GIAN Course on Distributed Network Algorithms

Distributed Synchronization

«A man with one clock knows what time it is –
a man with two is never sure.»

Synchronization often very helpful to simplify design and analyze algorithms for distributed system and provide consistency. But also challenging....

Distributed Synchronization

«A man with one clock knows what time it is –
a man with two is never sure.»

Synchronization often very helpful to simplify design and analyze algorithms for distributed system and provide consistency. But also challenging....

Distributed Synchronization

Remember BFS: easy and cheap in LOCAL model, more expensive (time and/or messages) in asynchronous model. E.g., D^2 runtime if we enforce synchronous rounds (Dijkstra algorithm).

Synchronization often very helpful to simplify design and analyze algorithms for distributed system and provide consistency. But also challenging....

Distributed Synchronization

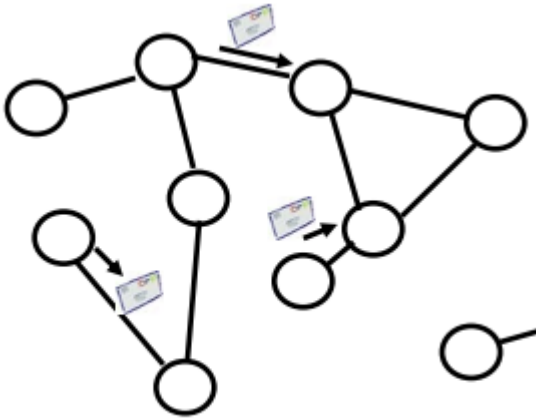
Remember BFS: easy and cheap in LOCAL model, more expensive (time and/or messages) in asynchronous model. E.g., D^2 runtime if we enforce synchronous rounds (Dijkstra algorithm).

Can we execute any LOCAL algorithm in asynchronous environment? Yes, with a synchronizer!

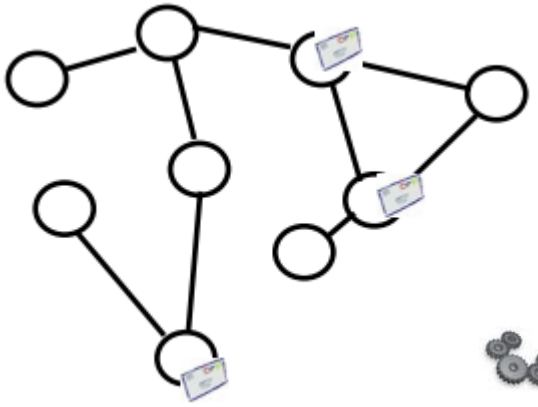
The LOCAL Model: A Synchronous Model

Synchronous LOCAL algorithms simple to design and reason about:

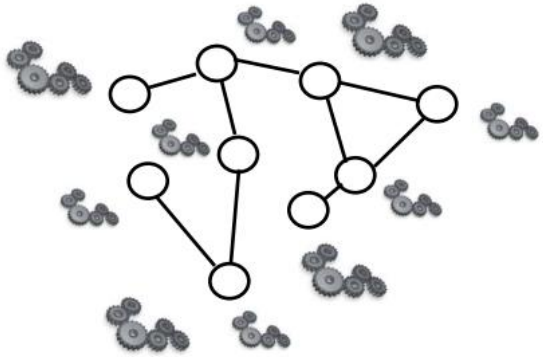
Send...



... receive...



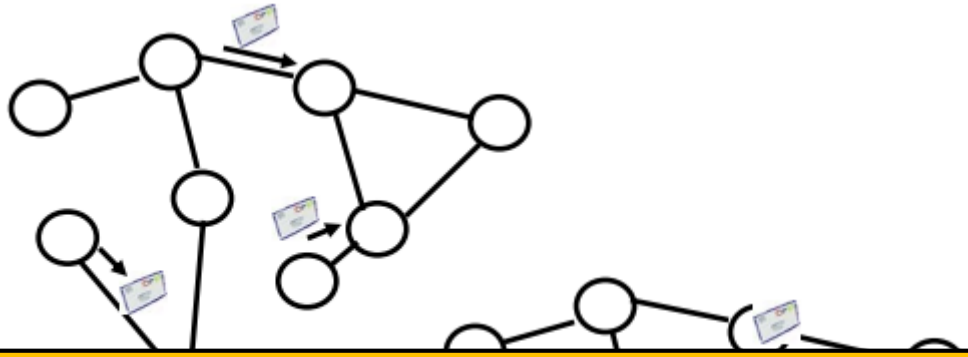
... compute.



The LOCAL Model: A Synchronous Model

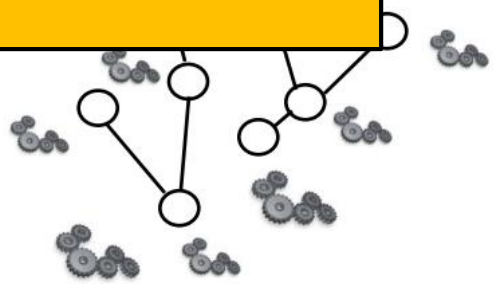
Synchronous LOCAL algorithms simple to design and reason about:

Send...



But how to render an asynchronous system synchronous? Run LOCAL algorithm in asynchronous environment: need a **distributed synchronizer**!

... compute.

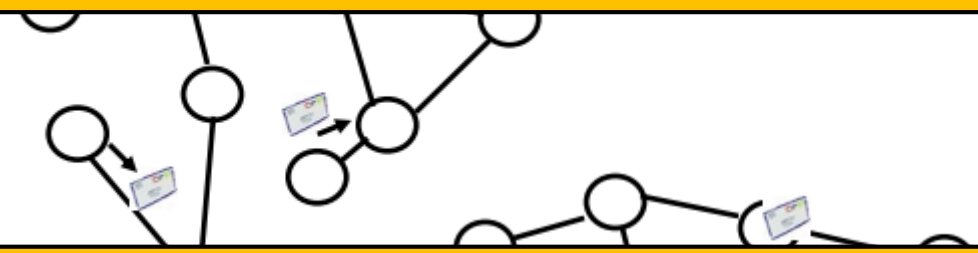


The LOCAL Model: A Synchronous Model

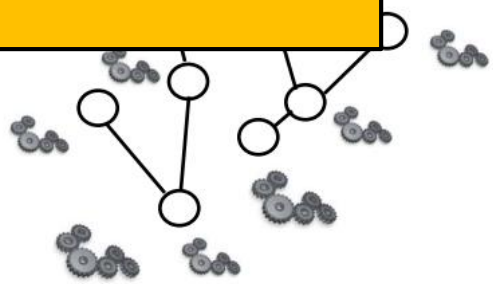
Synchrone[illegible]orithms simple to design and reason about:

Remember BFS: artificially synchronized protocol to make it use less messages in the worst case!

Send...



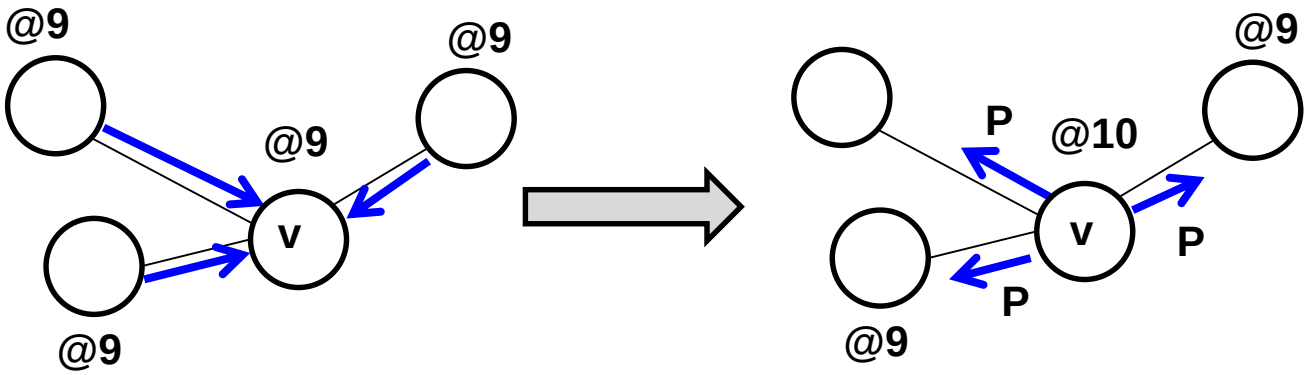
But how to render an asynchronous system synchronous? Run LOCAL algorithm in asynchronous environment: need a **distributed synchronizer**!



... compute.

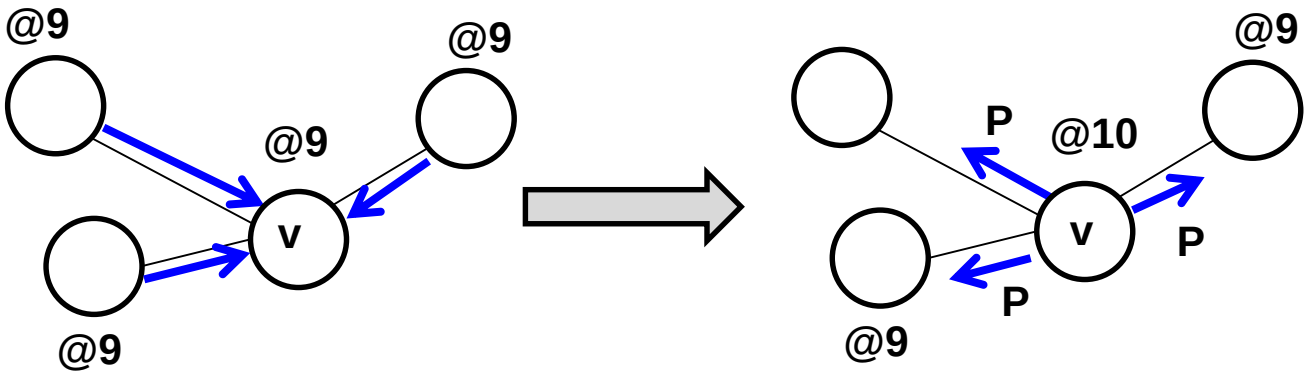
Synchronizer

A synchronizer is a distributed algorithm which generates clock pulses (PULSE) at each node.



Synchronizer

A synchronizer is a distributed algorithm which generates clock pulses (PULSE) at each node.

