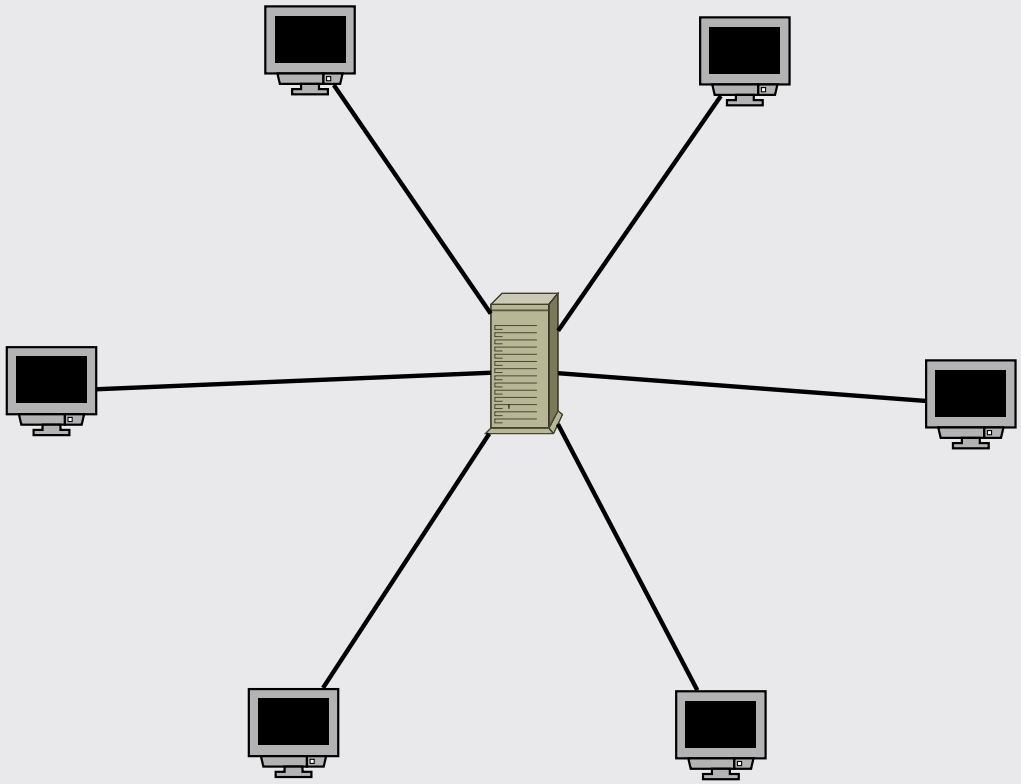
Motivation

- Build **scalable** distributed systems with **self-managing** behaviour and **transactional robust** storage
- Build **elastic** systems for **cloud computing**
- Decentralized architecture
  - Scalable: no central point of congestion
  - Fault-tolerant: no single point of failure
- Replicated storage
  - To achieve robustness
  - Dynamic allocation of replicas
  - Transactional support to update data

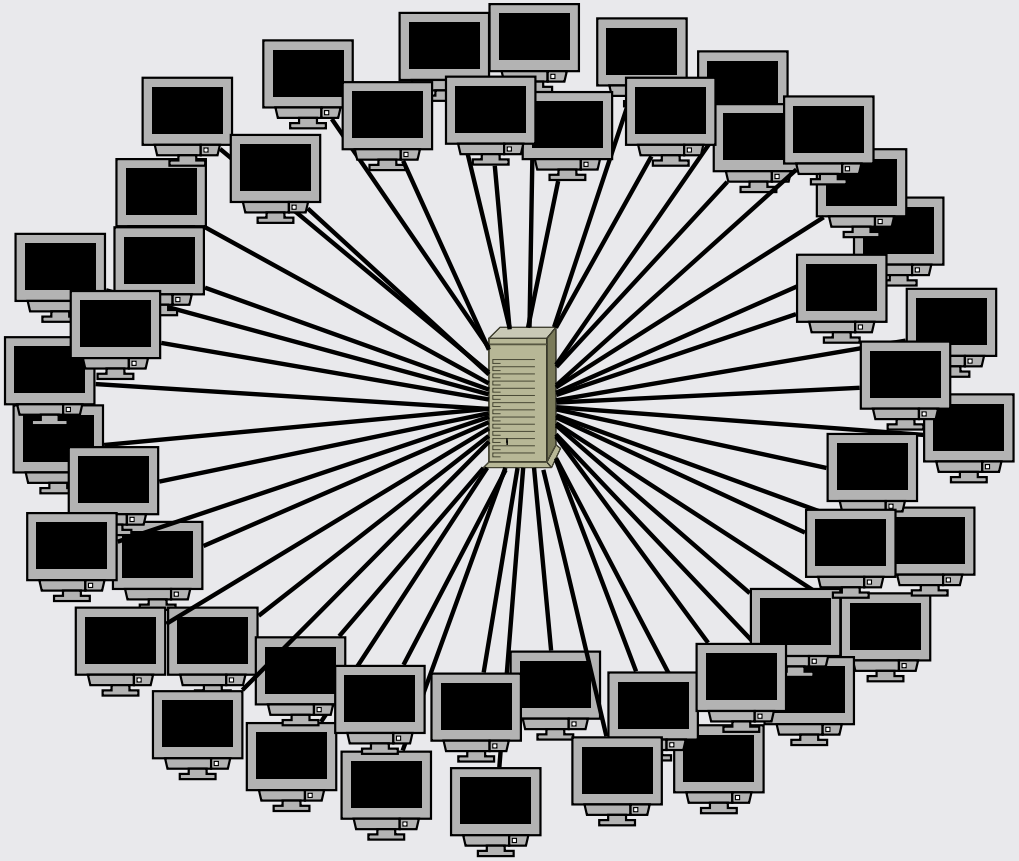
Motivation

- Build **scalable** distributed systems with **self-managing** behaviour and **transactional robust** storage
- Build **elastic** systems for **cloud computing**
- Decentralized architecture
  - Scalable: no central point of congestion
  - Fault-tolerant: no single point of failure
- Replicated storage
  - To achieve robustness
  - Dynamic allocation of replicas
  - Transactional support to update data
- Self management
  - Deals with complexity of decentralization
  - We want self-organization and self-healing behaviour.

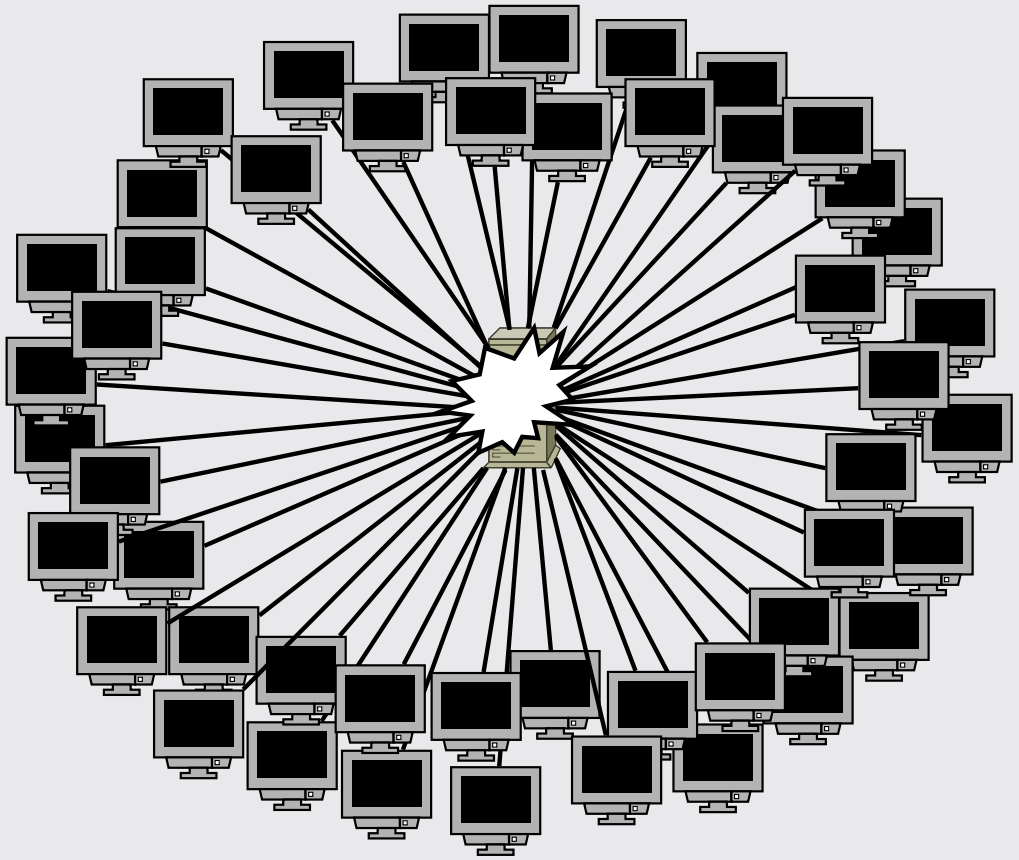
The good old client-server architecture



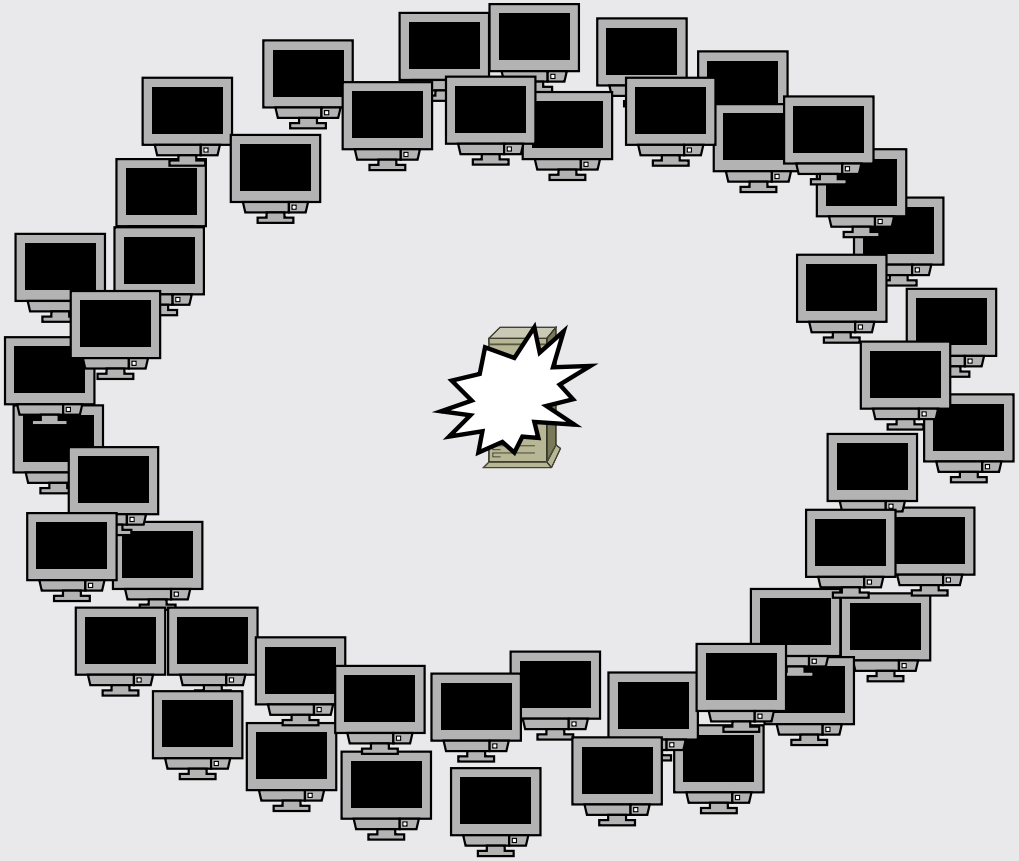
The good old client-server architecture



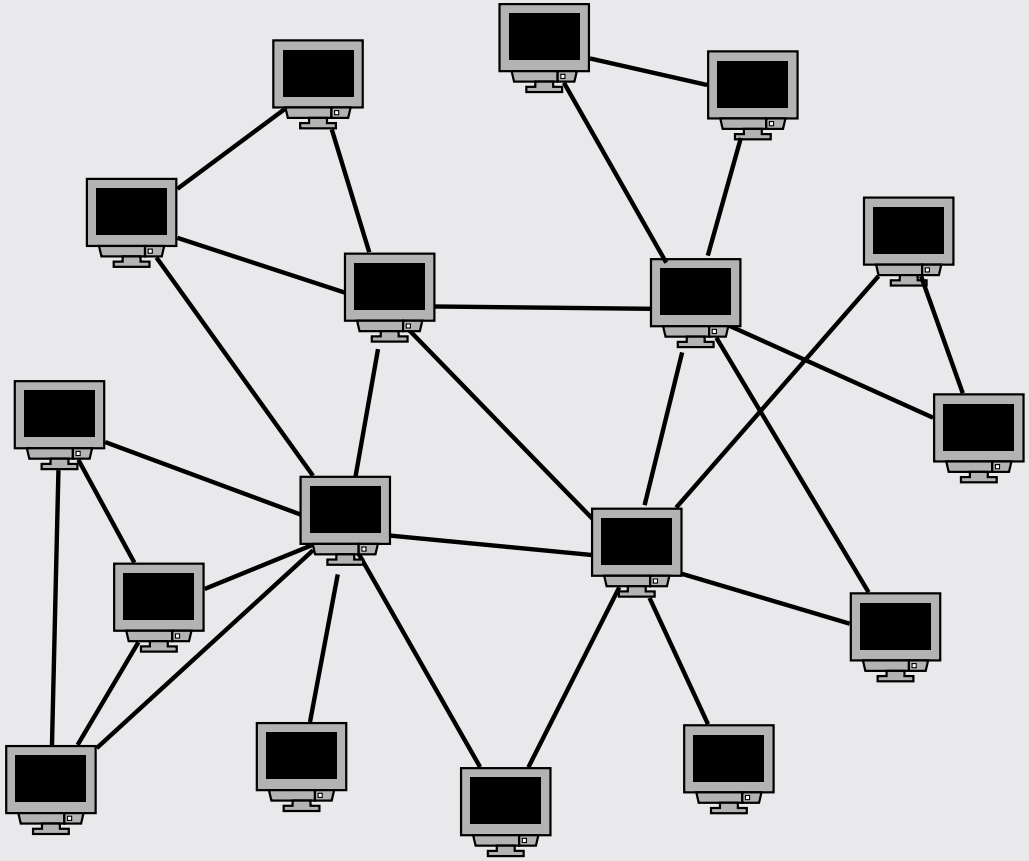
The good old client-server architecture



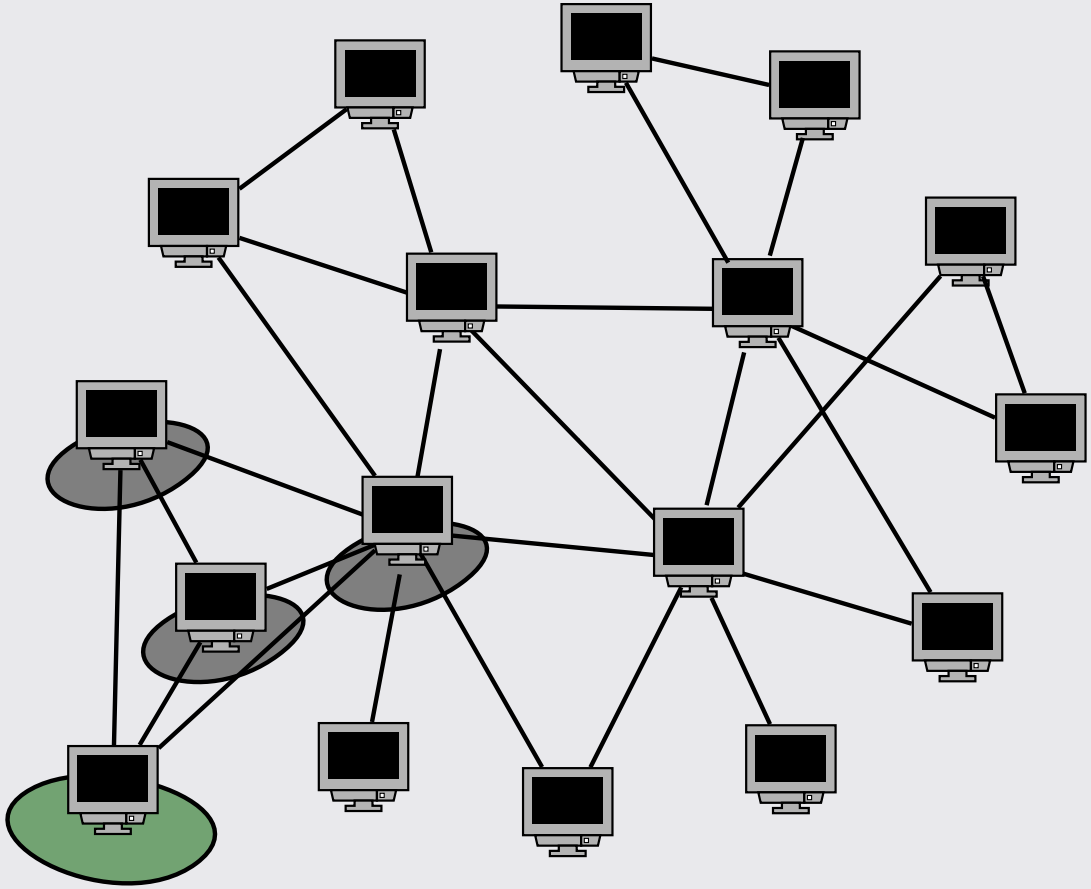
The good old client-server architecture



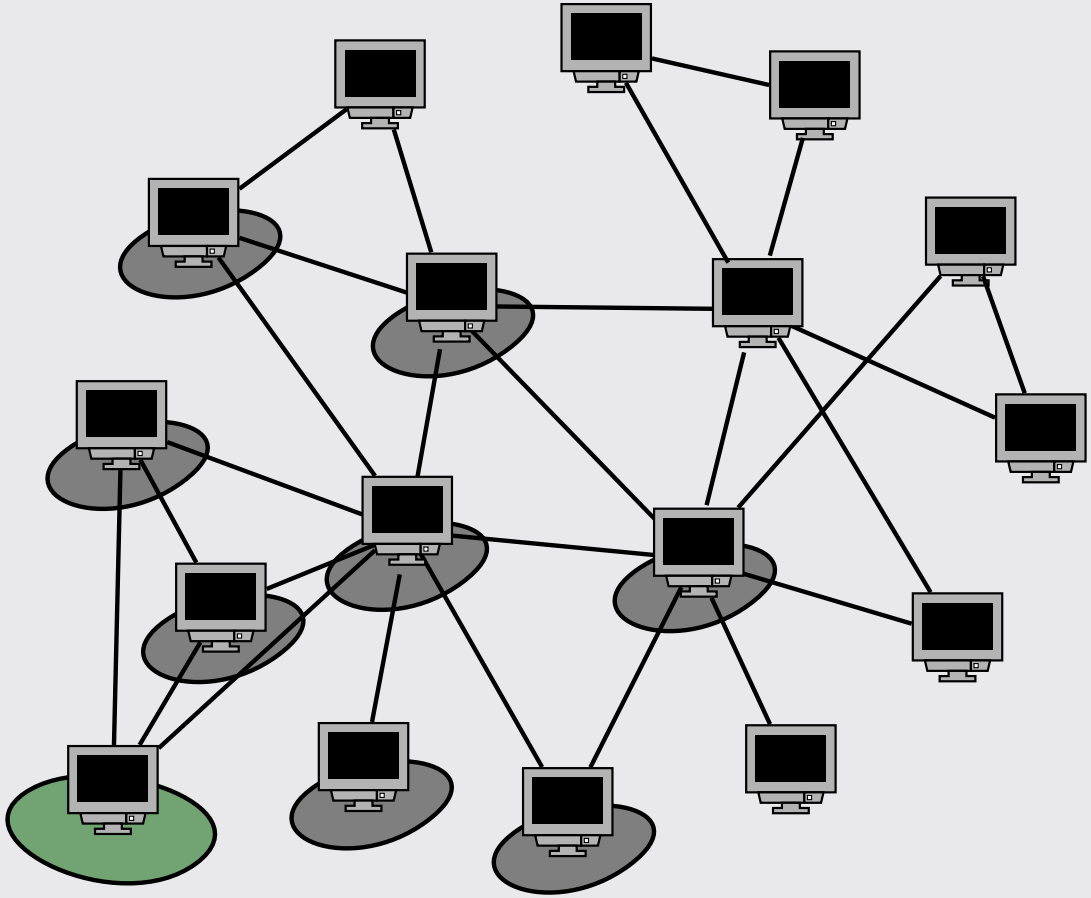
Decentralized systems (unstructured)



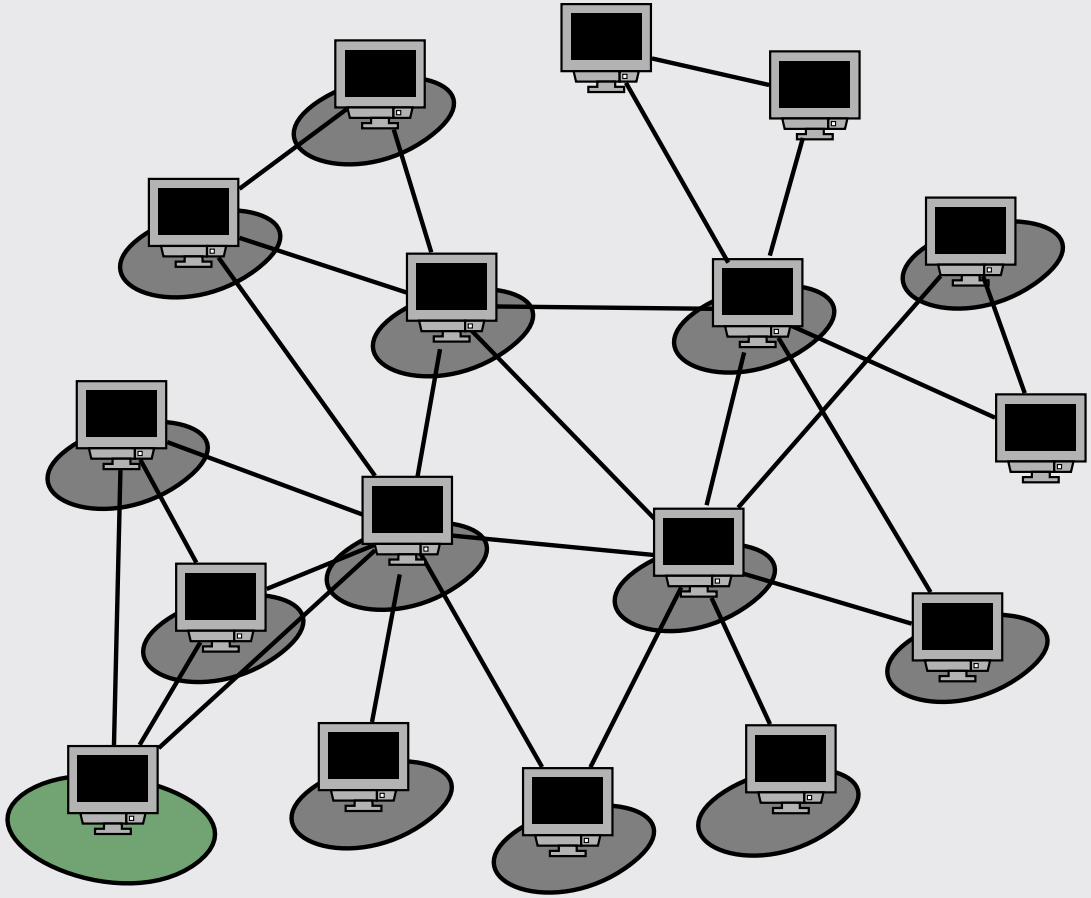
Decentralized systems (flooding)



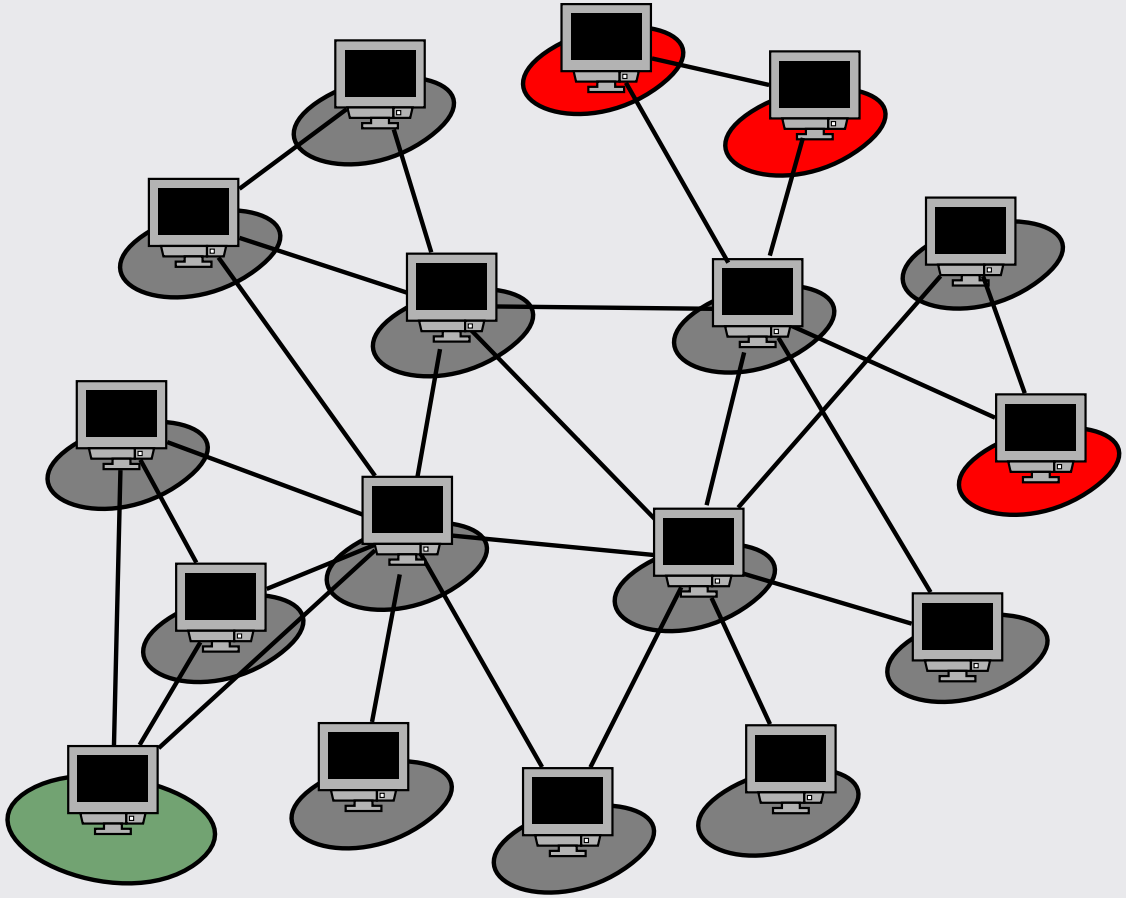
Decentralized systems (flooding)



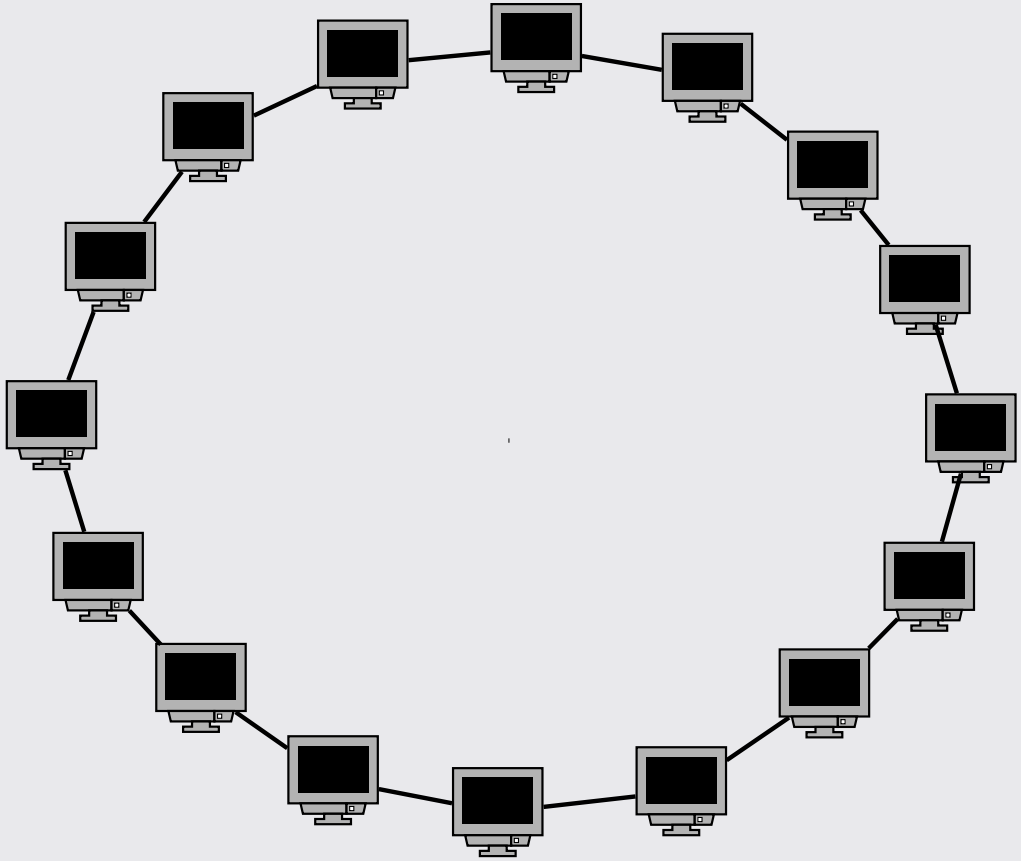
Decentralized systems (flooding)



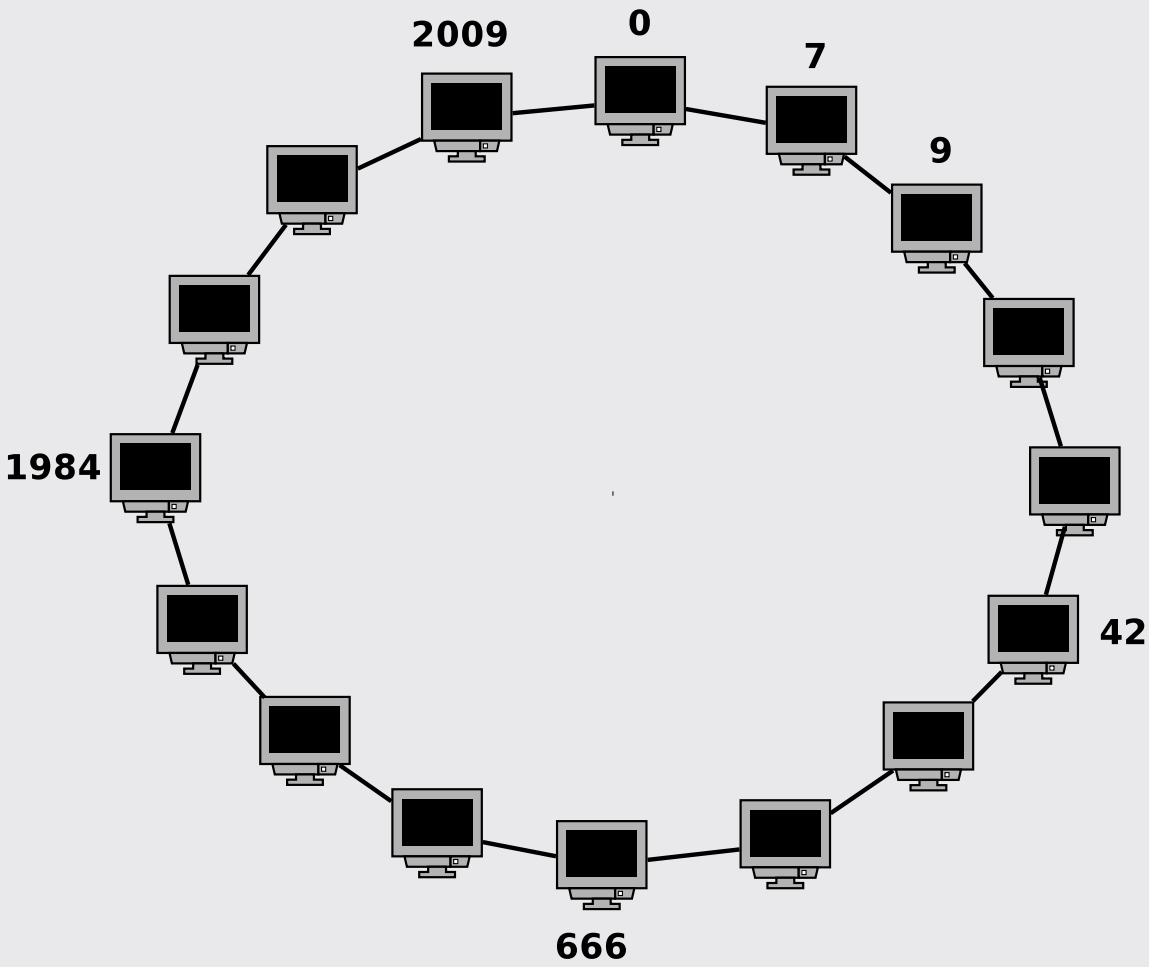
Decentralized systems (flooding)



