Seminar 5 – Population Protocols: General Computation and Leader Election

Distributed Computing in Anonymous Dynamic Systems

Giovanni Viglietta

Rome – March 12, 2024

Distributed Computing in Anonymous Dynamic Systems

Syllabus

- Anonymous Networks
  - Introduction and basic algorithms for static networks
  - Dynamicity and history trees
  - Optimal computation in networks with and without leaders
  - Computation in dynamic congested networks
- Population Protocols
  - Introduction and basic algorithmic techniques
  - Leader election in Mediated Population Protocols
- Mobile Robots
  - Gathering and Pattern Formation in the plane
  - Meeting in a polygon by oblivious robots

Exam

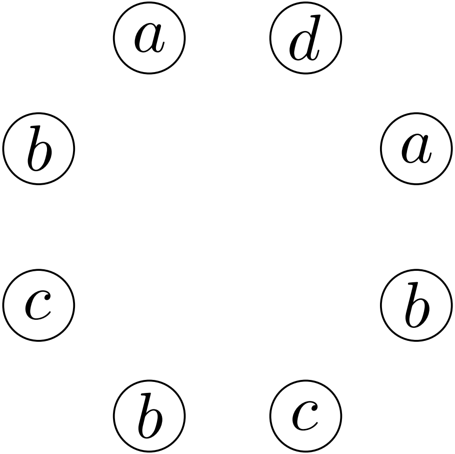
Pre-recorded 10-minute presentation video on one of the papers that will be suggested at the end of the course.

Today's seminar

- Introduction to Population Protocols
- Computable predicates
- Mediated Population Protocols
- One-way and faulty models

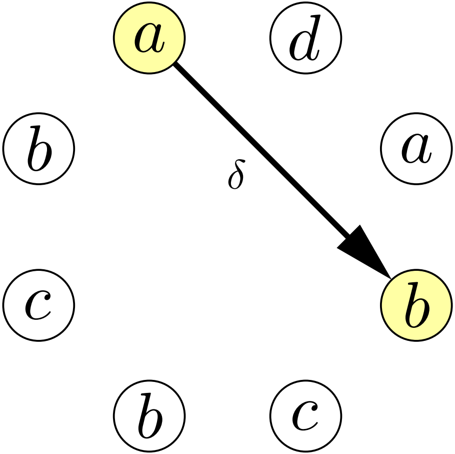
Population Protocols

Population Protocols



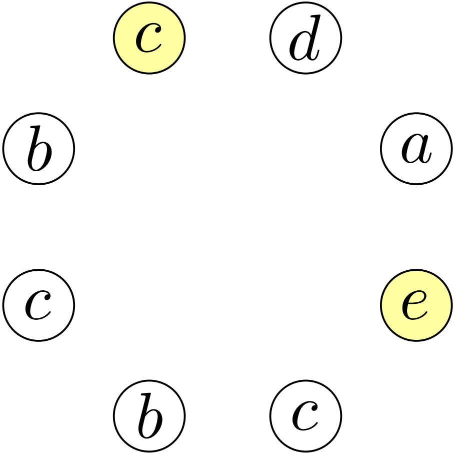
Setting: a set of finite-state agents.

Population Protocols



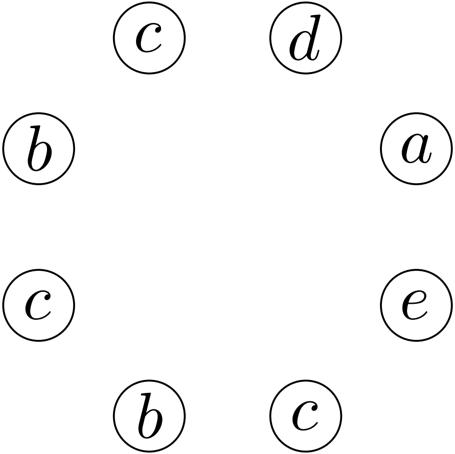
Pairs of agents interact in a non-deterministic order...

Population Protocols



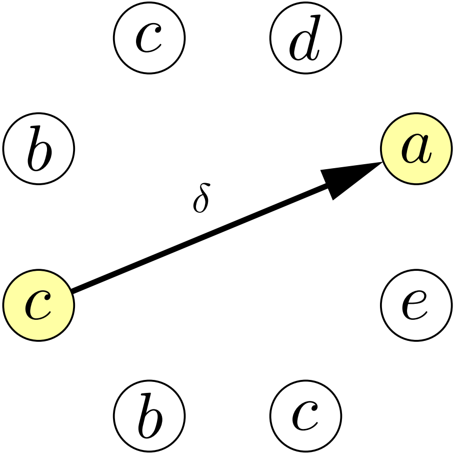
...and change states according to a transition function.

Population Protocols



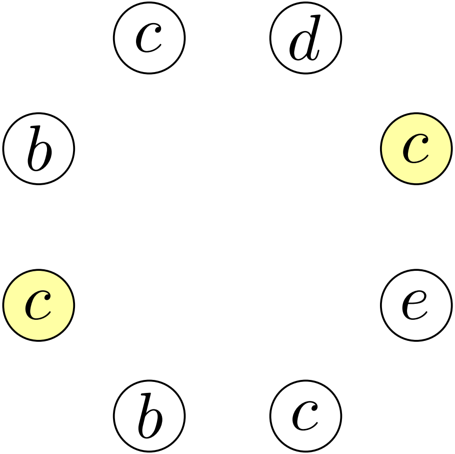
...and change states according to a transition function.

Population Protocols



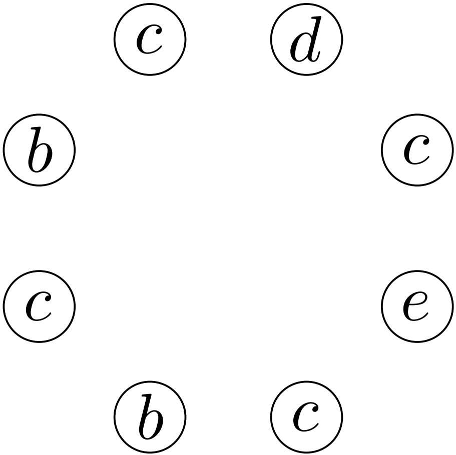
...and change states according to a transition function.

Population Protocols



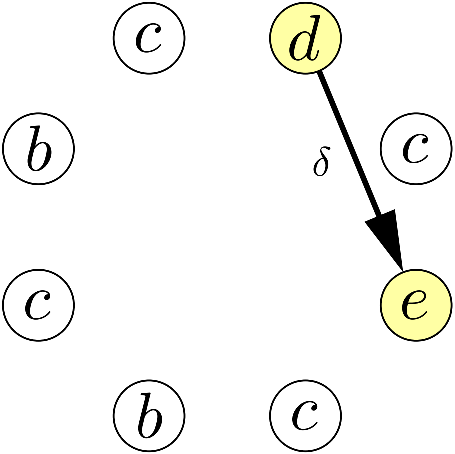
...and change states according to a transition function.

Population Protocols



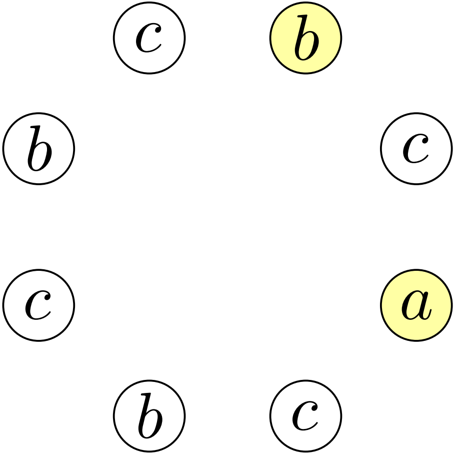
...and change states according to a transition function.

Population Protocols



...and change states according to a transition function.

Population Protocols



...and change states according to a transition function.

Population Protocols

A **Population Protocol** is a network of anonymous agents where:

- Communications are sequential (i.e., only one pair of agents interacts at any time),
- Interactions are asymmetric (i.e., there is an initiator and a responder).
- Each agent has a constant amount of internal memory.

So, if internal states belong to a finite set S, a *transition function* has the form $f\colon S \times S \to S \times S$.

There are two notions of *fairness* of interactions:

- Local fairness: Any pair of agents interacts infinitely often (in both ways).
- Global fairness: If a configuration of *all agents* is potentially reachable for infinitely many times, it is eventually reached.

Population Protocols: Leader Election

Example. Leader Election protocol rules:

$(L, L) \mapsto (L, N)$

$(L, N) \mapsto (L, N)$

$(N, L) \mapsto (N, L)$

$(N, N) \mapsto (N, N)$

If all agents start in state L, eventually there will be only one in state L (the *leader*). This protocol works under local fairness.

Population Protocols: Majority

Example. Majority protocol. Agents start in state "red" or "blue". Eventually, they reach a configuration where all agents are in state "yes" if # red > # blue, and in state "no" otherwise. Rules:

$(\text{red}, \text{blue}) \mapsto (\text{no}, \text{no})$ (eliminates all "red"s or all "blue"s)

$(\text{red}, \text{no}) \mapsto (\text{red}, \text{yes})$ ("red" changes answer to "yes")

$(\text{blue}, \text{yes}) \mapsto (\text{blue}, \text{no})$ ("blue" changes answer to "no")

$(\text{yes}, \text{no}) \mapsto (\text{no}, \text{no})$ (takes care of ties)

This protocol requires global fairness (why?).