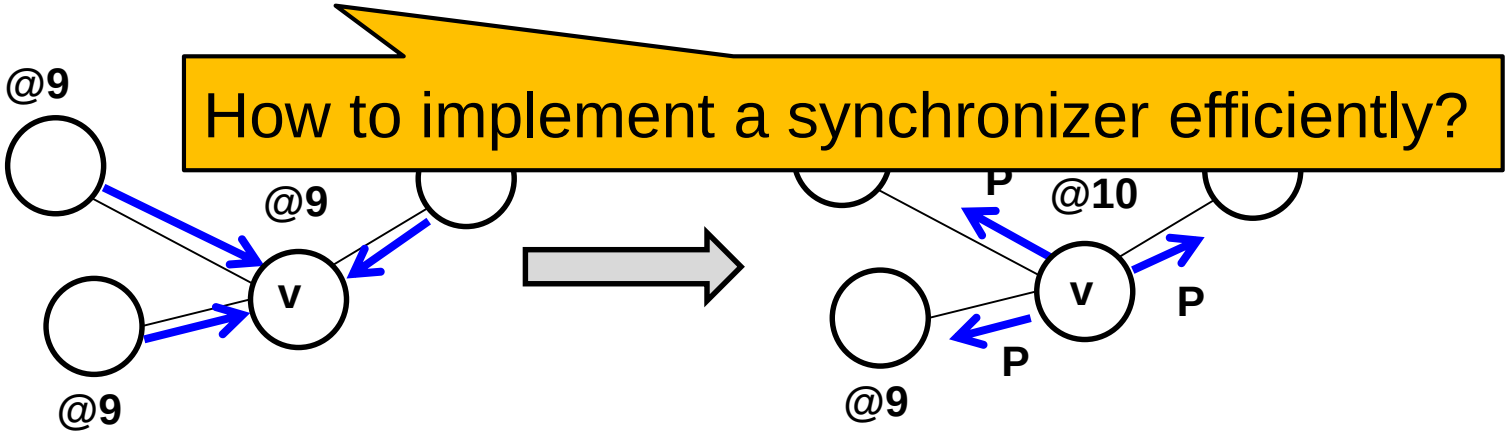


Valid Clock Pulse

A pulse generated at some node v is valid, iff it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in the previous pulses.

Synchronizer

A synchronizer is a distributed algorithm which generates clock pulses (PULSE) at each node.

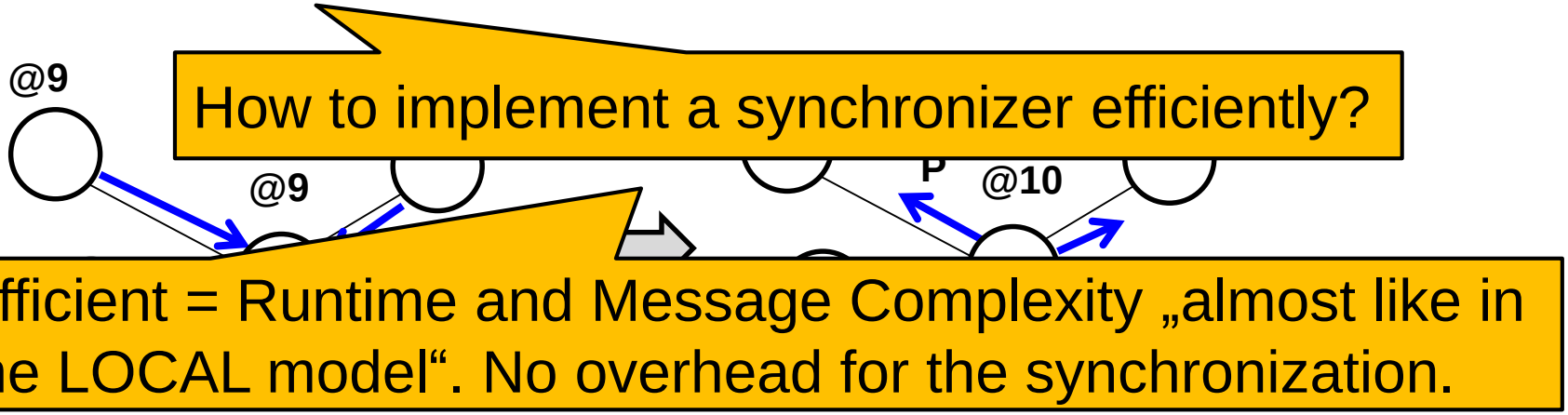


Valid Clock Pulse

A pulse generated at some node v is valid, iff it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in the previous pulses.

Synchronizer

A synchronizer is a distributed algorithm which generates clock pulses (PULSE) at each node.



Valid Clock Pulse

A pulse generated at some node v is valid, iff it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in the previous pulses.

Formally: Overhead of Using Synchronizers

Overhead: Message needed for each pulse, independent of protocol data!

Overhead

Say T(A) and M(A) are time and message complexity of synchronous algorithm A, and T(S) and M(S) are complexities of a synchronizer for each pulse. Moreover, $T_{init}(S)$ and $M_{init}(S)$ to set up synchronizer.
Then:

$$T = T_{init}(S) + T(A)*(1+T(S)),\ M = M_{init}(S) + M(A) + T(A)*M(S)$$

Some setup costs.

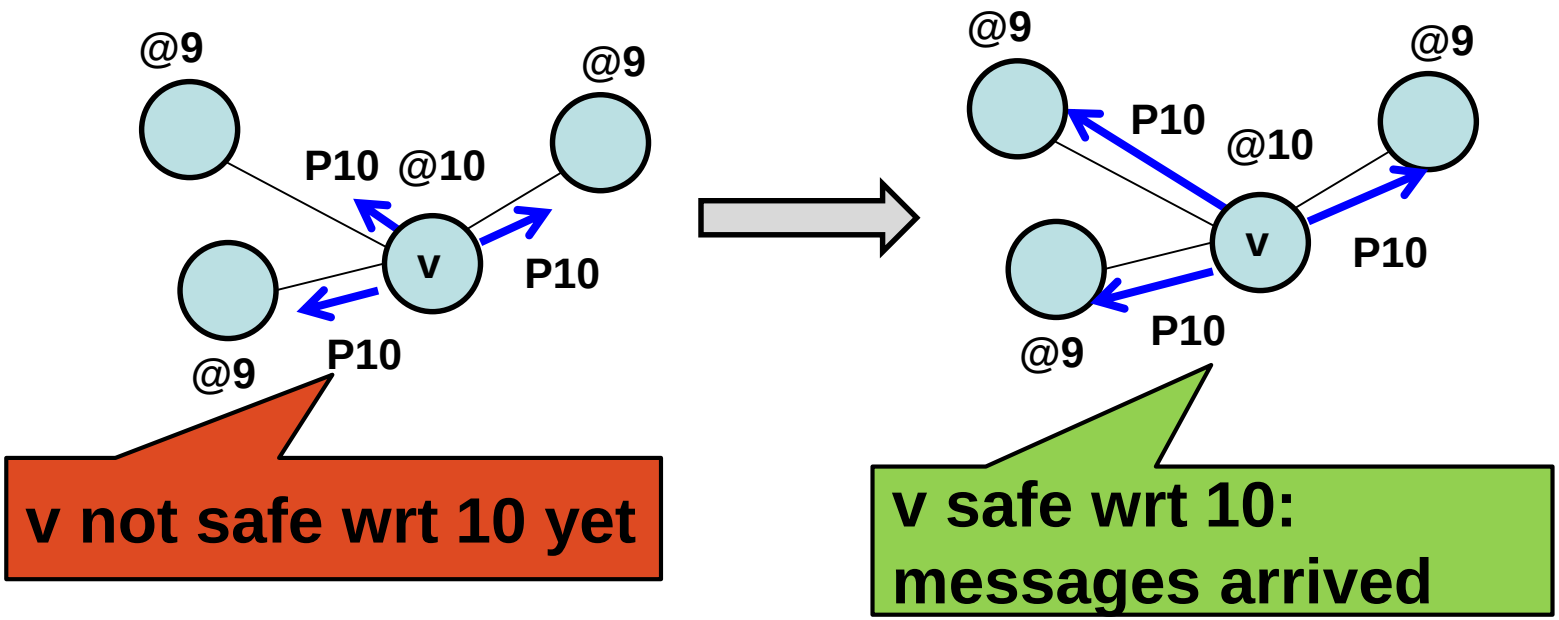
Extra rounds per LOCAL round.

Maybe additionally messages M(S) per each LOCAL round.

Safe Node

A node v is safe wrt certain clock pulse if all messages of the synchronous algorithm sent by v in that pulse have already arrived *at their destination*.

Idea: v at least knows what it sent itself!

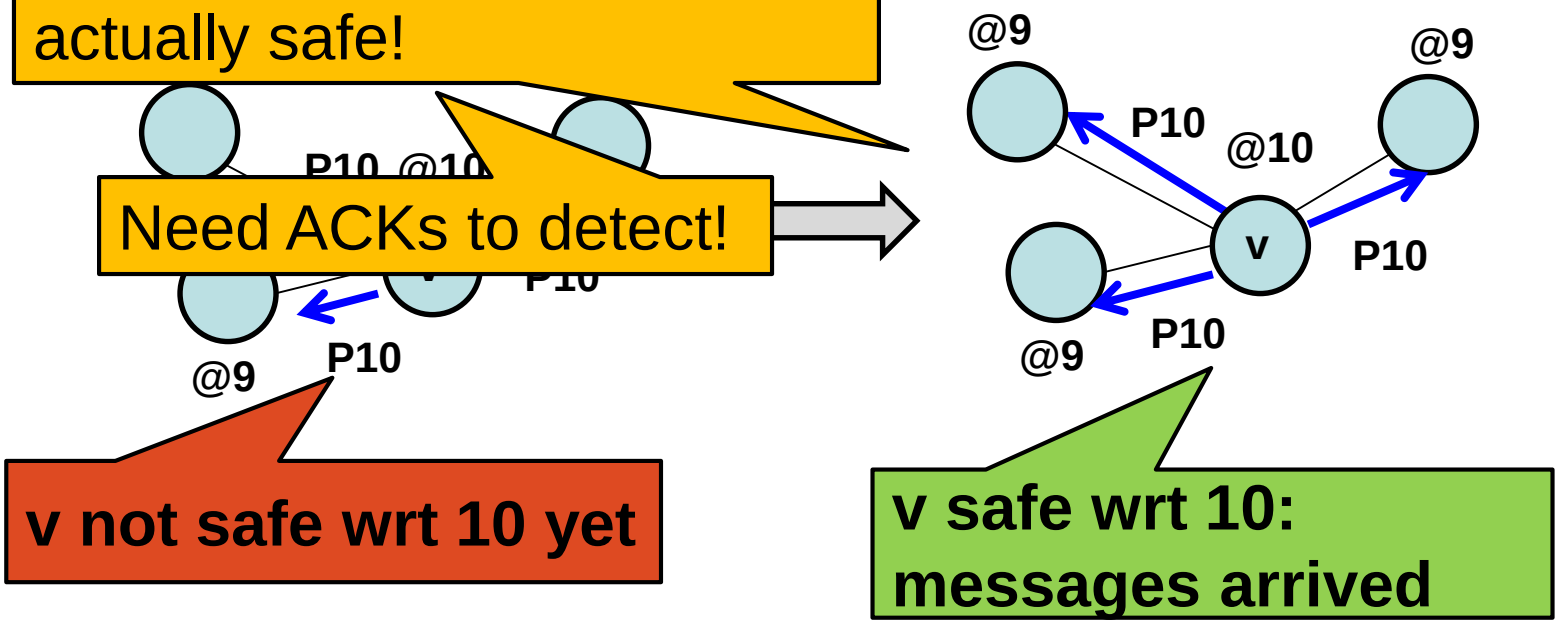


Safe Node

A node v is safe wrt certain clock pulse if all messages of the synchronous algorithm sent by v in that pulse have already arrived *at their destination*.