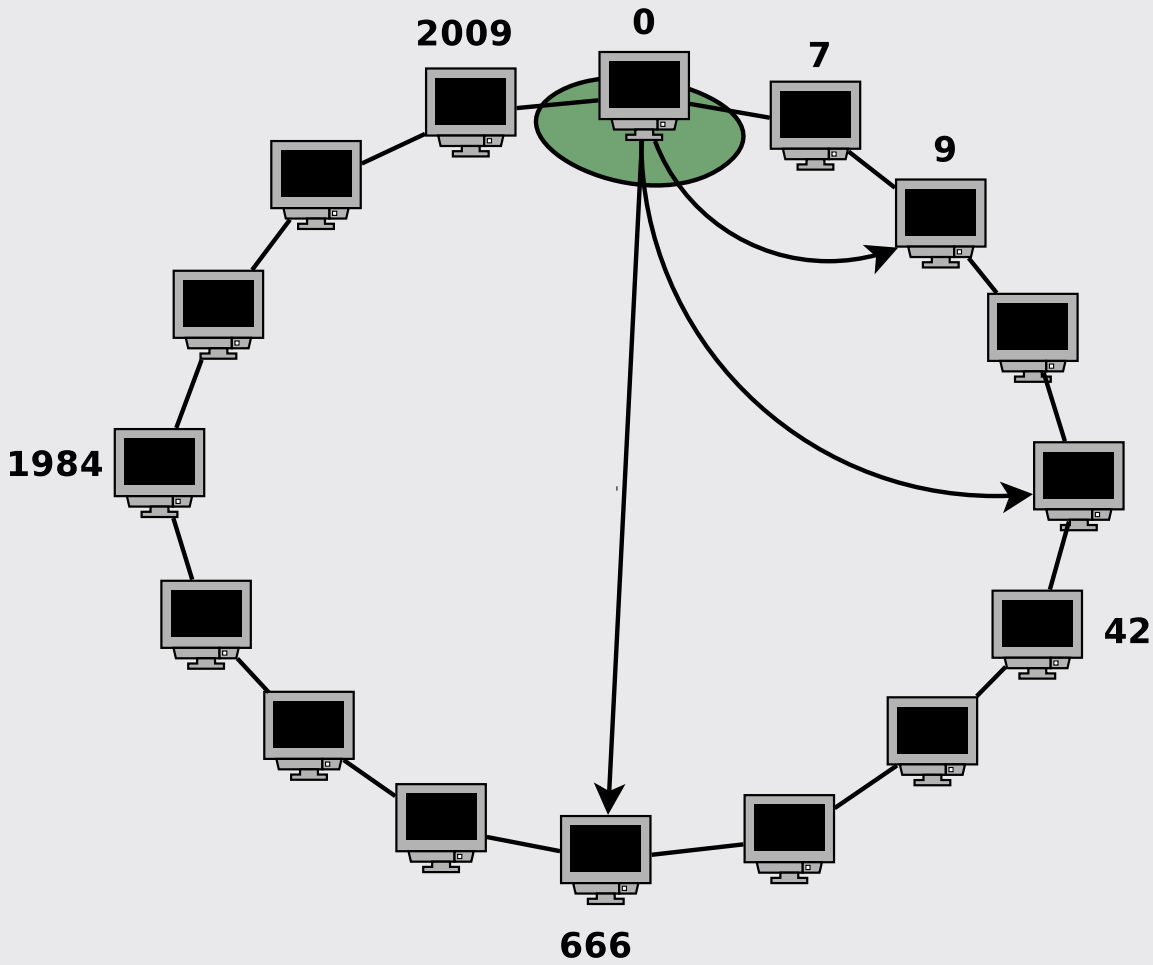
One ring to rule them all



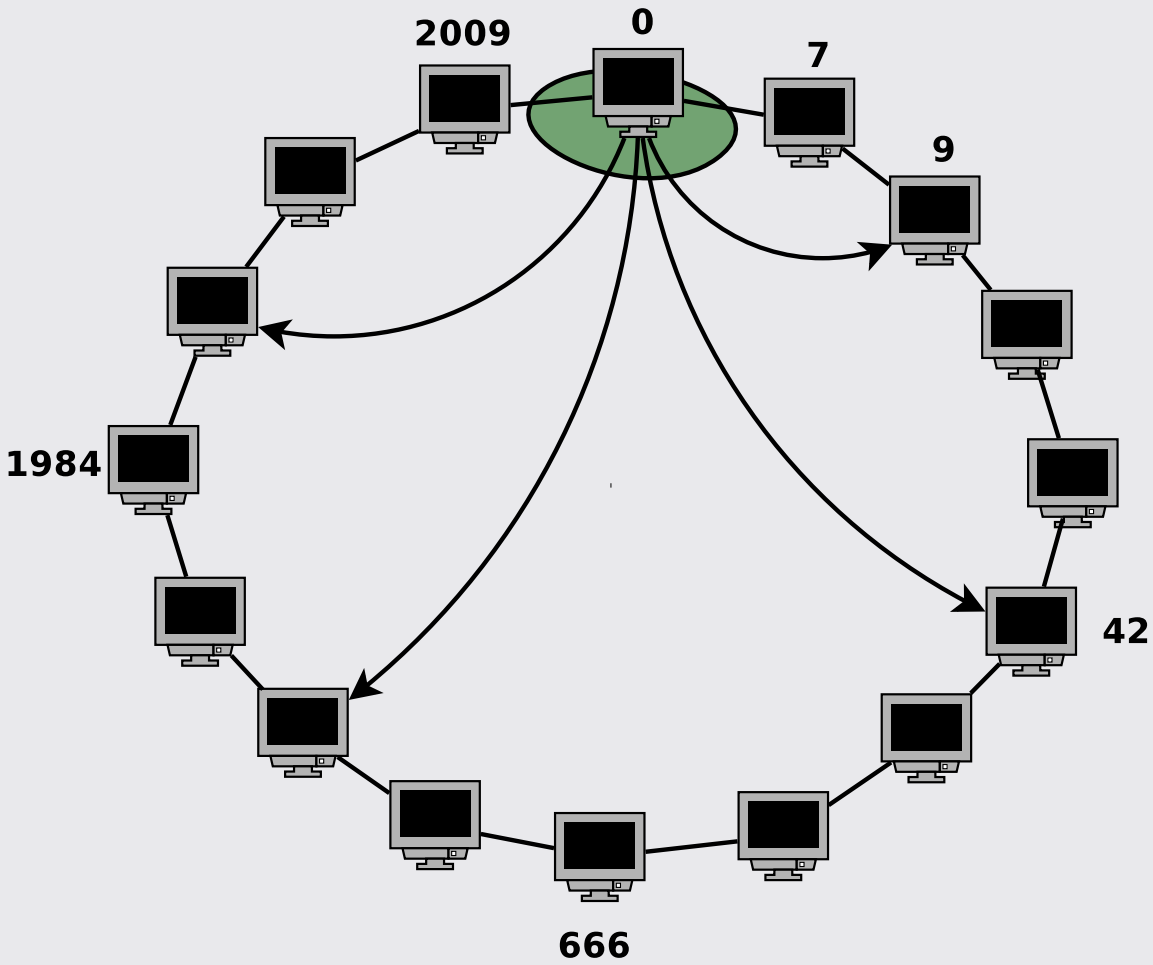
One ring to rule them all



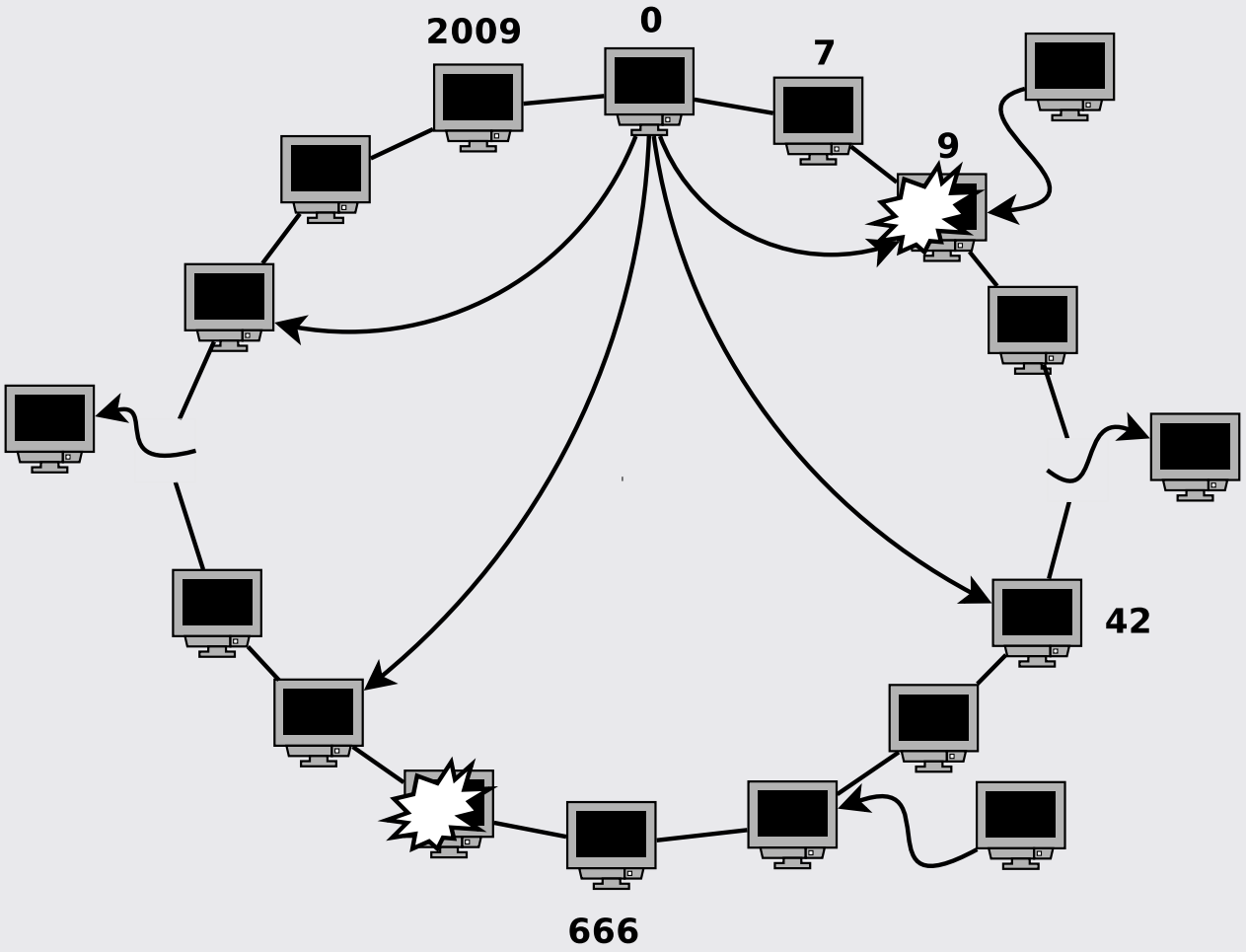
One ring to find them



One ring to find them



Need for self management



The Relaxed Approach

The relaxed approach is a design process for systematically relaxing system requirements

The Relaxed Approach

The relaxed approach is a design process for systematically relaxing system requirements to allow it to cope with the inherent asynchrony of distributed systems without sacrificing functionality.

The Relaxed Approach

The relaxed approach is a design process for systematically relaxing system requirements to allow it to cope with the inherent asynchrony of distributed systems without sacrificing functionality.

1. Identify the desired functionality and the requirements of the system to provide them.

The Relaxed Approach

The relaxed approach is a design process for systematically relaxing system requirements to allow it to cope with the inherent asynchrony of distributed systems without sacrificing functionality.

1. Identify the desired functionality and the requirements of the system to provide them.
2. Analyse the cost and feasibility of such requirements. If the requirements are too hard to meet, relax them to fit the inherent properties of the system.

The Relaxed Approach

The relaxed approach is a design process for systematically relaxing system requirements to allow it to cope with the inherent asynchrony of distributed systems without sacrificing functionality.

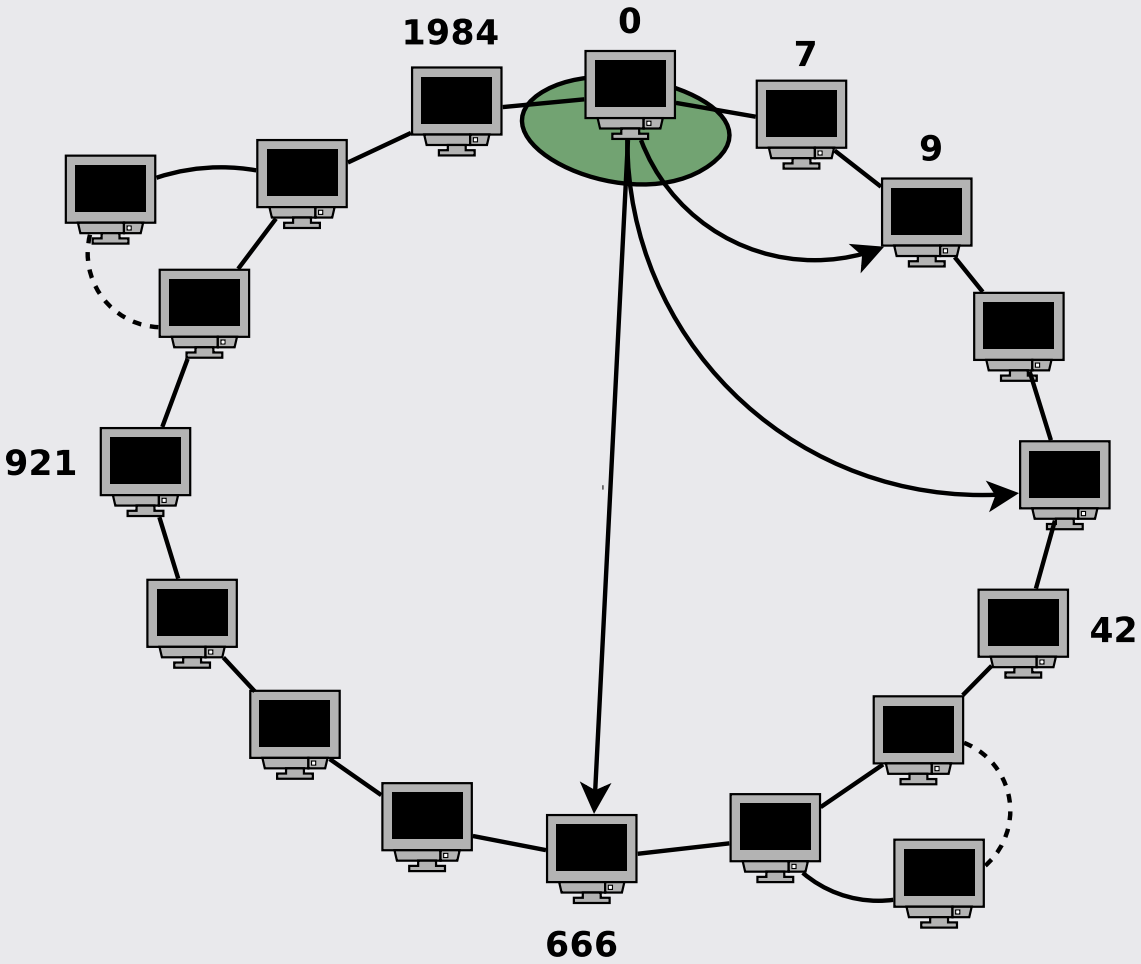
1. Identify the desired functionality and the requirements of the system to provide them.
2. Analyse the cost and feasibility of such requirements. If the requirements are too hard to meet, relax them to fit the inherent properties of the system.
3. Analyse the cost of the relaxation to avoid relaxing too much.

The rest of the presentation

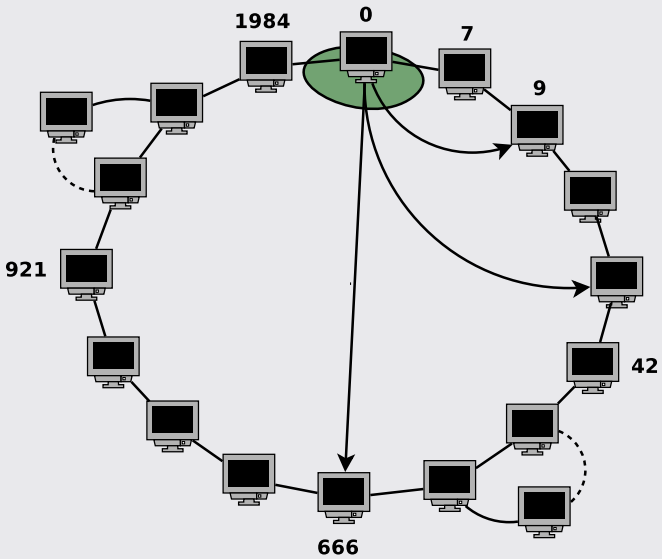
- The relaxed ring - overlay network
- Trappist - transactional storage
- Building applications

The Relaxed Ring

The relaxed ring



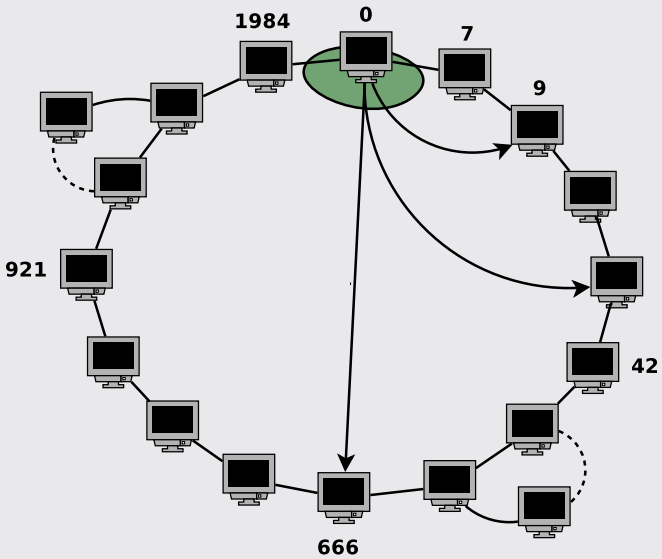
The relaxed ring



Distributed Hash Table

- `put(key, value)`
- `get(key)`
- `lookup(key)`

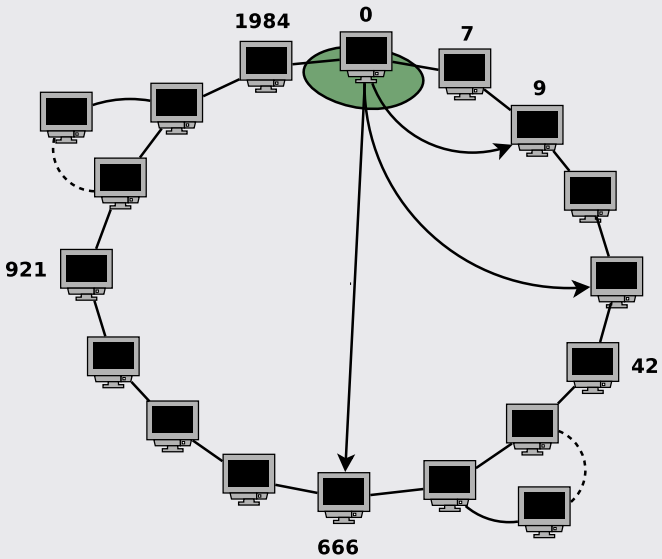
The relaxed ring



Distributed Hash Table

- `put(key, value)`
- `get(key)`
- `lookup(key)`
- Lookup consistency: at any given time, **only one peer is responsible** for a given key, or it is temporary unavailable.
- Responsibility given by `]pred, self]`

The relaxed ring



Distributed Hash Table

- `put(key, value)`
- `get(key)`
- `lookup(key)`
- Lookup consistency: at any given time, **only one peer is responsible** for a given key, or it is temporary unavailable.
- Responsibility given by `]pred, self]`
- Fingers provide efficient routing $O(log_k(N) + b)$

Applying the relaxed approach

1. **Functionality and requirements**:
   - DHT with lookup consistency $\Rightarrow$ perfect successor/predecessor links $\Rightarrow$ transitive connectivity
   - self organization $\Rightarrow$ decentralized architecture

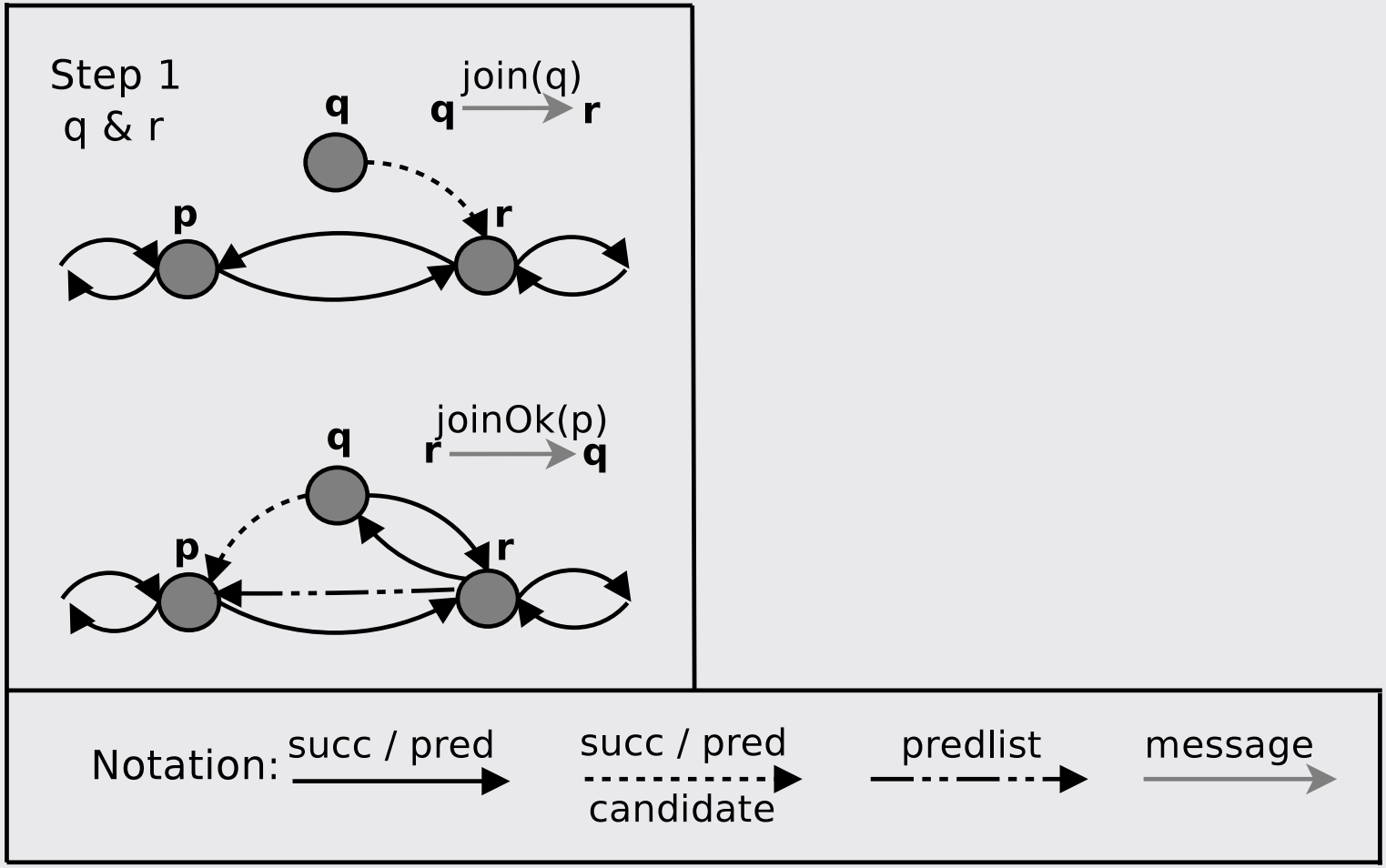
Applying the relaxed approach

1. **Functionality and requirements**:
   - DHT with lookup consistency $\Rightarrow$ perfect successor/predecessor links $\Rightarrow$ transitive connectivity
   - self organization $\Rightarrow$ decentralized architecture
2. **Feasibility of requirements**:
   - Beernet: NAT devices, high latency $\Rightarrow$ no transitive connectivity $\Rightarrow$ relax the ring
   - Chord: No atomic join/leave $\Rightarrow$ relax the join/leave and fix with periodic stabilization

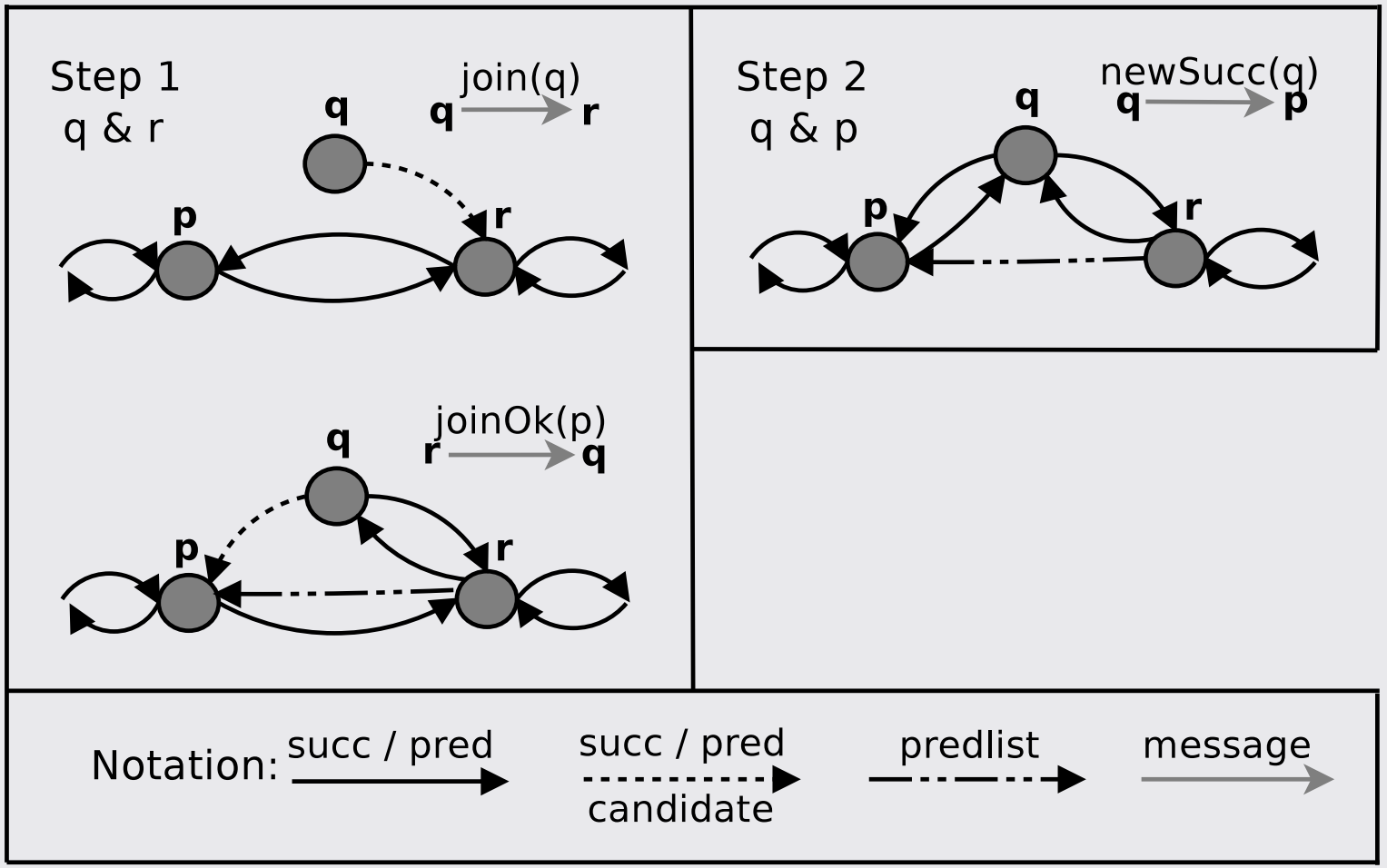
Applying the relaxed approach

1. **Functionality and requirements**:
   - DHT with lookup consistency $\Rightarrow$ perfect successor/predecessor links $\Rightarrow$ transitive connectivity
   - self organization $\Rightarrow$ decentralized architecture
2. **Feasibility of requirements**:
   - Beernet: NAT devices, high latency $\Rightarrow$ no transitive connectivity $\Rightarrow$ relax the ring
   - Chord: No atomic join/leave $\Rightarrow$ relax the join/leave and fix with periodic stabilization
3. **Cost of relaxation**:
   - Beernet: Extra routing to branches if any. Skewed distribution of address space in branches. Cost efficient ring-maintenance
   - Chord: Periodic stabilization is too expensive