Population Protocols: Computable Predicates

Assuming that the initial states are integers (modulo m), what predicates can be computed by Population Protocols?

A *predicate* is a function whose output is "yes" or "no", and all agents must eventually return the correct output based on the multiset of initial states.

Theorem (Angluin et al., PODC 2006)

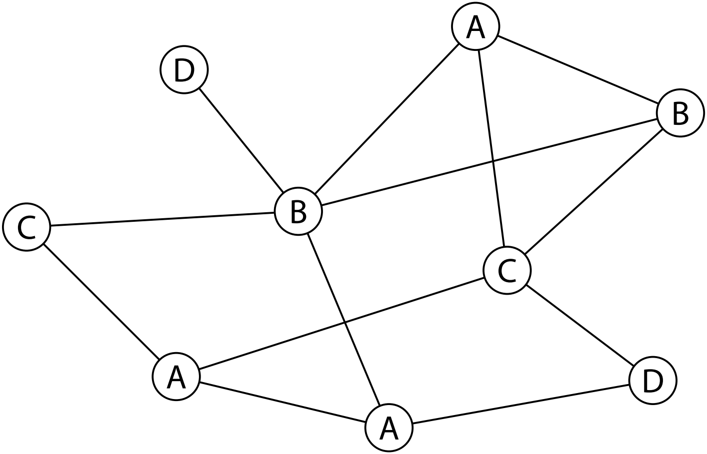
The only predicates computable by Population Protocols are:

- $\sum_i c_i x_i \geq a$, *where* a, c_i*'s are integer constants (generalization of Majority),*
- $\sum_i c_i x_i \equiv a \pmod{b}$, *where* a, b, c_i*'s are integer constants,*
- *Boolean combinations of the above predicates (*$\neg$, $\vee$, $\wedge$*).*

These can also be characterized in terms of *Presburger arithmetic*: the predicates in first-order logic using the symbols $+$, 0, 1, $\neg$, $\vee$, $\wedge$, $\forall$, $\exists$, $=$, $<$, $($, $)$, plus variables.

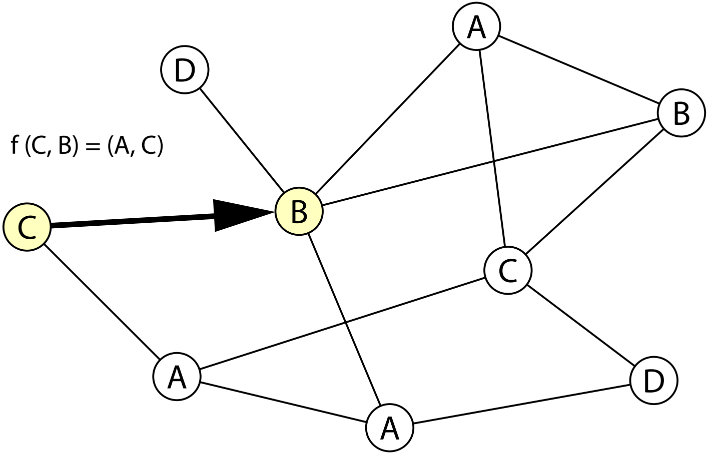
Mediated Population Protocols

Fixed-Network Population Protocol



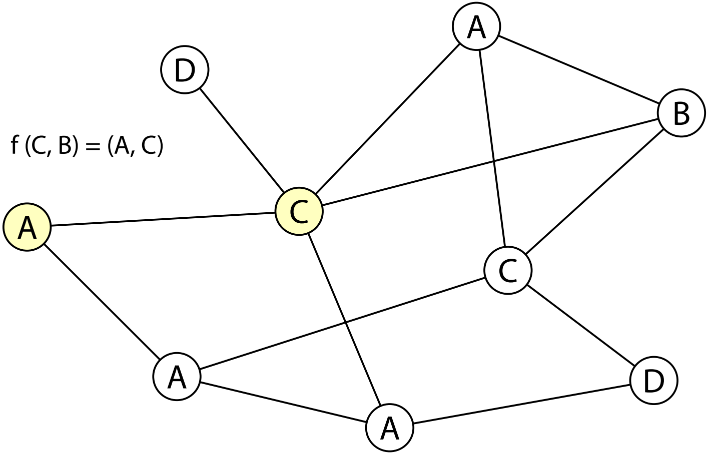
Assume that agents can only interact through specific links.

Fixed-Network Population Protocol



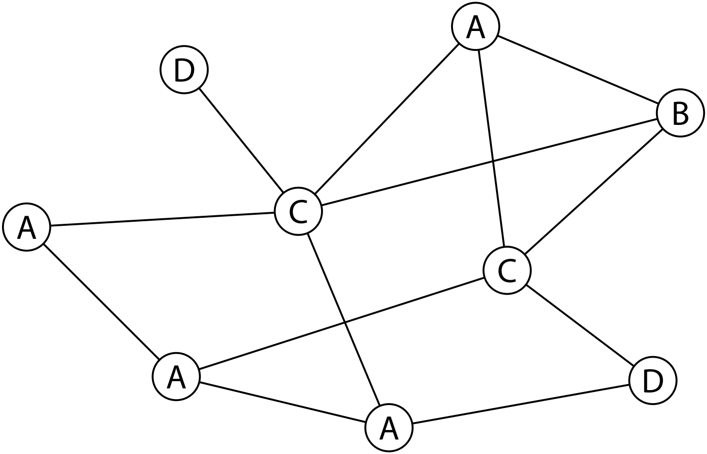
Pairs of adjacent agents interact in a non-deterministic order...

Fixed-Network Population Protocol



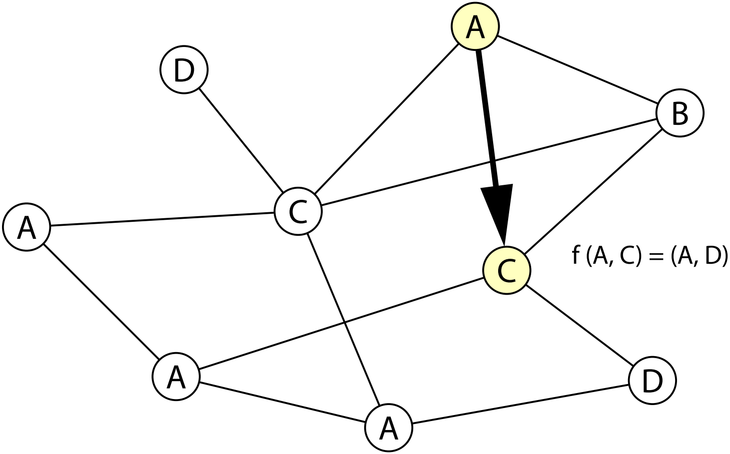
...and change states according to a transition function.

Fixed-Network Population Protocol



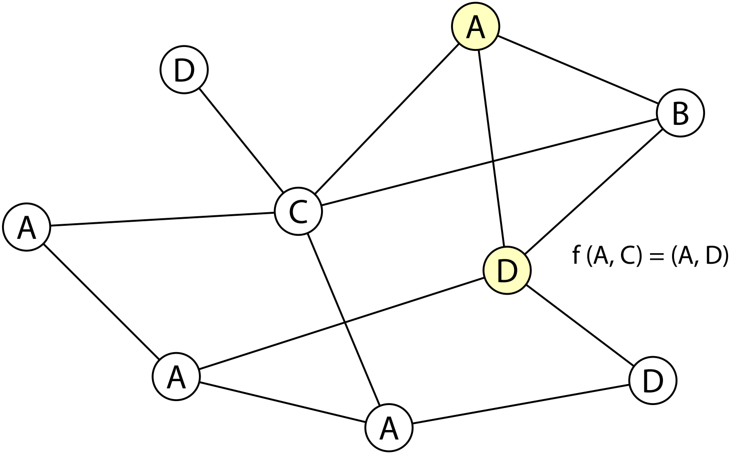
...and change states according to a transition function.

Fixed-Network Population Protocol



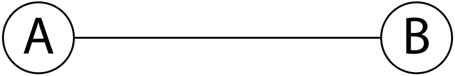
...and change states according to a transition function.

Fixed-Network Population Protocol



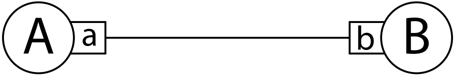
...and change states according to a transition function.

Mediated Population Protocol



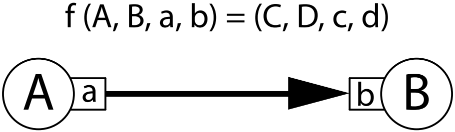
Mediated agents: we add ports with (finite) states.

Mediated Population Protocol



Mediated agents: we add ports with (finite) states.

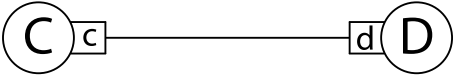
Mediated Population Protocol



The transition function affects both agent and port states.