How to detect? At this point, v cannot know that it is actually safe!

sent itself!



Note: Once all neighbors of v are safe, v can generate the next pulse: it has received their messages for this round. This pulse must be valid.

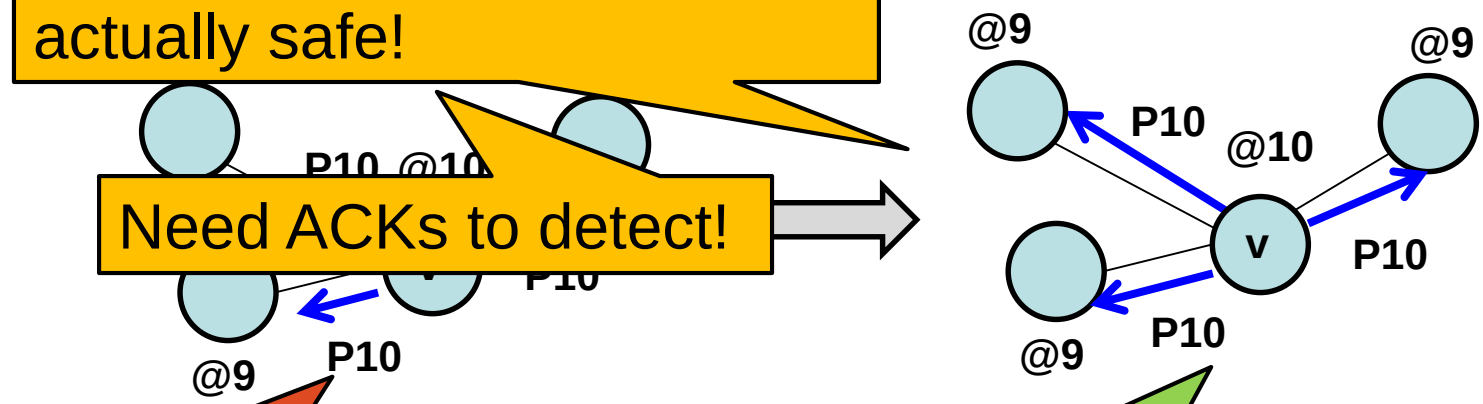
Safe Node

A node v is safe wrt certain pulse if all messages of the synchronous algorithm sent by v in that pulse have already arrived *at their destination*.

How to detect? At this point, v cannot know that it is actually safe!

sent itself!

Need ACKs to detect!



v not safe wrt 10 yet

v safe wrt 10: messages arrived

AKA Local Synchronizer α!

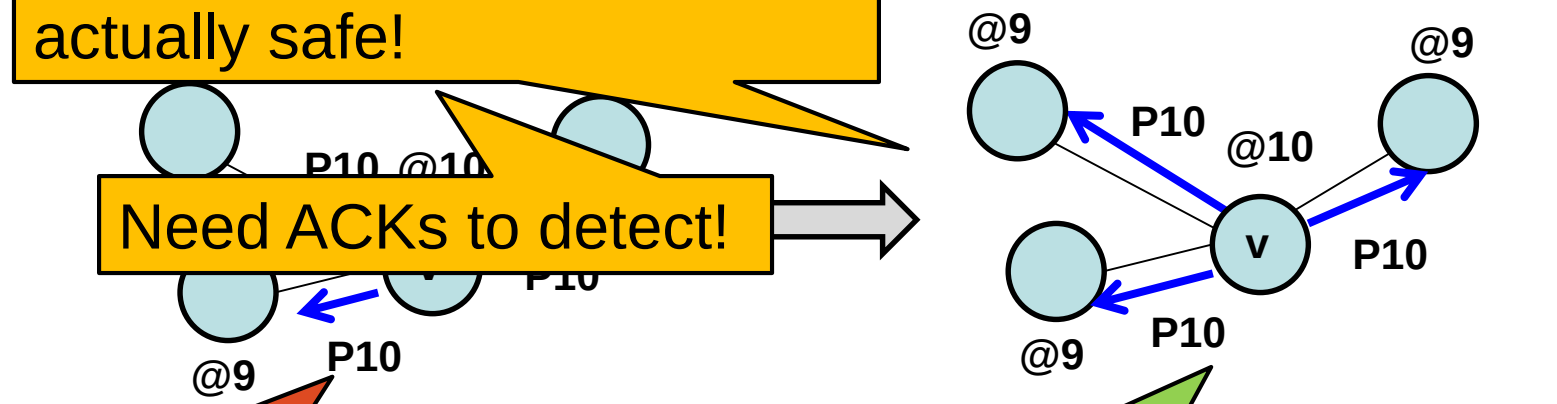
Once all neighbors of v
afe, v can generate the
next pulse: it has received
their messages for this round.
This pulse must be valid.

Safe Node

A node v is safe wrt certain
of the synchronous algorithm sent by v in that pulse have already arrived *at their destination*.

How to detect? At this point, v cannot know that it is actually safe!

sent itself!



Need ACKs to detect!

v not safe wrt 10 yet

v safe wrt 10: messages arrived

The Local Synchronizer α

Synchronizer α

v's messages arrived!

At node v:

wait until v is safe (learn via ACKs)
send SAFE to all neighbors
wait until v receives SAFE messages from all neighbors
start new PULSE

v got their messages!

We do this for each regular LOCAL round!

The Local Synchronizer α

Synchronizer α

v's messages arrived!

At node v:

wait until v is safe (learn via ACKs)
send SAFE to all neighbors
wait until v receives SAFE messages from all neighbors
start new PULSE

v got their messages!

We do this for each regular LOCAL round!

Overhead?

The Local Synchronizer α

Synchronizer α

At node v:

wait until v is safe (learn via ACKs)
send SAFE to all neighbors
wait until v receives SAFE messages from all neighbors
start new PULSE

Time good: just O(1) interactions with neighbors: no initialization and local!

Synchronizer α

Overhead per synchronous round:

$T(\alpha) = O(1)$
$M(\alpha) = O(m)$

The Local Synchronizer α

Synchronizer α

At node v:

wait until v is safe (learn via ACKs)
send SAFE to all neighbors
wait until v receives SAFE messages from all neighbors
start new PULSE

Time good: just O(1) interactions with neighbors: no initialization and local!

Synchronizer α

Overhead per synchronous round:

T(α) = O(1)
M(α) = O(m)

Messages not so good: *Every edge* sees 6 messages (PULSE, SAFE, ACK in both directions), in each round! Independently of whether LOCAL used this edge or not.

The Local Synchronizer α

Synch

At node v:

wai

send SAFE to all neighbors

wait until v receives SAFE messages from all neighbors

start new PULSE

The global synchronizer β provides the opposite tradeoff!

Time good: just O(1) interactions with neighbors: no initialization and local!

Sy **onizer α**

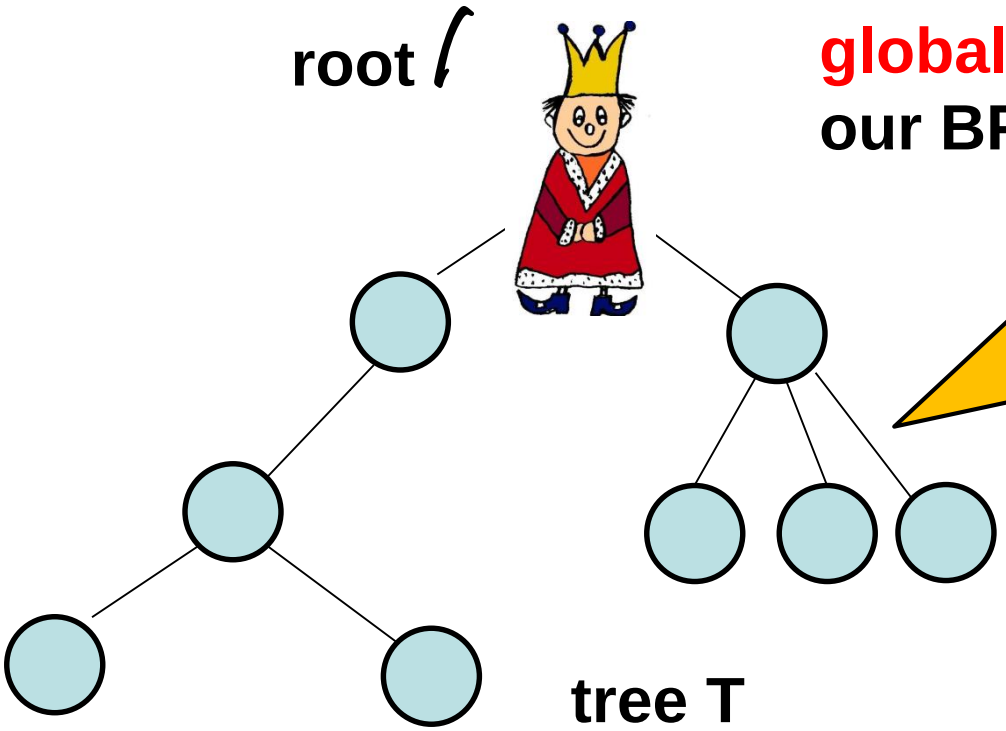
Overhead per synchron

T(α) = O(1)

M(α) = O(m)

Messages not so good: *Every edge* sees 6 messages (PULSE, SAFE, ACK in both directions), in each round! Independently of whether LOCAL used this edge or not.