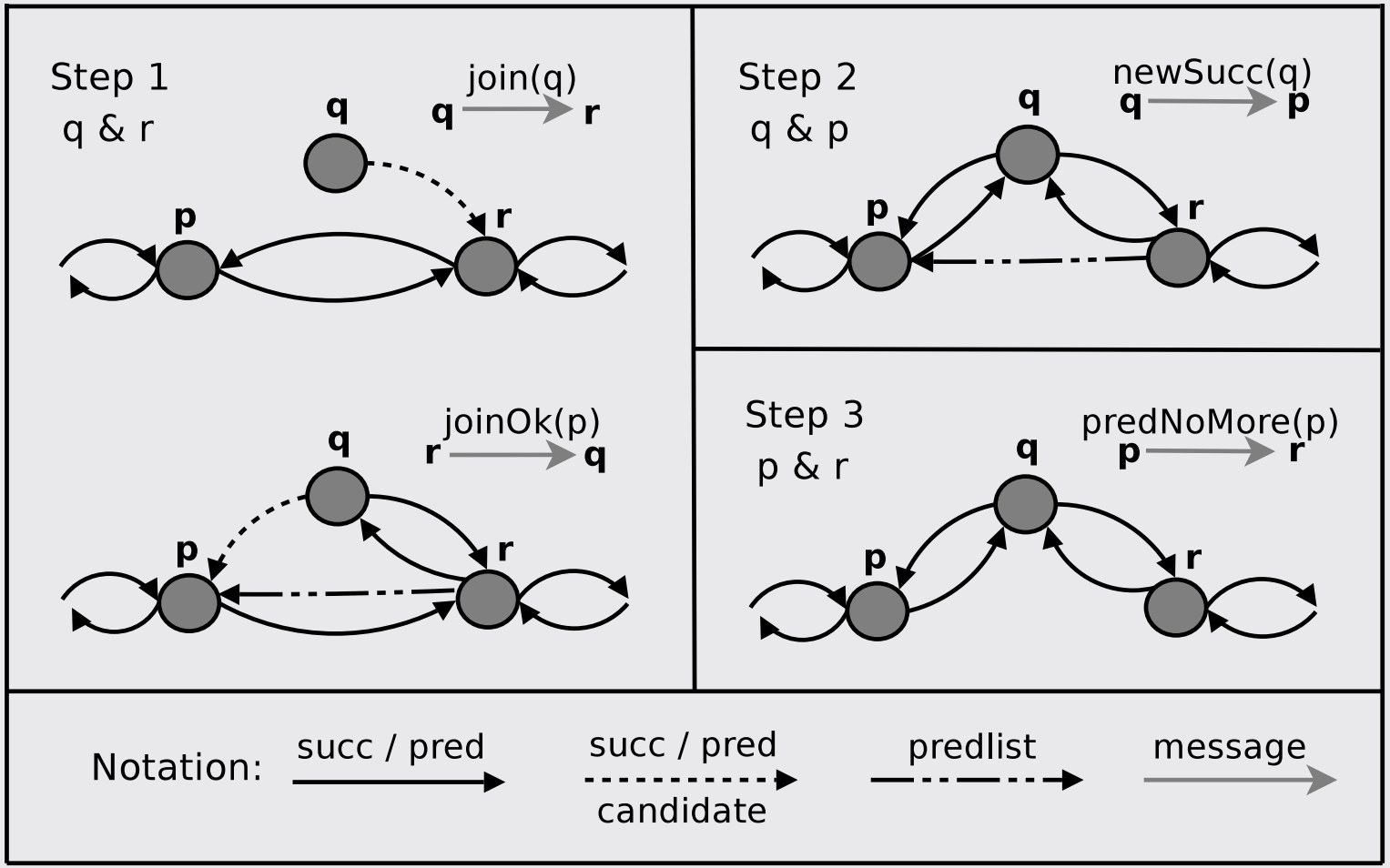
The join algorithm



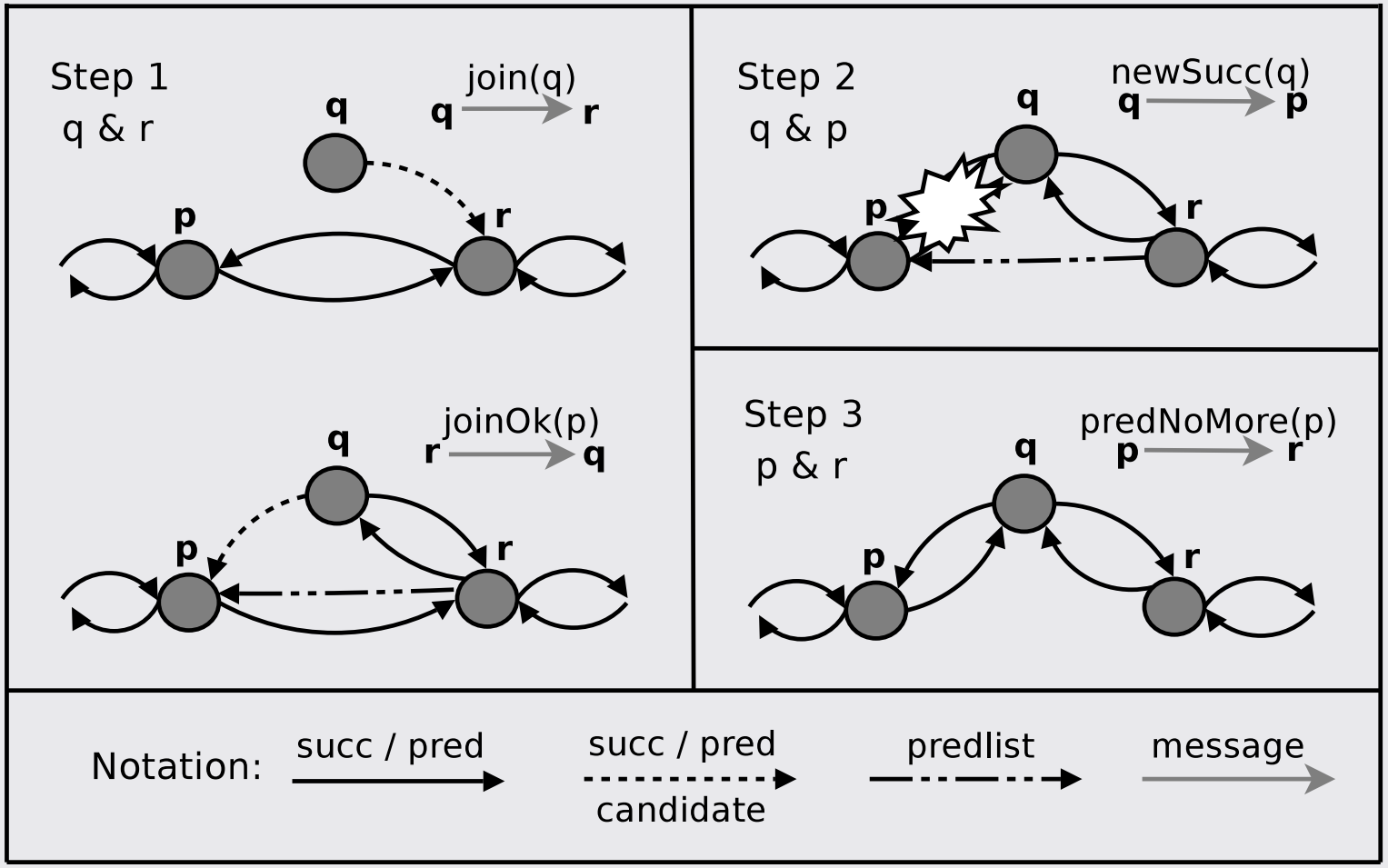
The join algorithm



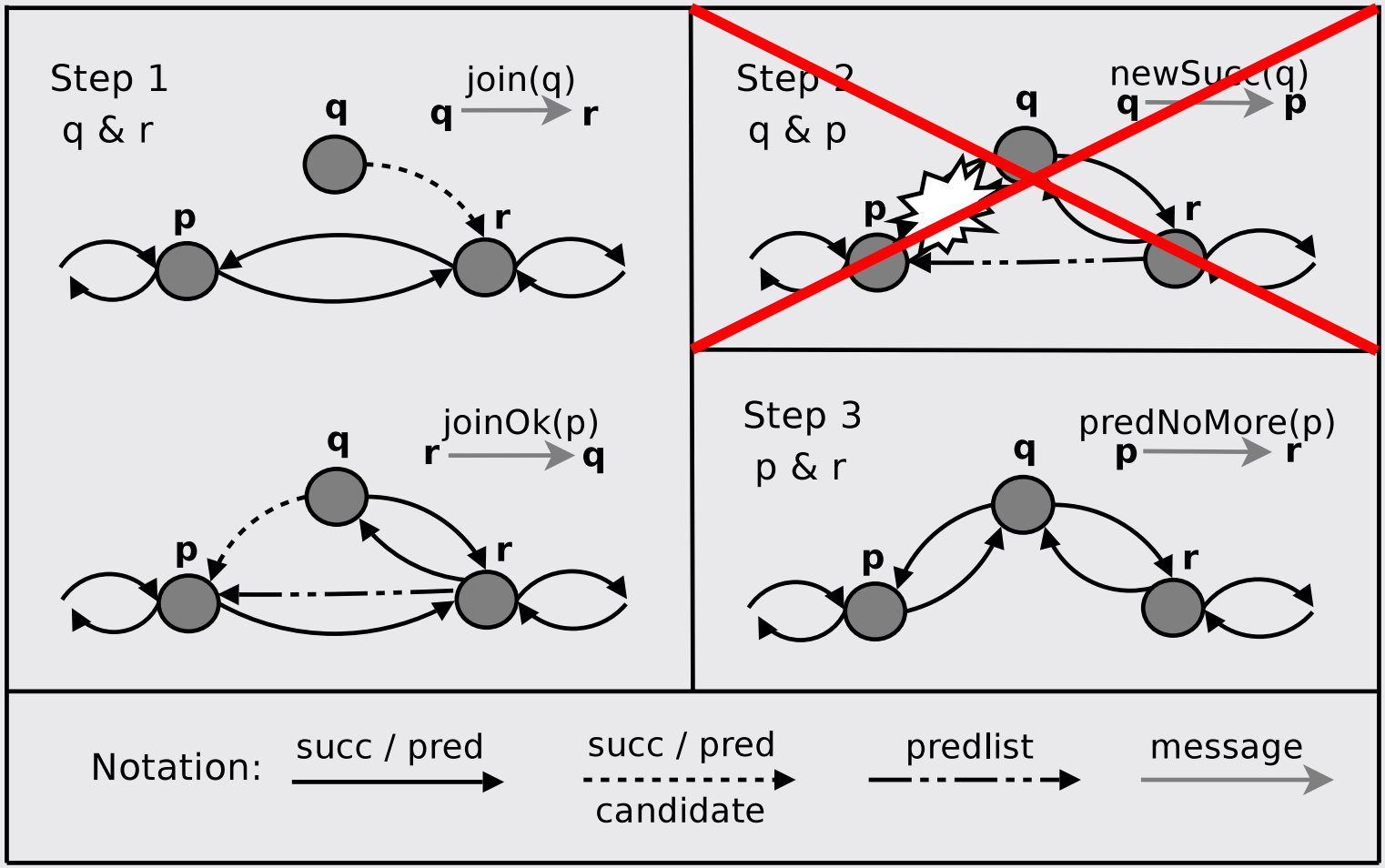
The join algorithm



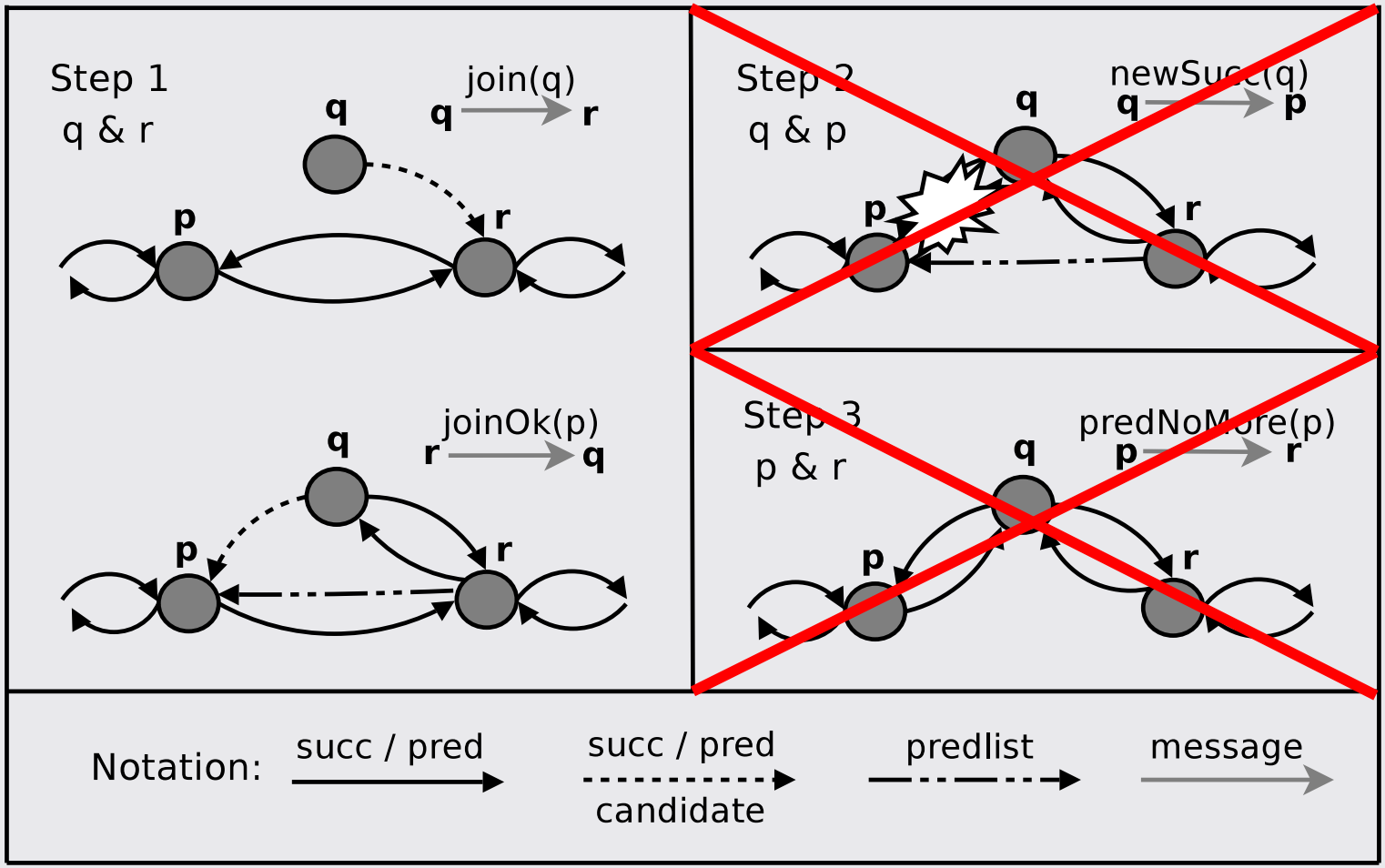
The join algorithm



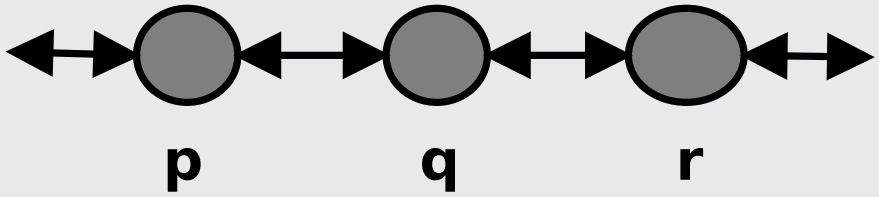
The join algorithm



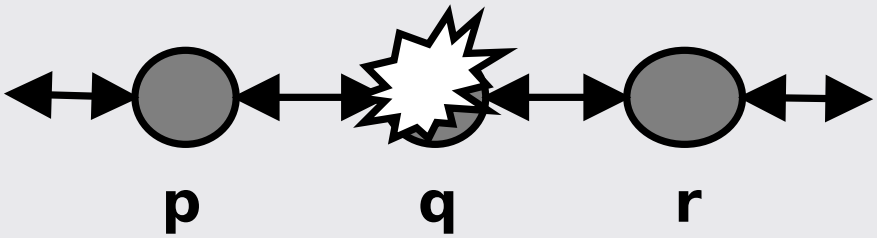
The join algorithm



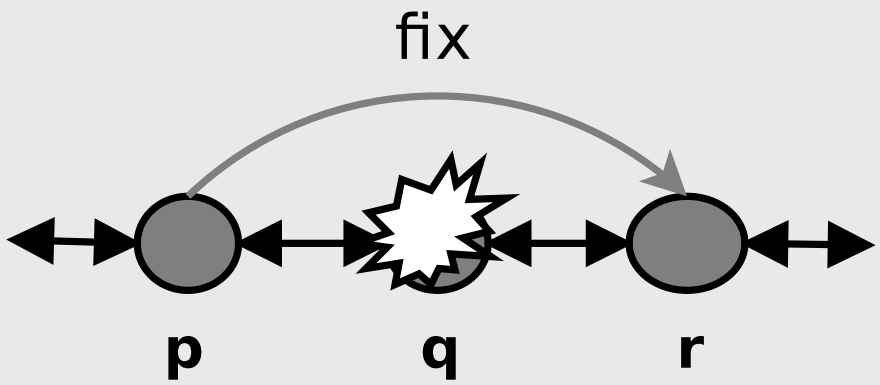
Failure recovery



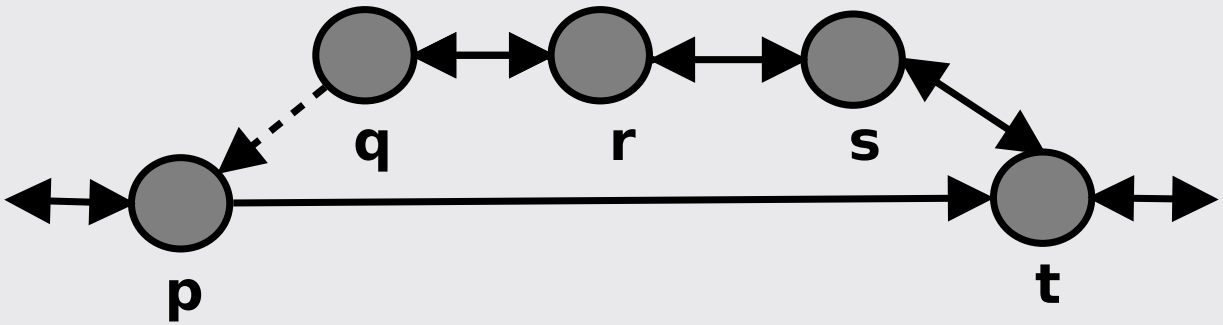
Failure recovery



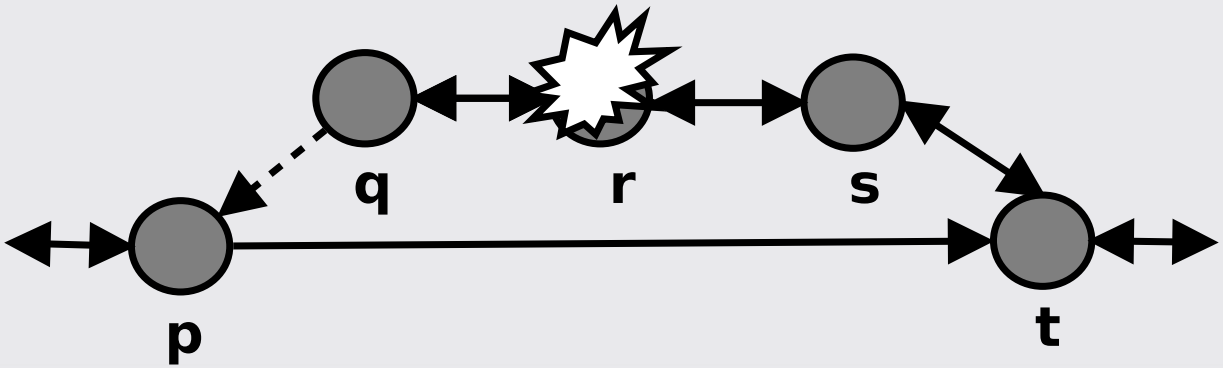
Failure recovery



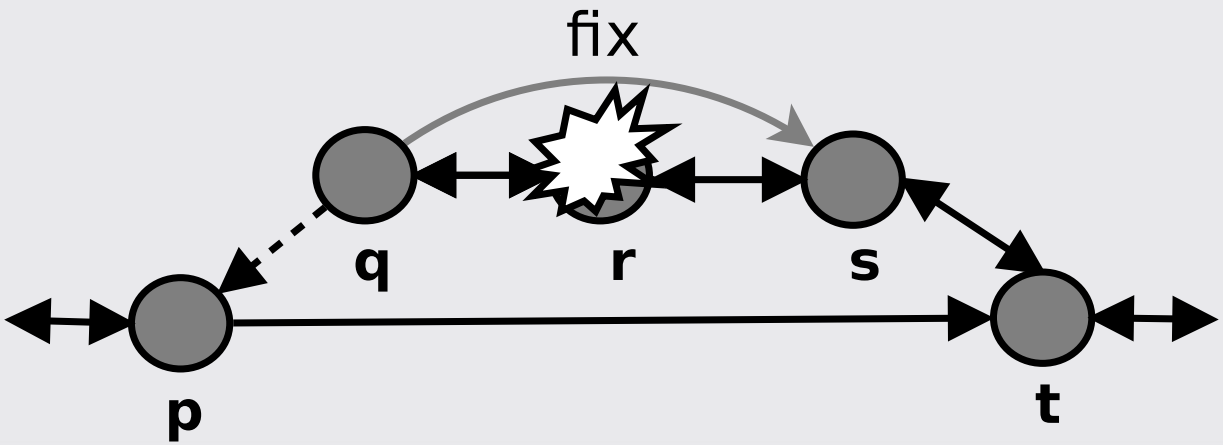
Failure recovery



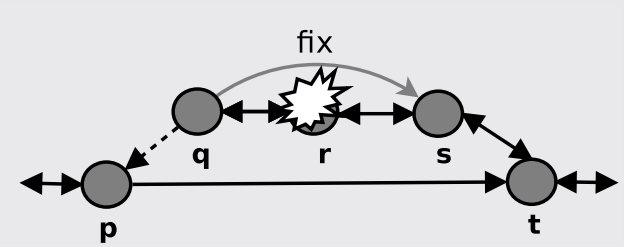
Failure recovery



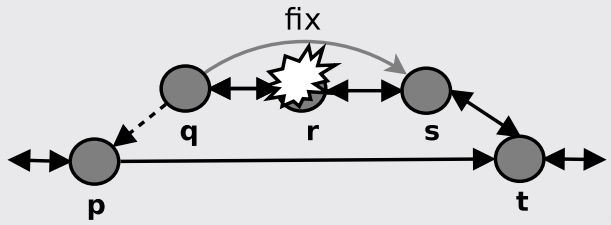
Failure recovery

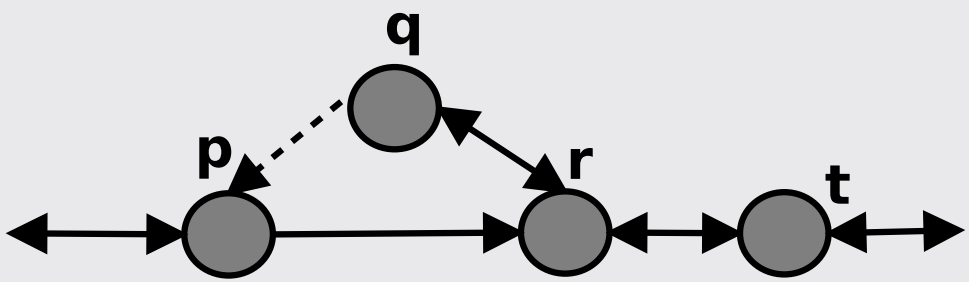


Failure recovery

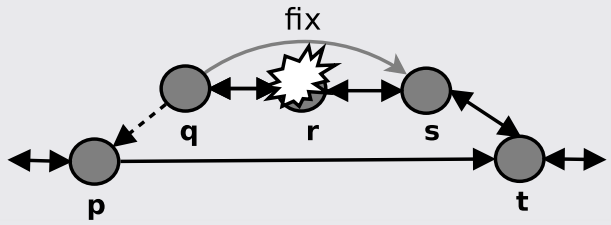


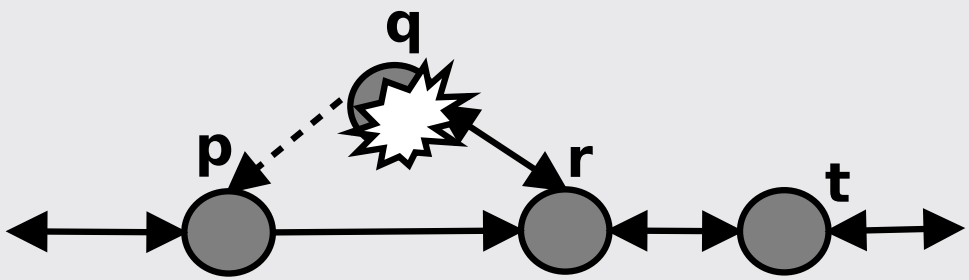
Failure recovery



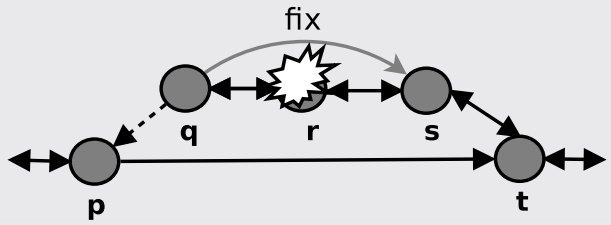


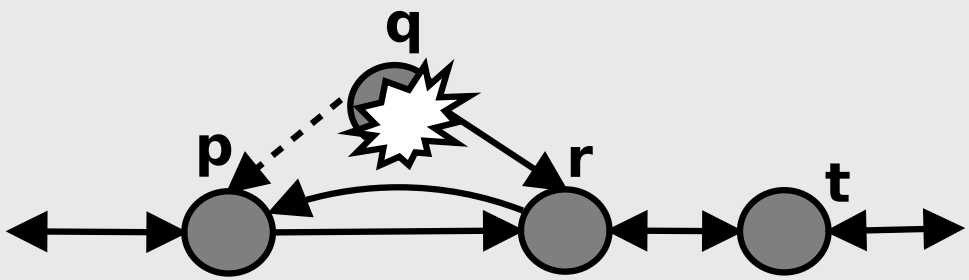
Failure recovery



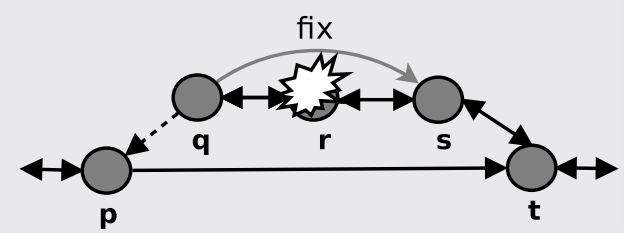


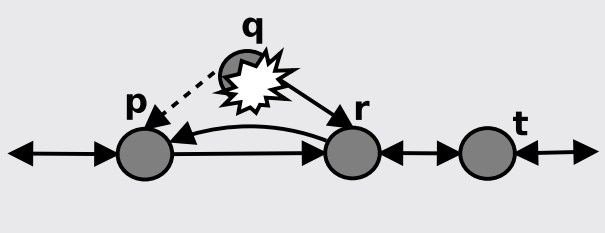
Failure recovery



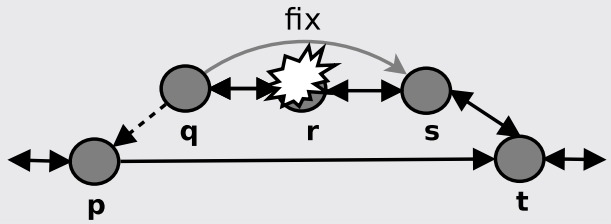


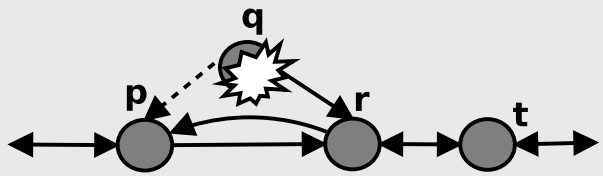
Failure recovery

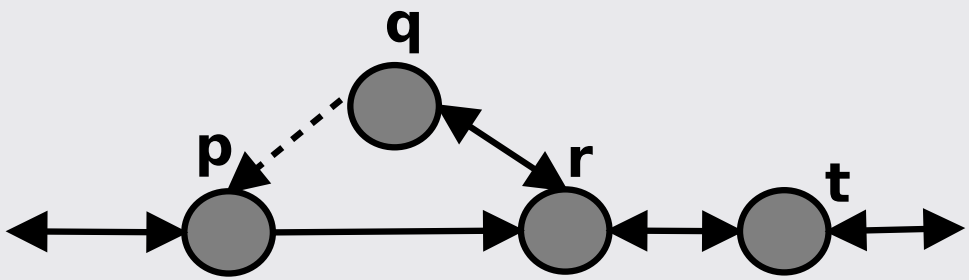




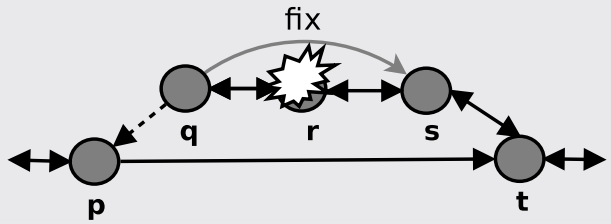
Failure recovery

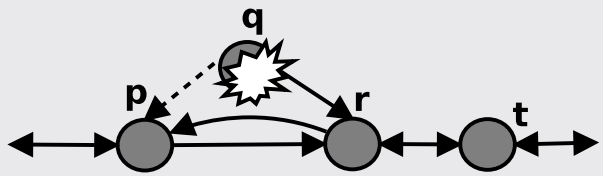


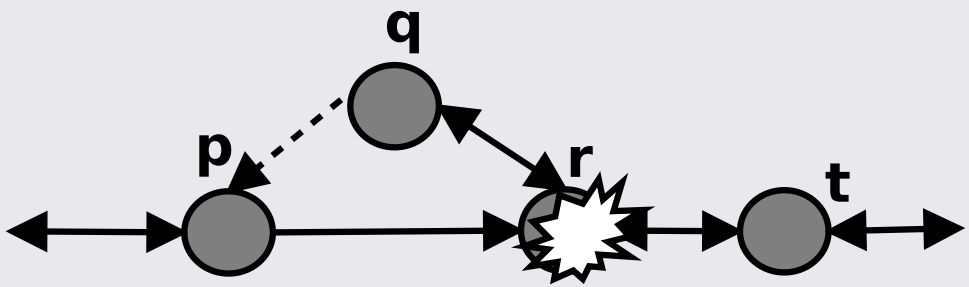




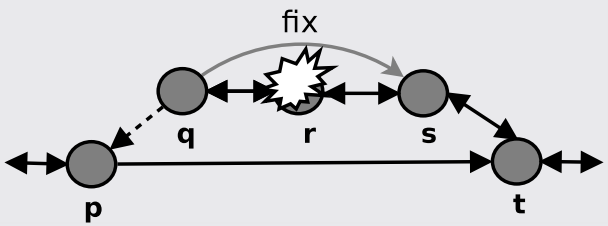
Failure recovery

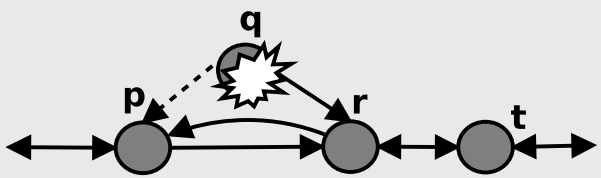


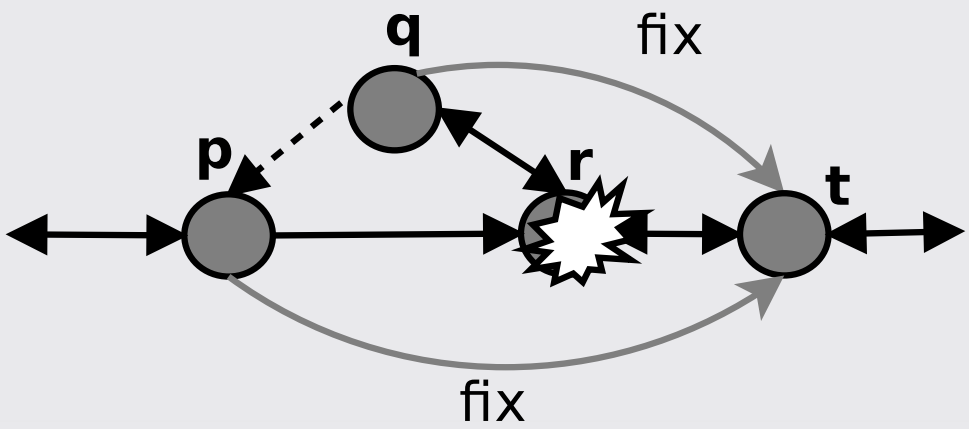




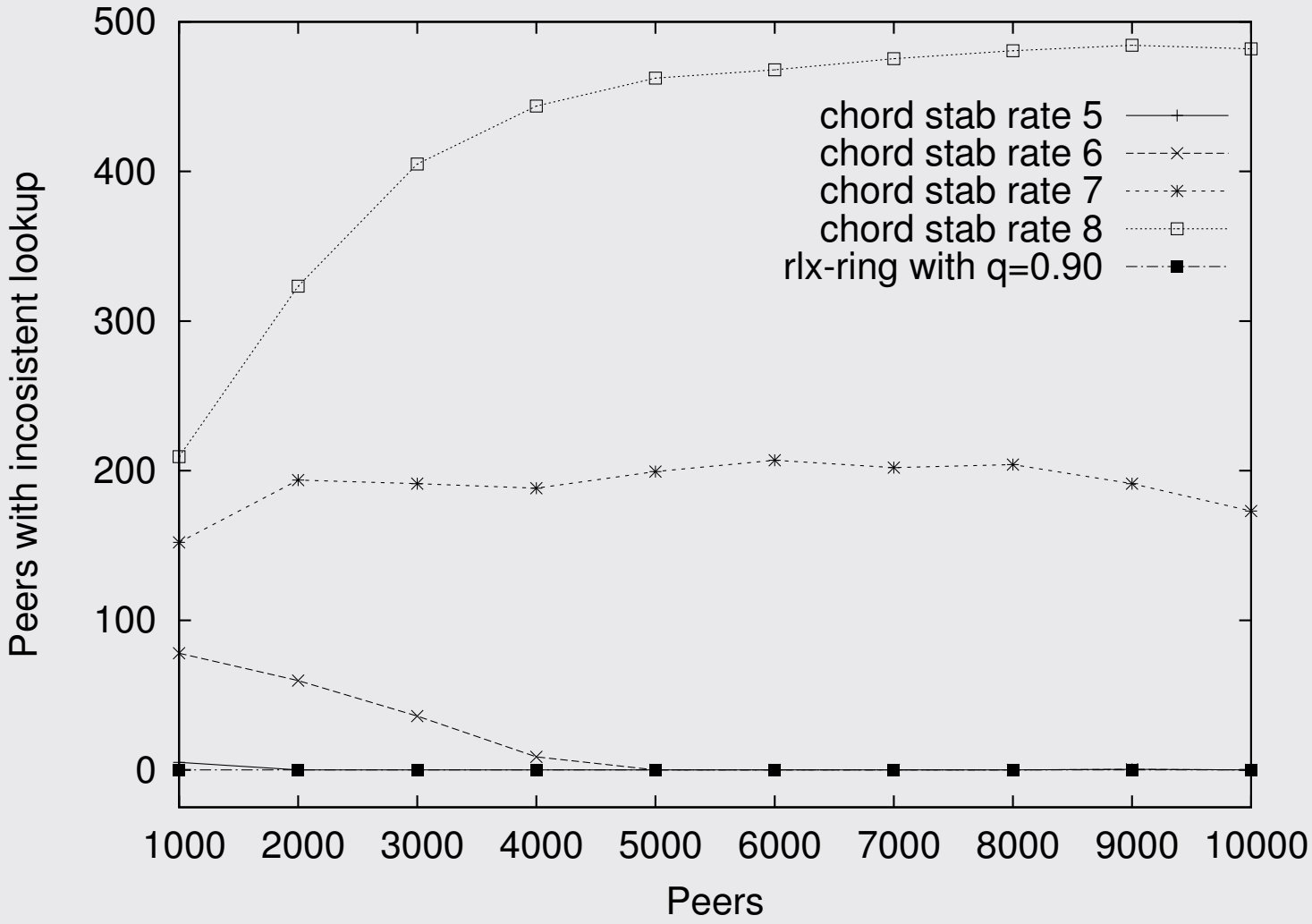
Failure recovery



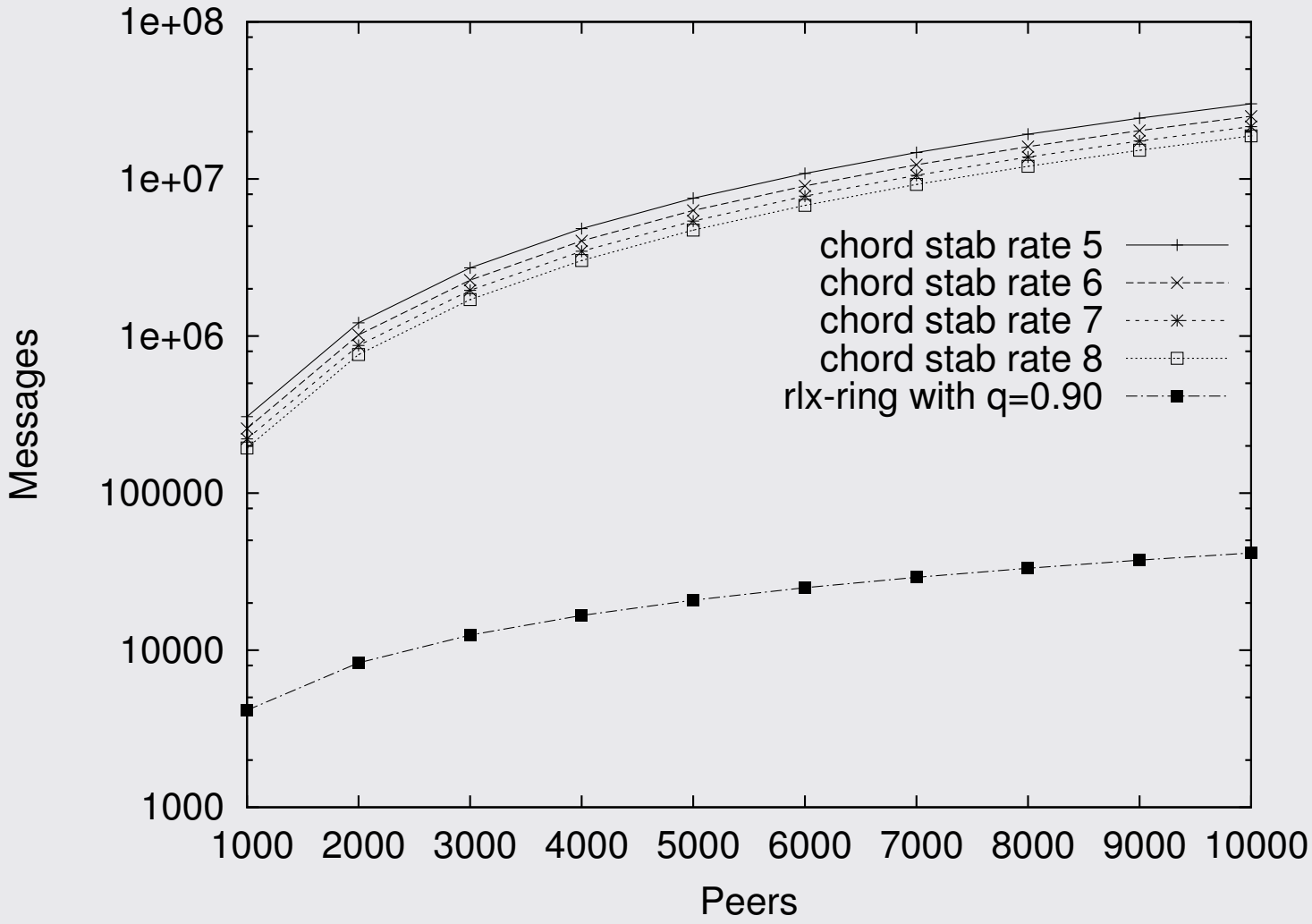




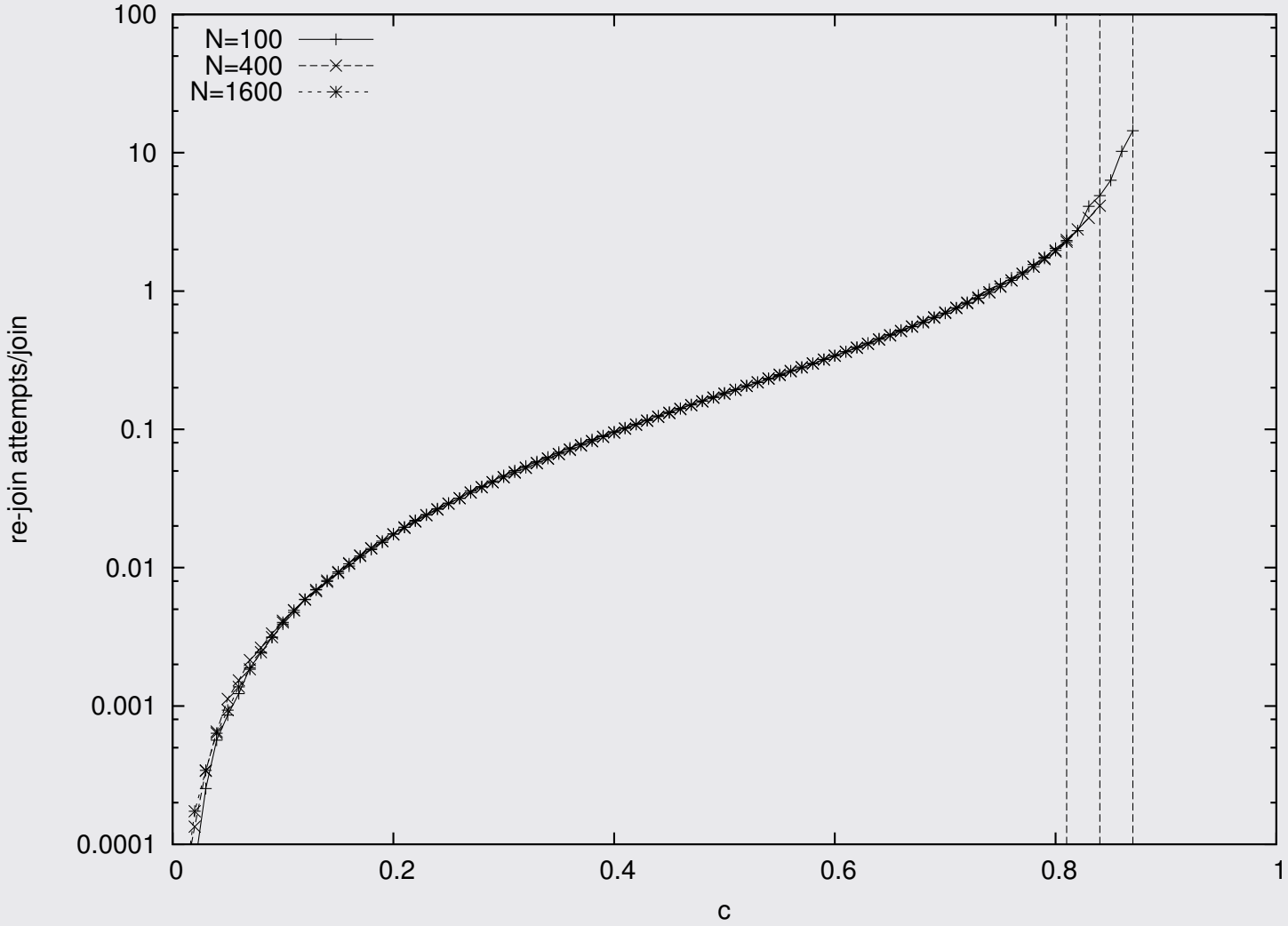
Lookup consistency



Cost efficient ring maintenance



In presence of NATed peers



Relaxed ring

- Scalable and fully decentralized
- Self-organized and fault tolerant (self-healing)

Relaxed ring

- Scalable and fully decentralized
- Self-organized and fault tolerant (self-healing)

- Relies only on point-to-point link
- No transitive connectivity needed:
  if $a \rightarrow b$ and $b \rightarrow c$ does not imply $a \rightarrow c$
- **join/fail** algorithms requires the agreement of only two nodes
  (3 steps with 2 nodes, instead of 1 step with 3 nodes)

Relaxed ring

- Scalable and fully decentralized
- Self-organized and fault tolerant (self-healing)

- Relies only on point-to-point link
- No transitive connectivity needed:
  if $a \rightarrow b$ and $b \rightarrow c$ does not imply $a \rightarrow c$
- **join/fail** algorithms requires the agreement of only two nodes
  (3 steps with 2 nodes, instead of 1 step with 3 nodes)

- No lookup inconsistency introduced by join events
- Cost-efficient ring maintenance (no periodic stabilization)
- Efficient routing $O(\log(N) + b)$

Related work

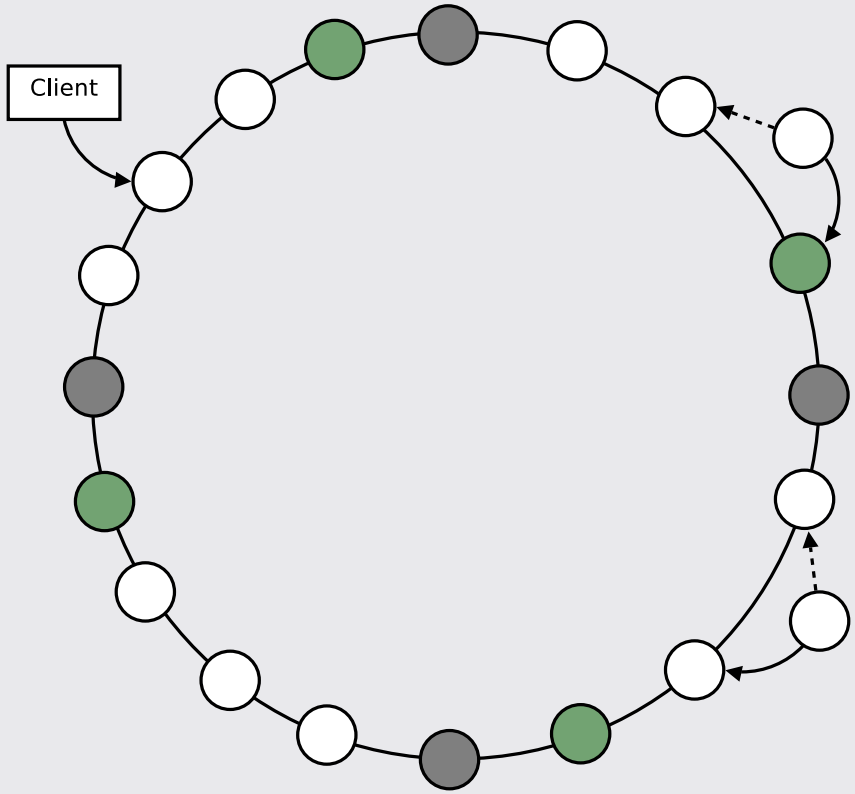
- **Chord**: Too much relaxation. Good idea to just let peers join or leave, but fixing pointers by doing *periodic stabilization* is too costly.
- **DKS atomic join/leave**: Requirements too hard to meet. By locking too nodes it achieves agreement of three nodes. Problems with non-transitive networks, latency and failures.

Trappist

Transactions over **p**eer-to-**p**eer with **is**ola**t**ion

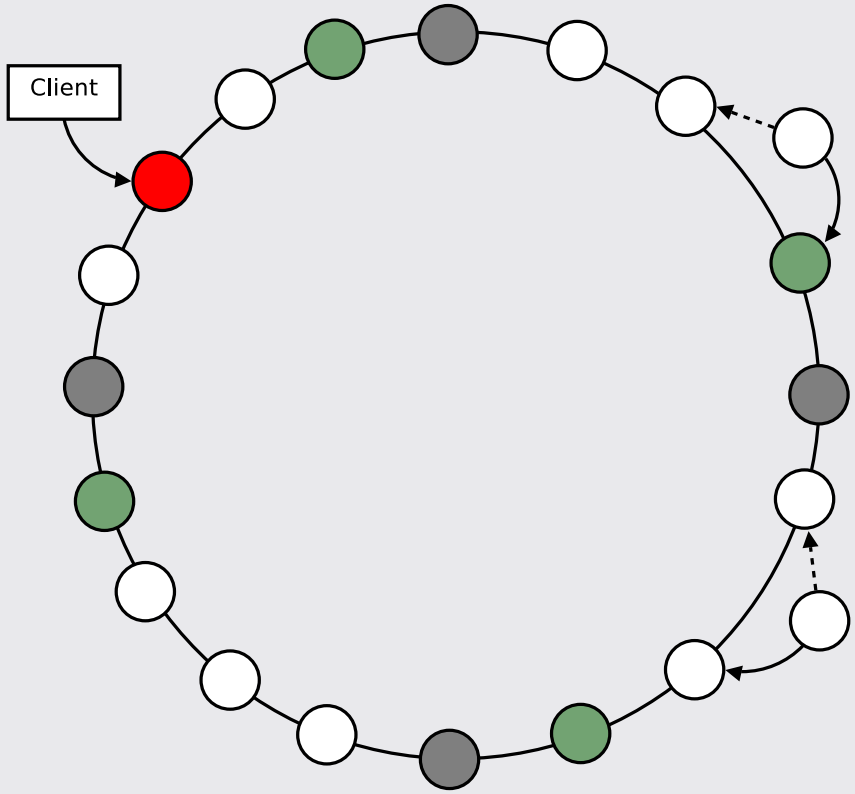
Symmetric replication and transactions

- Every item is symmetrically replicated



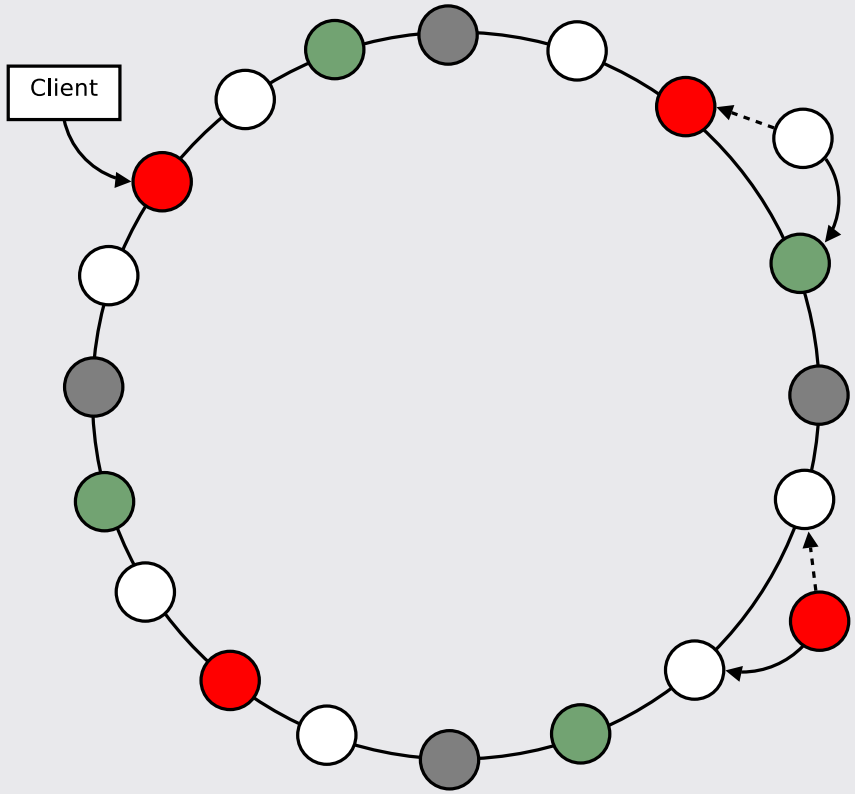
Symmetric replication and transactions

- Every item is symmetrically replicated
- Every transaction creates a transaction manager
- Two-phase commit strongly relies on the transaction manager and needs all replicas



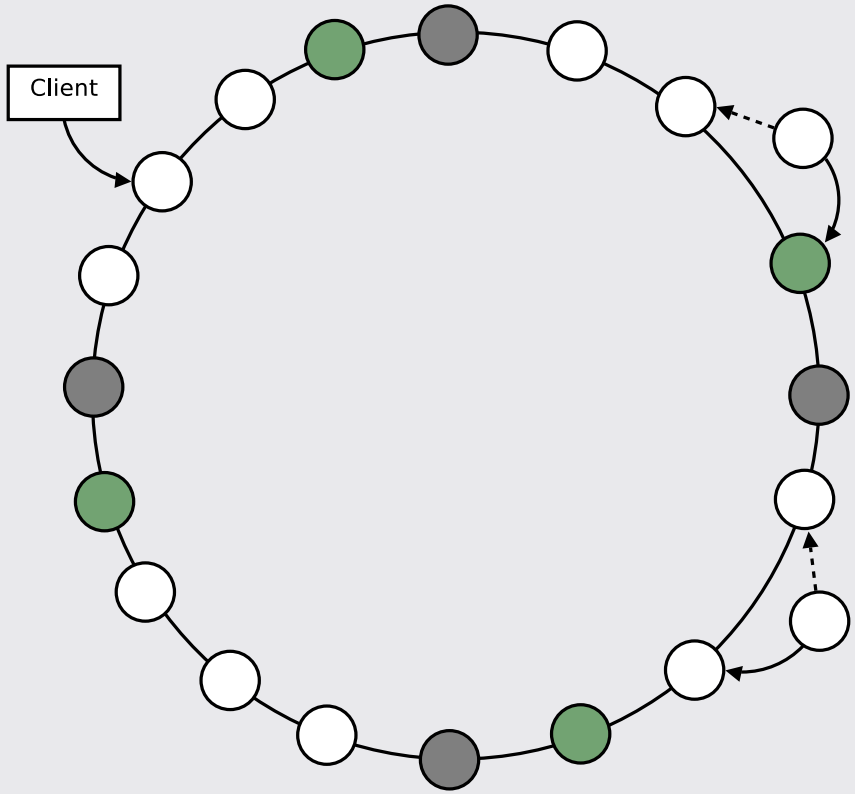
Symmetric replication and transactions

- Every item is symmetrically replicated
- Every transaction creates a transaction manager
- Two-phase commit strongly relies on the transaction manager and needs all replicas
- Paxos consensus algorithm:
  - Replicated transaction manager
  - The majority commit the transaction

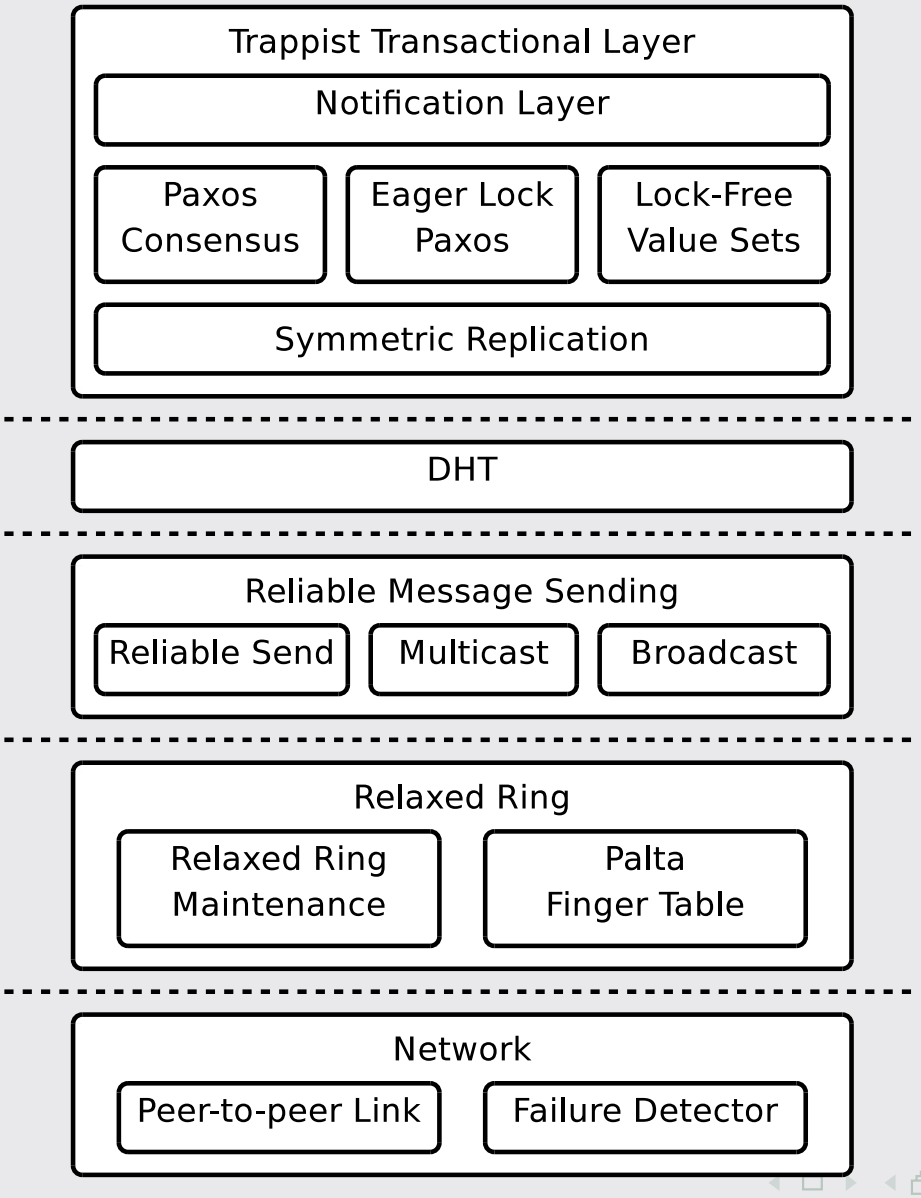


Symmetric replication and transactions