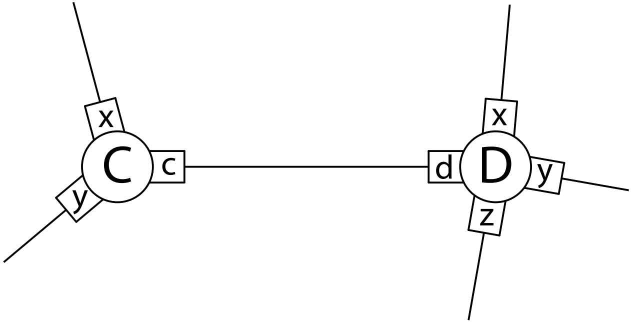
Mediated Population Protocol

$$f(A, B, a, b) = (C, D, c, d)$$



The transition function affects both agent and port states.

Mediated Population Protocol



Each agent has a port for each neighbor.

Schedulers

We distinguish two types of scheduler:

- **Recurrent:** each pair of neighboring agents interacts infinitely often (in both directions)
- k-**Bounded:** it is recurrent and, between two consecutive interactions of the same pair of agents, no other pair interacts more than k times

Note: under a 1-bounded scheduler, the sequence of interactions is periodic

Stability and Termination

A protocol can be:

- **Stable:** eventually, no agent changes state
- **Terminating:** eventually, all agents are in a *terminal state* (i.e., "explicit stability")

Usually, with the recurrent scheduler protocols are stable;
with the k-bounded schedulers, they are terminating

Overview

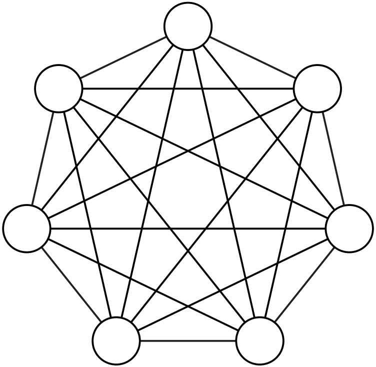
Leader Election Problem: all agents start in the same state, and eventually there is a unique agent in a *leader state*

- Complete graphs
- Complete bipartite graphs
- Trees

Applications of a Unique Leader:

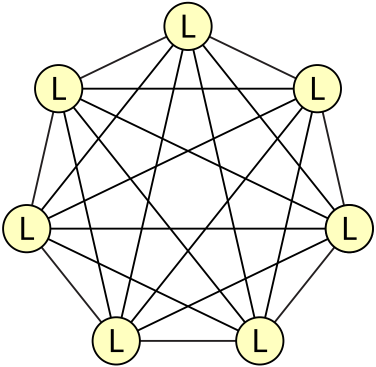
- Token circulation
- Construction of a shortest-path spanning tree
- Stability detection (turning stable protocols into terminating ones)
- Equivalence of k-bounded schedulers for all $k > 1$

Leader Election in a Complete Graph



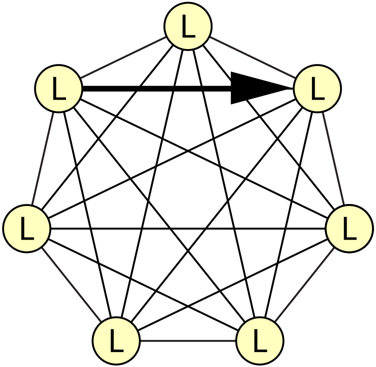
Theorem: in the complete graph K_n, it is possible to elect a leader under the recurrent scheduler.

Leader Election in a Complete Graph



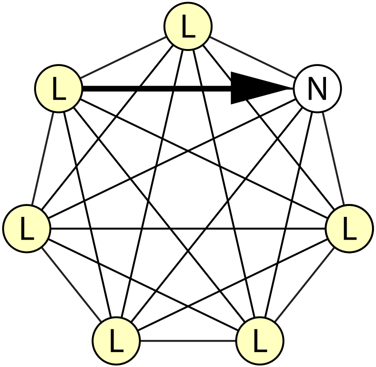
Initially, all agents have the leader state.

Leader Election in a Complete Graph



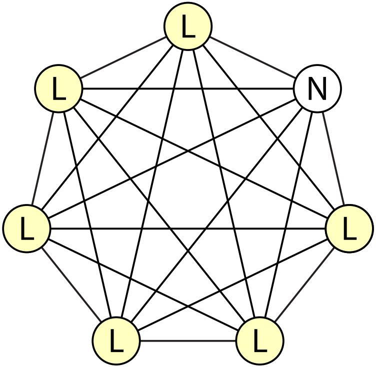
When two leaders interact, one is "eliminated".

Leader Election in a Complete Graph



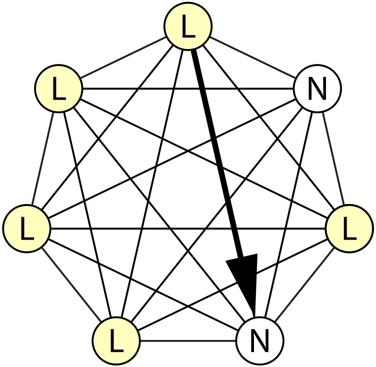
When two leaders interact, one is “eliminated”.

Leader Election in a Complete Graph



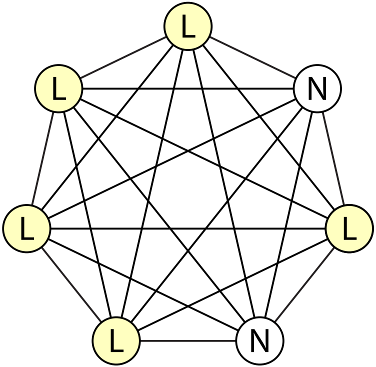
When two leaders interact, one is “eliminated”.

Leader Election in a Complete Graph



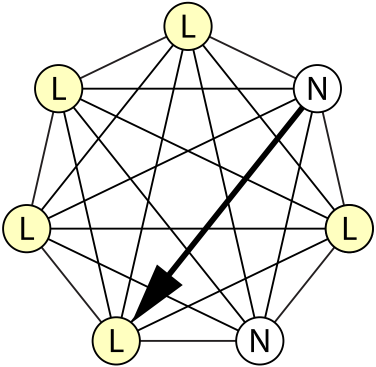
When two leaders interact, one is “eliminated”.

Leader Election in a Complete Graph



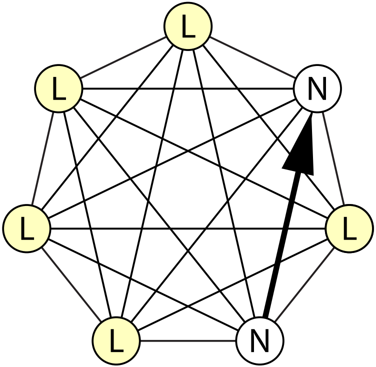
When two leaders interact, one is “eliminated”.

Leader Election in a Complete Graph



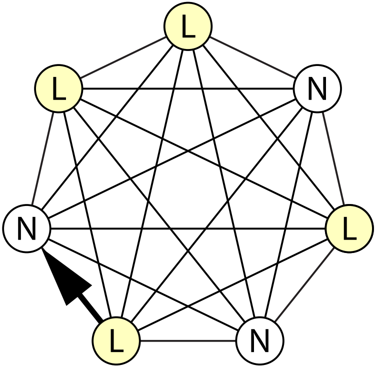
Otherwise, nothing happens.

Leader Election in a Complete Graph



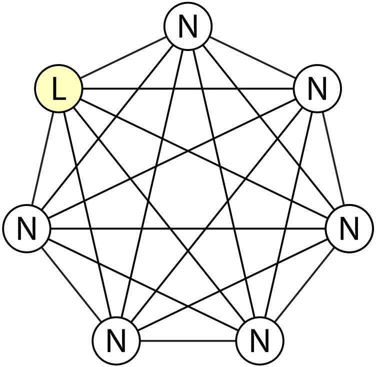
Otherwise, nothing happens.

Leader Election in a Complete Graph



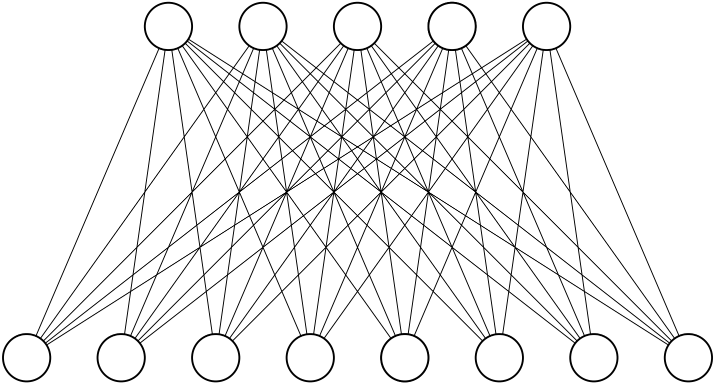
Otherwise, nothing happens.