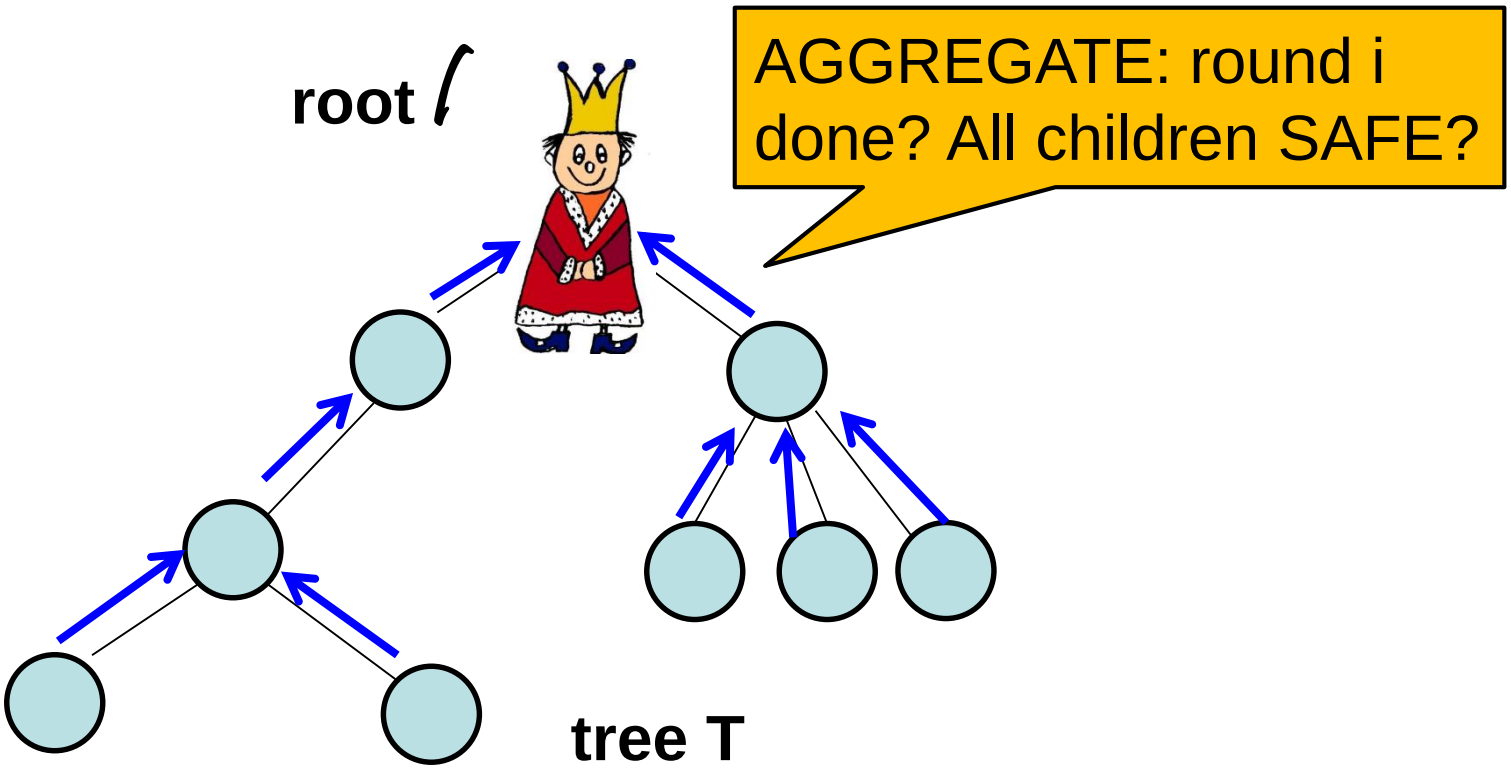
The Global Synchronizer β



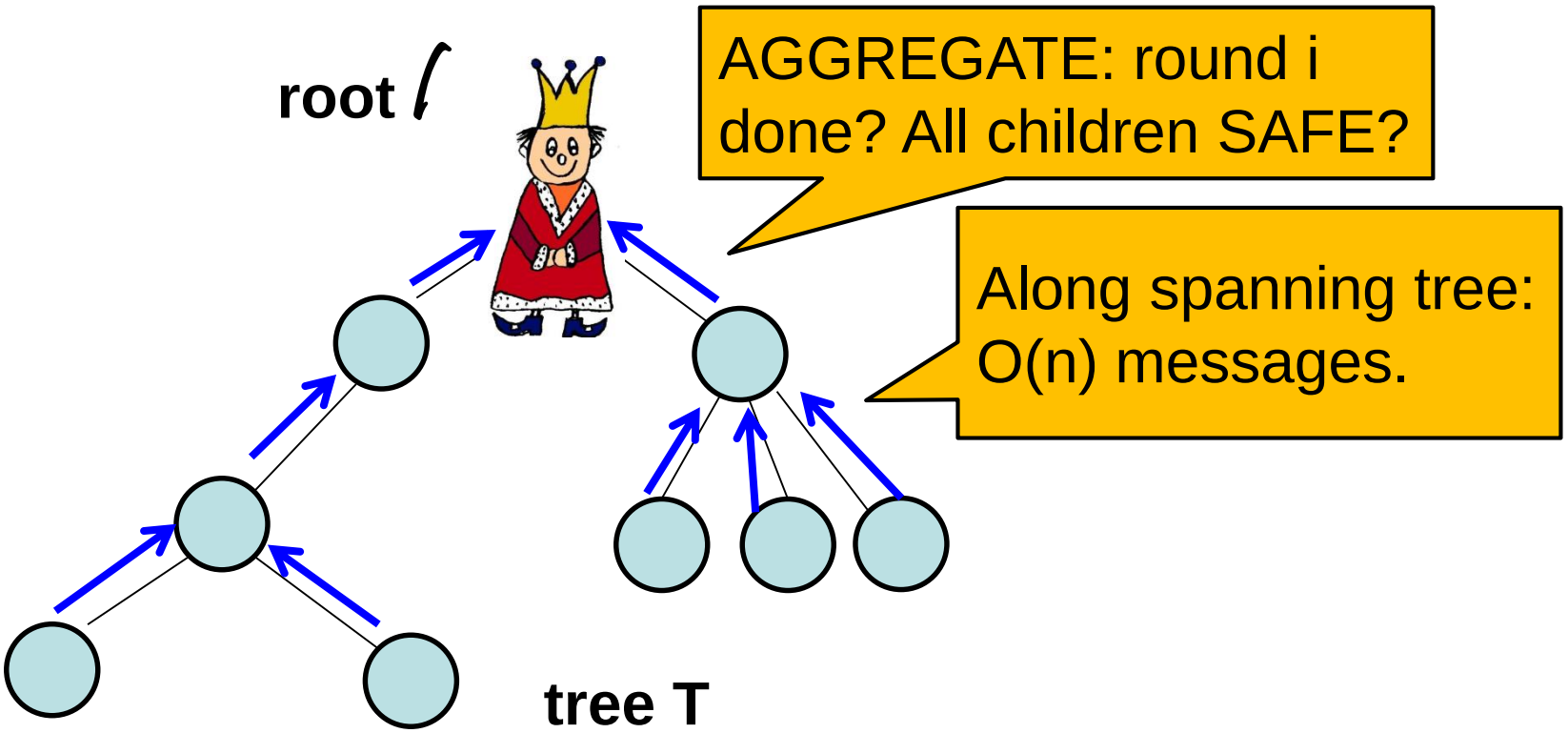
Idea: leader coordinates global phases: Remember our BFS algorithm!

In each round essentially a ConvergeCast: broadcast PULSE and aggregate SAFE subtrees!

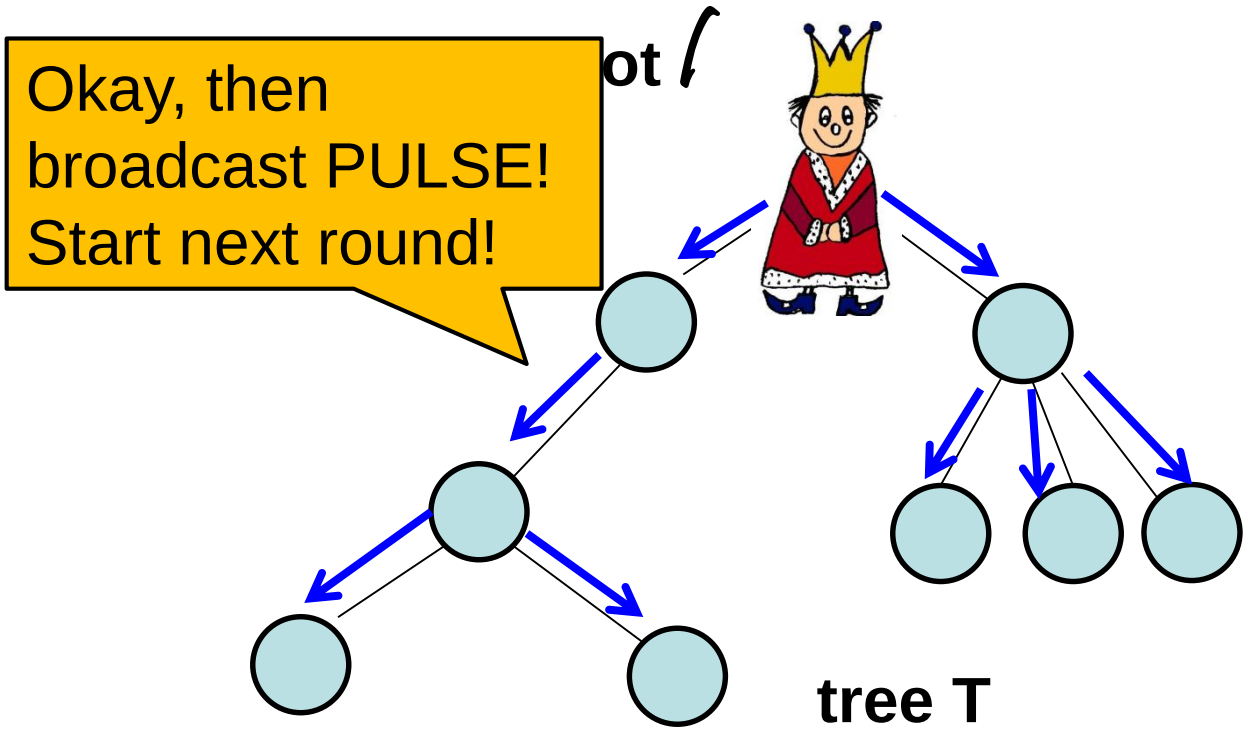
The Global Synchronizer β



The Global Synchronizer β



The Global Synchronizer β



Synchronizer β

At node v:

wait until v is safe
wait until v receives SAFE message from all its children in tree
only then send SAFE message to parent in T
wait until PULSE received from parent
send PULSE to children
start PULSE

Synchronizer β

At node v:

wait until v is safe
wait until v receives SAFE message from all its children in tree
only then send SAFE message to parent in T
wait until PULSE received from parent
send PULSE to children
start PULSE

Synchronizer β

Complexities per synchronous round:

$T(\beta) = O(\text{diam } T) = O(n)$
$M(\beta) = O(n)$

Synchronizer β

At node v:

wait until v is safe
wait until v receives SAFE message from all its children in tree
only then send SAFE message to parent in T
wait until PULSE received from parent
send PULSE to children
start PULSE

Time expensive: non-local convergecasts in each round!

