Distributed Synchronization

«A man with one clock knows what time it is – a man with two is never sure.»
The LOCAL Model: A Synchronous Model

Synchronous LOCAL algorithms simple to design and reason about:

Send...

... receive...

... compute.
Synchronous LOCAL algorithms simple to design and reason about:

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

Synchronous 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**.

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

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

**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.
Definitions

**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.
Definitions

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.

How to implement a synchronizer efficiently?

Efficient = Runtime and Message Complexity „almost like in the LOCAL model“. Don‘t want to send messages just for the sake of synchronization.
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, $T_{\text{init}}(S)$ and $M_{\text{init}}(S)$ to set up synchronizer. Then:

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

Some setup costs maybe.

Each round: additionally costs time $T(S)$ and messages $M(S)$. 
Definitions

**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!

$v$ not safe wrt 10

$v$ safe wrt 10
Definitions

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**.

Note: at this point, $v$ cannot know that it is actually safe. Need ACKs to detect!

$v$ not safe wrt 10

$v$ safe wrt 10
Definitions

**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*.

- Note: at this point, $v$ cannot know that it is actually safe.
- Need ACKs to detect!
- 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.

$v$ safe wrt 10
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.

Note: at this point, $v$ cannot know that it is actually safe. Need ACKs to detect!

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.
The Local Synchronizer $\alpha$

**Synchronizer $\alpha$**

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

Overhead?
The Local Synchronizer $\alpha$

**Synchronizer $\alpha$**

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

**Good:** just $O(1)$ interactions with neighbors: no initialization and local!

**Overhead per synchronous round:**
- $T(\alpha) = O(1)$
- $M(\alpha) = O(m)$
The Local Synchronizer $\alpha$

**Synchronizer $\alpha$**

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

**Good:** just $O(1)$ interactions with neighbors: no initialization and local!

**Not so good:** *Every edge sees 6 messages* (PULSE, SAFE, ACK in both directions), in each round!

Overhead per synchronous round:
- $T(\alpha) = O(1)$
- $M(\alpha) = O(m)$
The Local Synchronizer $\alpha$

**Synchronizer $\alpha$**

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

**Overhead per synchronous round:**
- $T(\alpha) = O(1)$
- $M(\alpha) = O(m)$

**Good:** just $O(1)$ interactions with neighbors: no initialization and local!

**Not so good:** Every edge sees 6 messages (PULSE, SAFE, ACK in both directions), in each round!

The global synchronizer $\beta$ provides the opposite tradeoff!
The Global Synchronizer $\beta$

Idea: make global rounds: leader coordinates global phases. Like in our BFS algorithm.
The Global Synchronizer $\beta$

Aggregate: round $i$ done? All children safe?
The Global Synchronizer $\beta$

Along spanning tree: $O(n)$ messages.

Aggregate: round $i$ done? All children safe?
The Global Synchronizer $\beta$

Okay, then PULSE! Start next round!
Synchronizer $\beta$

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 $\beta$

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 $\beta$

Complexities per synchronous round:
- $T(\beta) = O(\text{diam } T) = O(n)$
- $M(\beta) = O(n)$
Synchronizer $\beta$

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

Expensive: non-local convergecarts in each round!

Complexities per synchronous round:
- $T(\beta) = O(\text{diam } T) = O(n)$
- $M(\beta) = O(n)$

Cheap: convergecarts along spanning tree edges only!

Plus initialization: leader election!
A Tradeoff:

**Synchronizer $\alpha$**

Overhead per synchronous round:
- $T(\alpha) = O(1)$
- $M(\alpha) = O(m)$

Fast but many messages.

**Synchronizer $\beta$**

Complexities per synchronous round:
- $T(\beta) = O(\text{diam } T) = O(n)$
- $M(\beta) = O(n)$

Slow but message efficient.

Can we get the best of both worlds?
A Tradeoff:

**Synchronizer \( \alpha \)**

Overhead per synchronous round:
- \( T(\alpha) = O(1) \)
- \( M(\alpha) = O(m) \)

Fast but many messages.

Slow but message efficient.

Not so bad in sparse graphs!

