Hierarchical Leader Election Algorithm With Remoteness Constraint

Mohamed Tbarka

November 15, 2019

ENSIAS

Outline

ENSIAS

Outline

What is the problem of leader election ?

Leader election is an important primitive for distributed computing, useful as a sub-routine for any application that requires the selection of a unique processor among multiple candidate processors (video conferencing, multi-player games, ...).

What is distributed computing ?

Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance.

State of art

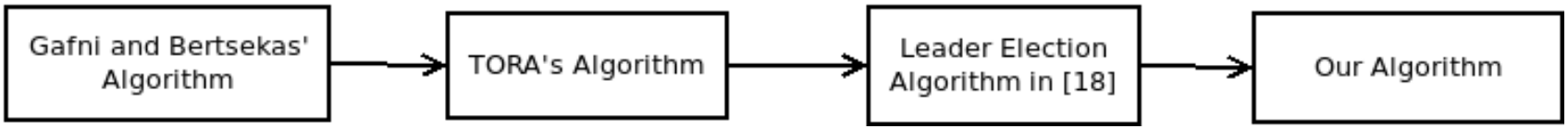


Figure: State Of Art

Outline

System Model

- The system is consisting of a set P of computing nodes and a set χ of directed communication channels from one node to another node.

System Model

- The system is consisting of a set P of computing nodes and a set χ of directed communication channels from one node to another node.
- The whole system as a set of (infinite) state machines that interact through shared events.

Asynchronous Dynamic Links' Model

The state of $Channel(u, v)$, which models the communication channel from node u to node v, consists of:

- a $status_{uv}$ variable;

Asynchronous Dynamic Links' Model

The state of $Channel(u, v)$, which models the communication channel from node u to node v, consists of:

- a $status_{uv}$ variable;
- and a queue $mqueue_{uv}$ of messages.

Configurations & Executions

- The notion of configuration is used to capture an instantaneous snapshot of the state of the entire system.

Configurations & Executions

- The notion of configuration is used to capture an instantaneous snapshot of the state of the entire system.
- A configuration is a vector of node states, one for each node in P, and a vector of channel states, one for each channel in χ.

Configurations & Executions

In an initial configuration:

- each node is in an initial state (according to its algorithm),

Configurations & Executions

In an initial configuration:

- each node is in an initial state (according to its algorithm),
- for each channel $Channel(u, v)$, $mqueue_{uv}$ is empty, and

Configurations & Executions

In an initial configuration:

- each node is in an initial state (according to its algorithm),
- for each channel $Channel(u, v)$, $mqueue_{uv}$ is empty, and
- for all nodes u and v, $status_{uv} = status_{vu}$ (i.e., either both channels between u and v are up, or both are down).

Configurations & Executions

An execution is an infinite sequence $C_0, e_1, C_1, e_2, C_2, \ldots$ of alternating configurations and events, starting with an initial configuration and, if finite, ending with a configuration.

Problem Definition

Each node u in the system should have after the last topology change :

- a local variable lid_u to hold the id of the supreme leader;

Problem Definition

Each node u in the system should have after the last topology change :

- a local variable lid_u to hold the *id* of the supreme leader;
- another local variable $slid_u$ to hold the identifier of the sub-leader whose remoteness towards u obeys the constraint.

Outline

Informal description

After a leader is gone, the algorithm consists on three waves:

Informal description

After a leader is gone, the algorithm consists on three waves:

- First wave : initiated by one of the lost leader's neighbors looking for it;

Informal description

After a leader is gone, the algorithm consists on three waves:

- First wave : initiated by one of the lost leader's neighbors looking for it;
- Second wave : initiated by the node located at the edge of the network if the search has hit a dead-end;

Informal description

After a leader is gone, the algorithm consists on three waves:

- First wave : initiated by one of the lost leader's neighbors looking for it;
- Second wave : initiated by the node located at the edge of the network if the search has hit a dead-end;
- Third wave : initiated by the same node which initiated the first wave updating the other nodes' heights and constructing the spanning tree.

Nodes, Neighbors and Heights

- When a node u gets a *ChannelUp* event for the channel from u to v, it puts v in a local set variable called $forming_u$.

Nodes, Neighbors and Heights

- When a node u gets a *ChannelUp* event for the channel from u to v, it puts v in a local set variable called $forming_u$.
- When u subsequently receives a message from v, it moves v from its $forming_u$ set to a local set variable called N_u (N for neighbor).