Complexities per synchronous round:

$T(\beta) = O(\text{diam } T) = O(n)$
$M(\beta) = O(n)$

Message cheap: convergecasts along spanning tree edges only!

Synchronizer β

At node v:

wait until v is safe
wait until v receives SAFE message from all its children in tree
only then send SAFE message to parent in T
wait until PULSE received from parent
send PULSE to children
start PULSE

Time expensive: non-local convergecasts in each round!

Complexities per synchronous round:

$T(\beta) = O(\text{diam } T) = O(n)$
$M(\beta) = O(n)$

Message cheap: convergecasts along spanning tree edges only!

Plus initialization: leader election!

A Tradeoff:

Synchronizer α

Overhead per synchronous round:

$T(\alpha) = O(1)$

$M(\alpha) = O(m)$

Fast but many messages.

Slow but message efficient.

Synchronizer β

Complexities per synchronous round:

$T(\beta) = O(\text{diam } T) = O(n)$

$M(\beta) = O(n)$

Can we get the best of both worlds?

A Tradeoff:

Synchronizer α

Overhead per synchronous round:

$T(\alpha) = O(1)$

$M(\alpha) = O(m)$

Fast but many messages.

Not so bad in sparse graphs: m small!

Slow but message efficient.

Not so bad in low diameter graphs! Diam small.

Synchronizer β

Complexities per synchronous round:

$T(\beta) = O(\text{diam } T) = O(n)$

$M(\beta) = O(n)$

Can we get the best of both worlds?

A Tradeoff:

Synchronizer α

Overhead per synchronous round:

$T(\alpha) = O(1)$

$M(\alpha) = O(m)$

Fast but many messages.

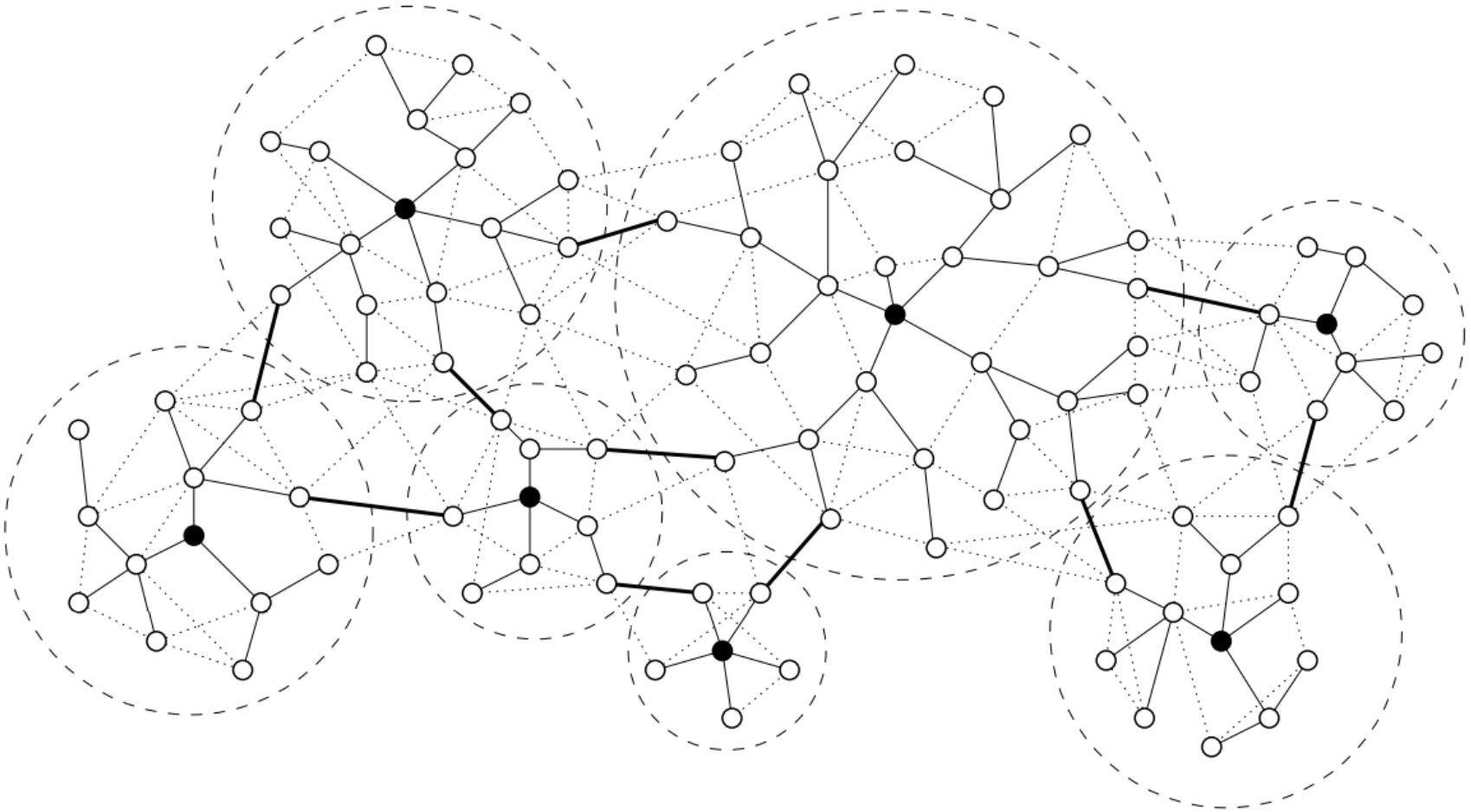
Not so bad in sparse graphs: m small!

Idea: partition the network into low-diameter clusters with sparse interconnections! Inside: can use **β** synchronizer (small diameter), across cluster can use **α** synchronizer (number of messages m small)!

Can we get the best of both worlds?

The Hybrid Synchronizer γ

Idea: Execute β intra- and α inter-cluster

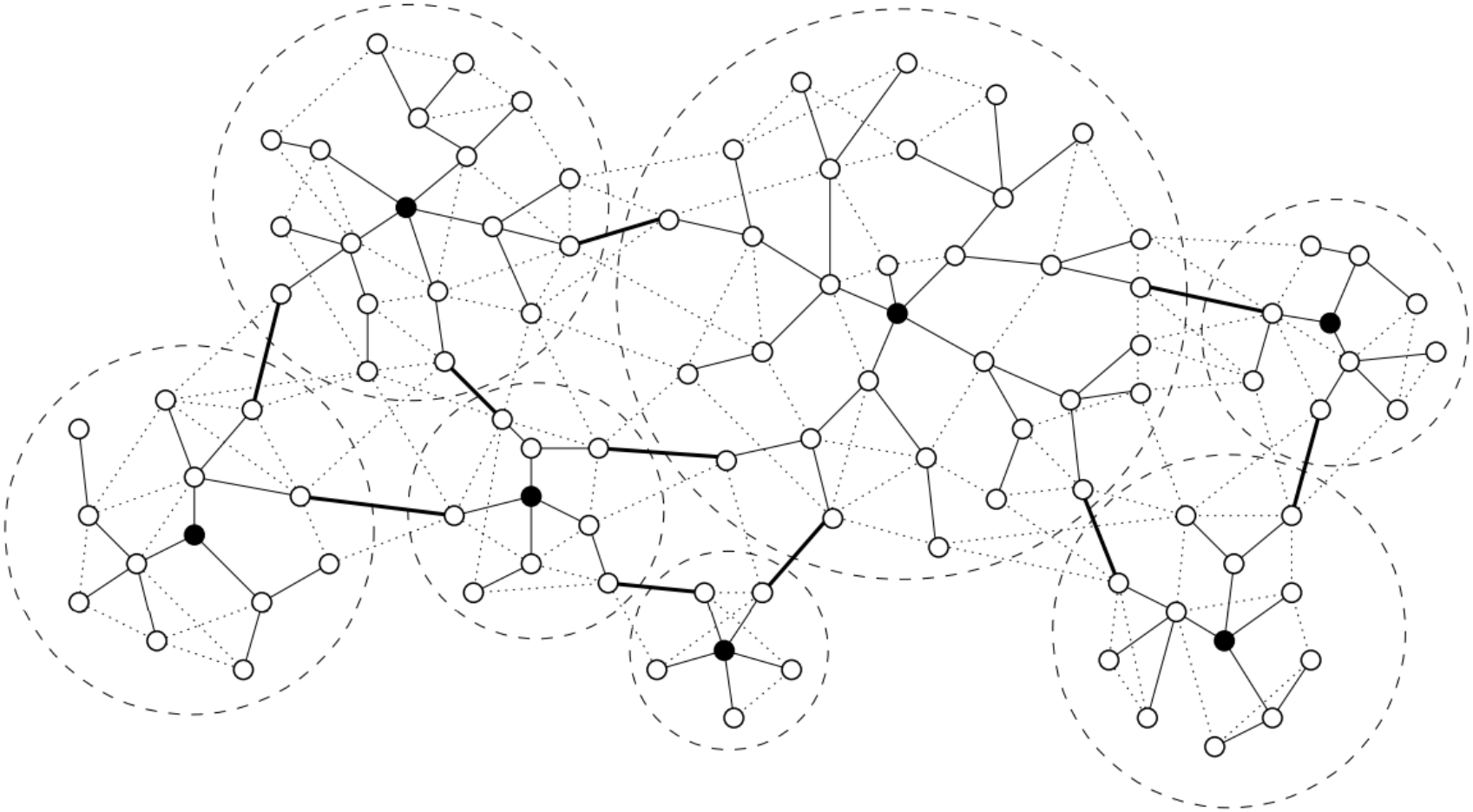


The … er γ

Small diameter: convergecast fast.