Leader Election in a Complete Graph



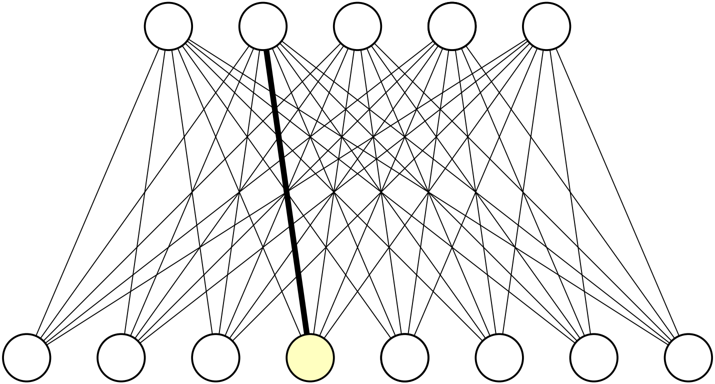
Eventually, only one leader remains.

Leader Election in a Complete Bipartite Graph



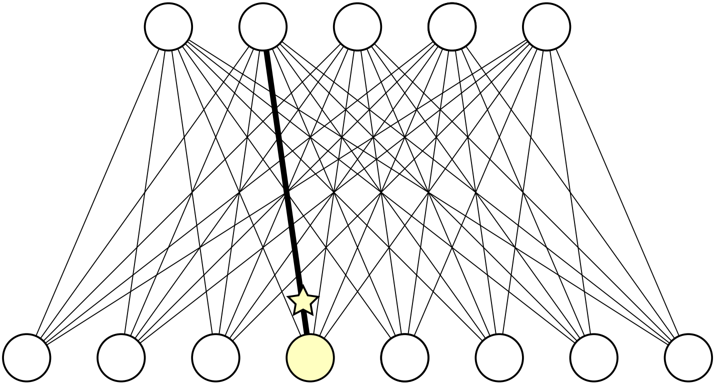
Theorem: in $K_{m,n}$, it is possible to elect a leader under the 1-bounded scheduler if and only if m and n are coprime.

Leader Election in a Complete Bipartite Graph



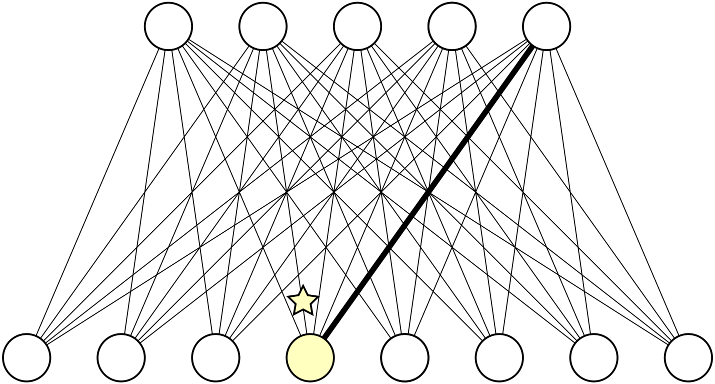
Suppose that m and n are coprime.

Leader Election in a Complete Bipartite Graph



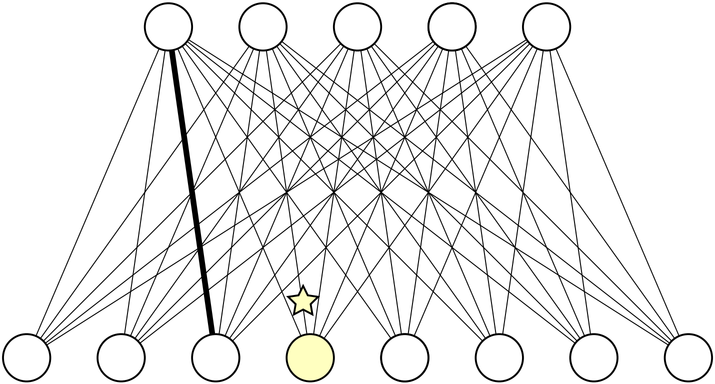
Since the 1-bounded scheduler is periodic, an agent can tell when a new period starts by marking the first edge that it "sees".

Leader Election in a Complete Bipartite Graph



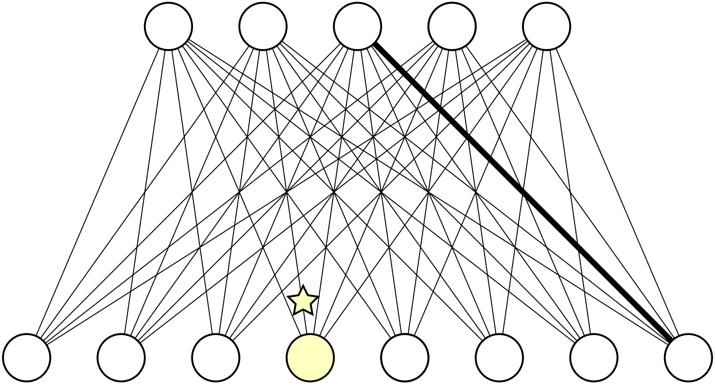
Since the 1-bounded scheduler is periodic, an agent can tell when a new period starts by marking the first edge that it "sees".

Leader Election in a Complete Bipartite Graph



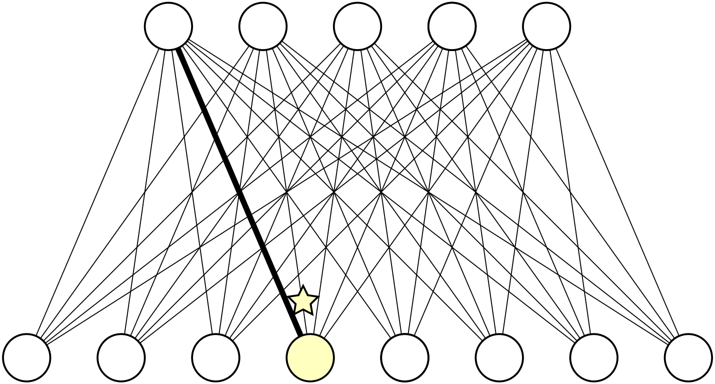
Since the 1-bounded scheduler is periodic, an agent can tell when a new period starts by marking the first edge that it "sees".

Leader Election in a Complete Bipartite Graph



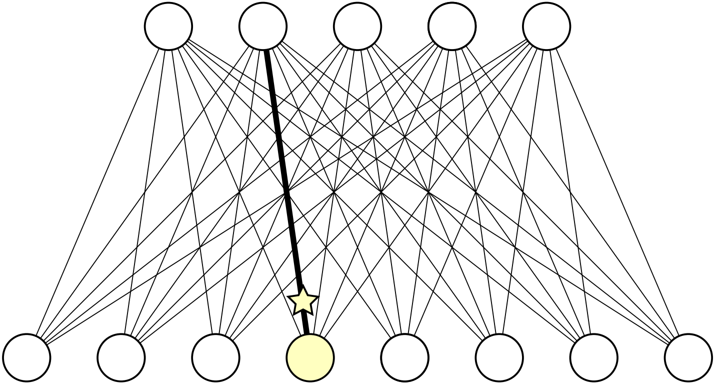
Since the 1-bounded scheduler is periodic, an agent can tell when a new period starts by marking the first edge that it "sees".

Leader Election in a Complete Bipartite Graph



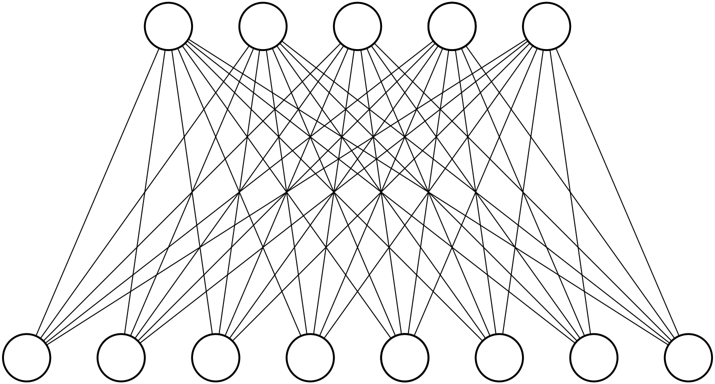
Since the 1-bounded scheduler is periodic, an agent can tell when a new period starts by marking the first edge that it “sees”.

Leader Election in a Complete Bipartite Graph



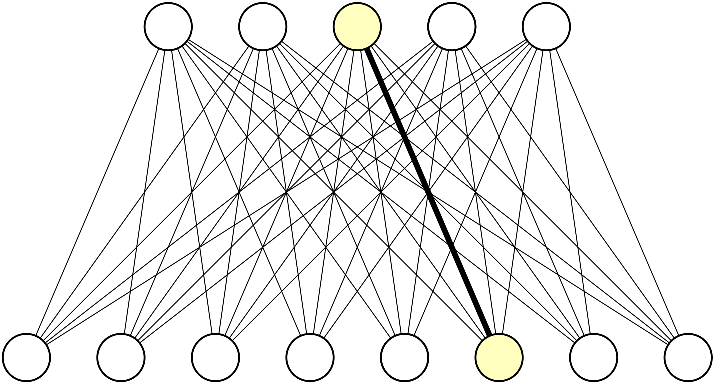
The next time it encounters the marked edge, it knows that a new period has started.