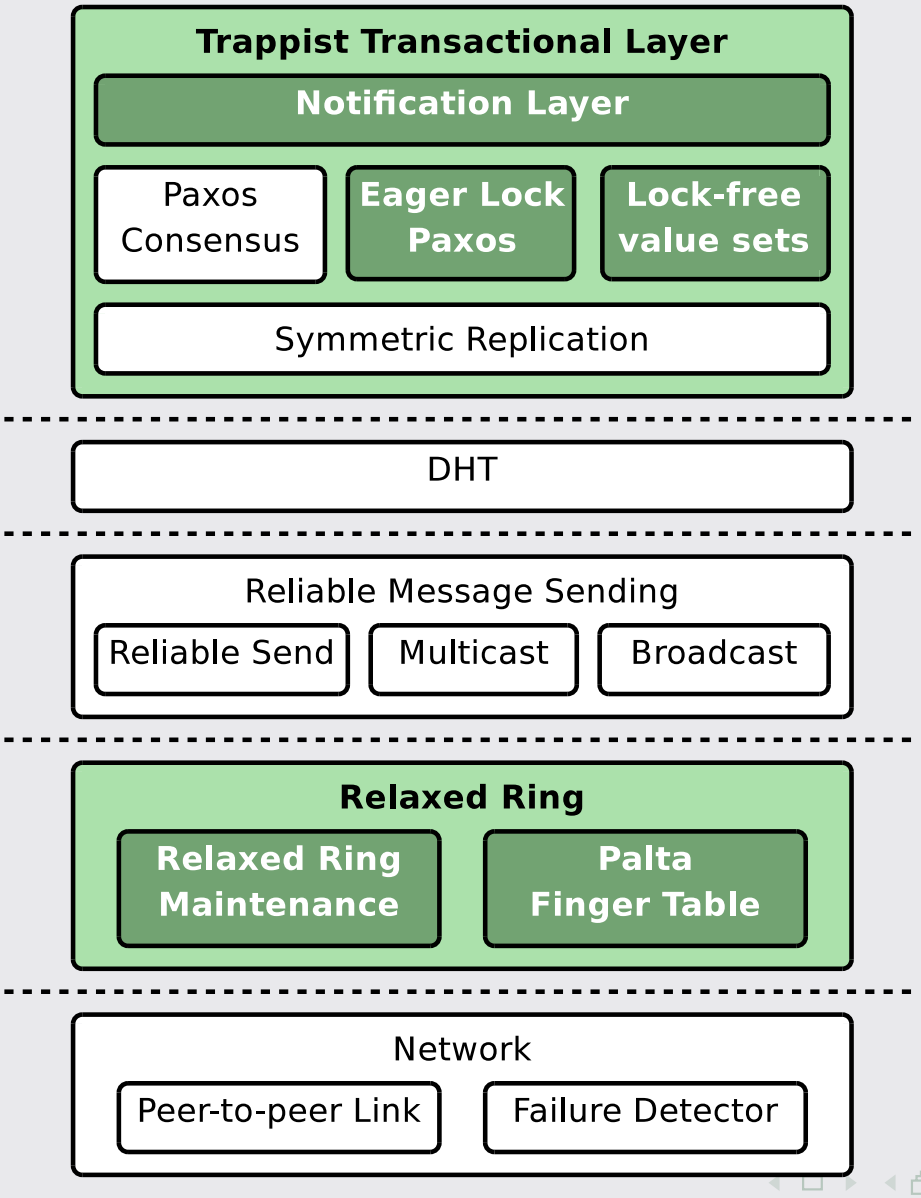
- Every item is symmetrically replicated
- Every transaction creates a transaction manager
- Two-phase commit strongly relies on the transaction manager and needs all replicas
- Paxos consensus algorithm:
  - Replicated transaction manager
  - The majority commit the transaction



Beernet's architecture



Beernet's architecture



Trappist protocols with strong consistency

- Paxos consensus
  - For asynchronous collaboration
  - It completes if the majority survives

Trappist protocols with strong consistency

- Paxos consensus
  - For asynchronous collaboration
  - It completes if the majority survives
- Eager locking
  - For synchronous collaboration
  - Still works with majority but it is more fragile

Trappist protocols with strong consistency

- Paxos consensus
  - For asynchronous collaboration
  - It completes if the majority survives
- Eager locking
  - For synchronous collaboration
  - Still works with majority but it is more fragile
- Lock-free key/value-sets
  - More fault-tolerant
  - Better performance
  - No versioning and no total order (no queues, no stacks)

Trappist's operations

- `write(foo, bar)`
  - Writes an item on the majority of the replicas where value *bar* is associated with key *foo*
- `read(foo)`
  - Returns the latest value stored with key *foo*

Trappist's operations

- `write(foo, bar)`
  - Writes an item on the majority of the replicas where value *bar* is associated with key *foo*
- `read(foo)`
  - Returns the latest value stored with key *foo*
- `add(key, value)`
  - Adding a new element stores it in the majority of replicas, and eventually in all of them
  - Adding an already added element has no effect

Trappist's operations

- `write(foo, bar)`
  - Writes an item on the majority of the replicas where value *bar* is associated with key *foo*
- `read(foo)`
  - Returns the latest value stored with key *foo*
- `add(key, value)`
  - Adding a new element stores it in the majority of replicas, and eventually in all of them
  - Adding an already added element has no effect
- `remove(key, value)`
  - Removing an element will removes it from the majority of replicas, and eventually from all of them
  - Removing a non-existing element has no effect

Trappist's operations

- `write(foo, bar)`
  - Writes an item on the majority of the replicas where value *bar* is associated with key *foo*
- `read(foo)`
  - Returns the latest value stored with key *foo*
- `add(key, value)`
  - Adding a new element stores it in the majority of replicas, and eventually in all of them
  - Adding an already added element has no effect
- `remove(key, value)`
  - Removing an element will removes it from the majority of replicas, and eventually from all of them
  - Removing a non-existing element has no effect
- `readSet(key)`
  - Reads the value-set associated to *key*

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{p, r}	$\rightarrow$ {a, b}		
t_2	readSet(k)	{q, r}		$\rightarrow$ {a, b}	

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{p, r}	$\rightarrow$ {a, b}		
t_2	readSet(k)	{q, r}	$\rightarrow$ {a, b}		
t_3	add(k, c)	{p, q, r}	{a, b, c}	{a, c}	{b, c}

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{p, r}	$\rightarrow$ {a, b}		
t_2	readSet(k)	{q, r}	$\rightarrow$ {a, b}		
t_3	add(k, c)	{p, q, r}	{a, b, c}	{a, c}	{b, c}
t_4	remove(k, c)	{p, q}	{a, b}	{a}	{b, c}

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{p, r}	$\rightarrow$ {a, b}		
t_2	readSet(k)	{q, r}	$\rightarrow$ {a, b}		
t_3	add(k, c)	{p, q, r}	{a, b, c}	{a, c}	{b, c}
t_4	remove(k, c)	{p, q}	{a, b}	{a}	{b, c}
t_5	readSet(k)	{q, r}	$\rightarrow$ {a, b} or {a, b, c}?		

Deducing the set

Time	Operation	Majority	p	q	r
t_0	add(k, a)	{p, q}	{a}	{a}	ϕ
t_1	add(k, b)	{p, r}	{a, b}	{a}	{b}
t_2	readSet(k)	{p, r}	$\rightarrow$ {a, b}		
t_2	readSet(k)	{q, r}	$\rightarrow$ {a, b}		
t_3	add(k, c)	{p, q, r}	{a, b, c}	{a, c}	{b, c}
t_4	remove(k, c)	{p, q}	{a, b}	{a}	{b, c}
t_5	readSet(k)	{q, r}	$\rightarrow$ {a, b} or {a, b, c}?		
t_5	readSet(k)	{p, q, r}	$\rightarrow$ {a, b} or {a, b, c}?		