Nodes, Neighbors and Heights

- When a node u gets a *ChannelUp* event for the channel from u to v, it puts v in a local set variable called $forming_u$.
- When u subsequently receives a message from v, it moves v from its $forming_u$ set to a local set variable called N_u (N for neighbor).
- If u gets a message from a node which is neither in its forming set, nor in N_u, it ignores that message.

Nodes, Neighbors and Heights

- When a node u gets a *ChannelUp* event for the channel from u to v, it puts v in a local set variable called $forming_u$.
- When u subsequently receives a message from v, it moves v from its $forming_u$ set to a local set variable called N_u (N for neighbor).
- If u gets a message from a node which is neither in its forming set, nor in N_u, it ignores that message.
- And when u gets a *ChannelDown* event for the channel from u to v, it removes v from $forming_u$ or N_u , as appropriate.

Nodes, Neighbors and Heights

- Nodes assign virtual directions to their links using variables called heights.

Nodes, Neighbors and Heights

- Nodes assign virtual directions to their links using variables called heights.
- For each link (u, v) of node u, u considers the link as incoming if the height that u has recorded for v is larger than $u's$ own height; otherwise u considers the link as outgoing (directed from u to v).

Nodes, Neighbors and Heights

- Nodes assign virtual directions to their links using variables called heights.
- For each link (u, v) of node u, u considers the link as incoming if the height that u has recorded for v is larger than $u's$ own height; otherwise u considers the link as outgoing (directed from u to v).
- Heights are compared using lexicographic ordering.

Nodes, Neighbors and Heights

The height for each node is a 7-tuple of integers $((\tau, oid, r), \delta, (nlts, lid), id)$ where :

- τ, a non-negative timestamp which is either 0 or the value of the causal clock time when the current search for an alternate path to the leader was initiated.

Nodes, Neighbors and Heights

The height for each node is a 7-tuple of integers $((\tau, oid, r), \delta, (nlts, lid), id)$ where :

- τ, a non-negative timestamp which is either 0 or the value of the causal clock time when the current search for an alternate path to the leader was initiated.
- *oid*, is a non-negative value that is either 0 or the id of the node that started the current search (we assume node ids are positive integers).

Nodes, Neighbors and Heights

The height for each node is a 7-tuple of integers $((\tau, oid, r), \delta, (nlts, lid), id)$ where :

- τ, a non-negative timestamp which is either 0 or the value of the causal clock time when the current search for an alternate path to the leader was initiated.
- oid, is a non-negative value that is either 0 or the id of the node that started the current search (we assume node ids are positive integers).
- r, a bit that is set to 0 when the current search is initiated and set to 1 when the current search hits a dead end.

Nodes, Neighbors and Heights

- δ, an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components.

Nodes, Neighbors and Heights

- δ, an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components.
- *nlts*, a non-positive timestamp whose absolute value is the causal clock time when the current leader was elected.

Nodes, Neighbors and Heights

- δ, an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components.
- *nlts*, a non-positive timestamp whose absolute value is the causal clock time when the current leader was elected.
- *lid*, the *id* of the current leader.

Nodes, Neighbors and Heights

- δ, an integer that is set to ensure that links are directed appropriately to neighbors with the same first three components.
- *nlts*, a non-positive timestamp whose absolute value is the causal clock time when the current leader was elected.
- *lid*, the *id* of the current leader.
- *id*, the node's unique ID.

Initial State

– $forming_u$ is empty,
N_u equals the set of neighbors of u in G^{init}_{chan}

Initial State

- $forming_u$ is empty,
  N_u equals the set of neighbors of u in G^{init}_{chan}
- $height_u[u] = (0, 0, 0, \delta_u, 0, l, u)$ where l is the id of a fixed node in $u's$ connected component in G^{init}_{chan} (the current leader), and δ_u equals the distance from u to l in G^{init}_{chan},

Initial State

- $forming_u$ is empty,
  N_u equals the set of neighbors of u in G^{init}_{chan}
- $height_u[u] = (0, 0, 0, \delta_u, 0, l, u)$ where l is the id of a fixed node in $u's$ connected component in G^{init}_{chan} (the current leader), and δ_u equals the distance from u to l in G^{init}_{chan},
- for each v in N_u, $height_u[v] = height_v[v]$ (i.e., u has accurate information about v's height), and

Initial State

- $forming_u$ is empty,
  N_u equals the set of neighbors of u in G^{init}_{chan}
- $height_u[u] = (0, 0, 0, \delta_u, 0, l, u)$ where l is the id of a fixed node in $u's$ connected component in G^{init}_{chan} (the current leader), and δ_u equals the distance from u to l in G^{init}_{chan},
- for each v in N_u, $height_u[v] = height_v[v]$ (i.e., u has accurate information about v's height), and
- $\mathcal{T}_u$ is initialized properly with respect to the definition of causal clocks.