Leader Election in a Complete Bipartite Graph



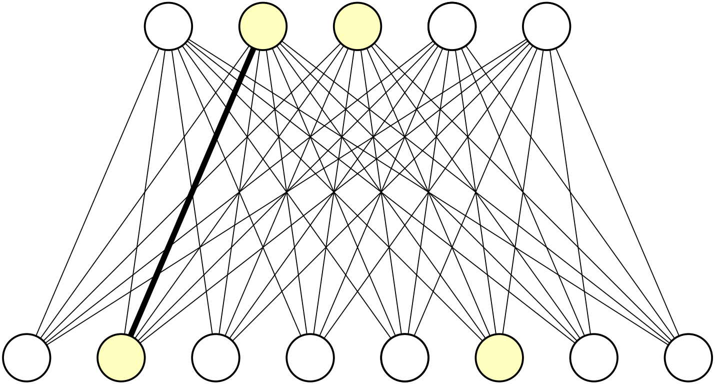
In the first phase, we construct a maximal matching.

Leader Election in a Complete Bipartite Graph



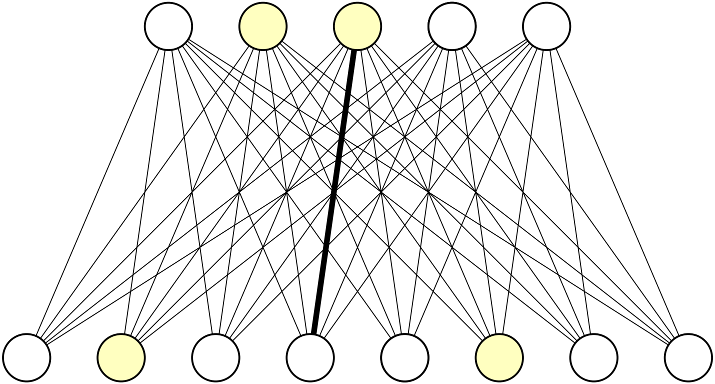
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



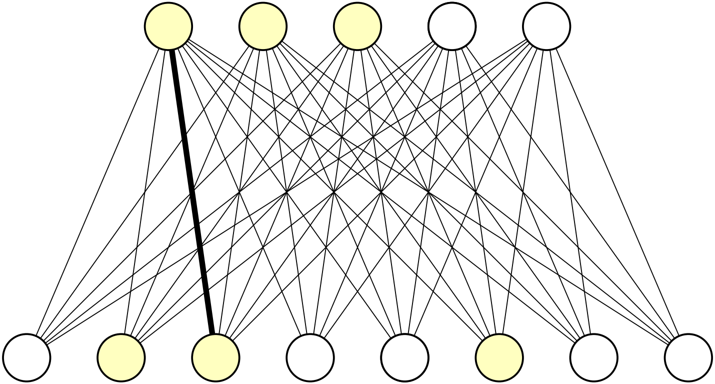
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



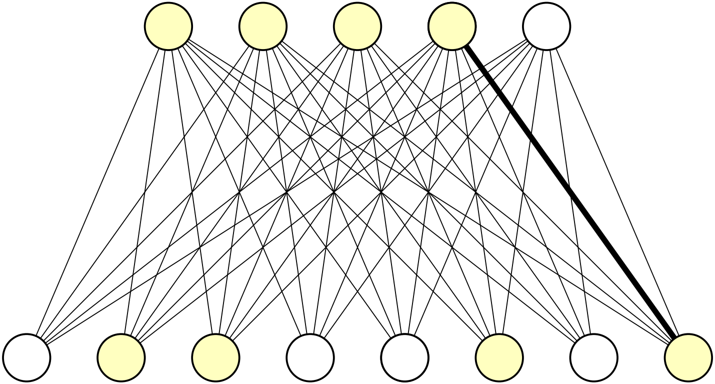
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



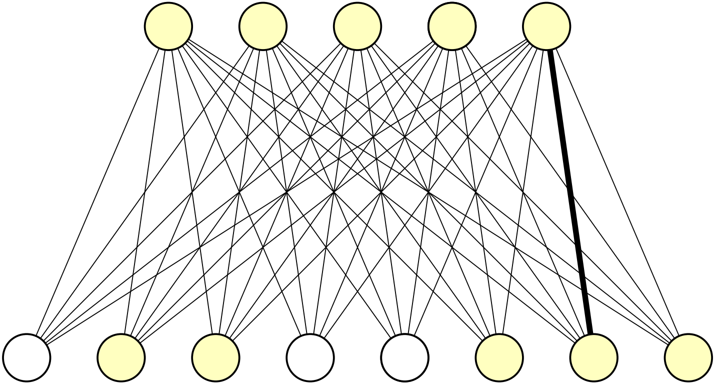
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



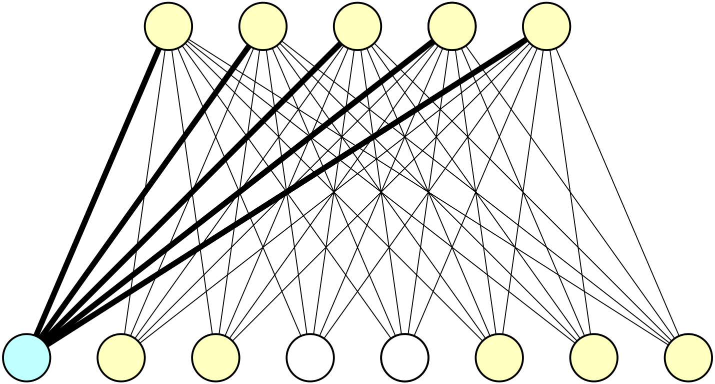
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



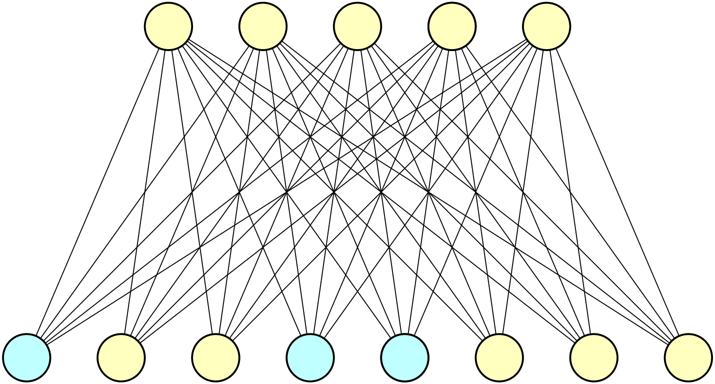
When two unmatched agents meet, they become matched.

Leader Election in a Complete Bipartite Graph



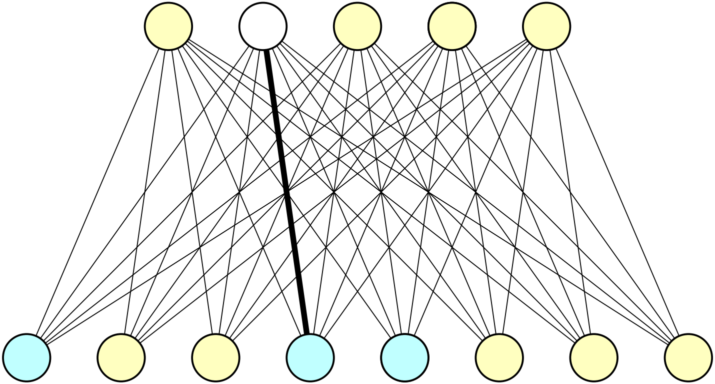
When an unmatched agent sees only matched agents for an entire period, it knows that the matching is maximal.

Leader Election in a Complete Bipartite Graph



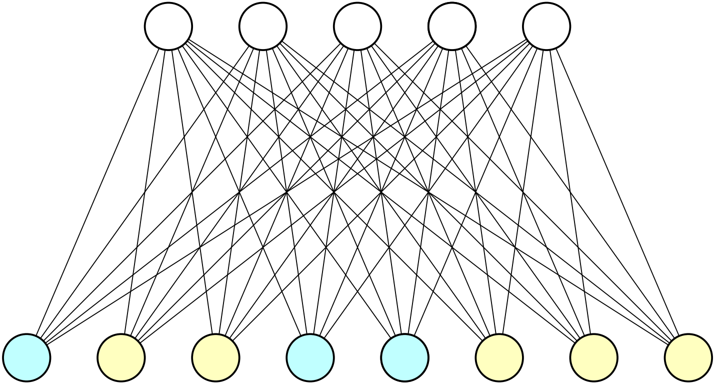
These unmatched agents assume a “reset” state.