Identifying every operation

t	Operation	Majority	p	q	r
t_0	i: add(k, a)	{p, q}	(i)	(i)	ϕ

Identifying every operation

t	Operation	Majority	p	q	r
t_0	*i*: add(k, a)	{p, q}	(*i*)	(*i*)	ϕ
t_1	*ii*: add(k, c)	{p, q, r}	(*i*, *ii*)	(*i*, *ii*)	(*ii*)

Identifying every operation

t	Operation	Majority	p	q	r
t_0	i: add(k, a)	{p, q}	(i)	(i)	ϕ
t_1	ii: add(k, c)	{p, q, r}	(i, ii)	(i, ii)	(ii)
t_2	ii': remove(k, c)	{p, q}	(i, ii, ii')	(i, ii, ii')	(ii)

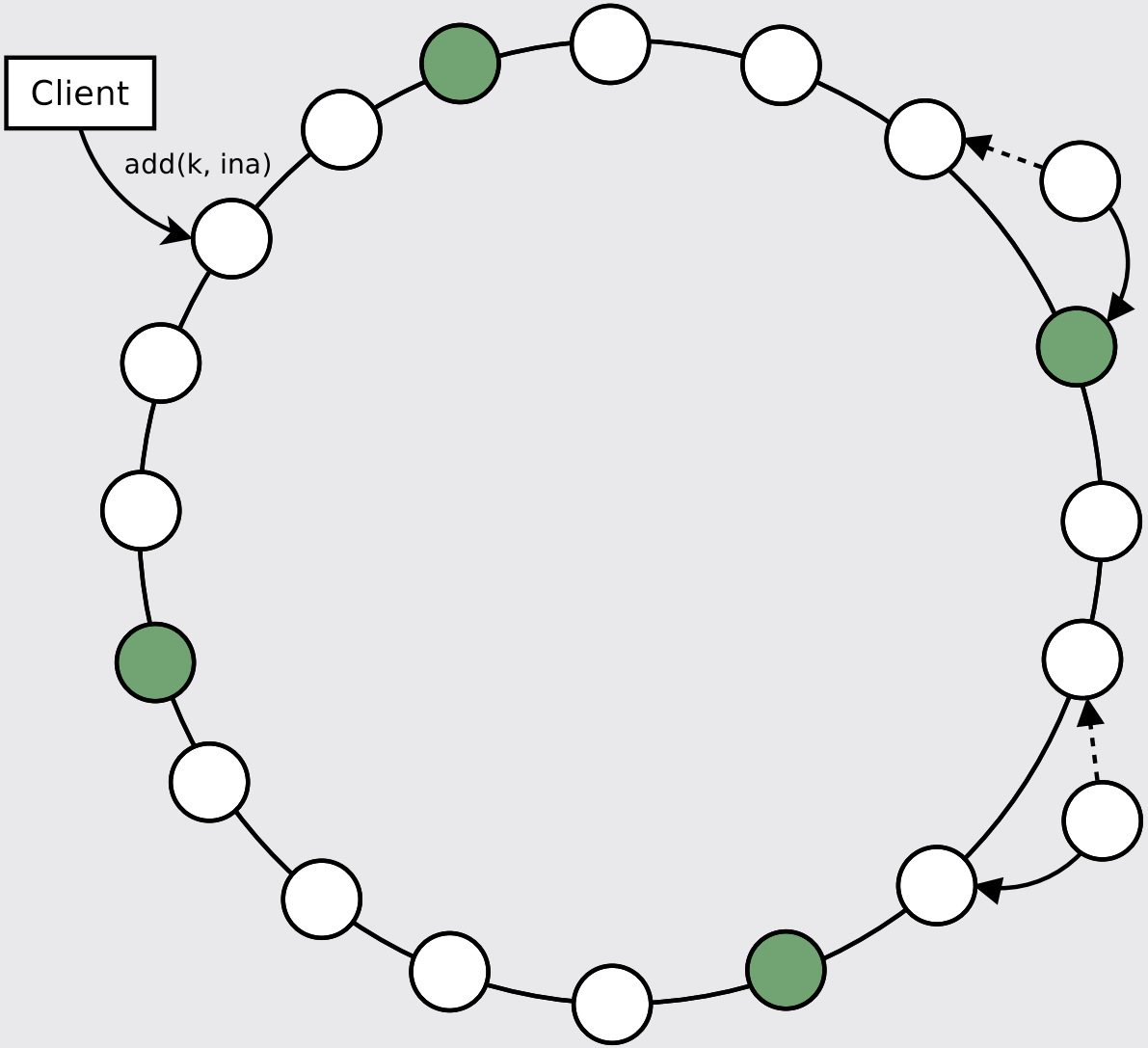
Identifying every operation

t	Operation	Majority	p	q	r
t_0	i: add(k, a)	{p, q}	(i)	(i)	ϕ
t_1	ii: add(k, c)	{p, q, r}	(i, ii)	(i, ii)	(ii)
t_2	ii': remove(k, c)	{p, q}	(i, ii, ii')	(i, ii, ii')	(ii)
t_3	readSet(k)	{q, r}	$\rightarrow$ {a}		

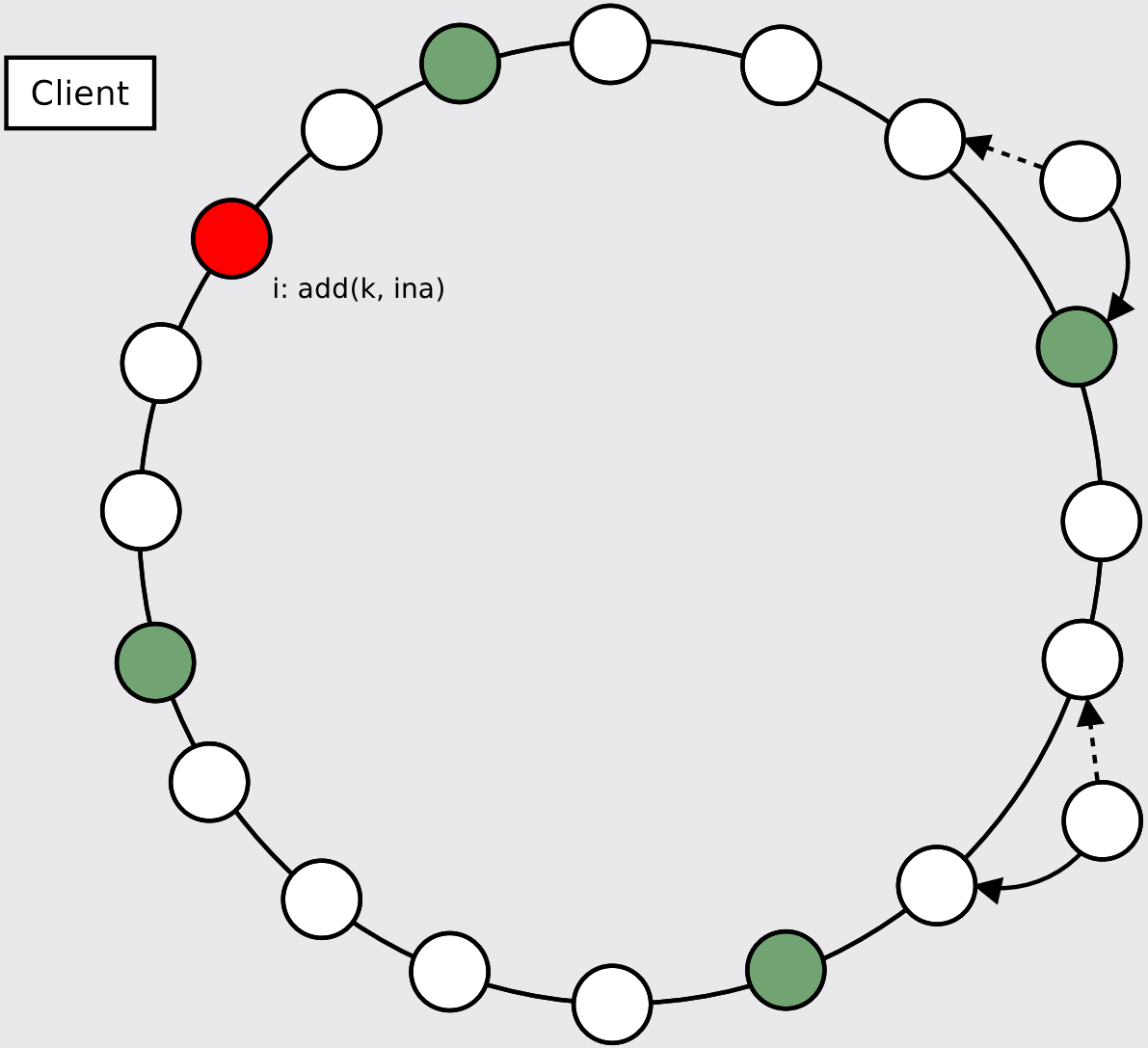
Identifying every operation

t	Operation	Majority	p	q	r
t_0	i: add(k, a)	{p, q}	(i)	(i)	ϕ
t_1	ii: add(k, c)	{p, q, r}	(i, ii)	(i, ii)	(ii)
t_2	ii': remove(k, c)	{p, q}	(i, ii, ii')	(i, ii, ii')	(ii)
t_3	readSet(k)	{q, r}		$\rightarrow$ {a}	
t_3	readSet(k)	{p, q, r}		$\rightarrow$ {a}	