Algorithm 1 When $ChannelDown_{uv}$ event occurs:

1: $N := N \setminus v$
2: $forming := forming \setminus v$
3: **if** $N = \emptyset$ **then**
4: $\quad ELECTSELF$
5: $\quad$ send $Update(height[u]$ *to all* $w \in forming)$
6: **else if** $SINK$ **then**
7: $\quad STARTNEWREFLEVEL$
8: $\quad$ send Update$(height[u])$ to all $w \in (N \cup forming))$
9: **else if** $id^v = pred^u$ **and** $id^v = slid^u$ **then**
10: $\quad pred^u = \min\{k \mid k \in N^u \text{ and } \delta^k = \delta^u\}$
11: $\quad slid^u = \min\{k \mid k \in N^u \text{ and } \delta^k = \delta^u\}$
12: **else if** $id^v = pred^u$ **and** $id^v \neq slid^u$ **then**
13: $\quad pred^u = \min\{k \mid k \in N_u \text{ and } \delta^k = \delta^u\}$
14: $\quad slid^u = slid^{\min\{k \mid k \in N_u \text{ and } \delta^k = \delta^u\}}$
15: **end if**

Algorithm 2 When $ChannelUp_{uv}$ event occurs:

1: $forming := forming \cup v$
2: send Update$(height[u])$ to v

Figure: Code triggered by Topology Changes

Algorithm 3 When node u receives $Update(h)$ from node $v \in forming \cup N$:

1: $height[v] := h$ ▷ if v is in neither forming nor N, message is ignored
2: $forming := forming \setminus v$
3: $N := N \cup v$
4: $myOldHeight := height[u]$
5: **if** $(nlts_u, lid_u) = (nlts_v, lid_v)$ **then** ▷ leader pair are the same
6: **if** $SINK$ **then**
7: **if** $(\exists(\tau, oid, r) \mid (\tau_w, oid_w, r_w) = (\tau, oid, r)\ \forall\ w \in N)$ **then**
8: **if** $(\tau > 0)$ *and* $(r = 0)$ **then**
9: $REFLECTREFLEVEL$
10: **else if** $(\tau > 0)$ *and* $(r = 1)$ *and* $(oid = u)$ **then**
11: $ELECTSELF$
12: **else** ▷ $(\tau = 0)$ *or* $(\tau > 0$ *and* $r = 1$ *and* $oid \neq u)$
13: $STARTNEWREFLEVEL$
14: **end if**
15: **else** ▷ neighbors have different ref levels
16: $PROPAGATELARGESTREFLEVEL$
17: **end if**
18: **end if** ▷ else not sink, do nothing
19: **else** ▷ leader pairs are different
20: $ADOPTLPIFPRIORITY(v)$
21: **end if**
22: **if** $myOldHeight \neq height[u]$ **then**
23: send Update($height[u]$) to all $w \in (N \cup forming)$
24: **end if**