Leader Election in a Complete Bipartite Graph



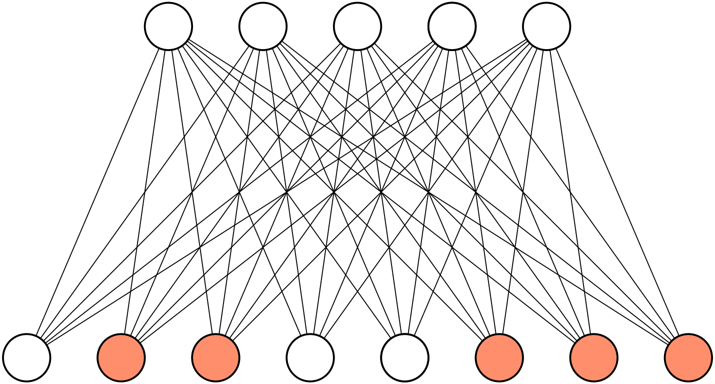
After another period, whoever sees a "reset" agent becomes unmatched again.

Leader Election in a Complete Bipartite Graph



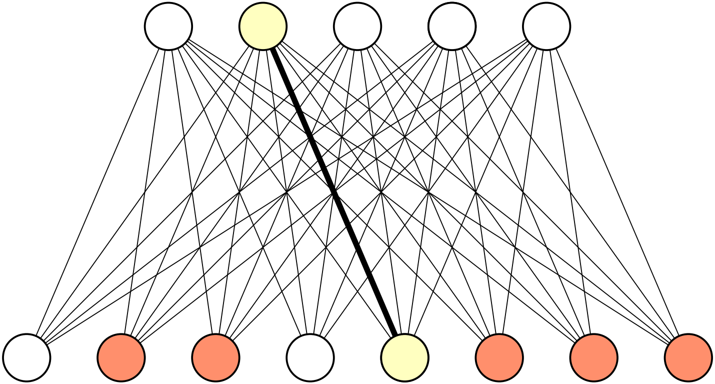
After another period, whoever sees a "reset" agent becomes unmatched again.

Leader Election in a Complete Bipartite Graph



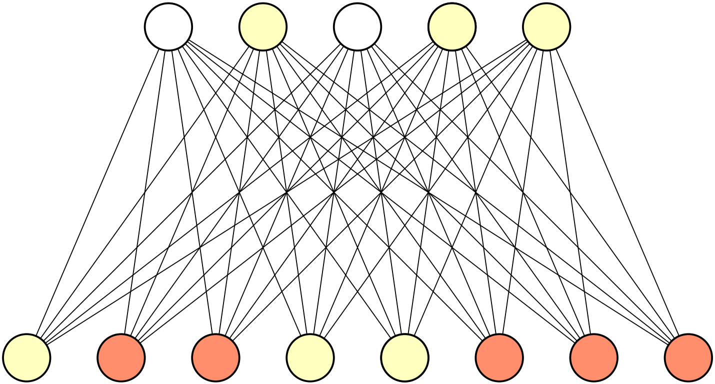
In the next period, the agents that are still matched become "eliminated".

Leader Election in a Complete Bipartite Graph



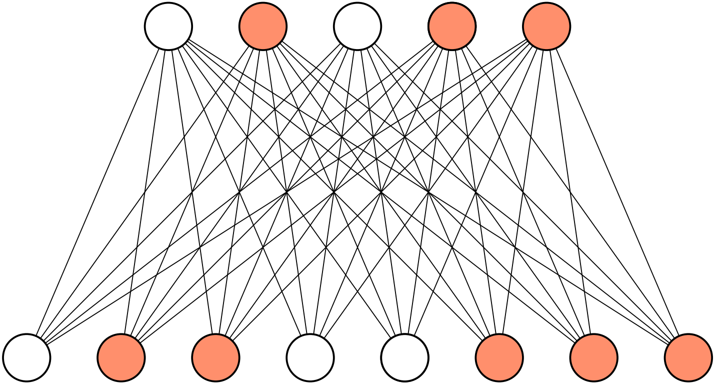
Then another matching phase starts, but the “eliminated” agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



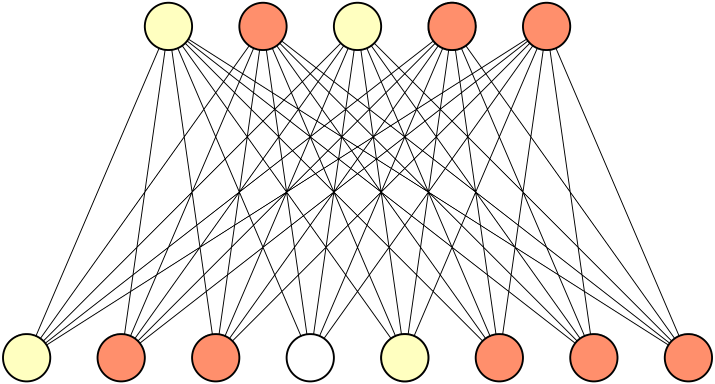
Then another matching phase starts, but the “eliminated” agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



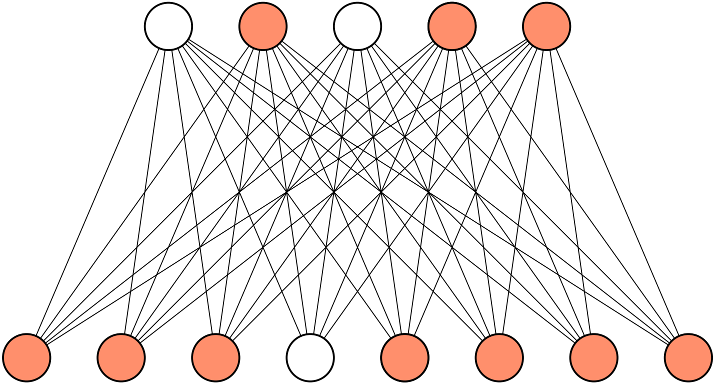
Then another matching phase starts, but the "eliminated" agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



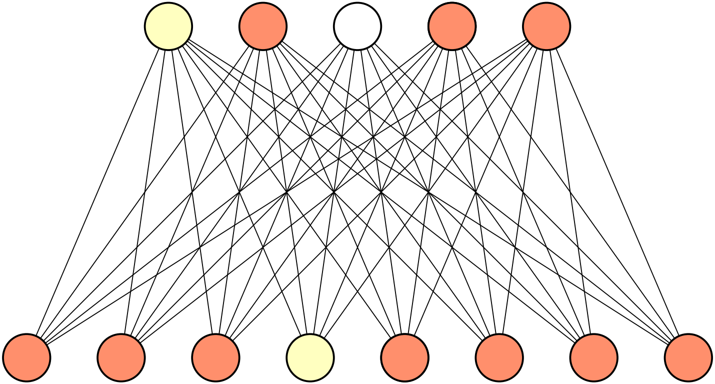
Then another matching phase starts, but the "eliminated" agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



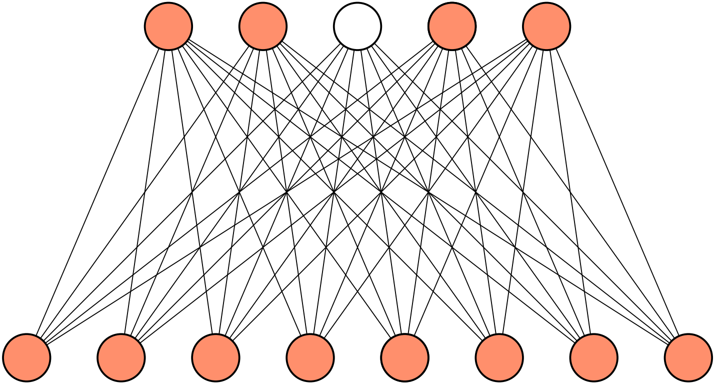
Then another matching phase starts, but the “eliminated” agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



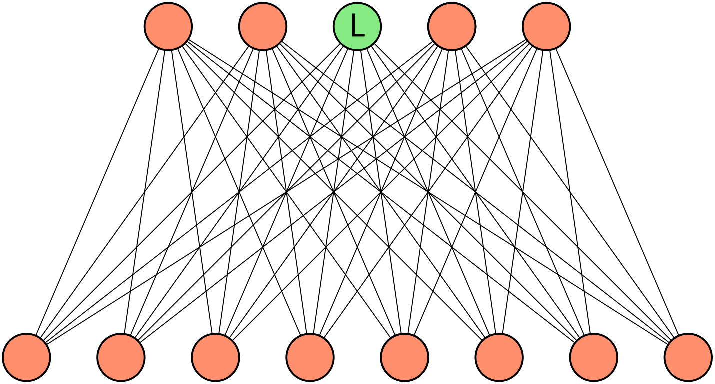
Then another matching phase starts, but the “eliminated” agents are ignored. The same protocol is repeated.

Leader Election in a Complete Bipartite Graph



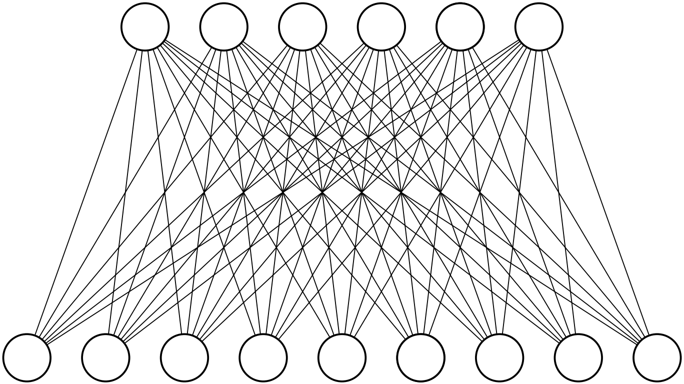
Since m and n are coprime, eventually only one agent will remain available.