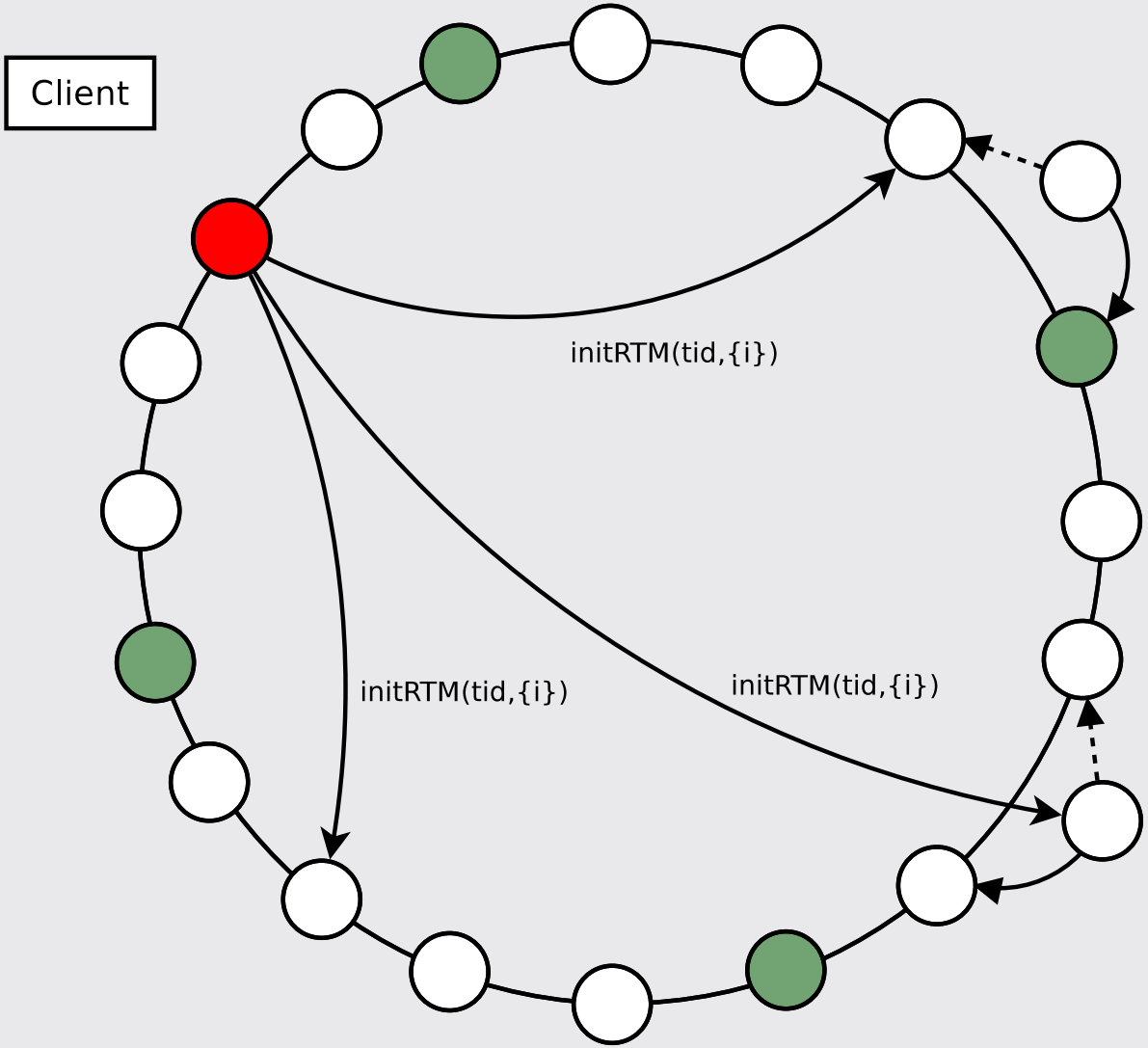
Lock-free key/value-sets



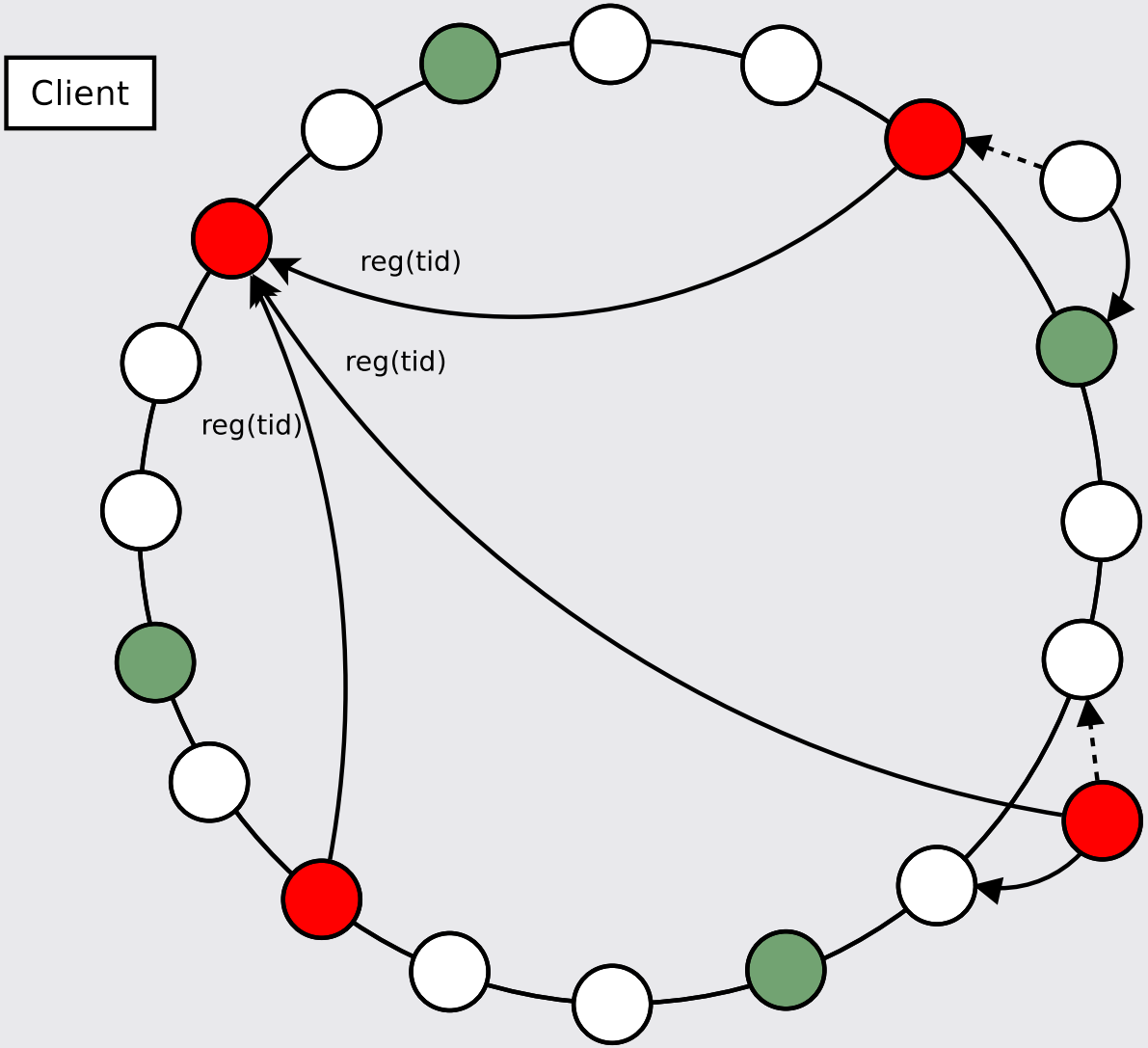
Lock-free key/value-sets



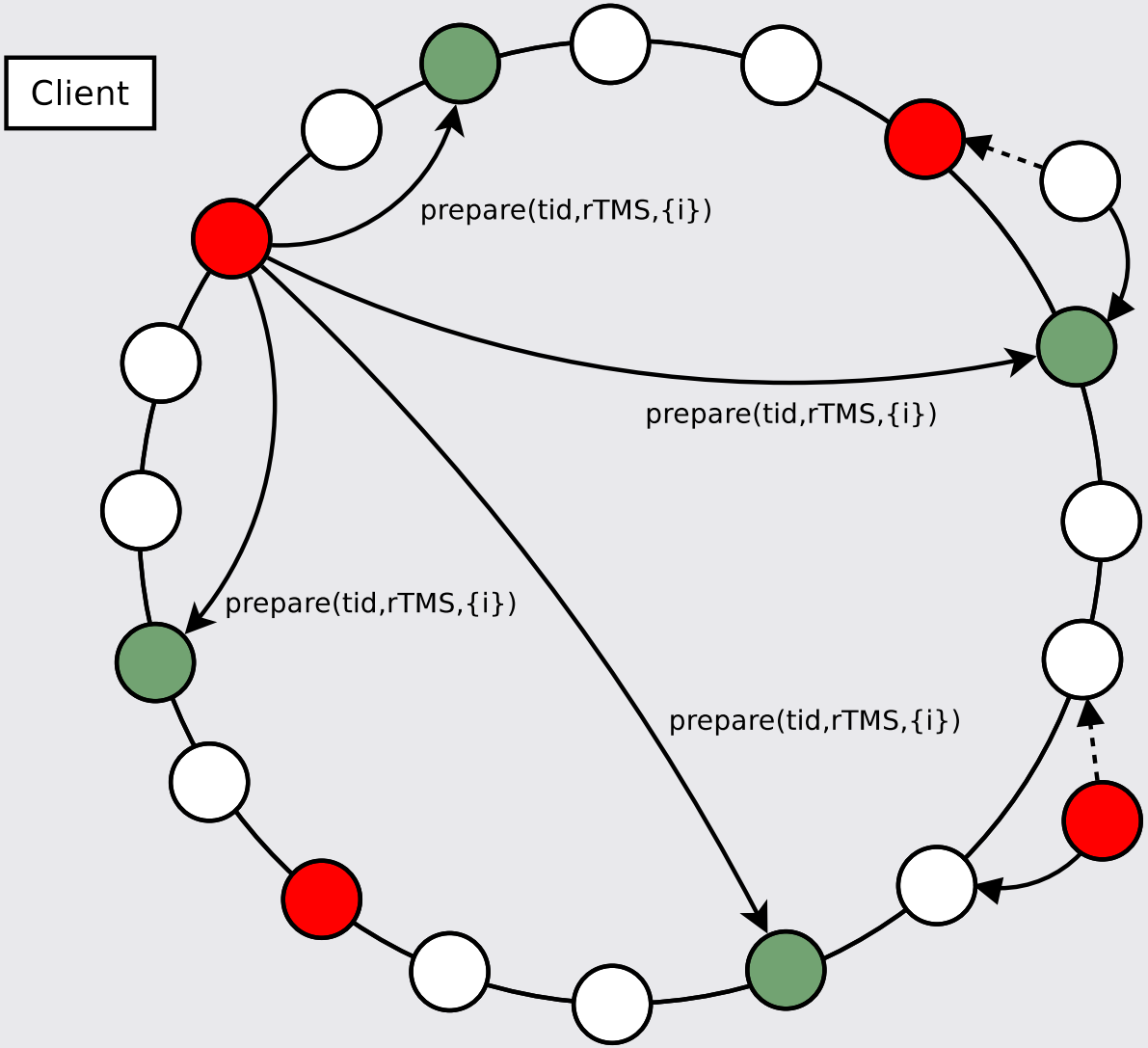
Lock-free key/value-sets



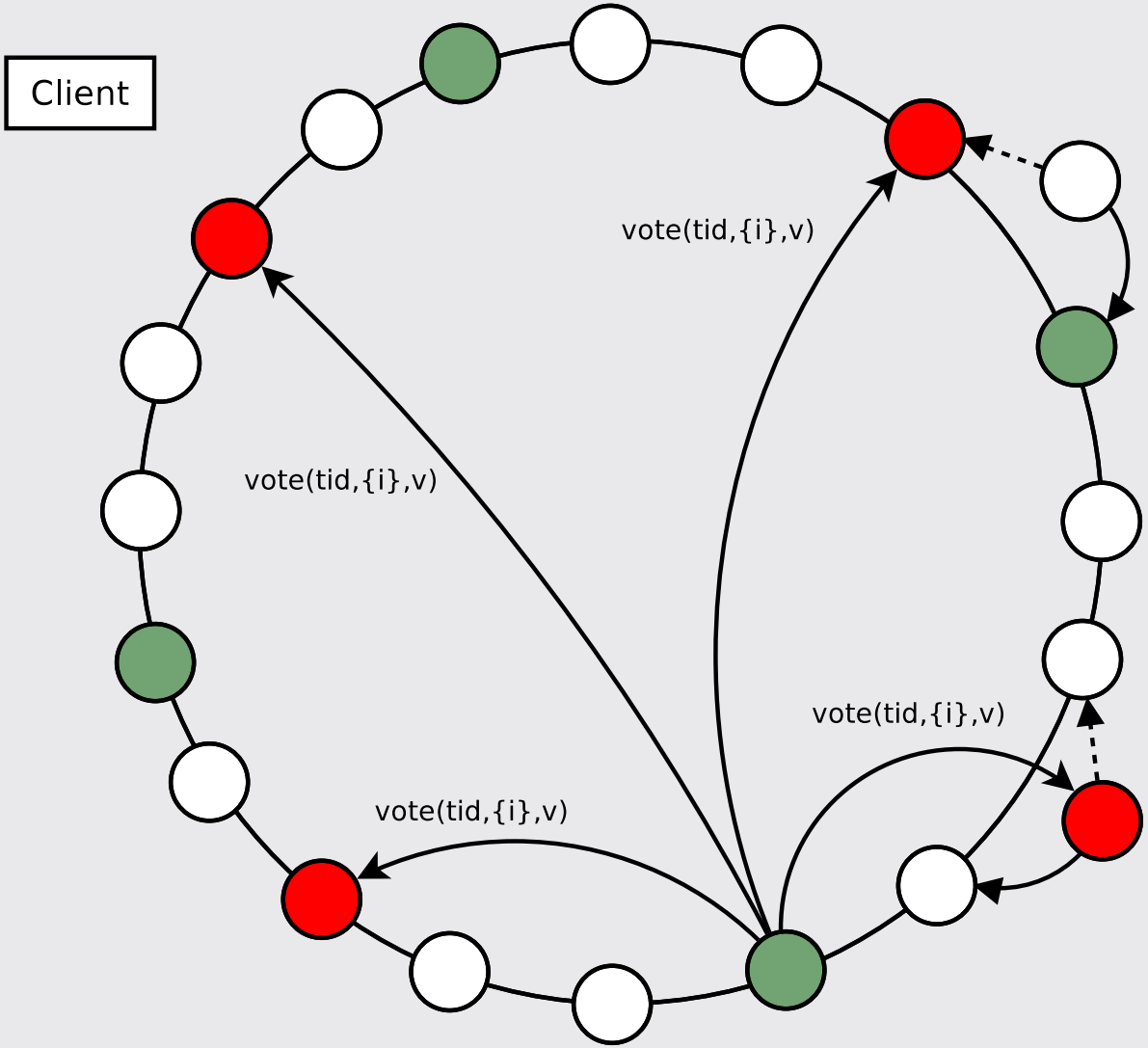
Lock-free key/value-sets



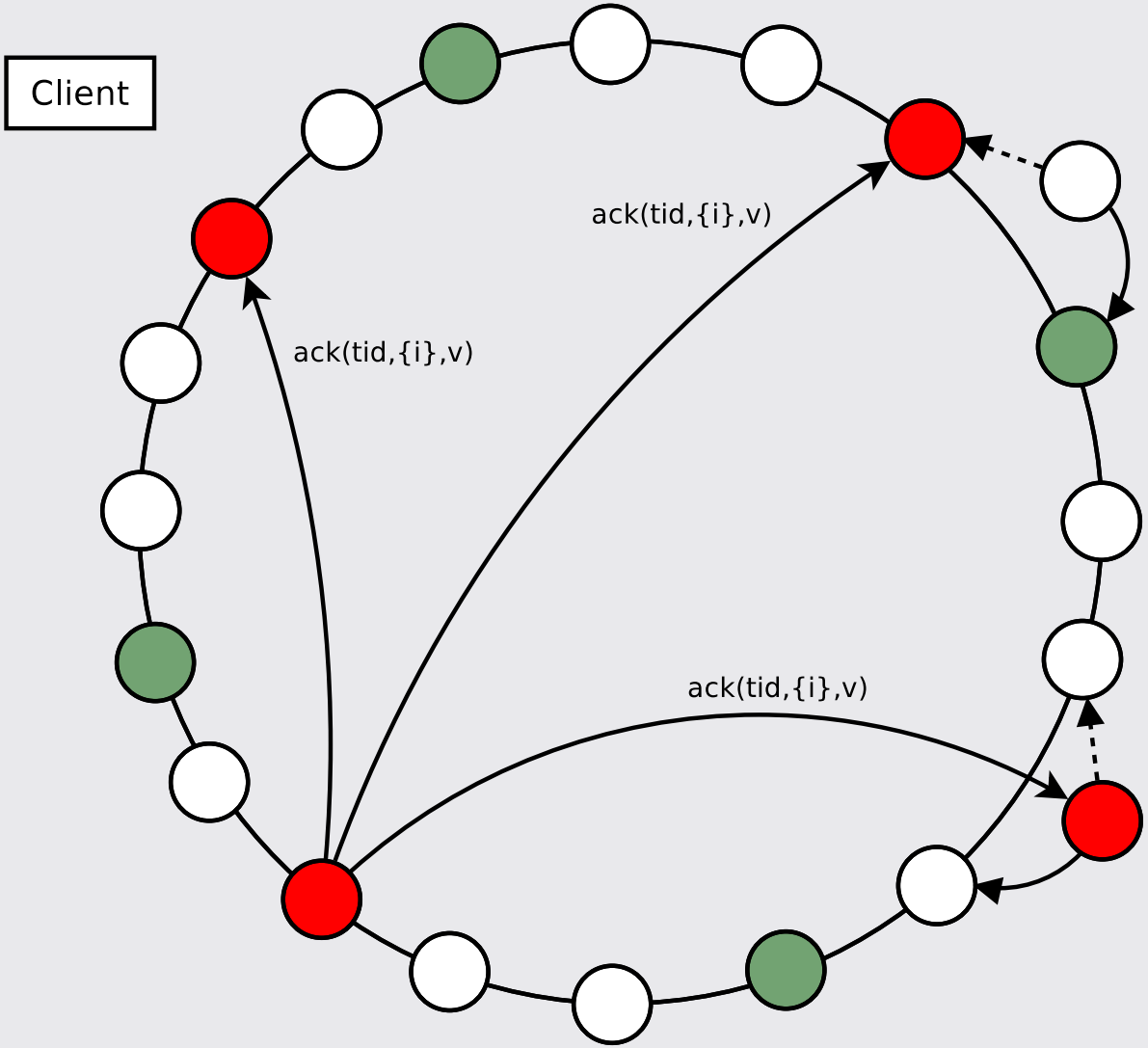
Lock-free key/value-sets



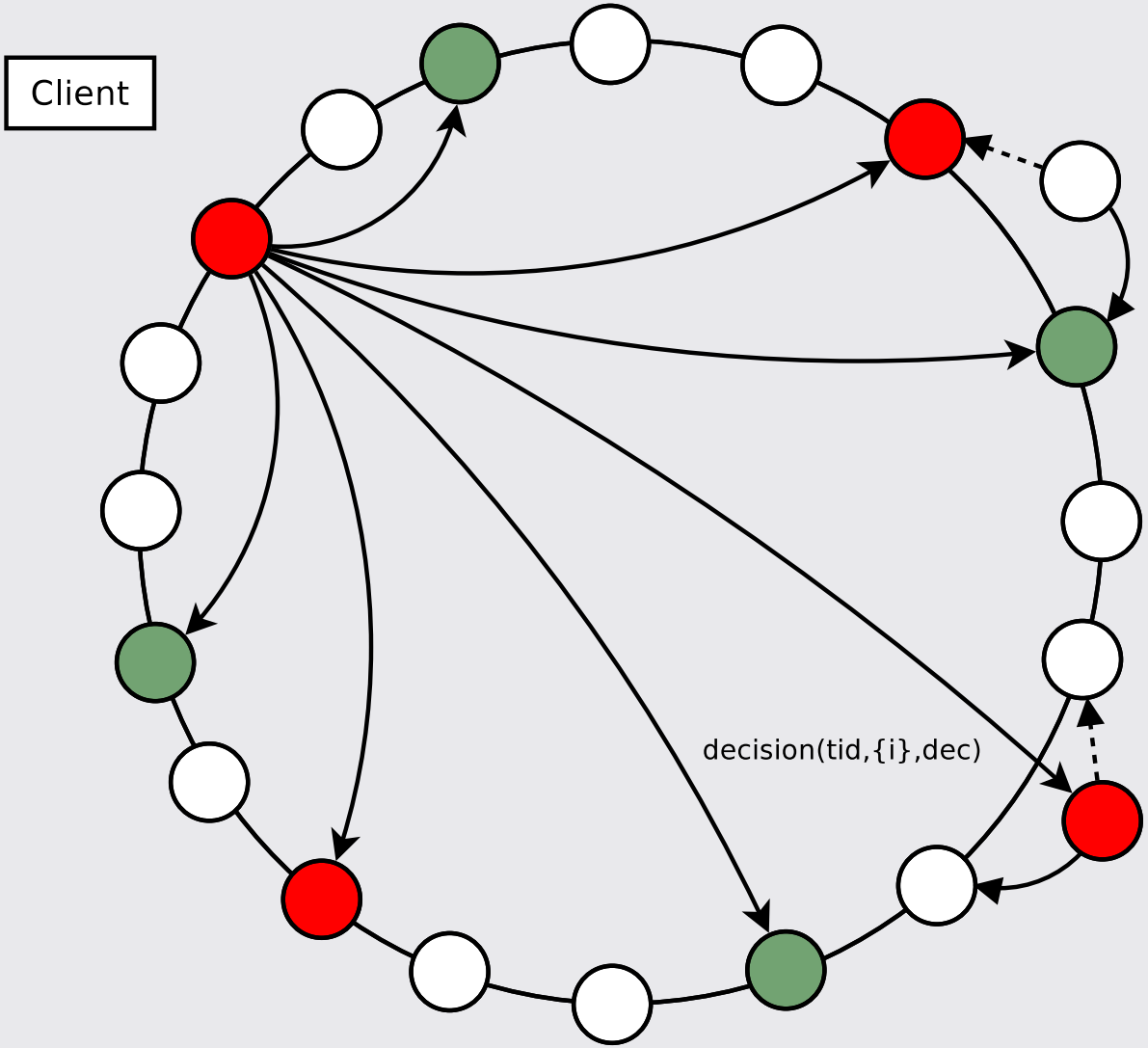
Lock-free key/value-sets



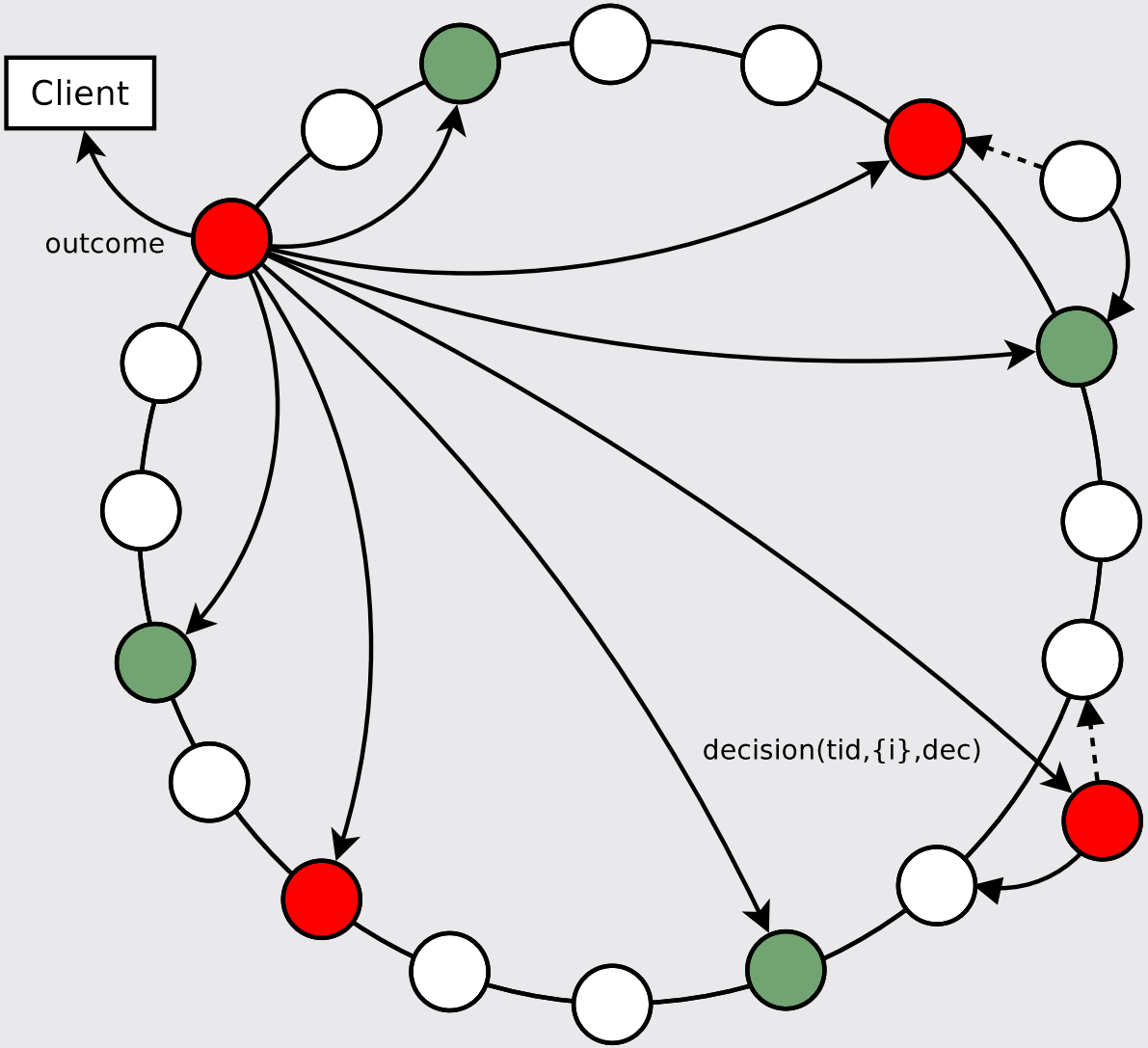
Lock-free key/value-sets



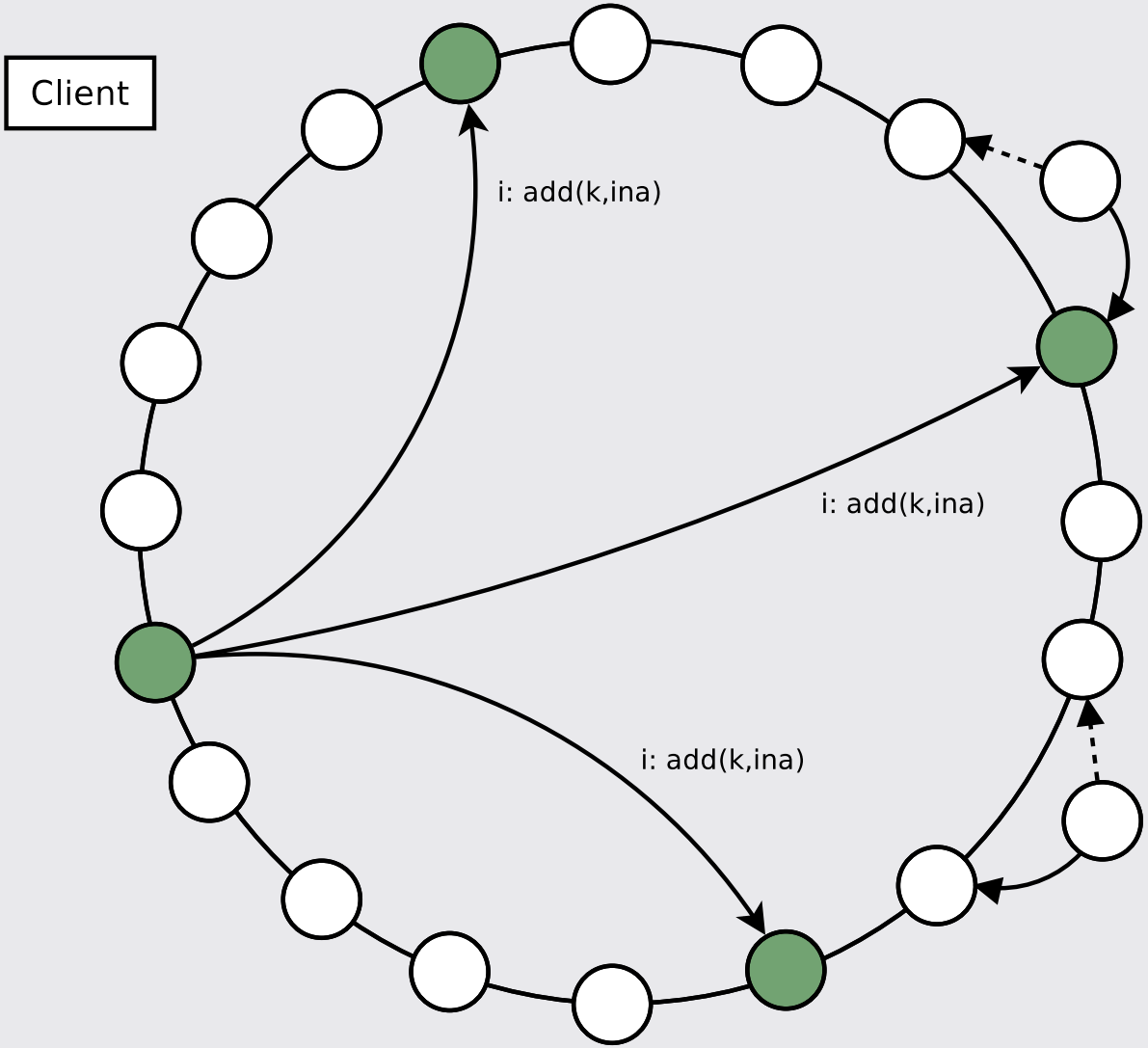
Lock-free key/value-sets



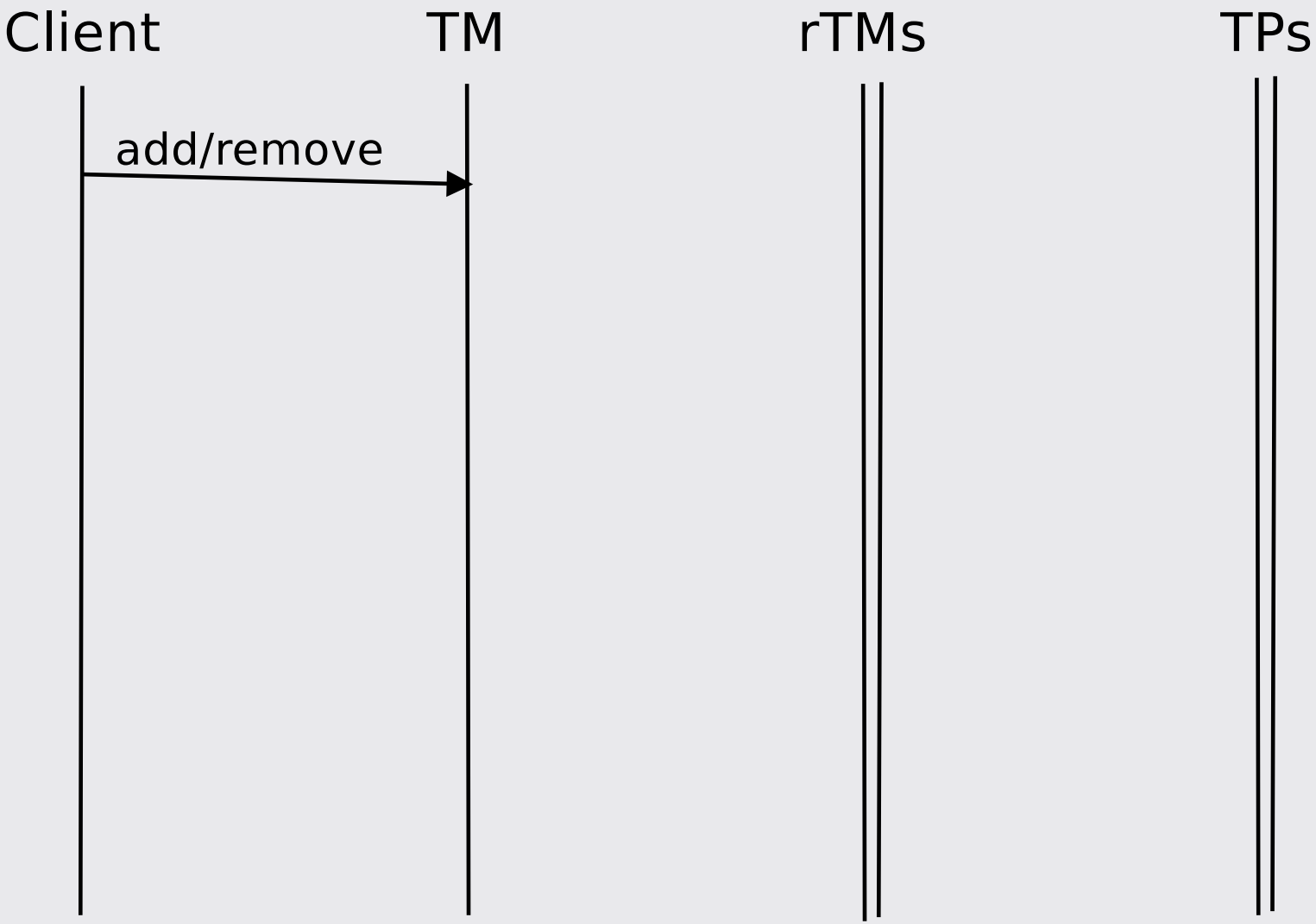
Lock-free key/value-sets



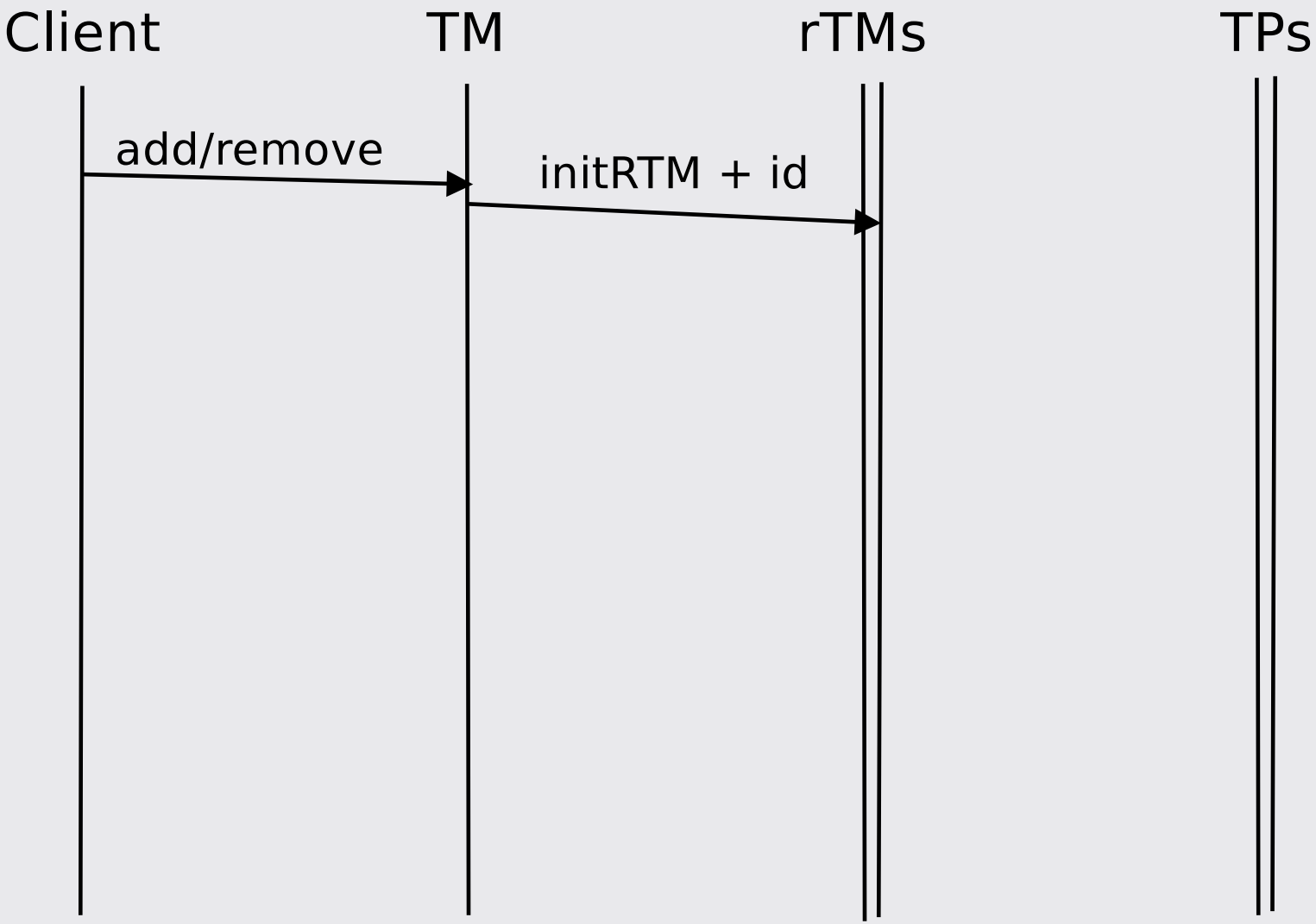
Lock-free key/value-sets



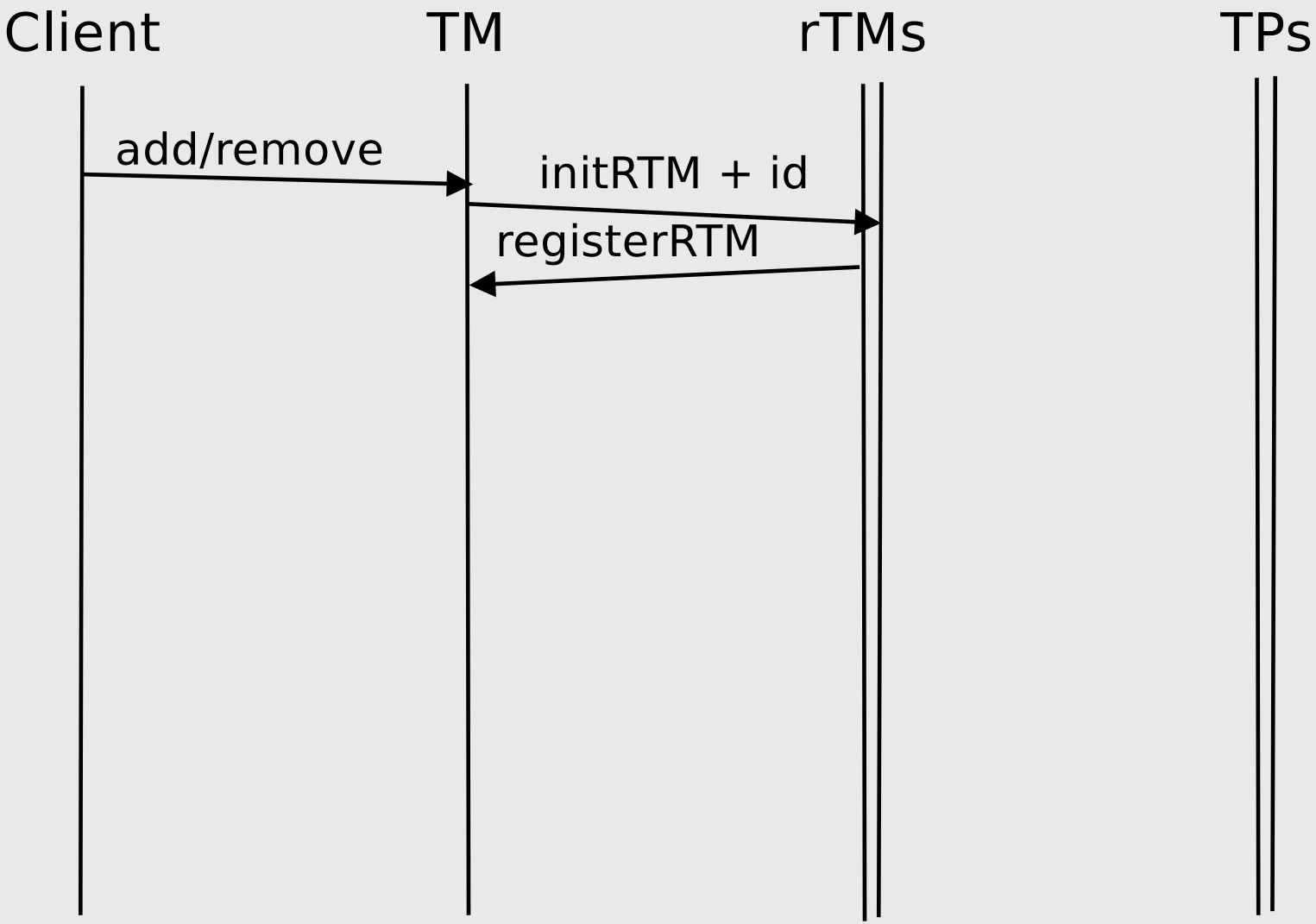
Lock-free key/value-sets



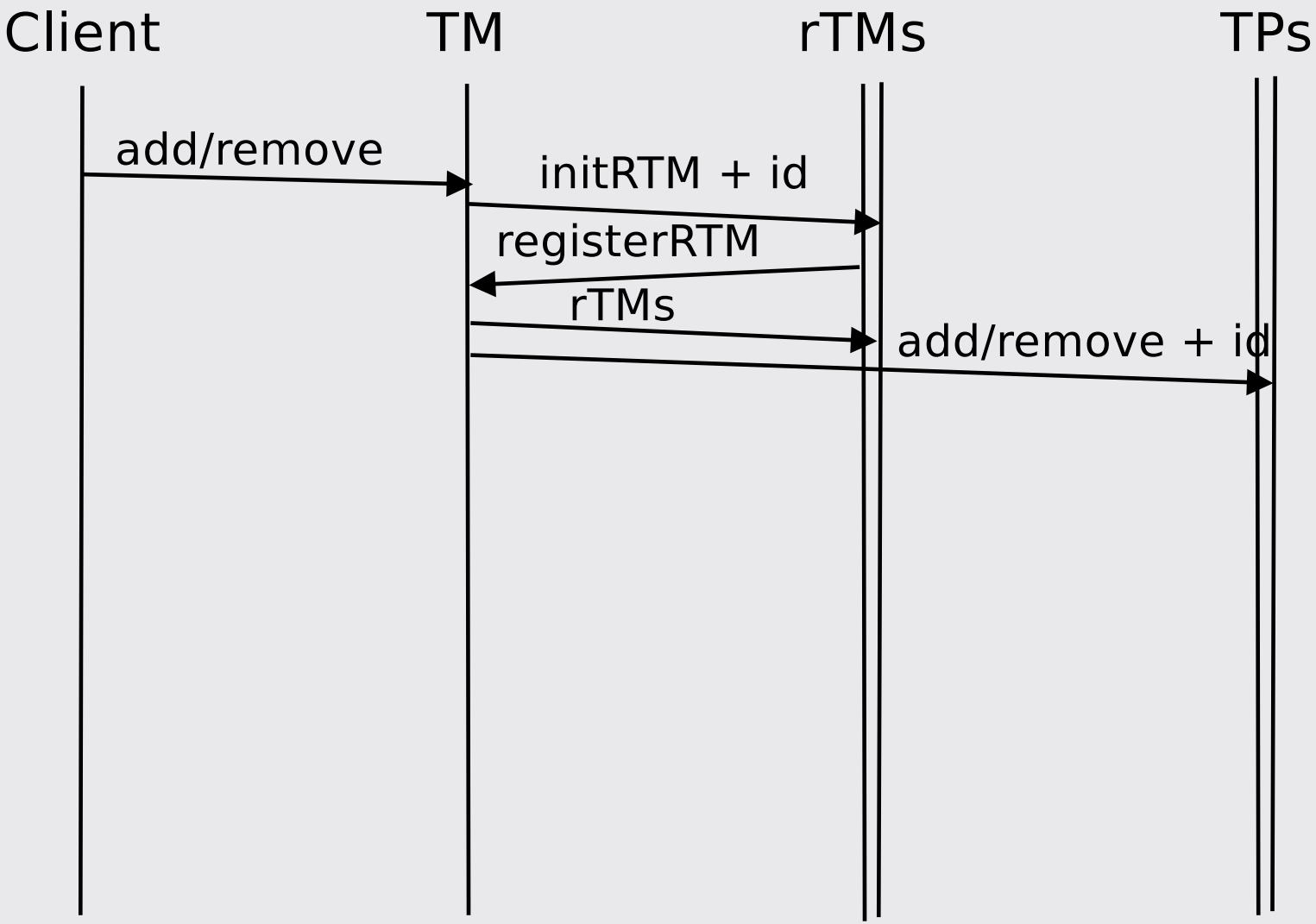
Lock-free key/value-sets



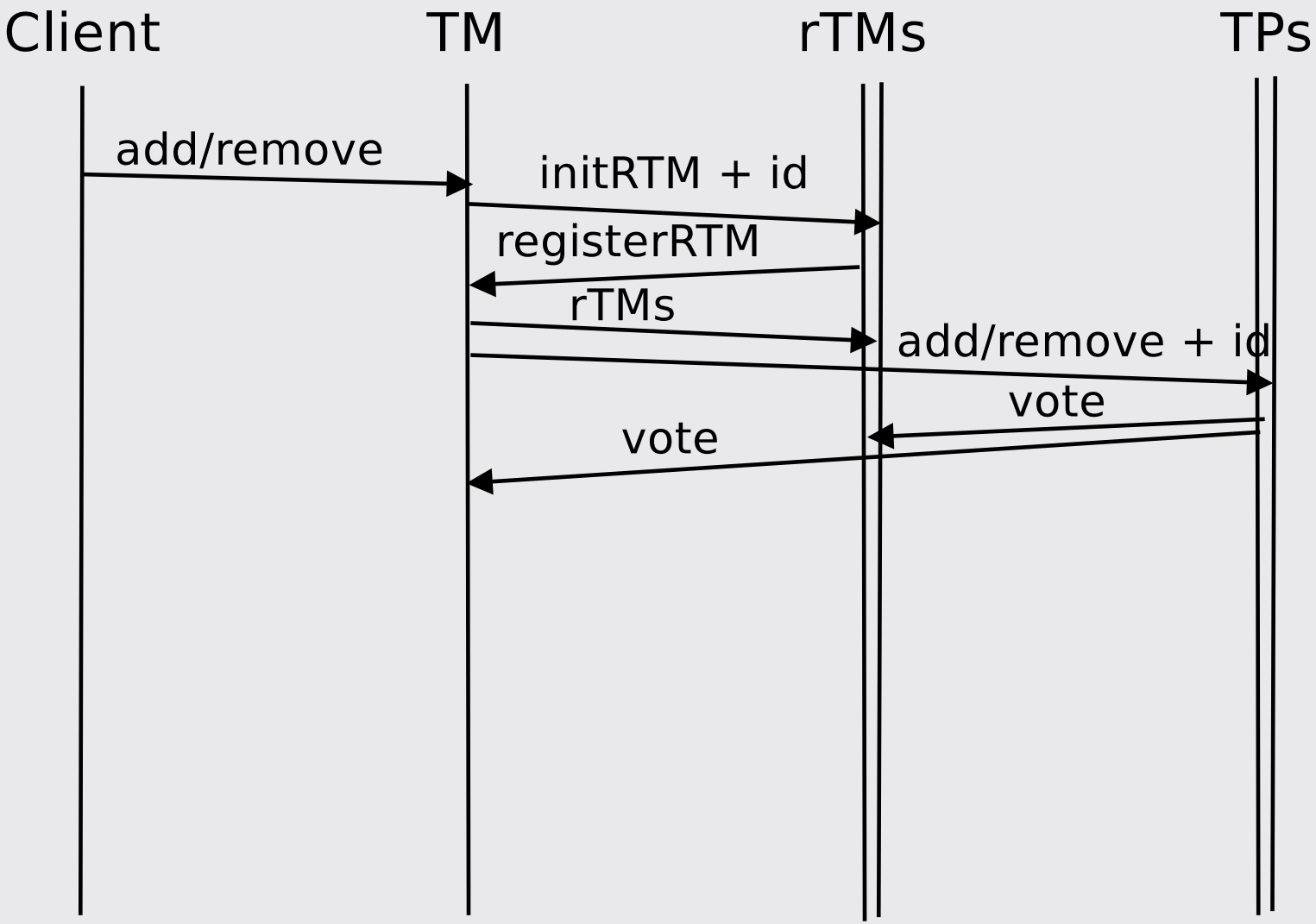
Lock-free key/value-sets



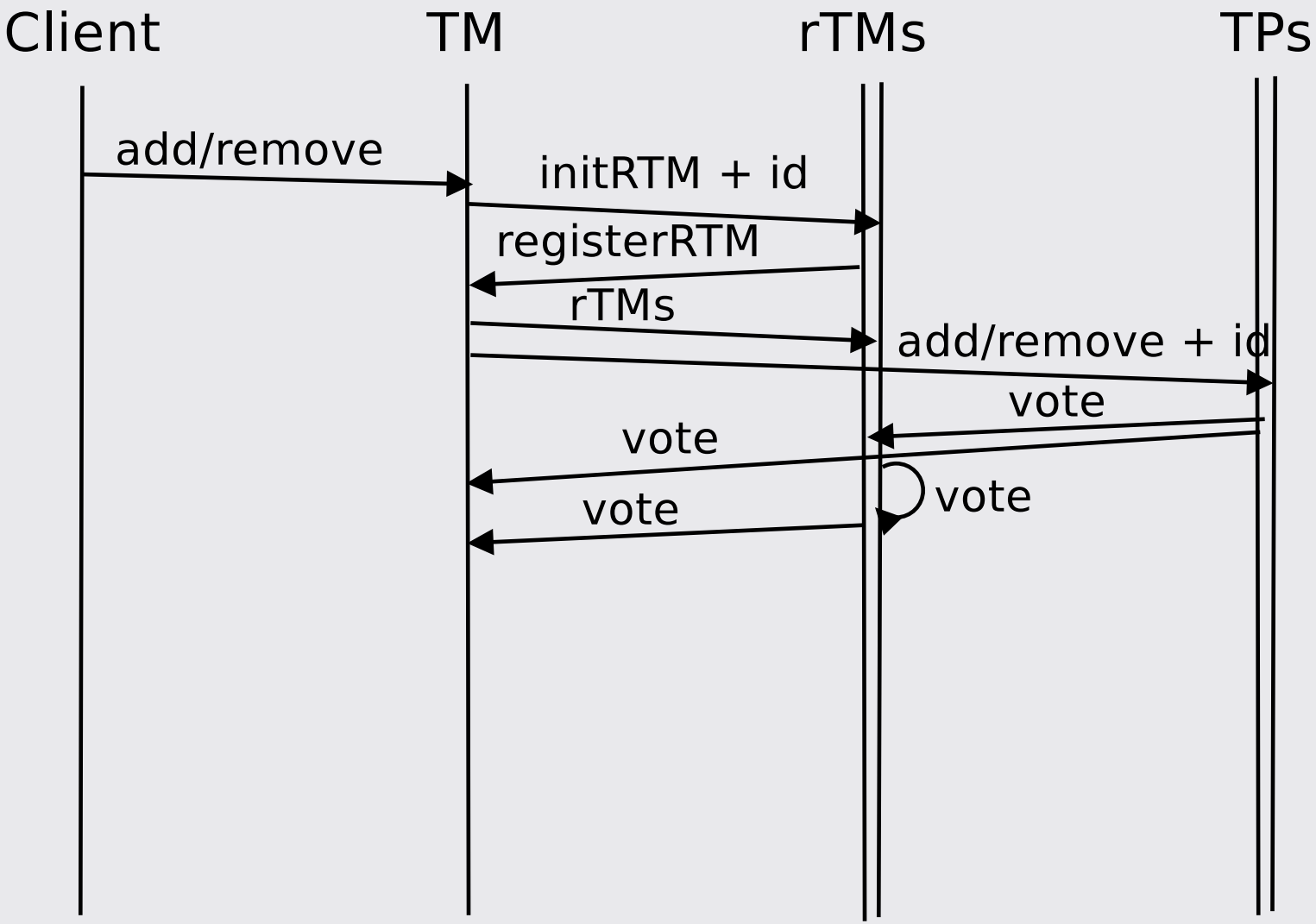
Lock-free key/value-sets



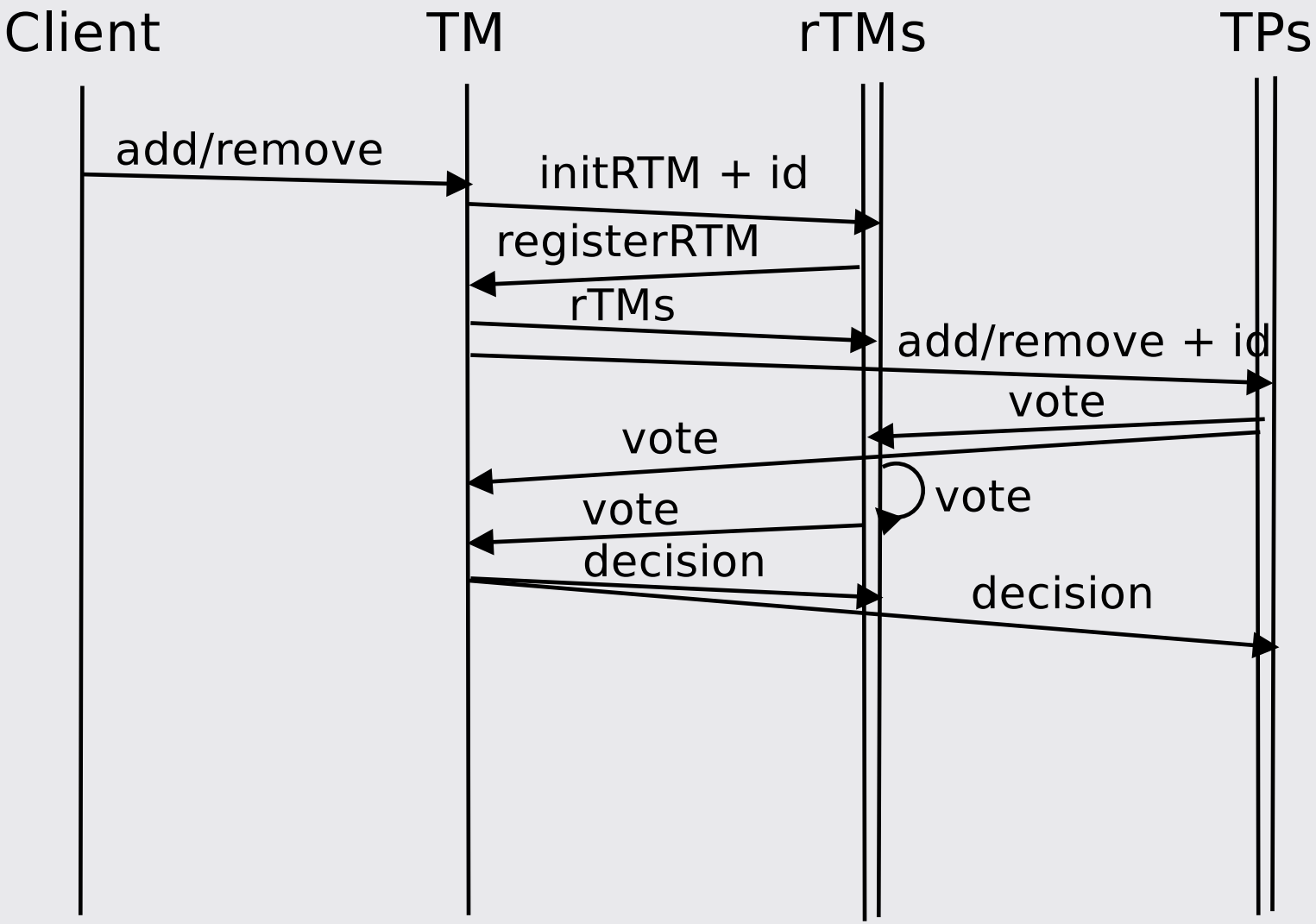
Lock-free key/value-sets



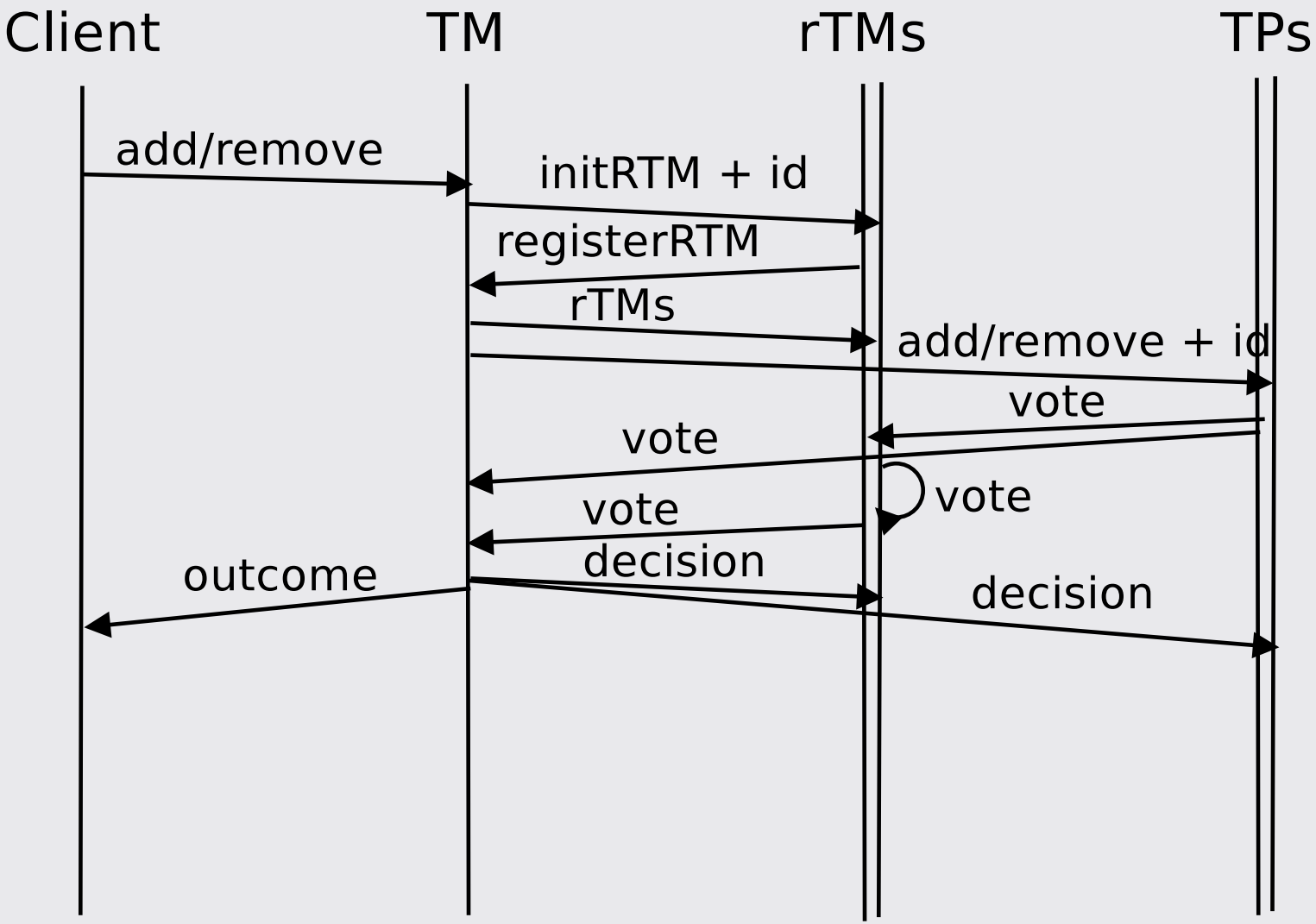
Lock-free key/value-sets



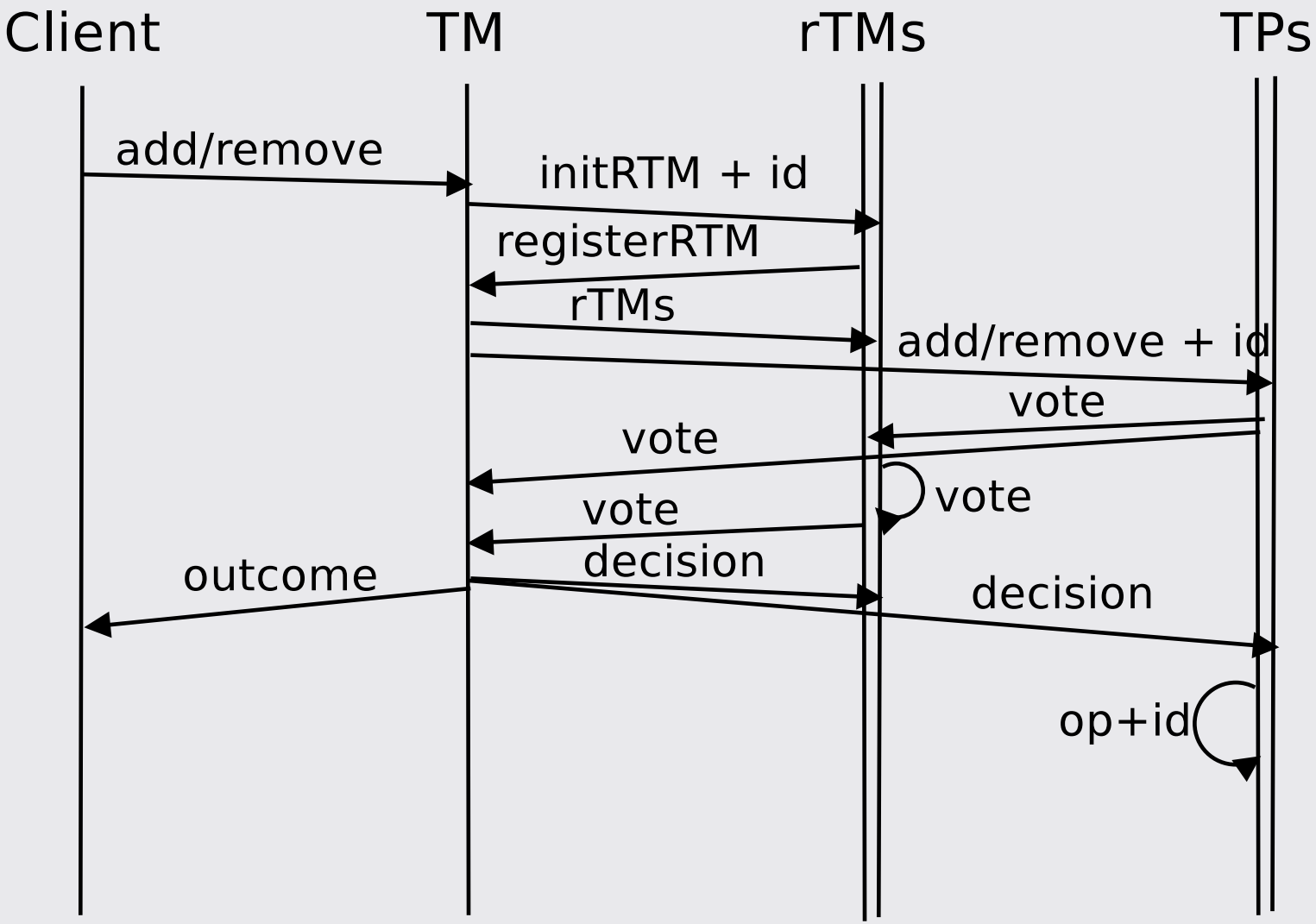
Lock-free key/value-sets



Lock-free key/value-sets



Lock-free key/value-sets



Taking the decision