Figure: Code triggered by Update Message

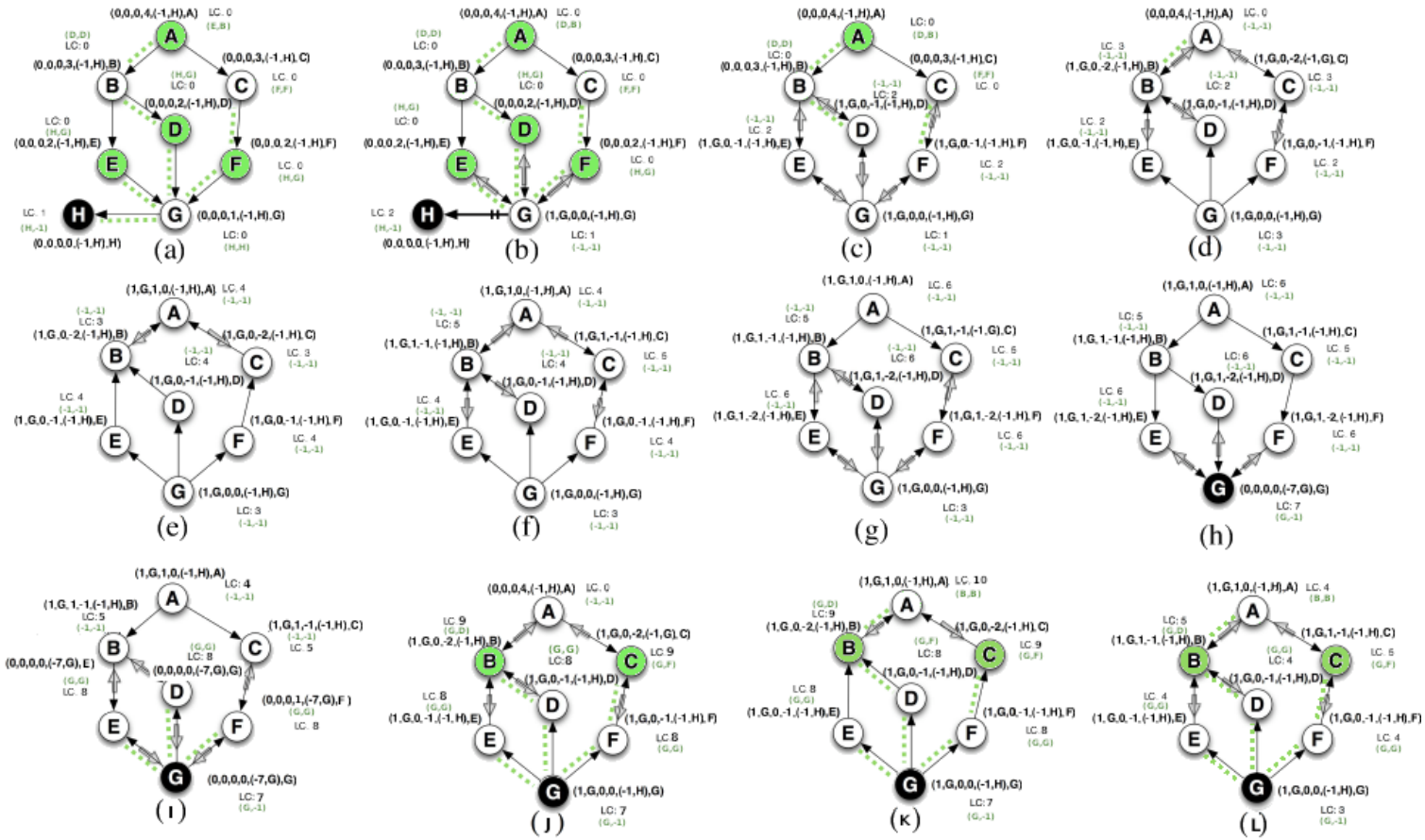


Figure: Sample Execution

Algorithm 4 *ELECTSELF*

1: $height[u] := (0, 0, 0, 0, -\mathcal{T}_u, u, u)$
2: $SLP_u := (-1, -1)$

Algorithm 5 *REFLECTREFLEVEL*

1: $height[u] := (\tau, oid, 1, 0, nlts^u, lid^u, u)$
2: $SLP_u := (-1, -1)$

Algorithm 6 *PROPAGATELARGESTREFLEVEL*

1: $(\tau^u, oid^u, r^u) := max\{(\tau^k, oid^k, r^k) \mid k \in N\}$
2: $\delta^u := min\{\delta^k \mid k \in N \text{ and } (\tau^u, oid^u, r^u) = (\tau^k, oid^k, r^k)\} - 1$
3: $SLP_u := (-1, -1)$

Algorithm 7 *STARTNEWREFLEVEL*

1: $height[u] := (\tau, oid, 1, 0, nlts^u, lid^u, u)$
2: $SLP_u := (-1, -1)$

Algorithm 8 *ADOPTLPIFPRIORITY*(v)

1: **if** $nlts^v < nlts^u$ **or** $nlts^v = nlts^u$ **and** $lid^v < lid^u$ **or** $nlts^v = nlts^u$ **and** $lid^v = lid^u$ **and** $id^v < pred^u$ **then**
2: $height[u] := (\tau^v, oid^v, r^v, \delta^v + 1, nlts^v, lid^v, u)$
3: **if** $(\delta^v \mod D \neq 0)$ **then**
4: $SLP_u = (slid^v, id^v)$
5: **else**
6: $SLP_u = (id^v, id^v)$
7: **end if**
8: **end if**

Figure: Subroutines

Outline

How the proof is dealt with ?