Leader Election in a Complete Bipartite Graph



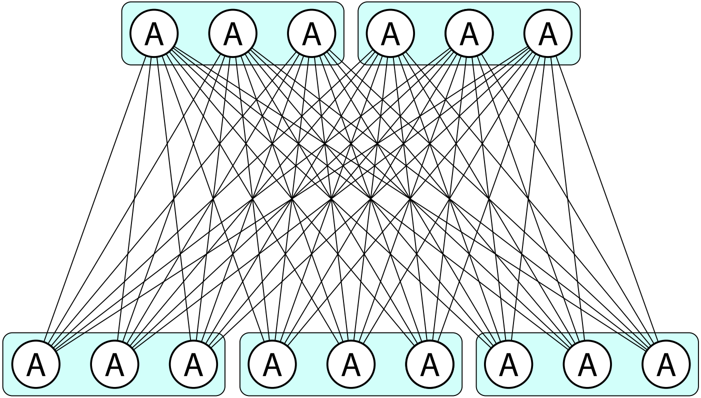
When this agent sees that all its neighbors are “eliminated”, it becomes the leader. This protocol is terminating.

Leader Election in a Complete Bipartite Graph



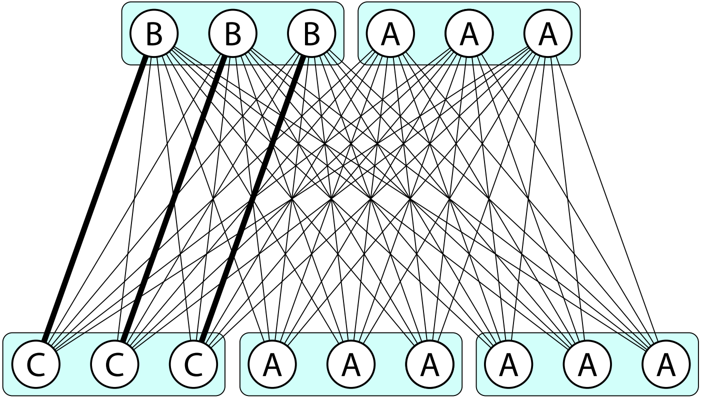
Suppose that m and n have a common divisor $d > 1$.

Leader Election in a Complete Bipartite Graph



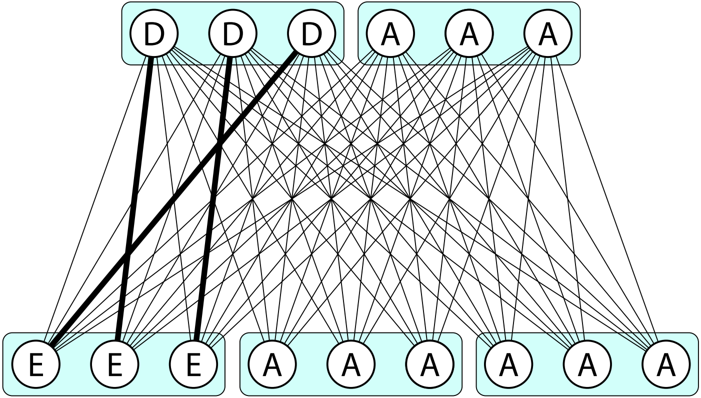
We partition each side of the network into groups of d agents, and we assign all agents the same initial state.

Leader Election in a Complete Bipartite Graph



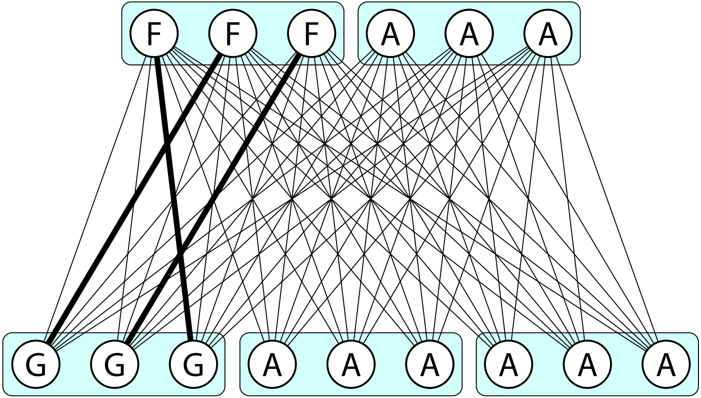
The scheduler chooses two groups on opposite sides, and activates the agents according to a perfect matching.

Leader Election in a Complete Bipartite Graph



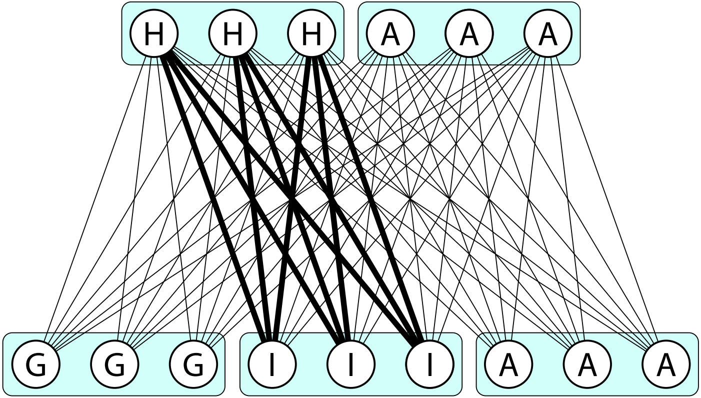
Then it chooses another perfect matching, and so on, until all pairs of neighbors have been activated.

Leader Election in a Complete Bipartite Graph



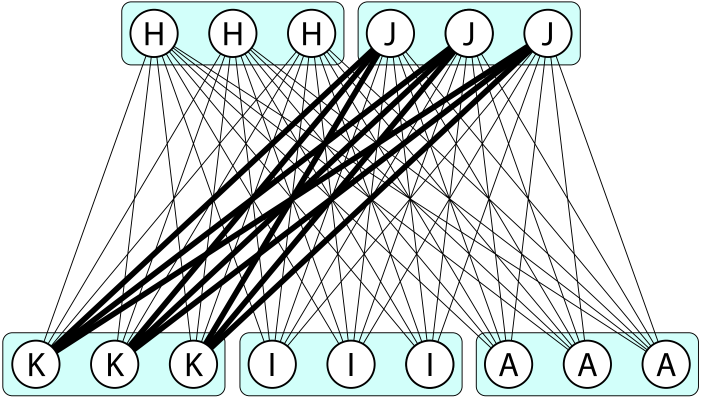
Then it chooses another perfect matching, and so on, until all pairs of neighbors have been activated.

Leader Election in a Complete Bipartite Graph



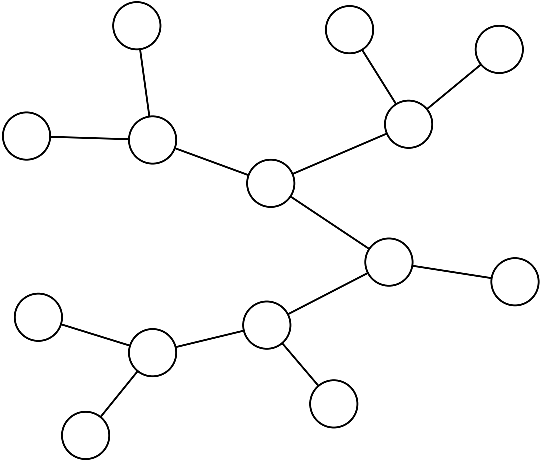
Then it does the same with two other groups, and so on, until all pairs of neighbors have been activated.

Leader Election in a Complete Bipartite Graph



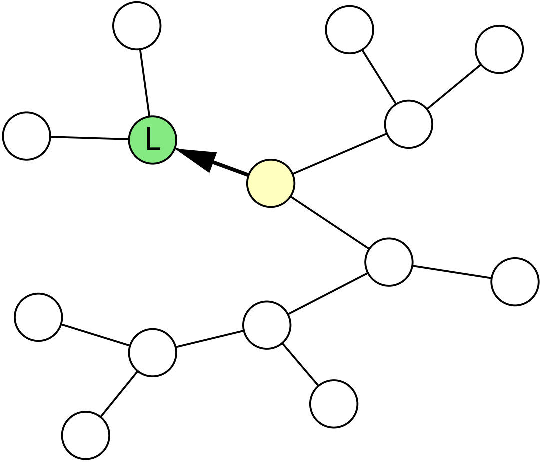
Every d interactions, all agents in the same group have the same state. Hence a leader cannot be elected.