vote	decision	outcome
ok	commit	commit

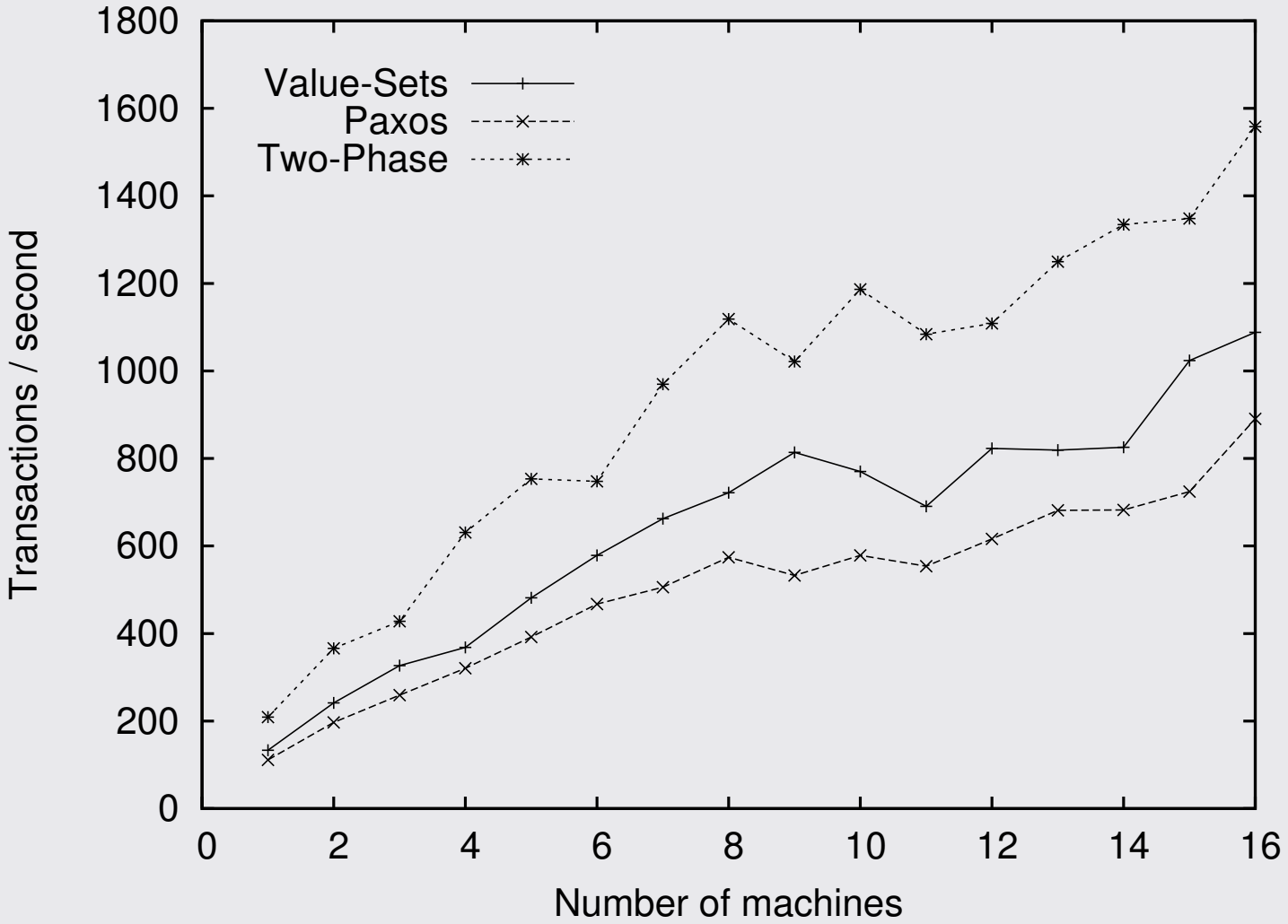
Taking the decision

vote	decision	outcome
ok	commit	commit
duplicated	discard	commit
not_found	discard	commit

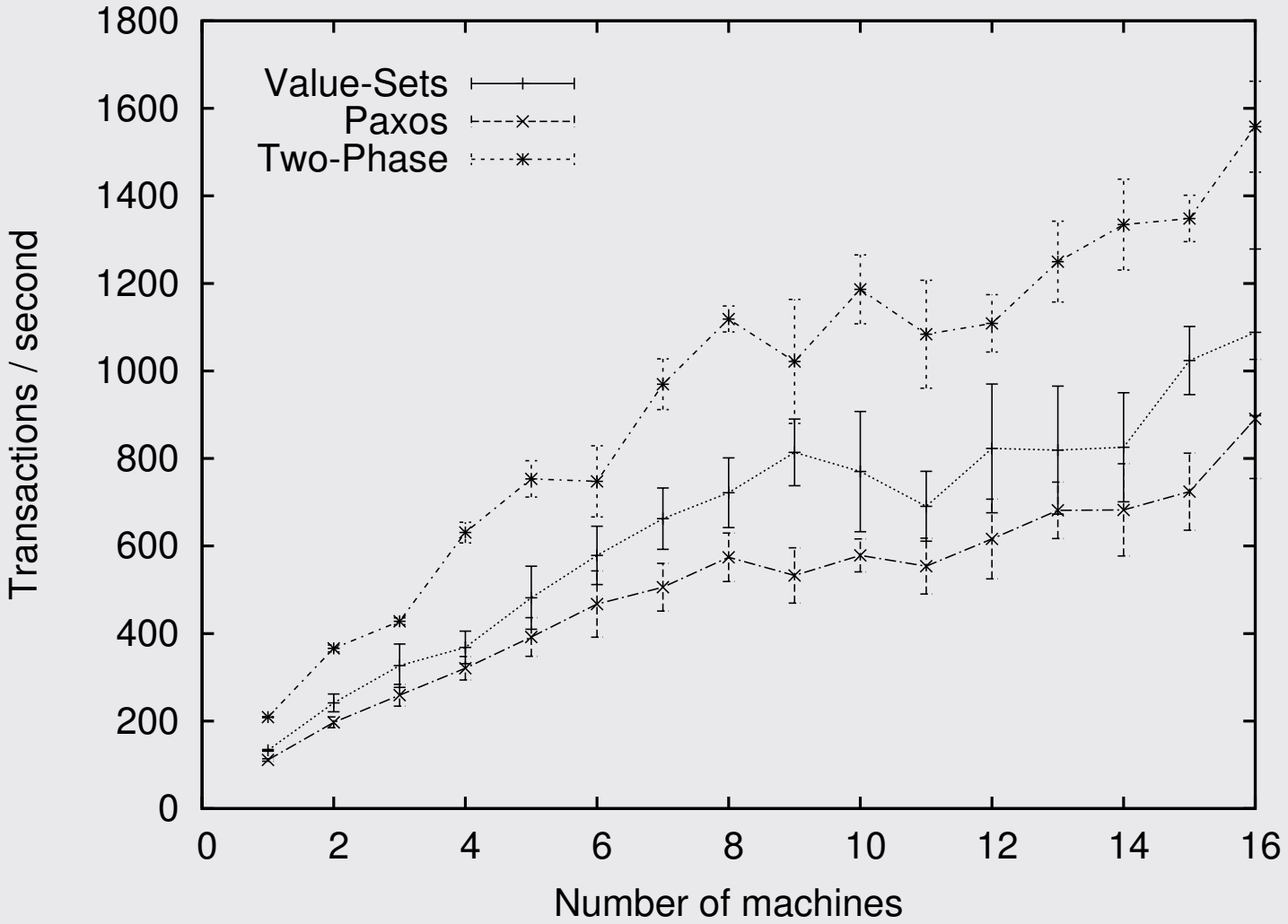
Taking the decision

vote	decision	outcome
ok	commit	commit
duplicated	discard	commit
not_found	discard	commit
concurrent	*retry*	*no outcome*
no majority	abort	abort

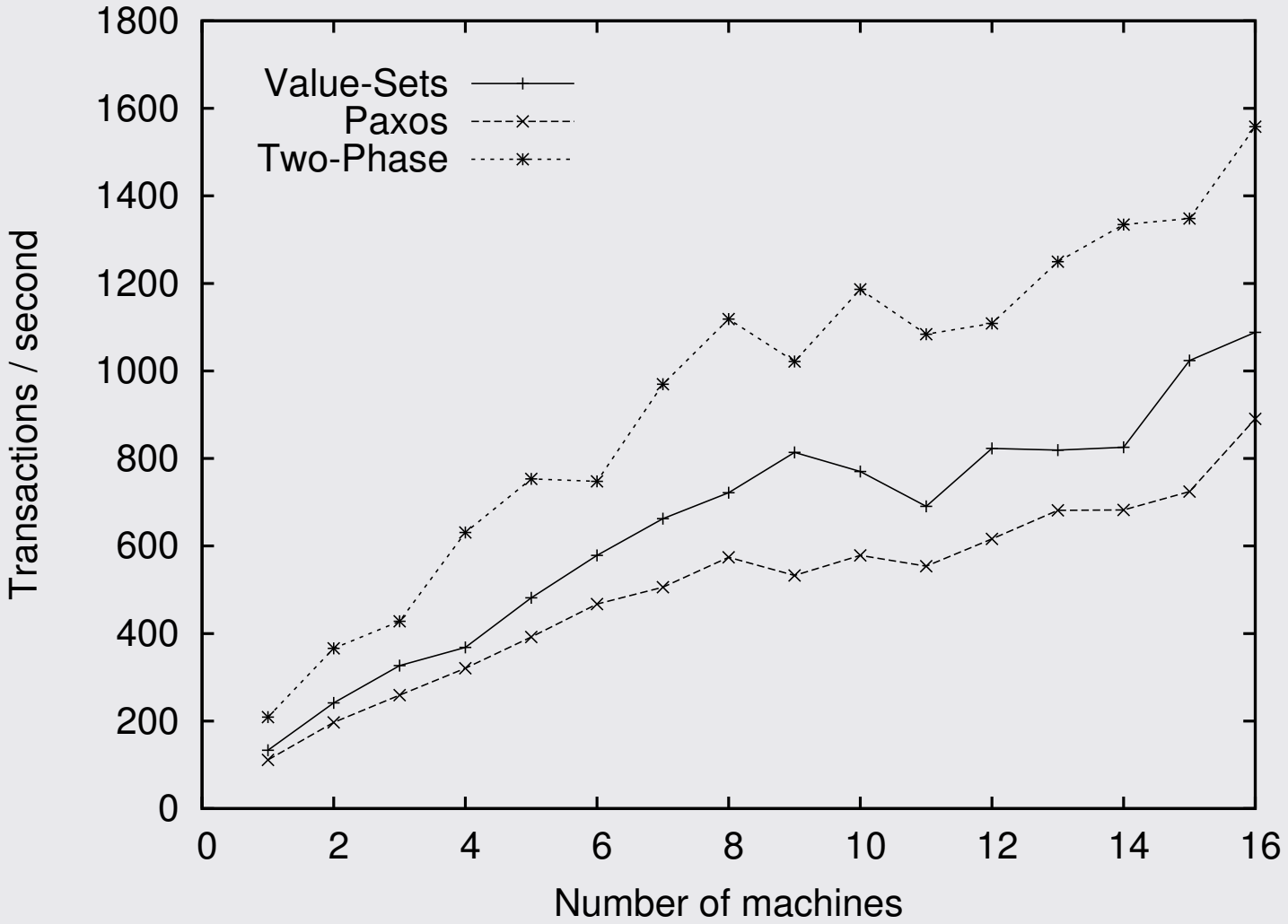
Trappist's Evaluation - Average Performance



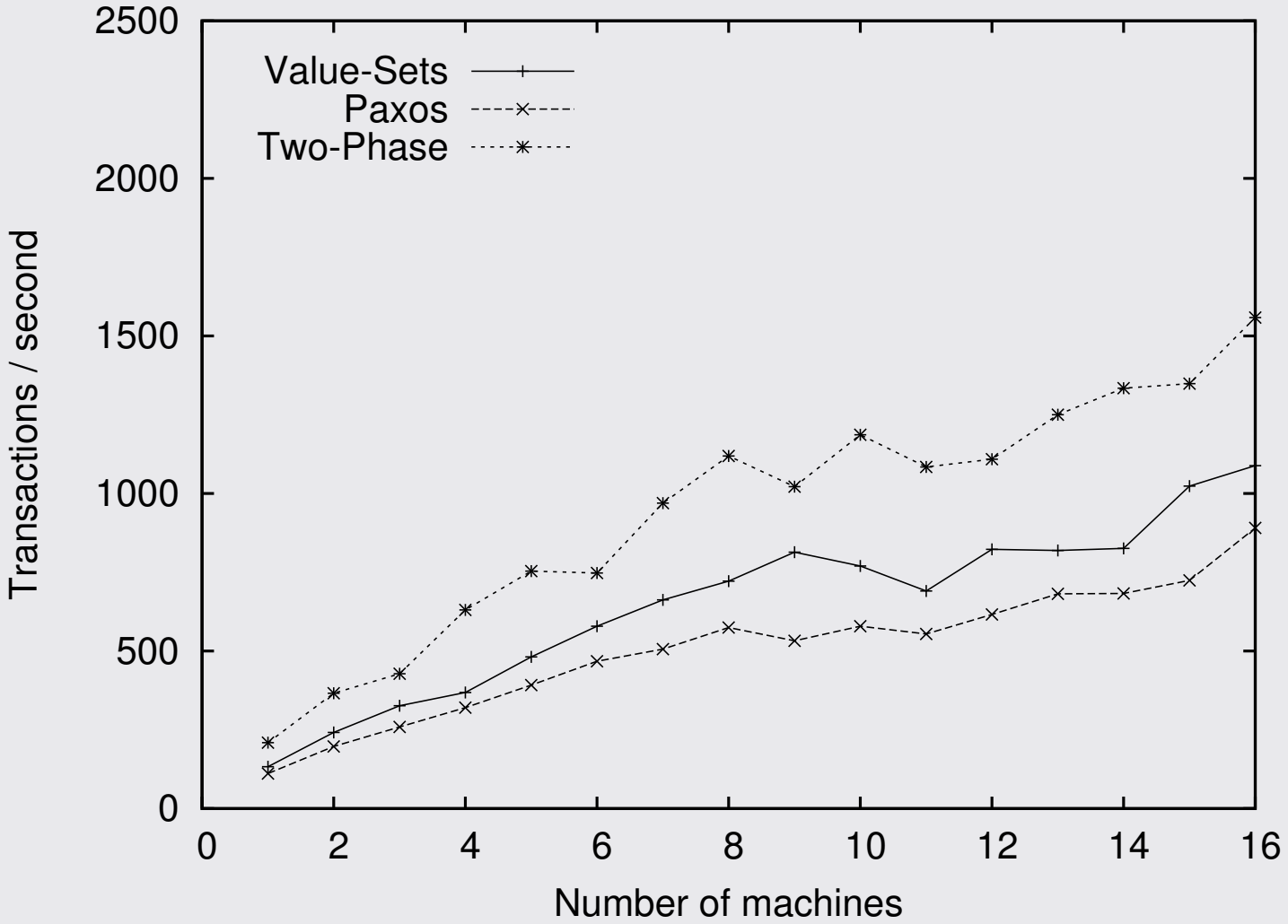
Trappist's Evaluation - Average Performance



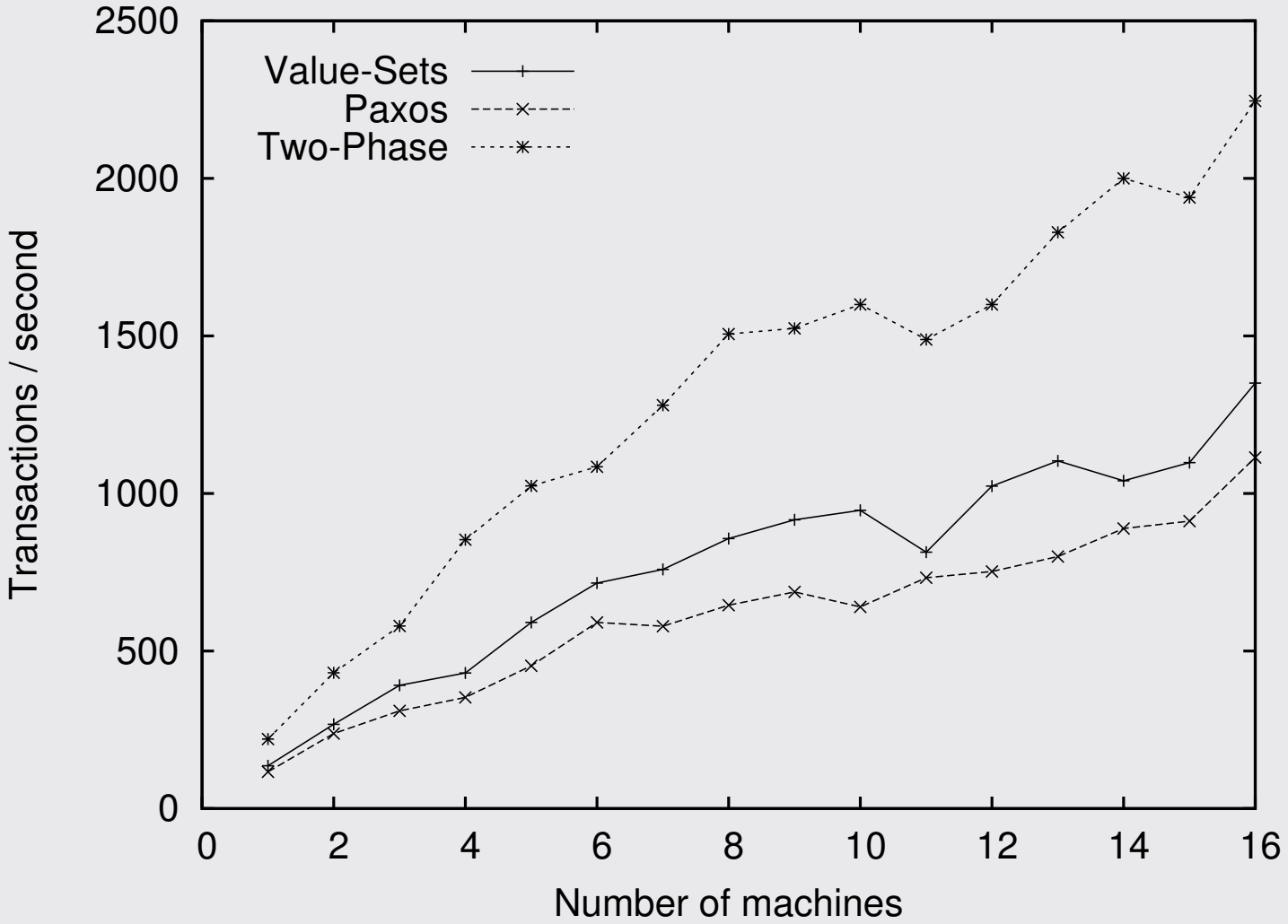
Trappist's Evaluation - Average Performance



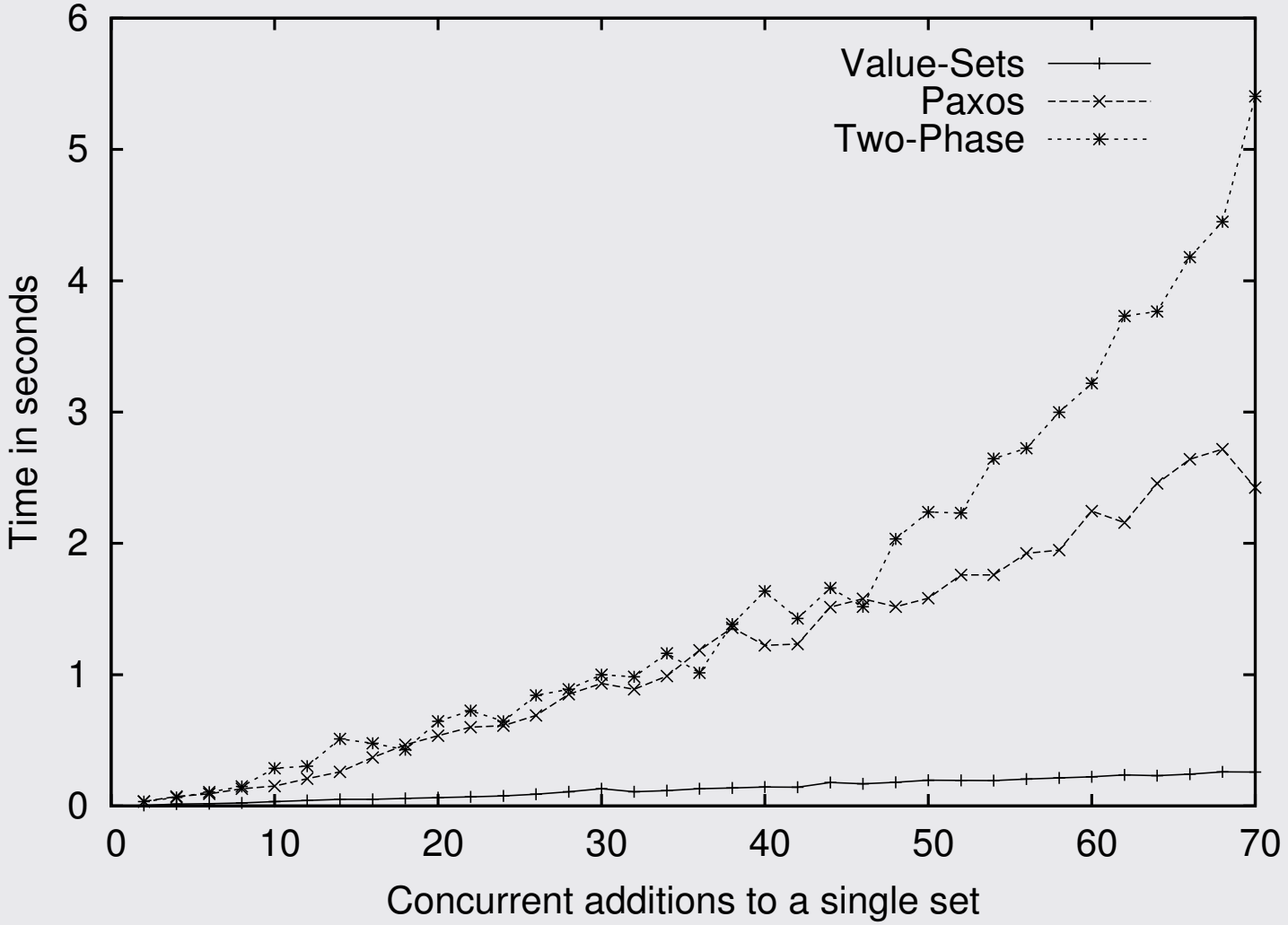
Trappist's Evaluation - Average Performance



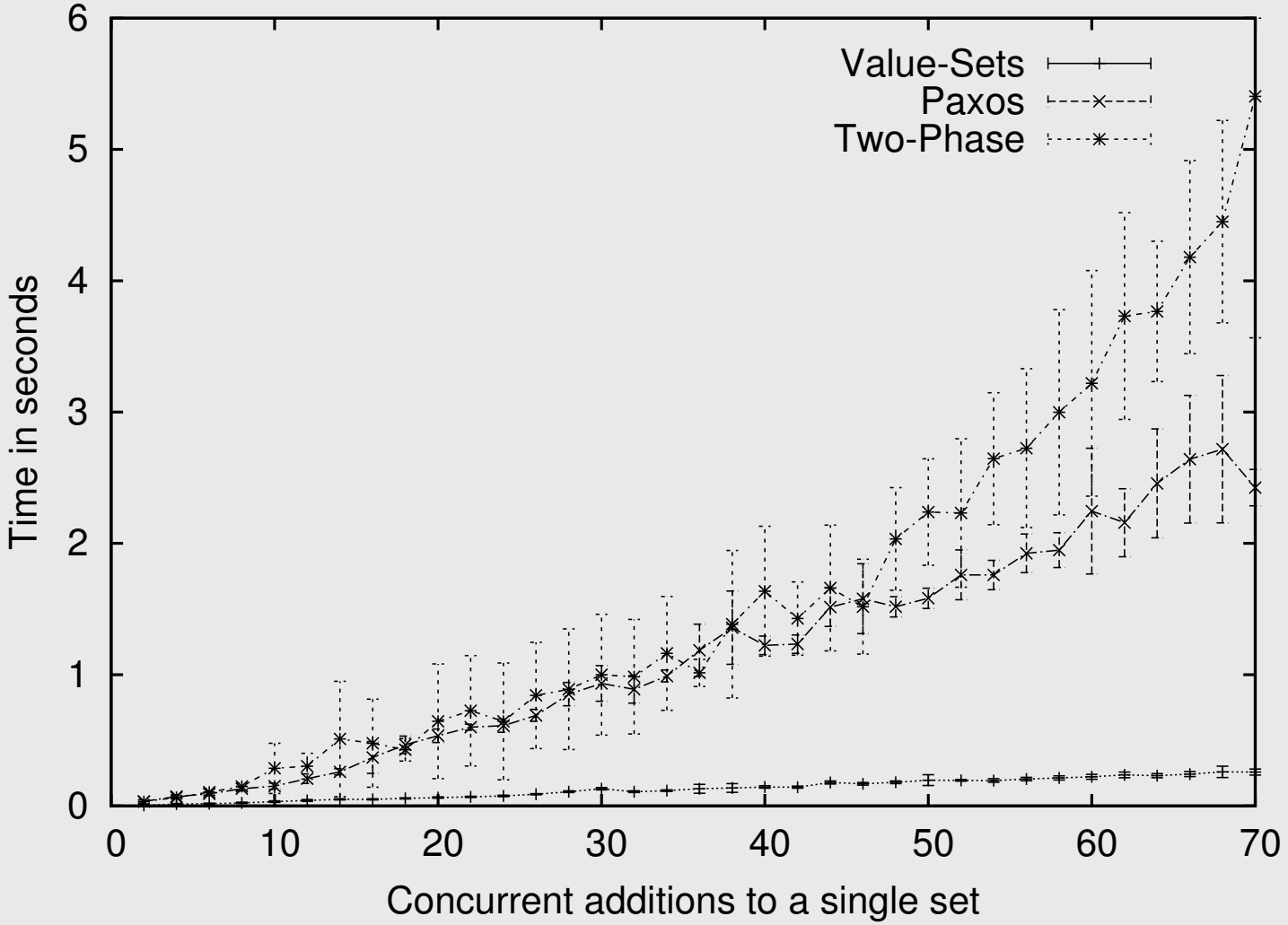
Trappist's Evaluation - Maximal Performance



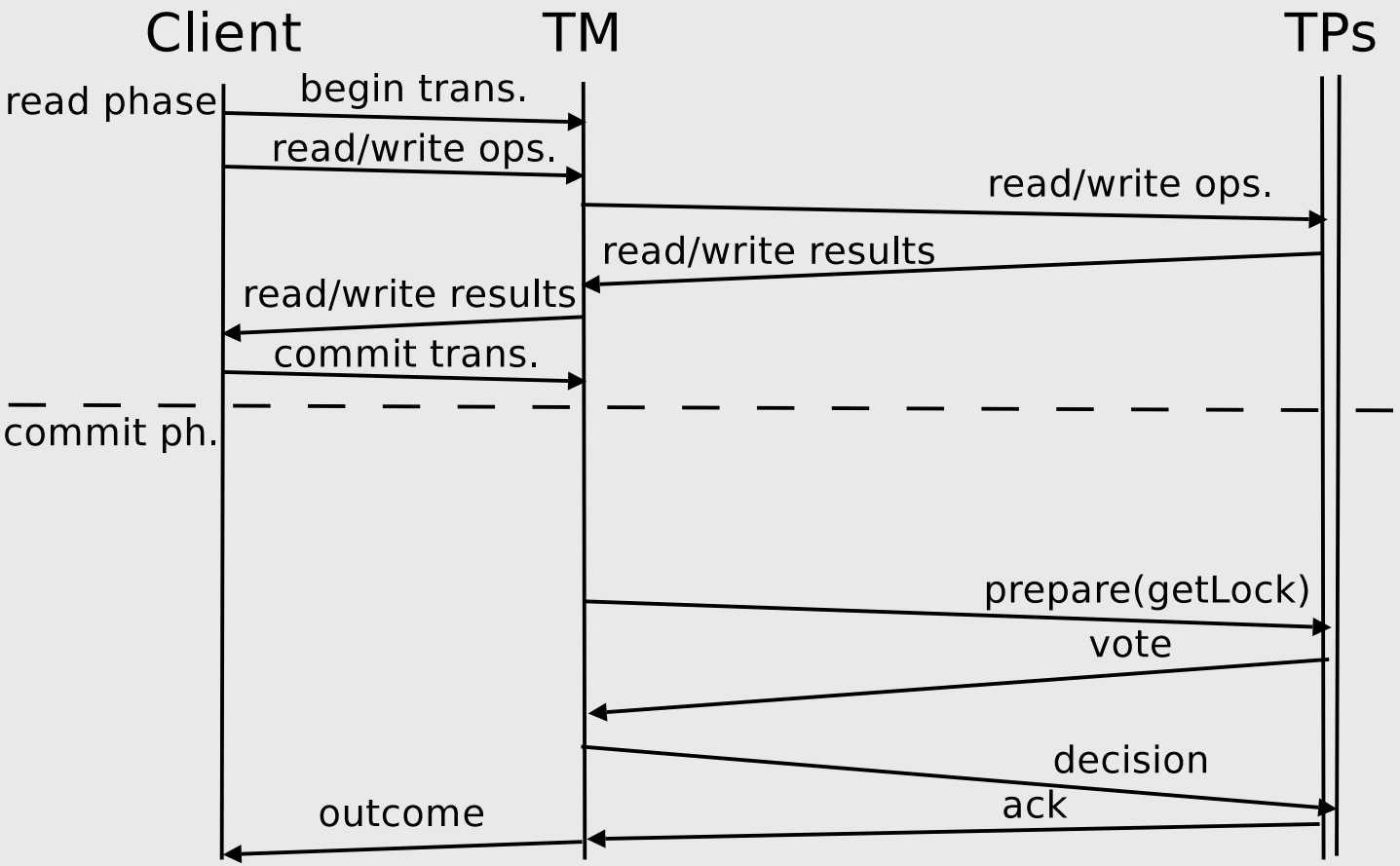
Trappist's Evaluation - Race Conditions



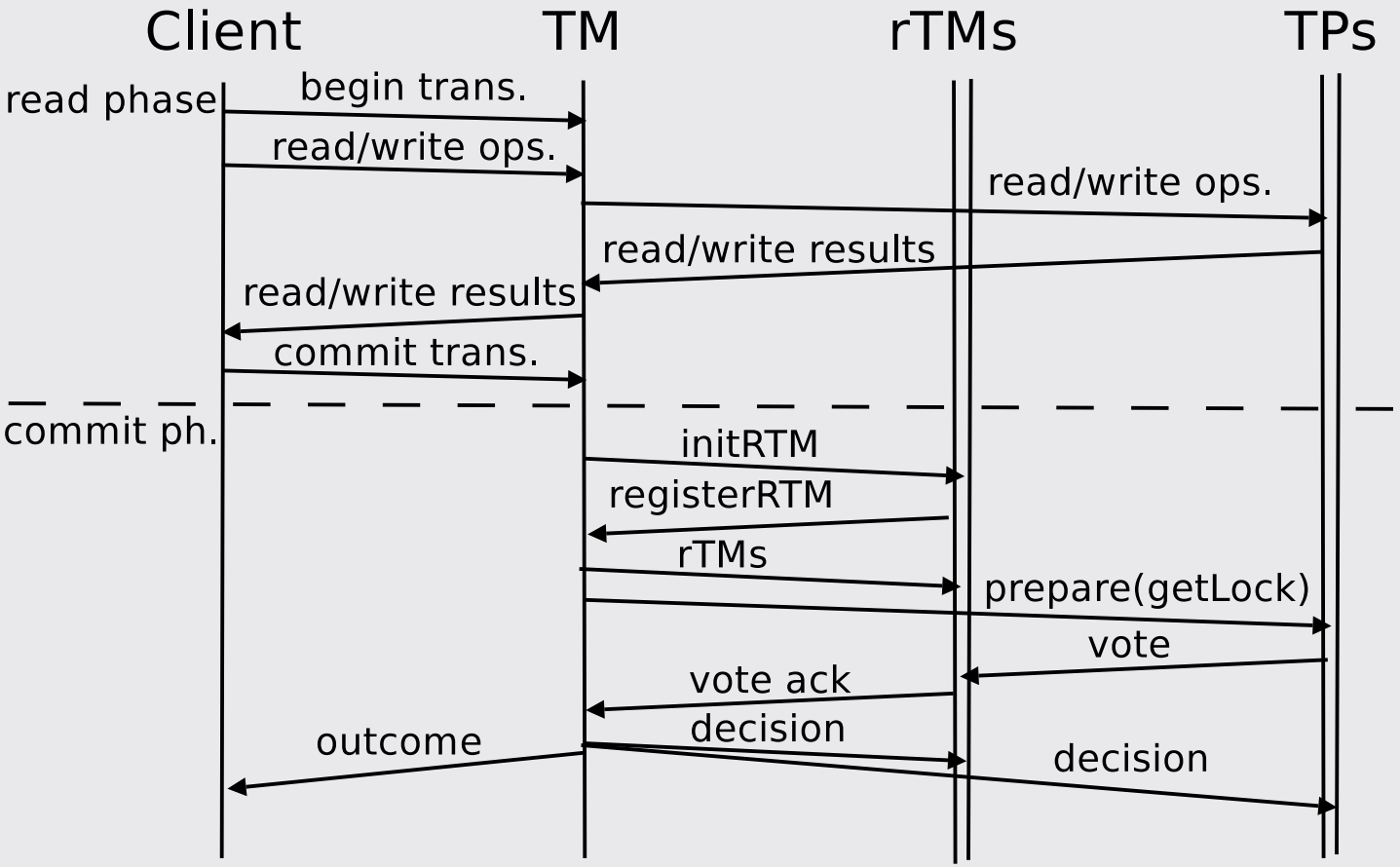
Trappist's Evaluation - Race Conditions



Two-Phase Commit



Atomic Paxos Commit



What about relaxation

- **Two-phase commit**:
  - The TM and all replicas must survive the transaction
    $\Rightarrow$ requirements too hard to meet in peer-to-peer systems.