The proof uses a number of invariants, denoted as “Properties”, which are shown to hold in every configuration of every execution; each one is proved (separately) by induction on the configurations occurring in an execution.

Bounding the Number of Elections

We bound, in Lemma 3, the number of elections that can occur after the last topology change; this result relies on the fact, shown in Lemma 2, that once a node u adopts a leader that was elected after the last topology change, u never becomes a sink again.

Bounding the Number of Elections

We bound, in Lemma 4, the number of new reference levels that are started after the last topology change; the proof of this result relies on several additional properties.

Bounding the Number of Messages

We show, in Lemmas 5, 6, and 7, that eventually there are no messages in transit and every node has an accurate view of its neighbors' heights.

Proving The Leader-Oriented DAG Aspect

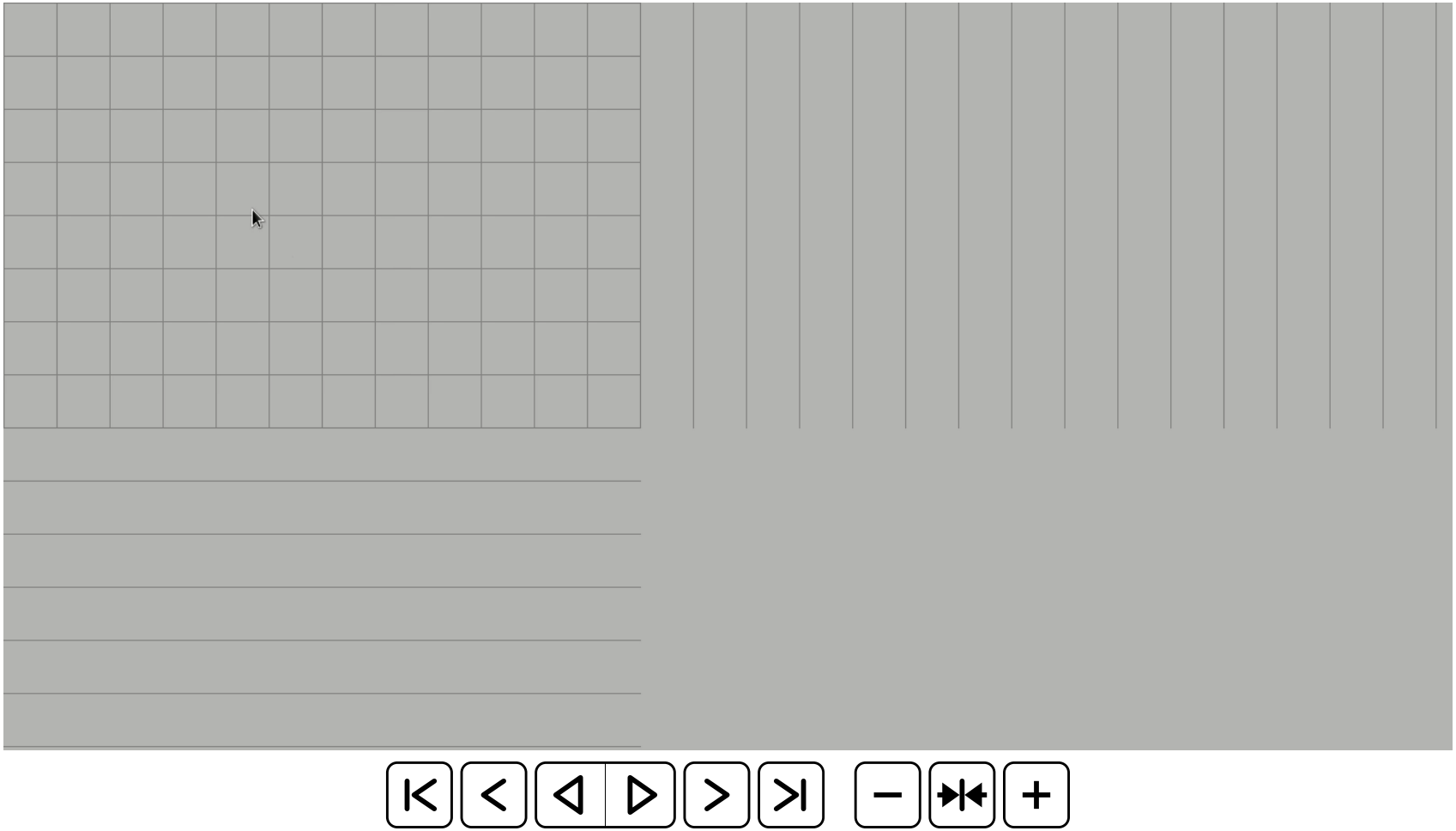
All the pieces are put together in Theorem 1 of Section 4.6 to show that eventually we have a leader-oriented connected component; a couple of additional properties are needed for this result.