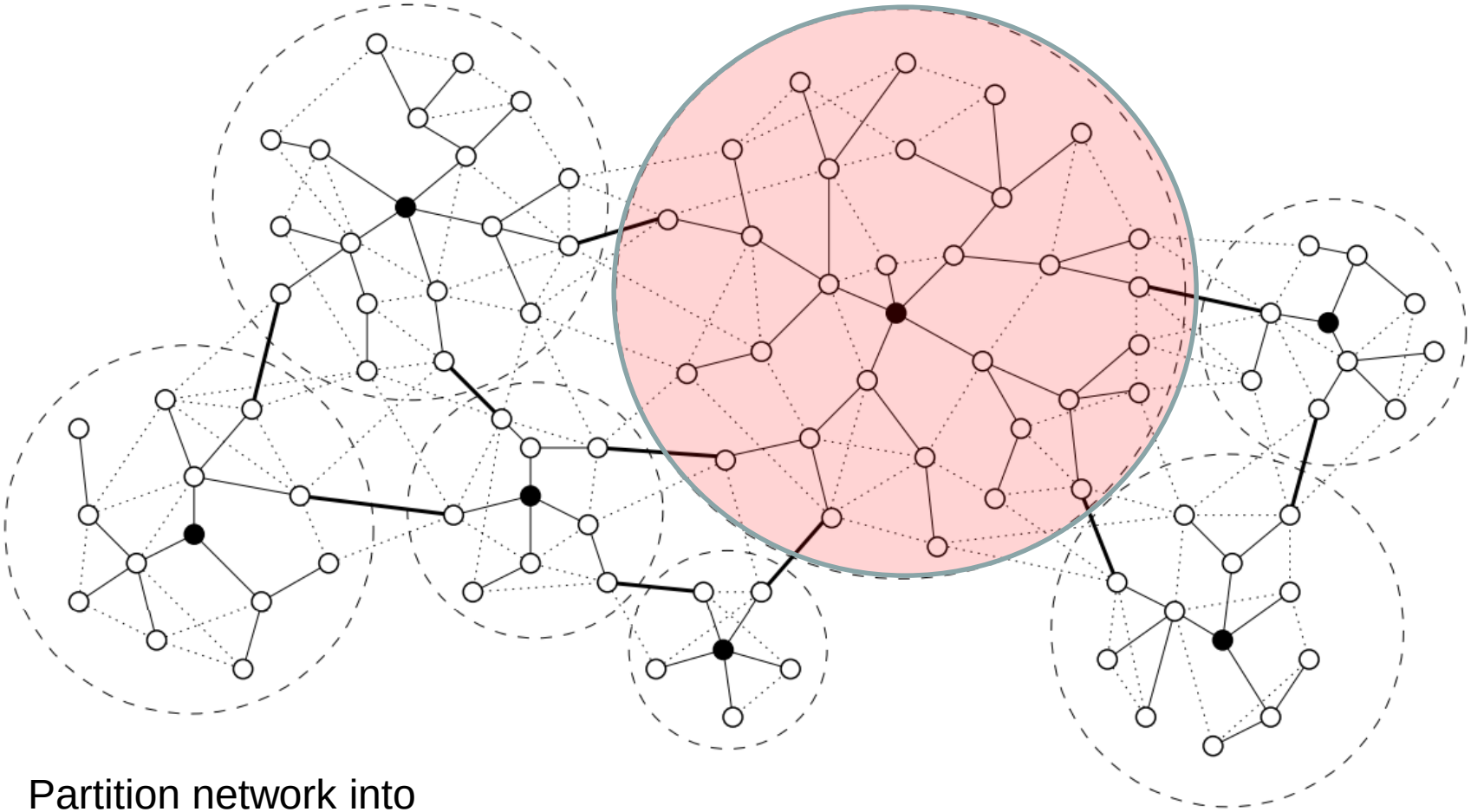
Number of edges m: small.

Idea: Execute β intra- and α inter-cluster



The Hybrid Synchronizer γ

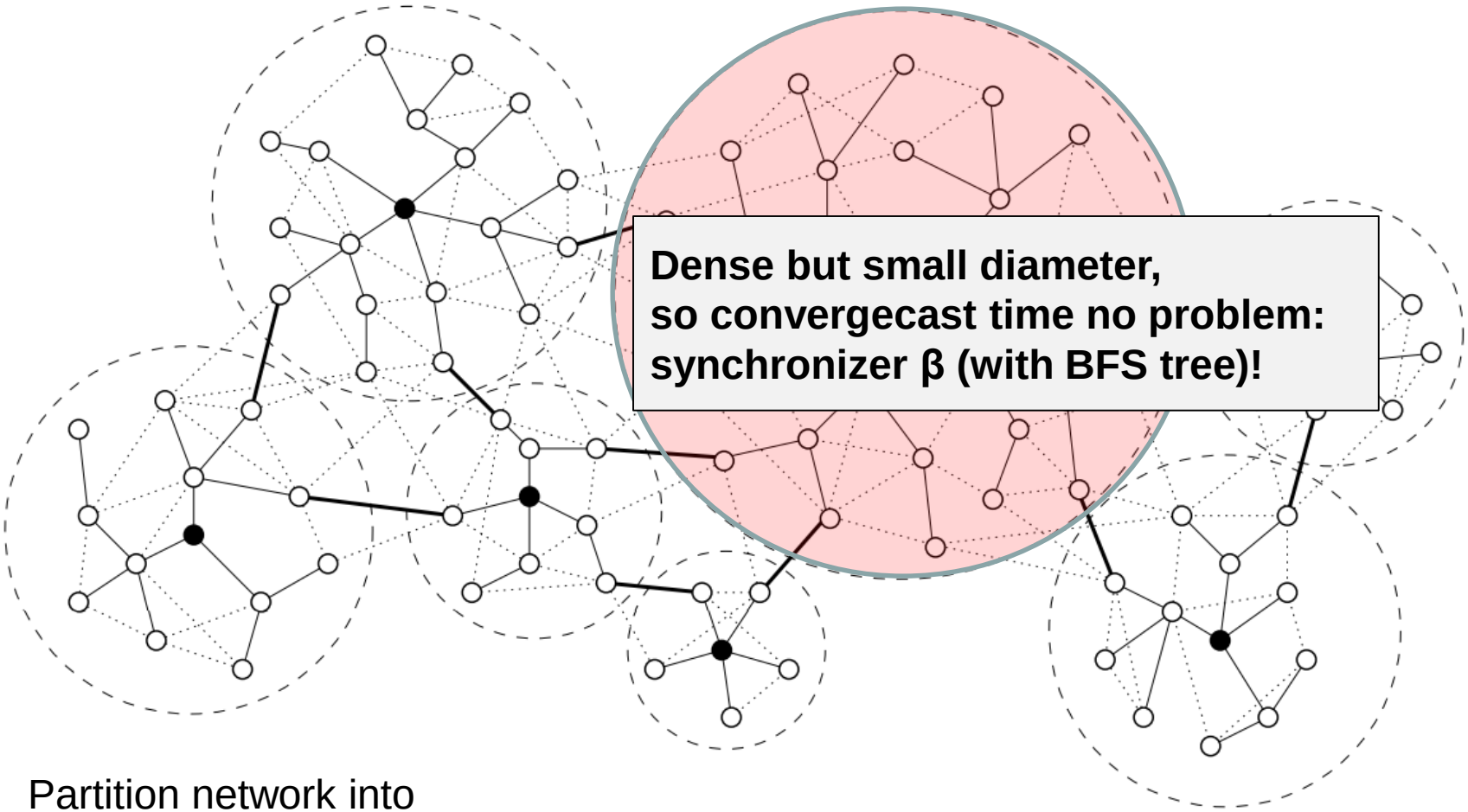
Idea:



Partition network into
small-diameter clusters.

The Hybrid Synchronizer γ

Idea:

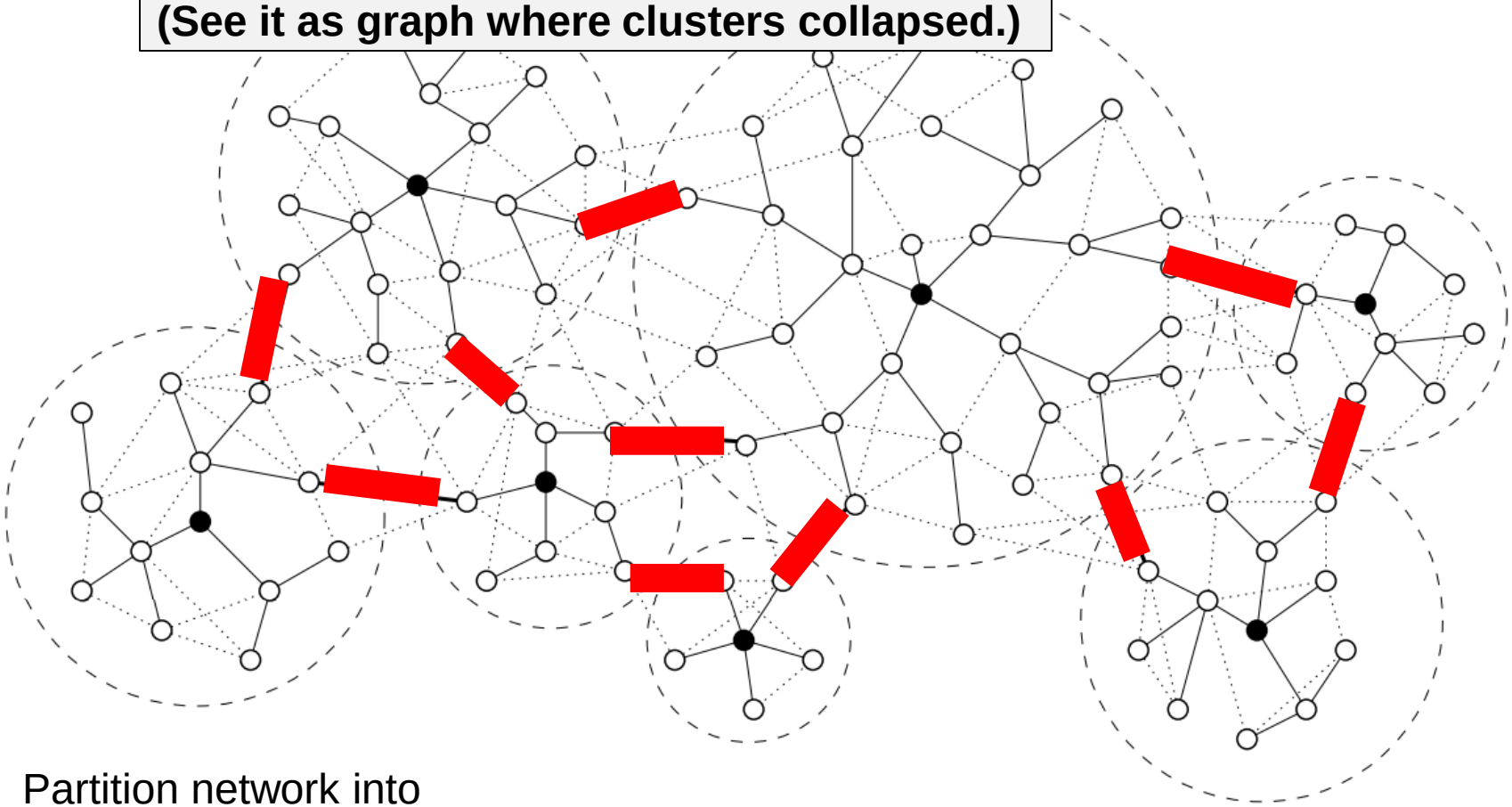


Partition network into
small-diameter clusters.