**Synchronizer \( \beta \)**

Complexities per synchronous round:
- \( T(\beta) = O(\text{diam } T) = O(n) \)
- \( M(\beta) = O(n) \)

Not so bad in low diameter graphs!

Can we get the best of both worlds?
A Tradeoff:

**Synchronizer \( \alpha \)**

Overhead per synchronous round:

\[ T(\alpha) = O(1) \]
\[ M(\alpha) = O(m) \]

- Fast but many messages.
- Not so bad in sparse graphs!

Idea: partition the network into low-diameter clusters with sparse interconnections! Inside: can use \( \beta \) synchronizer (runtime not critical), across cluster can use \( \alpha \) synchronizer (messages not critical)!

Can we get the best of both worlds?
The Hybrid Synchronizer $\gamma$

Idea: Execute $\beta$ intra- and $\alpha$ inter-cluster
The Hybrid Synchronizer $\gamma$

Idea:

Partition network into small-diameter clusters.
The Hybrid Synchronizer $\gamma$

Idea:

Partition network into small-diameter clusters.

Dense but small diameter, so diameter time no problem: synchronizer $\beta$ (with BFS tree)!
The Hybrid Synchronizer $\gamma$

Idea:

Between clusters, local synchronizer $\alpha$!
(See it as graph where clusters collapsed.)

Partition network into small-diameter clusters.
The Hybrid Synchronizer γ

Idea:

Each cluster has leader and BFS spanning tree!

Cluster leaders responsible for β synchronizer convergecast
Edge Types in the Hybrid Synchronizer γ

Idea:

- **Intra-cluster tree edge**
- **Intra-cluster edge** (not tree)
- **Inter-cluster edge**
- **Edge between clusters**
The Hybrid Synchronizer $\gamma$

Idea:

Partition network into small-diameter clusters.

Cluster safe if all its nodes safe.

Idea: First make cluster safe, then make inter-cluster safe.
The Hybrid Synchronizer $\gamma$

Idea:
1. Phase 1: Apply Synchronizer $\beta$ in each cluster; when done inform leaders in neighbor clusters
2. Phase 2: Generate next pulse when neighbor clusters are safe (Synchronizer $\alpha$)

Synchronizer $\gamma$

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

Idea: First make cluster safe, then make inter-cluster safe.
Synchronizer $\gamma$

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

$T(\gamma) = O(k)$

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

Global synchronizer time at most the cluster radius!

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

**Synchronizer $\gamma$**

Let $m_c$ be number of inter-cluster edges and let $k$ be the maximum cluster radius (max dist leaf to leader).

Then:

$T(\gamma) = O(k)$

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

Global synchronizer time at most the cluster radius!

Local synchronizer for inter cluster at most many $m_c$ 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!

Greedily grow cluster as long as increasing the radius gives many new nodes: a growth of at least a factor \( \rho \). So likely to have low diameter and little edges at edge once we stop!
Cluster Construction

while unprocessed nodes:
    select arbitrary unprocessed node $v$
    $r:=0$
    while $|B(v,r+1)| > \rho \times |B(v,r)|$ do
        $r := r+1$
    end while
    makeCluster($B(v,r)$)
end while

Idea:
1. Construct one cluster after another; start cluster at random non-covered node
2. Grow as long as “growth significant” (factor $\rho$)
Define: $B(v,r)$ = Ball of radius $r$ around $v$
Quality of Partition

Partition Properties

The resulting network partition:

1. consists of clusters of radius at most \( \log_\rho n \)
2. at most \((\rho - 1) \times n\) intercluster edges
Radius grows only if cluster size increases by factor $\rho$. 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
Radius grows only if cluster size increases by factor \( \rho \). 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 \( B \):

\[ |B(v, r+1)| \leq \rho \cdot |B(v, r)| \]

So the size of the “border of the cluster” is at most

\[ |B(v, r+1) \setminus B(v, r)| \leq \rho \cdot |C| - |C| \]

Summing over all clusters (\( n \) nodes in total, in worst case each one is a cluster):

\[ \sum (\rho - 1) \cdot |C| = (\rho - 1) \cdot \sum |C| = (\rho - 1) \cdot n \]
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!
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!

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