Outline

What's JBotSim ?

JBoTSim is a java library that offers basic primitives for proto-typing, running, and visualizing distributed algorithms in dynamic networks.

Simulation



Performance Characteristics

1. **Latency** /. The number of rounds necessary for a component to elect a leader (become stable) after a topology change (average over all initial topologies).

Performance Characteristics

1. **Latency** l. The number of rounds necessary for a component to elect a leader (become stable) after a topology change (average over all initial topologies).
2. **Sensitivity** s. The number of nodes that updated their heights in response to a topology change (average over all possible topology changes).

Performance Characteristics

1. **Latency** l. The number of rounds necessary for a component to elect a leader (become stable) after a topology change (average over all initial topologies).
2. **Sensitivity** s. The number of nodes that updated their heights in response to a topology change (average over all possible topology changes).
3. **Resilience** r. The maximal fraction of links which can go down in a stable component without reelecting the current leader.

Performance Characteristics

- Merging two stable components:

Performance Characteristics

- Merging two stable components:
  - Given two fully connected (every two nodes are neighbours) components: $l \simeq 2$.

Performance Characteristics

- Merging two stable components:
  - Given two fully connected (every two nodes are neighbours) components: $l \simeq 2$.
  - Given two poorly connected (a node can have not more than two neigh- bours) components: $l \simeq n$.

Performance Characteristics

- Merging two stable components:
  - Given two fully connected (every two nodes are neighbours) components: $l \simeq 2$.
  - Given two poorly connected (a node can have not more than two neigh- bours) components: $l \simeq n$.
- Partitioning a stable component:

Performance Characteristics

- Merging two stable components:
  - Given two fully connected (every two nodes are neighbours) components: $l \simeq 2$.
  - Given two poorly connected (a node can have not more than two neigh- bours) components: $l \simeq n$.
- Partitioning a stable component:
  - Getting two fully connected component: $l = 2$.

Performance Characteristics

- Merging two stable components:
  - Given two fully connected (every two nodes are neighbours) components: $l \simeq 2$.
  - Given two poorly connected (a node can have not more than two neigh- bours) components: $l \simeq n$.
- Partitioning a stable component:
  - Getting two fully connected component: $l = 2$.
  - Getting two poorly connected component: $l = 2n$.

- Topology change in a “small world-like” stable component (every node has O(log n) neighbours) without partitioning:

- Topology change in a "small world-like" stable component (every node has O(log n) neighbours) without partitioning:
  - Latency $l = O(1)$.

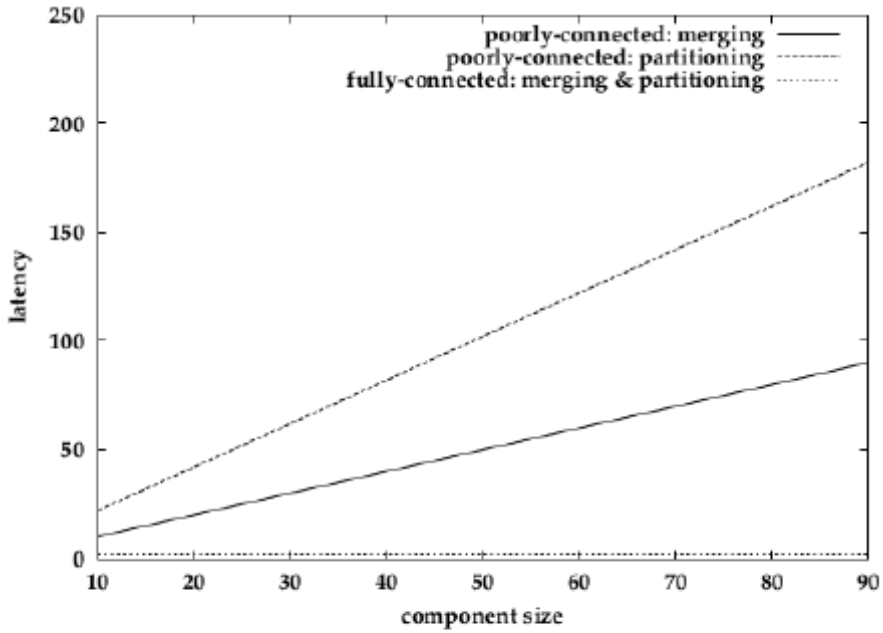
- Topology change in a "small world-like" stable component (every node has O(log n) neighbours) without partitioning:
  - Latency $l = O(1)$.
  - Sensitivity $s = O(log(n))$.