The Hybrid Synchronizer γ

Idea:

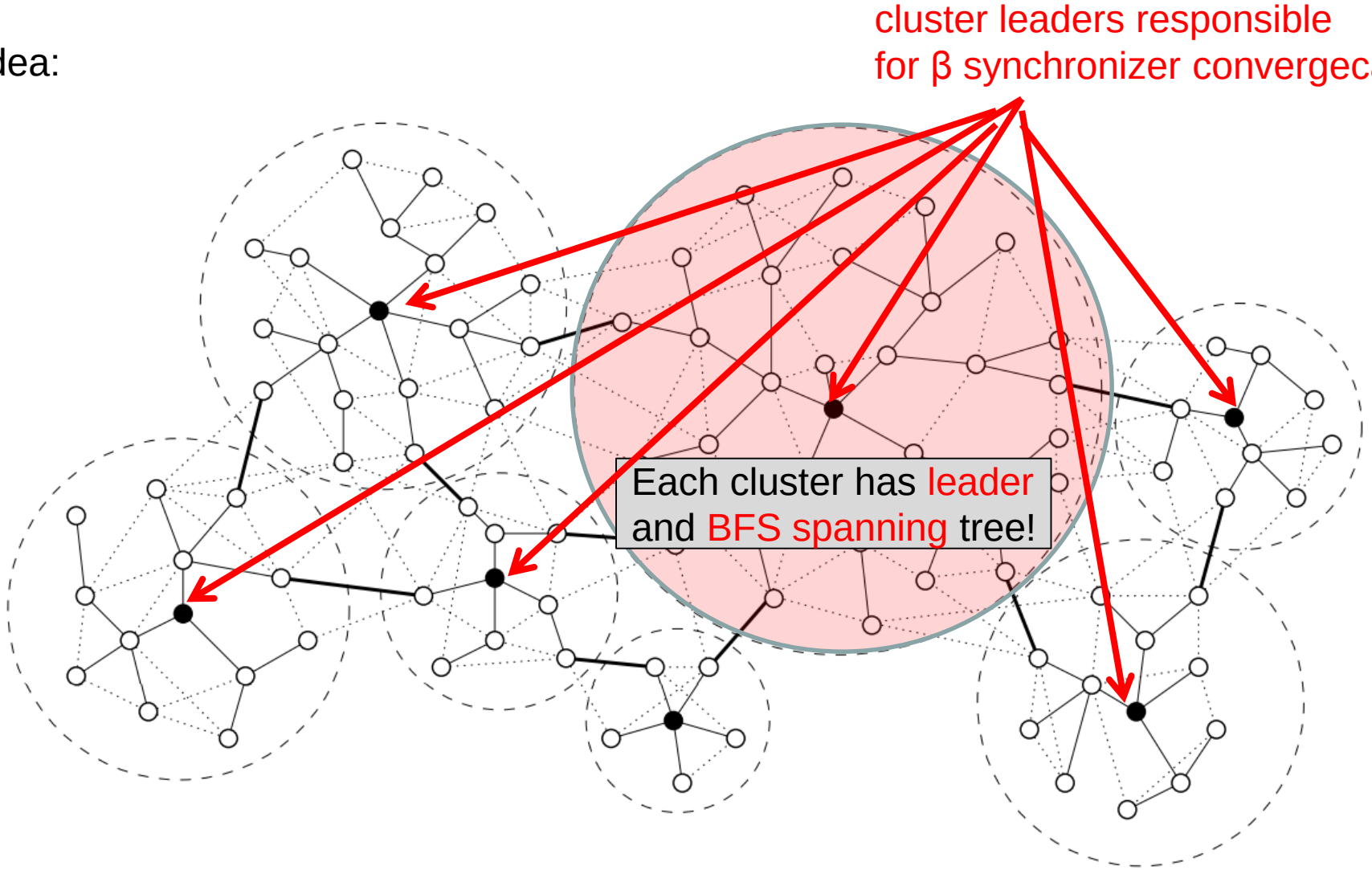
Between clusters, local synchronizer α!
Small number of edges.
(See it as graph where clusters collapsed.)



Partition network into
small-diameter clusters.

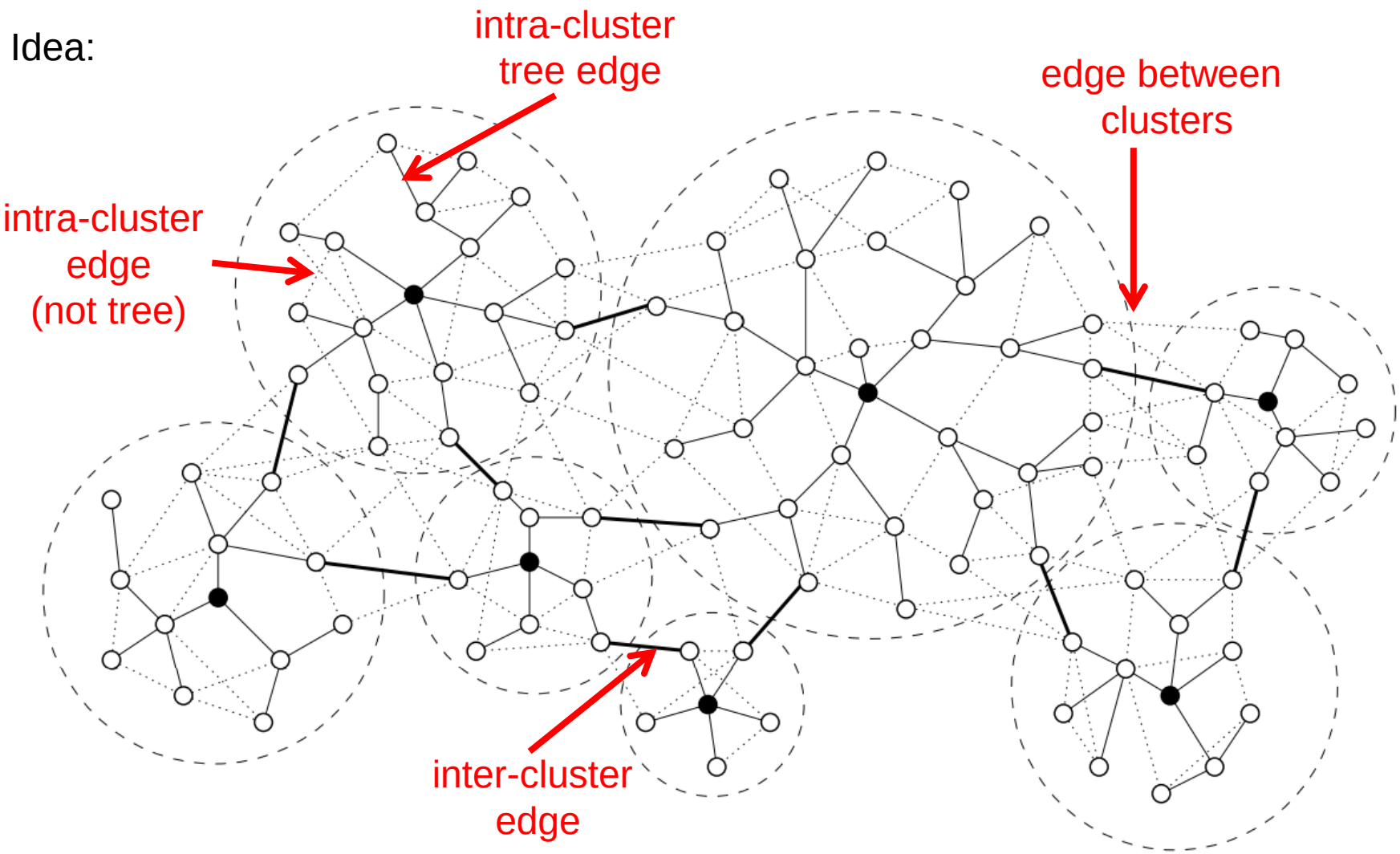
The Hybrid Synchronizer γ

Idea:



Edge Types in the Hybrid Synchronizer γ

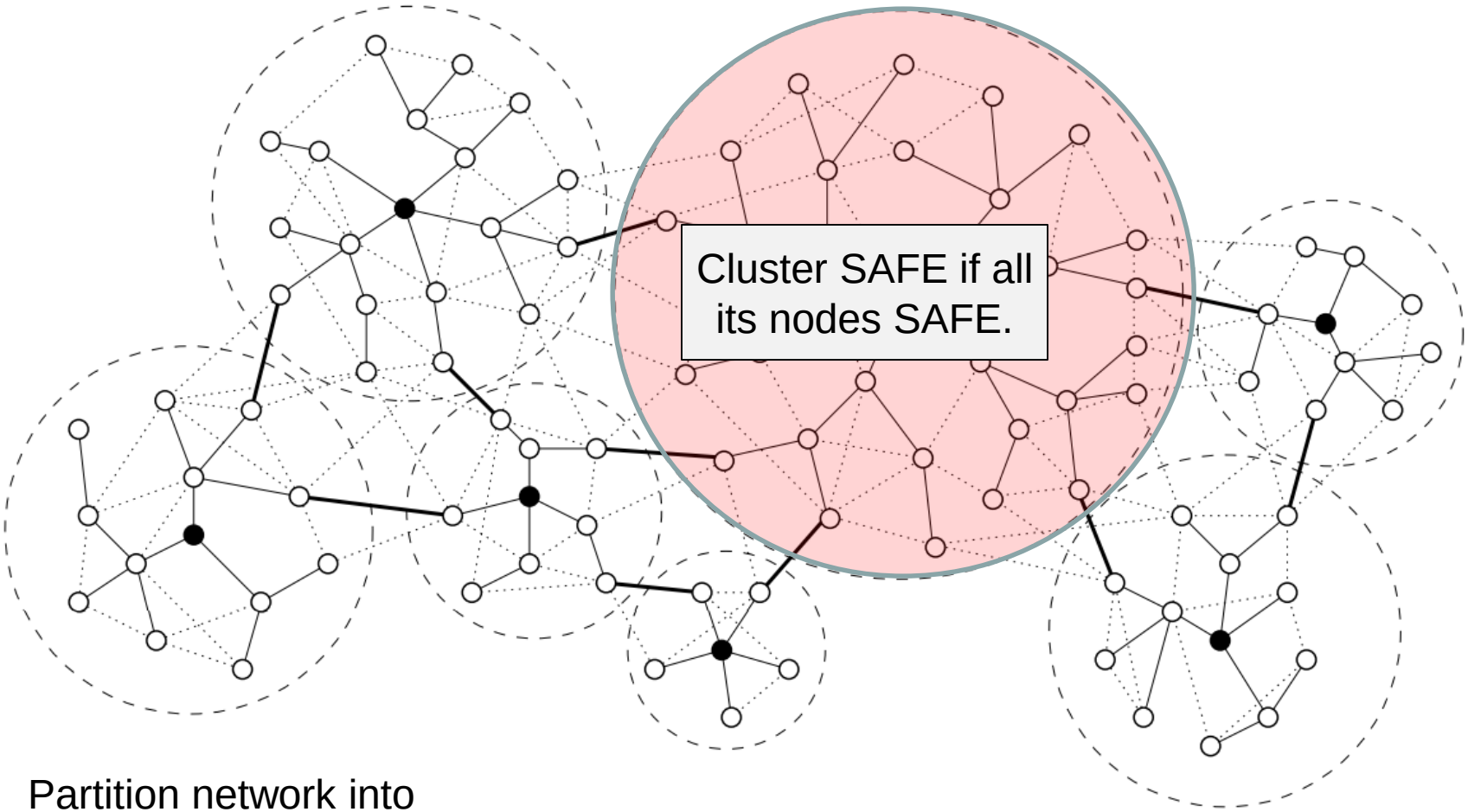
Idea:



The Hybrid Synchronizer γ

Idea: First make cluster safe, then make inter-cluster safe.