What about relaxation

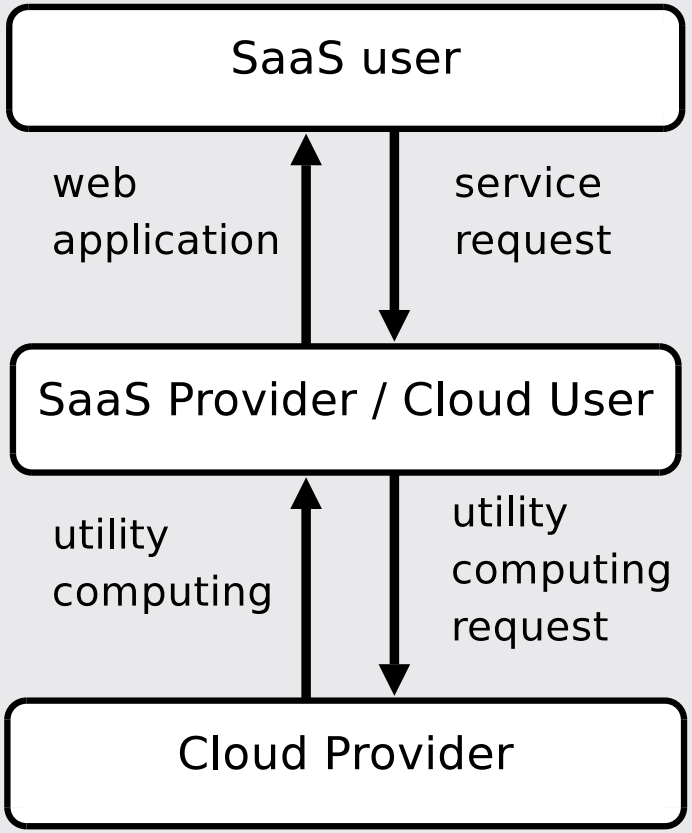
- **Two-phase commit**:
  - The TM and all replicas must survive the transaction
    $\Rightarrow$ requirements too hard to meet in peer-to-peer systems.
- **Paxos Consensus**:
  - Relaxes the condition on the survival of the TM by introducing rTMs
  - Relaxes the condition over all replicas by introducing consensus working with the majority.
  - $\Rightarrow$ Extra cost in performance and bandwidth usage, but the cost is constant and makes it suitable for peer-to-peer systems.

What about relaxation

- **Two-phase commit**:
  - The TM and all replicas must survive the transaction
    $\Rightarrow$ requirements too hard to meet in peer-to-peer systems.
- **Paxos Consensus**:
  - Relaxes the condition on the survival of the TM by introducing rTMs
  - Relaxes the condition over all replicas by introducing consensus working with the majority.
  - $\Rightarrow$ Extra cost in performance and bandwidth usage, but the cost is constant and makes it suitable for peer-to-peer systems.
- **Key/value-sets**
  - Relaxes versioning and ordering of elements
    $\Rightarrow$ get rid of locks, better performance and less race conditions.

Applications

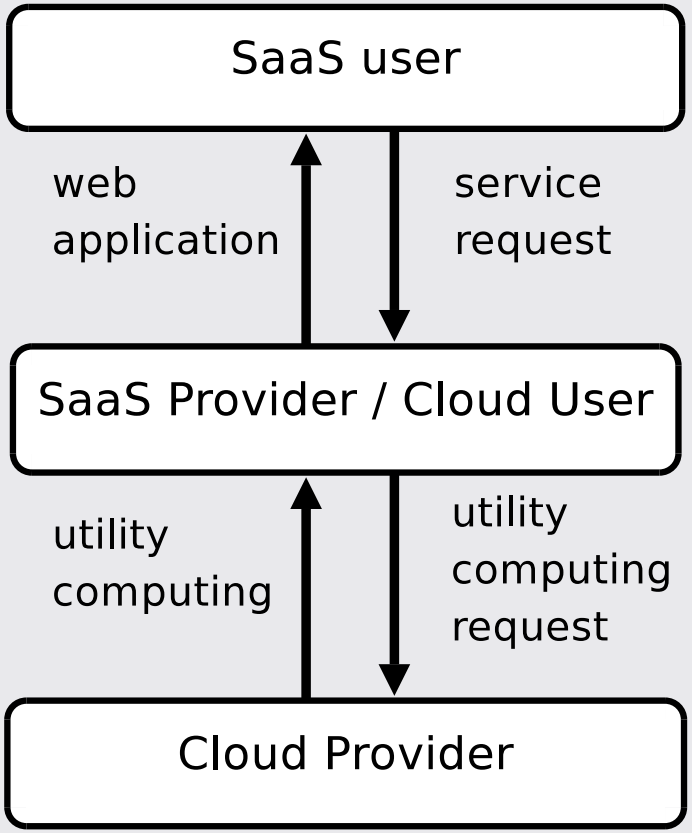
A Note on Cloud Computing and Elasticity



Berkeley's view of cloud computing

- **Elasticity** helps to take advantage of the cloud

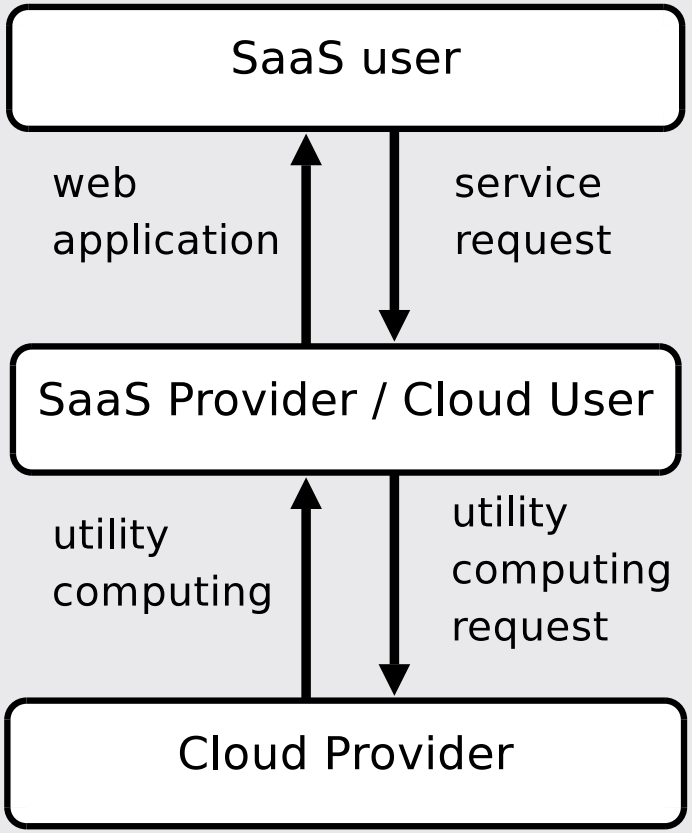
A Note on Cloud Computing and Elasticity



Berkeley's view of cloud computing

- **Elasticity** helps to take advantage of the cloud
- **Scale up** to provide good performance if service request increases (no Twitter's "fail whale")

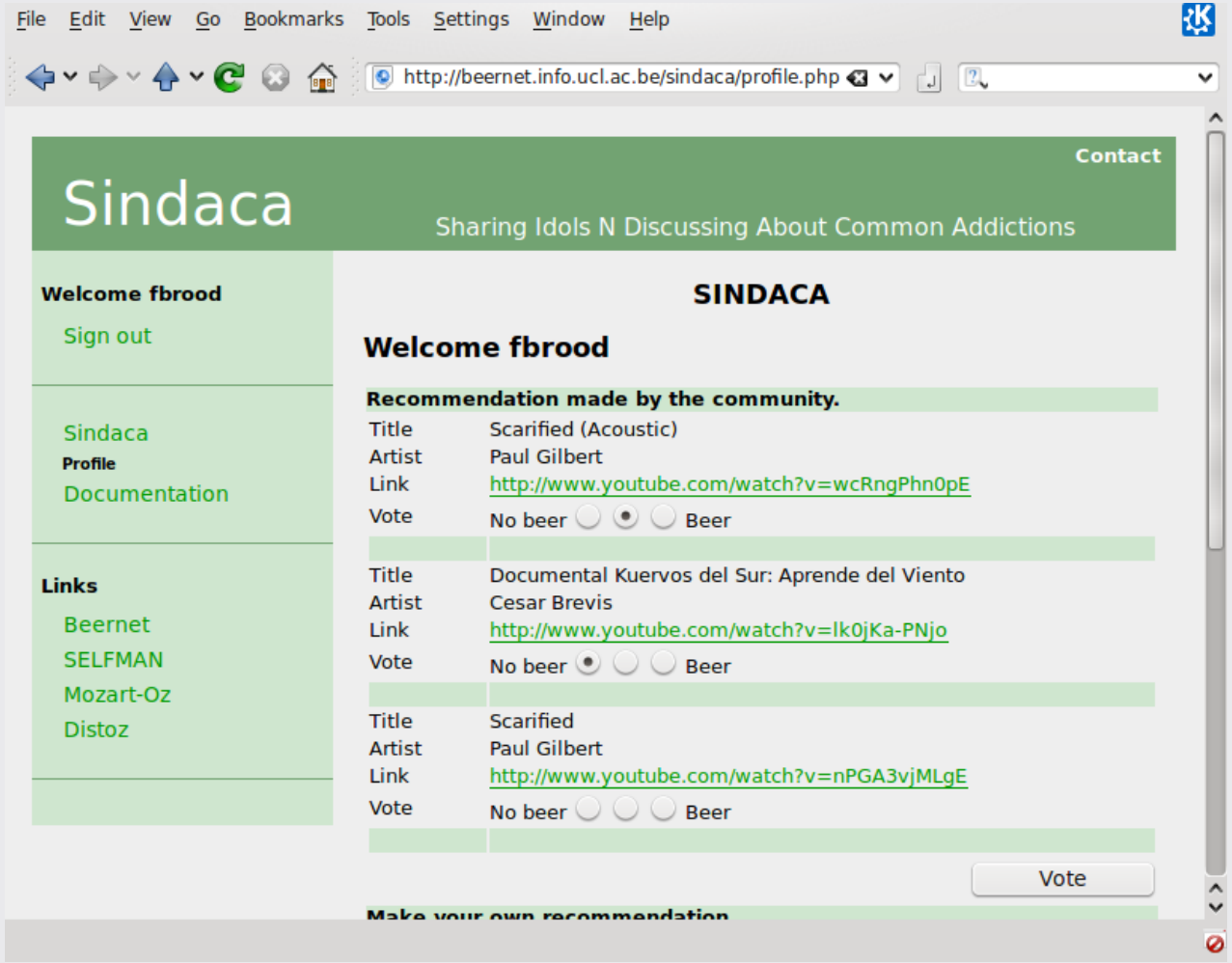
A Note on Cloud Computing and Elasticity



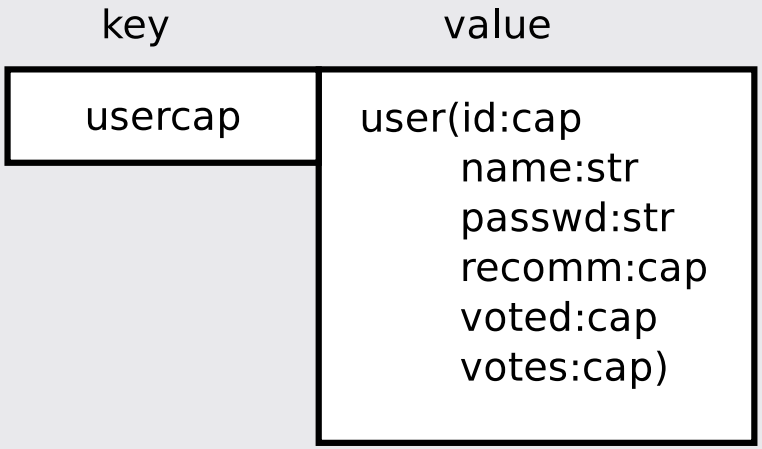
Berkeley's view of cloud computing

- **Elasticity** helps to take advantage of the cloud
- **Scale up** to provide good performance if service request increases (no Twitter's "fail whale")
- **Scale down** to minimize utility computing request

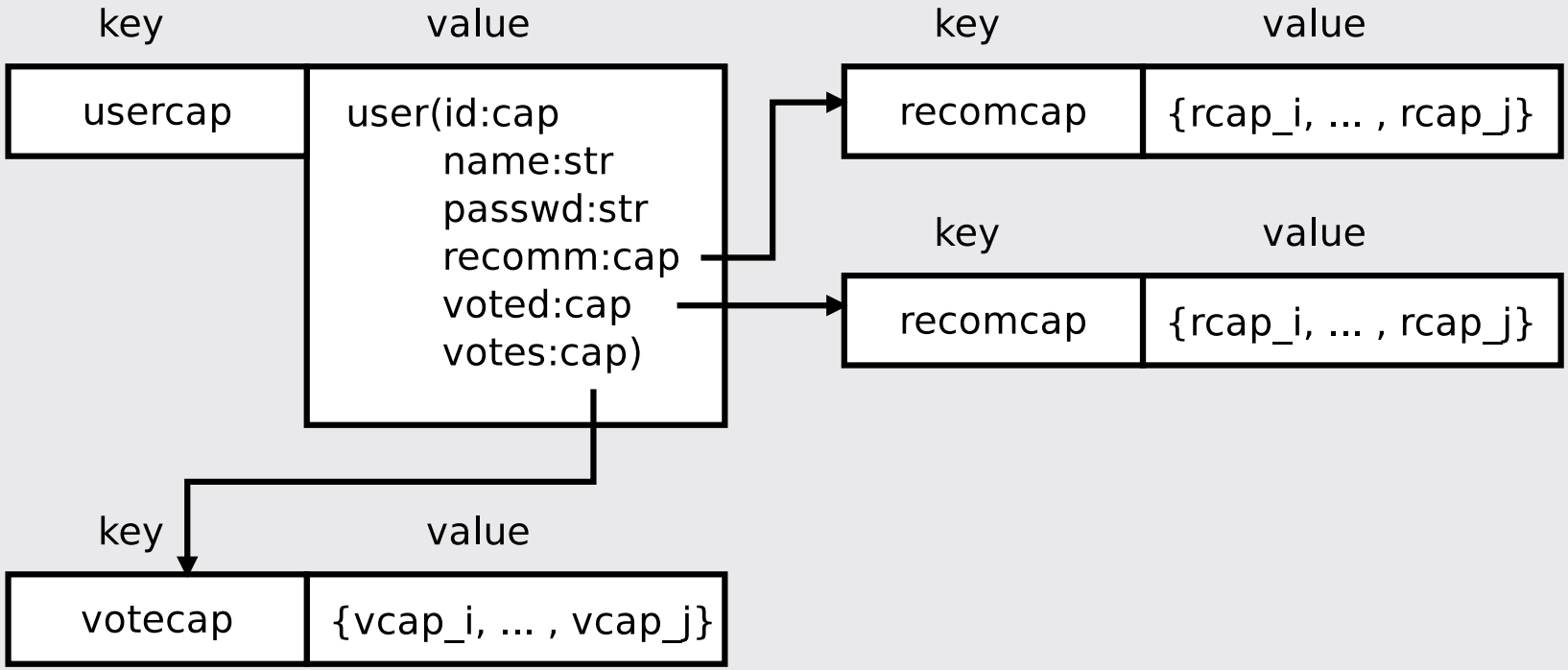
Sindaca - Community driven recommendation system



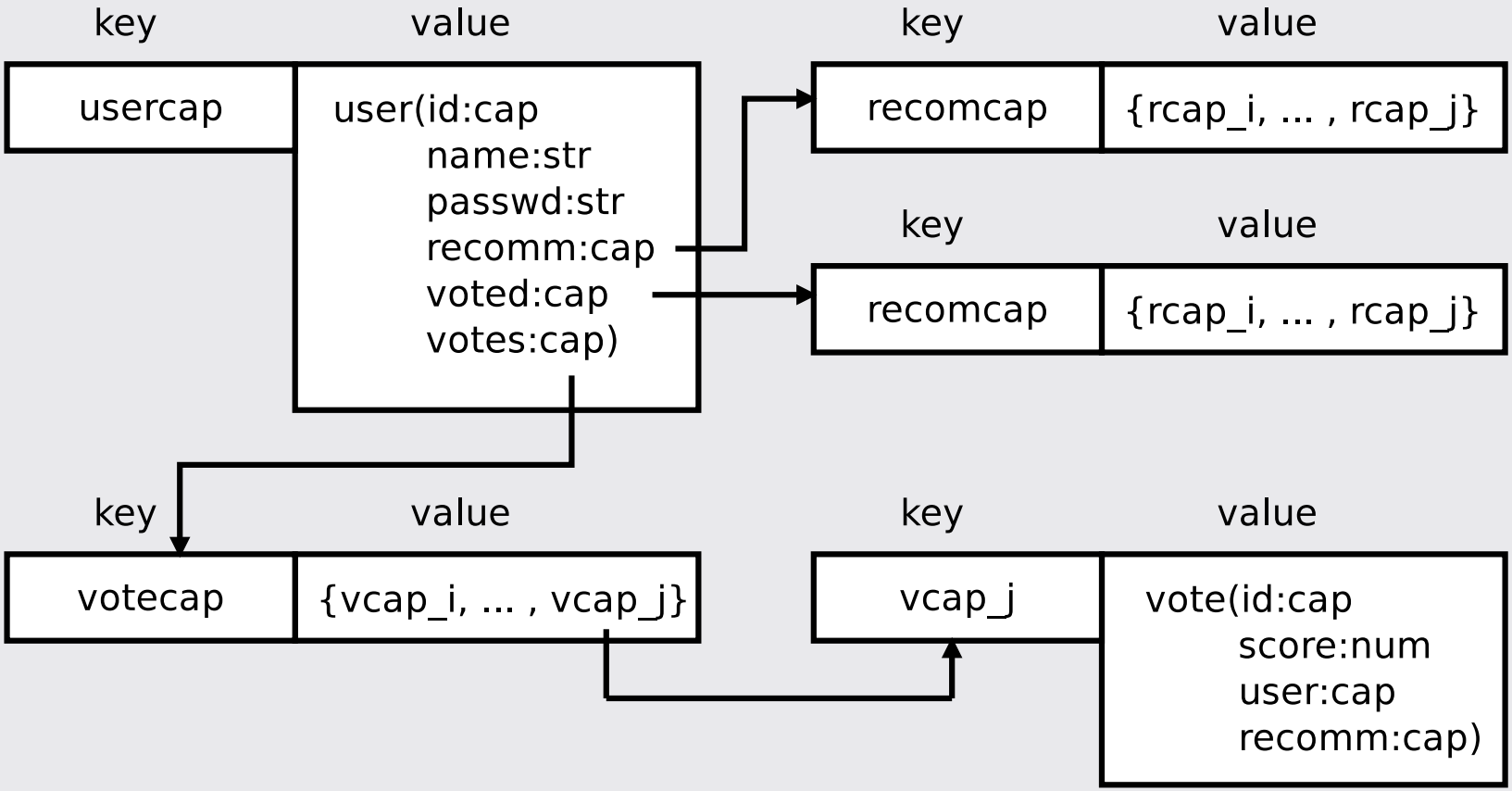
Sindaca's key/value store



Sindaca's key/value store



Sindaca's key/value store



WikipediOz

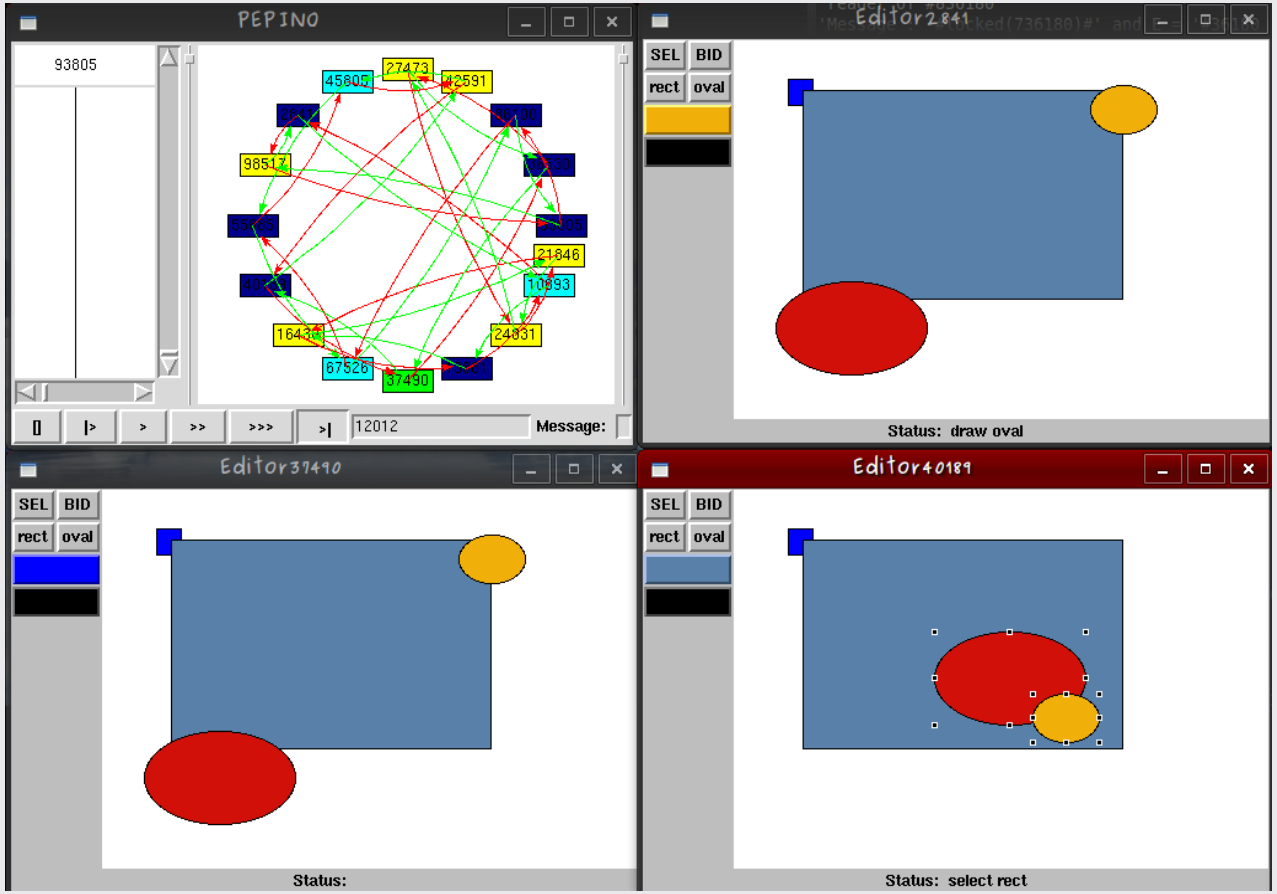
Thanks to Quentin Pirmez and Laurent Pierson

Programming with transactions

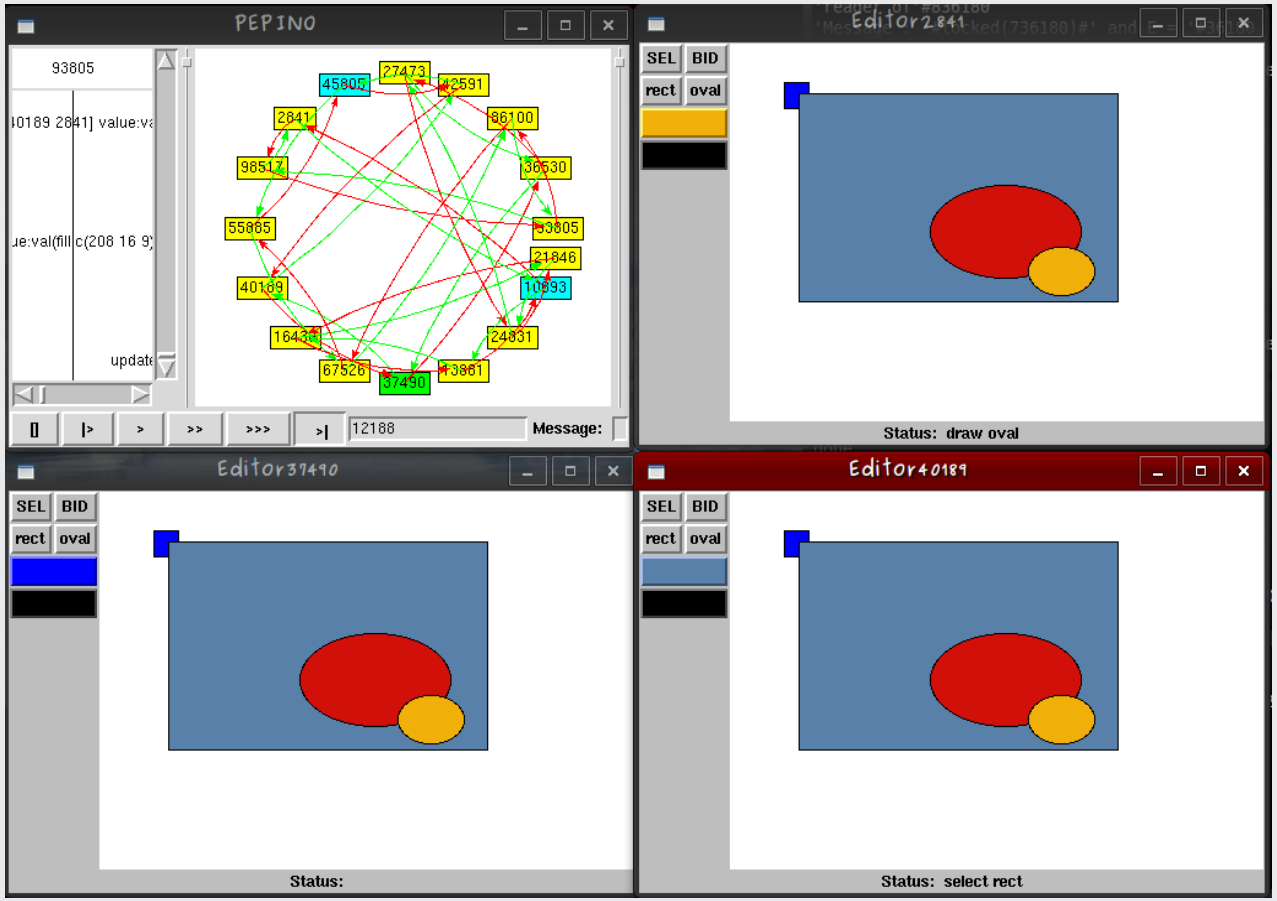
```
proc {UpdateArticle ToUpdate ToDelete}
   proc {Trans TM}
      for UpdPar in ToUpdate do
         {TM write(UpdPar.id UpdPar.text)}
      end
      for DelPar in ToDelete do
         {TM remove(DelPar.id)}
      end
      {TM commit}
   end
in
   {PBeer executeTransaction(Trans Client paxos)}
end
```

Thanks to Alexandre Bultot and Laurent Herbin

DeTransDraw - Decentralized Transactional Drawing



DeTransDraw - Decentralized Transactional Drawing



Conclusions

- We successfully apply the **relax approach** to build scalable distributed systems with self-managing behaviour and transactional robust storage.

Conclusions

- We successfully apply the **relax approach** to build scalable distributed systems with self-managing behaviour and transactional robust storage.
- One **relaxed ring** to rule them all
  - Self organizing
  - Self healing
  - Tolerates non-transitive connectivity

Conclusions

- We successfully apply the **relax approach** to build scalable distributed systems with self-managing behaviour and transactional robust storage.
- One **relaxed ring** to rule them all
  - Self organizing
  - Self healing
  - Tolerates non-transitive connectivity
- **Transactional DHT**
  - Symmetric replication
  - Pessimistic and optimistic locking
  - Lock-free key/value-sets
  - Notification layer

Conclusions

- We successfully apply the **relax approach** to build scalable distributed systems with self-managing behaviour and transactional robust storage.
- One **relaxed ring** to rule them all
  - Self organizing
  - Self healing
  - Tolerates non-transitive connectivity
- **Transactional DHT**
  - Symmetric replication
  - Pessimistic and optimistic locking
  - Lock-free key/value-sets
  - Notification layer
- **Software**
  - Free/open source software release: Beernet-0.8
  - Several applications

Future Work

- Technical improvements
  - Improve performance of Trappist's protocols
  - Study data reallocation performance on high churn
  - Improve semantics and API of storage operations to allow several applications running on the same net

Future Work

- Technical improvements
  - Improve performance of Trappist's protocols
  - Study data reallocation performance on high churn
  - Improve semantics and API of storage operations to allow several applications running on the same net
- Further study how peer-to-peer systems can provide elasticity in cloud computing
  - Beer in the Clouds
  - Social Interactions based on Beer(net)

That's it. . .

A beer a day keeps the doctor away

That's it. . .

A beer a day keeps the doctor away
Relaxing a bit can help to find the way

http://beernet.info.ucl.ac.be