- Topology change in a "small world-like" stable component (every node has O(log n) neighbours) without partitioning:
  - Latency $l = O(1)$.
  - Sensitivity $s = O(log(n))$.
- Resilience of a stable component:

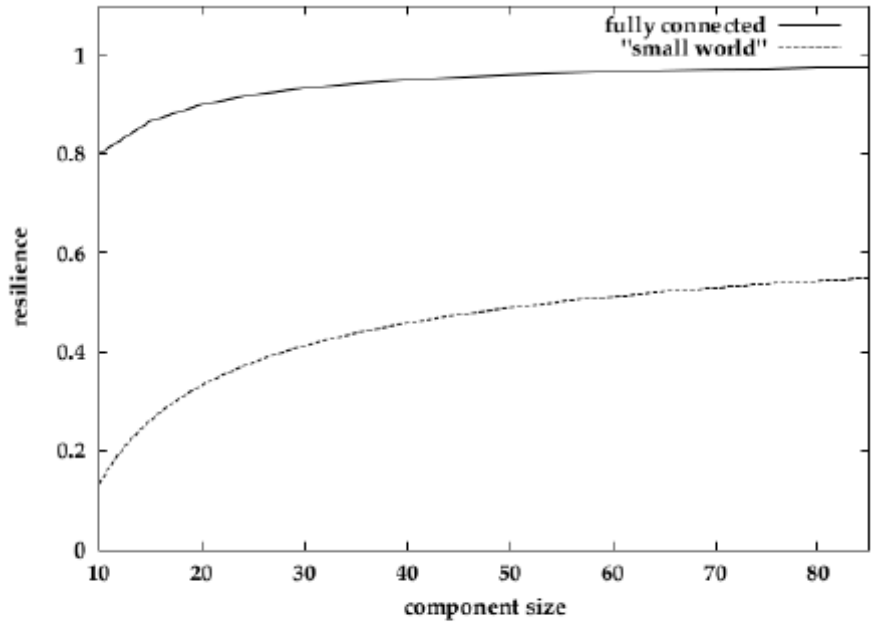
- Topology change in a "small world-like" stable component (every node has O(log n) neighbours) without partitioning:
  - Latency $l = O(1)$.
  - Sensitivity $s = O(log(n))$.
- Resilience of a stable component:
  - Given a fully connected component: $r \simeq 1 - 2/n$.

- Topology change in a "small world-like" stable component (every node has O(log n) neighbours) without partitioning:
  - Latency $l = O(1)$.
  - Sensitivity $s = O(log(n))$.
- Resilience of a stable component:
  - Given a fully connected component: $r \simeq 1 - 2/n$.
  - Given a "small world-like" component: $r \simeq 1 - 2/log(n)$.

How is our algorithm's performance ?



(a) Latency



(b) Resilience

Figure: Simulation Results

Outline

Is The Algorithm Perfect ?

- An open question is how to extend our algorithm and its analysis to handle a wider range of clocks, such as approximately synchronized clocks and vector clocks.

Is The Algorithm Perfect ?

- An open question is how to extend our algorithm and its analysis to handle a wider range of clocks, such as approximately synchronized clocks and vector clocks.
- Another question could be asked concerning the possibility of building up an optimized spanning tree using some algorithms such as Kruskal's Minimum Spanning Tree Algorithm (MST).

Thank you for your attention

All your questions and remarks are welcomed.