Idea:



Partition network into small-diameter clusters.

The Hybrid Synchronizer γ

Idea: First make cluster safe, then make inter-cluster safe.

Idea:

1. Phase 1: Apply Synchronizer **β** inside each cluster;
   when done inform leaders in neighbor clusters
2. Phase 2: Generate next pulse when neighbor clusters are SAFE (Synchronizer **α**)

Synchronizer γ

For node v:

wait until v is SAFE
wait until v receives SAFE from all children in intra-cluster tree
send SAFE to parent in tree
wait for CLUSTERSAFE message from parent
send CLUSTERSAFE to children
wait until NEIGHBORSAFE received from all incident
inter-cluster edges and children in intra-cluster
send NEIGHBORSAFE to parent
wait for PULSE and forward

Synchronizer γ

Let mc be number of inter-cluster edges and let k be the maximum cluster radius (max dist leaf to leader in BFS). Then:

$$T(\gamma) = O(k)$$
$$M(\gamma) = O(n+m_c)$$

Synchronizer γ

Let mc be number of inter-cluster edges and let k be the maximum cluster radius.
Then:

$T(\gamma) = O(k)$
$M(\gamma) = O(n+m_c)$

Dominant is the global **β** synchronizer: time at most the cluster radius k! O(1) time for **α** synchronizer.

Local synchronizer **α** for inter cluster at most m_c many messages. Intra-cluster along spanning tree at most n.

Synchronizer γ

Let mc be number of inter-cluster edges and let k be the maximum cluster radius.
Then:

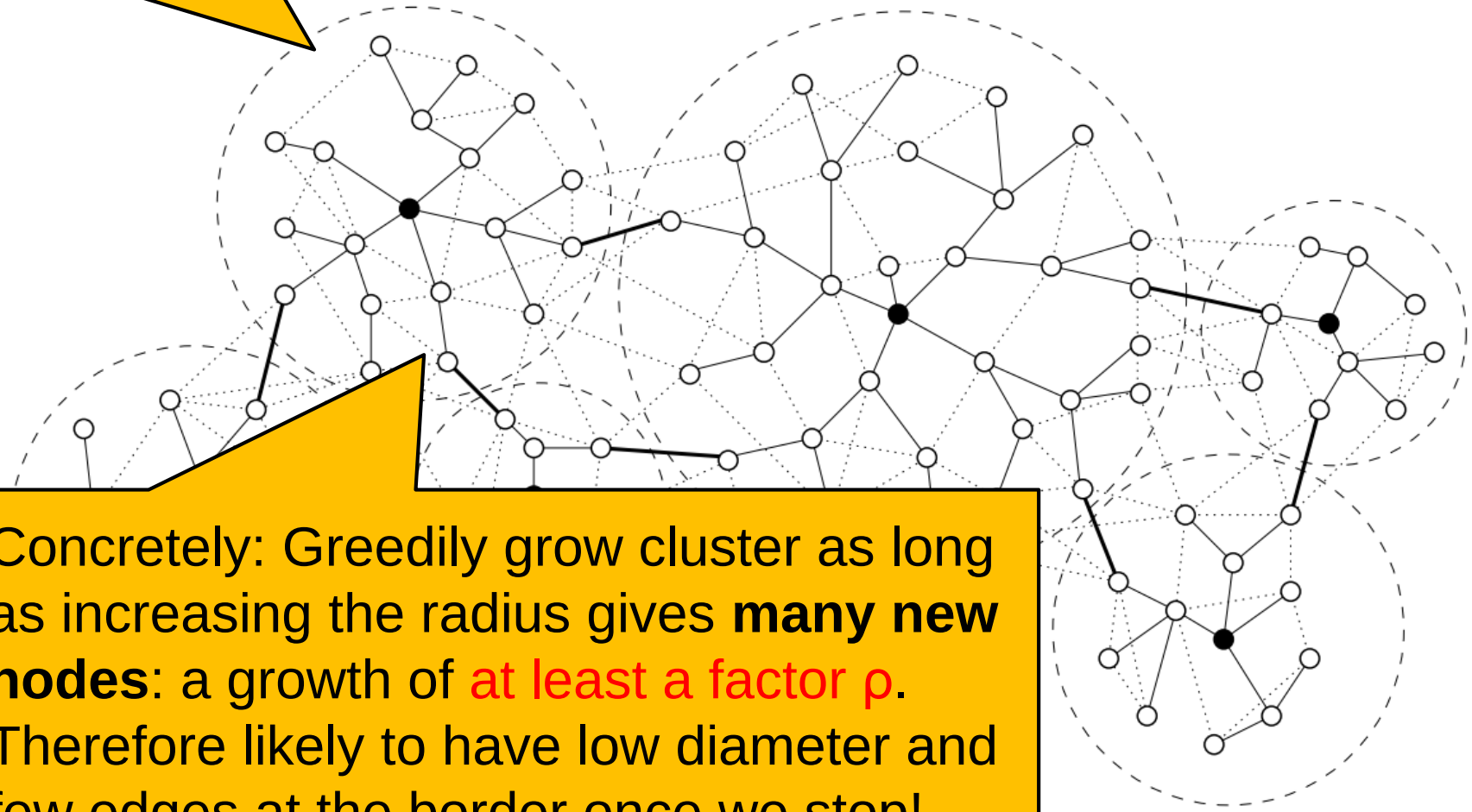
$T(\gamma) = O(k)$
$M(\gamma) = O(n+m_c)$

Dominant is the global **β** synchronizer: time at most the cluster radius k! O(1) time for **α** synchronizer.

Local synchronizer **α** for inter cluster at most m_c many messages. Intra-cluster along spanning tree at most n.

How to cluster the network so that m_c and k are minimal?

Idea: grow clusters one by one!



Concretely: Greedily grow cluster as long as increasing the radius gives **many new nodes**: a growth of at least a factor ρ. Therefore likely to have low diameter and few edges at the border once we stop!

Network Partition Algorithm

Idea:
1. Construct one cluster after another; start cluster at random non-covered node
2. Grow as long as “growth significant” (factor ρ)

Define: B(v,r) = Ball of radius r around v

Cluster Construction

```
while unprocessed nodes:
        select arbitrary unprocessed node v
        r:=0
        while |B(v,r+1)| > ρ * |B(v,r)| do
                r := r+1
        end while
        makeCluster(B(v,r))
end while
```

Partition Properties

The resulting network partition:

(1) consists of clusters of radius at most $\log_\rho n$

(2) at most $(\rho - 1)*n$ intercluster edges

Radius grows only if cluster size increases by factor ρ.
As there are at most n nodes, this can happen at most $\log_\rho n$ times.

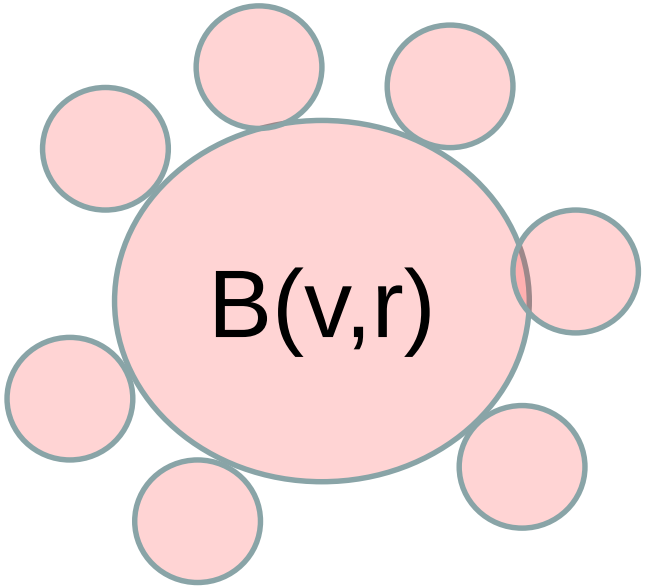
Partition Properties

The resulting network partition:

(1) consists of clusters of radius at most $\log_\rho n$

(2) at most $(\rho - 1)*n$ intercluster edges

We know that for ball B:
$|B(v,r+1)| \leq \rho * |B(v,r)|$



Define $|C| := |B(v,r)|$. The size of the "border of the cluster" is at most
$| B(v,r+1) \setminus B(v,r) | \leq \rho * |C| - |C|$.

Summing over all clusters (n nodes in total, in worst case each one is a cluster): at most $\sum (\rho - 1)* |C| = (\rho - 1)* \sum |C| = (\rho - 1)* n$ inter-cluster edges

Quality of Partition

Partition Properties

The resulting network partition:

(1) consists of clusters of radius at most $\log_\rho n$

(2) at most $(\rho - 1)*n$ intercluster edges

Asymptotically optimal tradeoff!

Partition Properties

The resulting network partition:

(1) consists of clusters of radius at most $\log_\rho n$

(2) at most $(\rho - 1)*n$ intercluster edges

Asymptotically optimal tradeoff!

Example: $\rho = 2$

log(n) time synchronization overhead,
but only O(n) inter-cluster edges (messages)

Example: $\rho = n^{1/k}$

k time overhead, $O(n^{1+1/k})$ inter-cluster edges

End of Lecture