Leader Election in a Tree



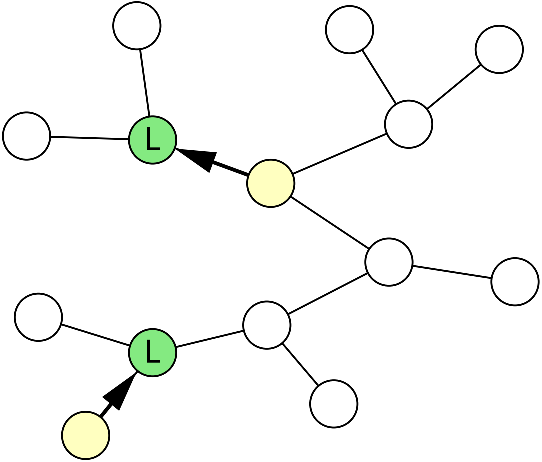
Theorem: in a tree, it is possible to elect a leader under the recurrent scheduler.

Leader Election in a Tree



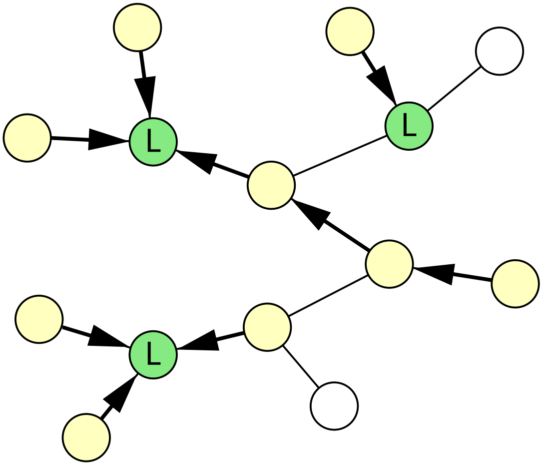
When two agents meet, they form a small rooted tree, where the root is a leader.

Leader Election in a Tree



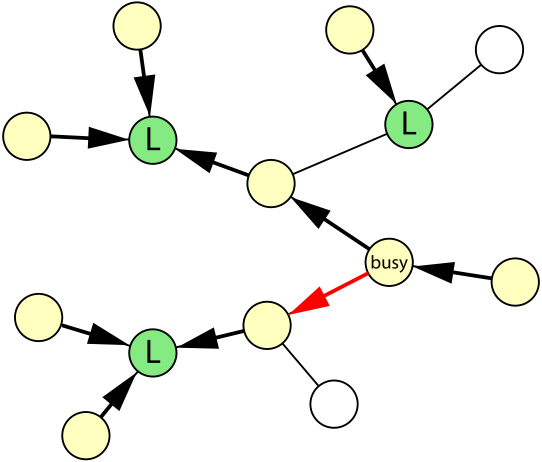
Arrows are encoded as port states.

Leader Election in a Tree



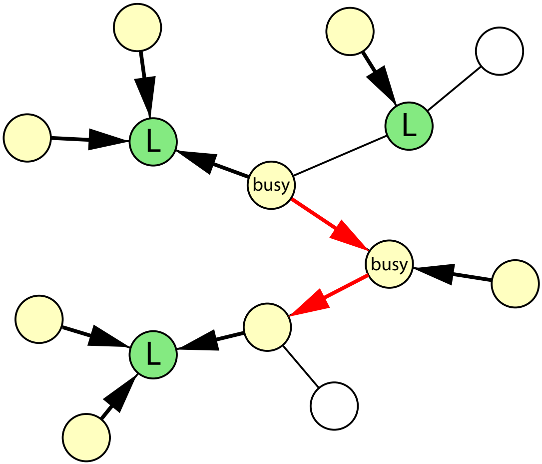
New agents may join existing trees, and a forest is formed.

Leader Election in a Tree



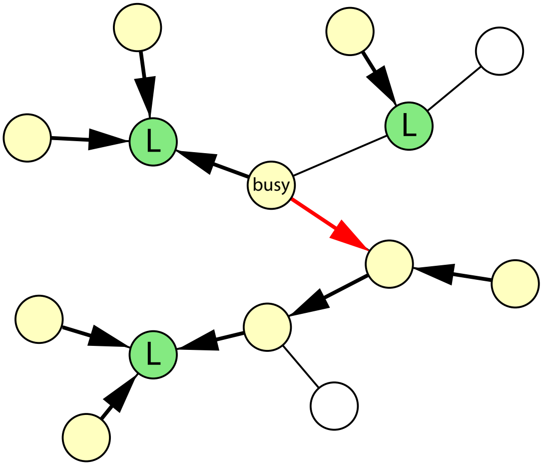
When two trees merge, one agent becomes "busy". Its task is to tell its leader that it is no longer a leader.

Leader Election in a Tree



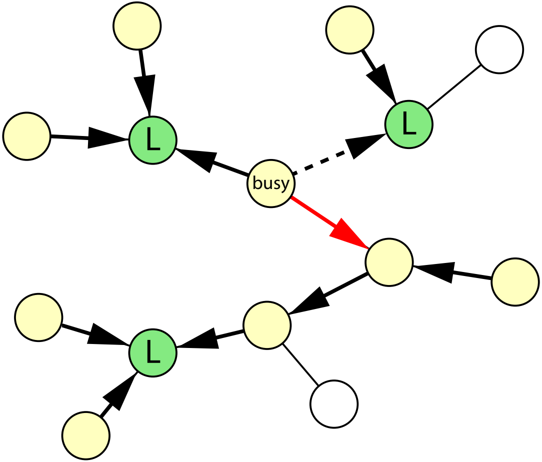
The parent of a busy agent becomes busy too, and reverses the corresponding arrow.

Leader Election in a Tree



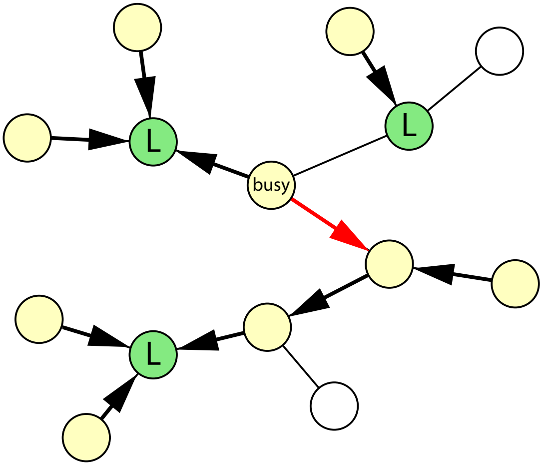
The child then ceases to be busy.

Leader Election in a Tree



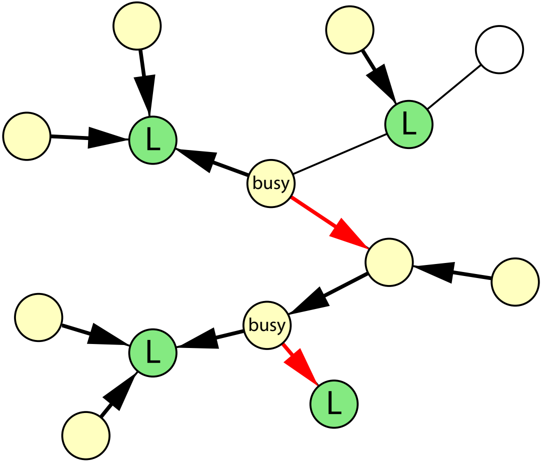
A busy agent rejects all requests to merge.

Leader Election in a Tree



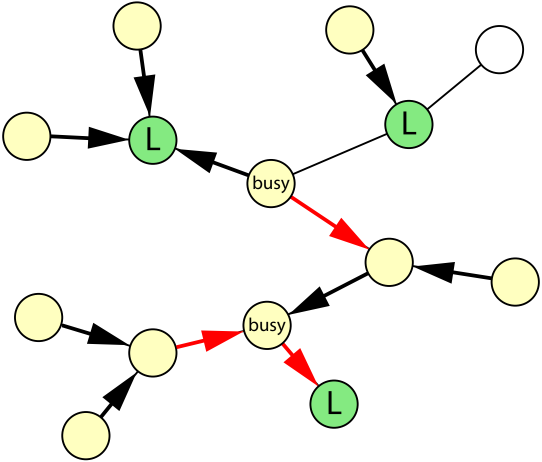
A busy agent rejects all requests to merge.

Leader Election in a Tree



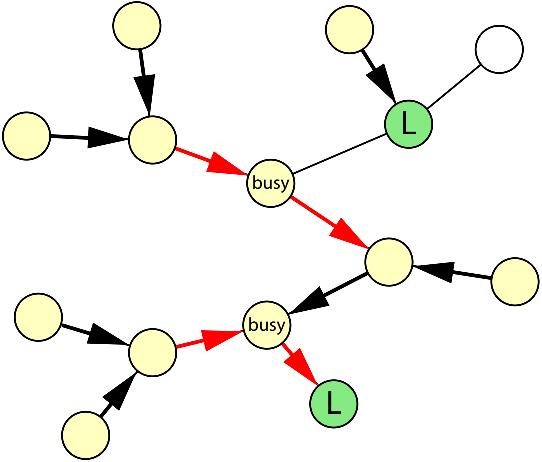
When a leader notices that one of its children is busy, it stops being a leader.

Leader Election in a Tree



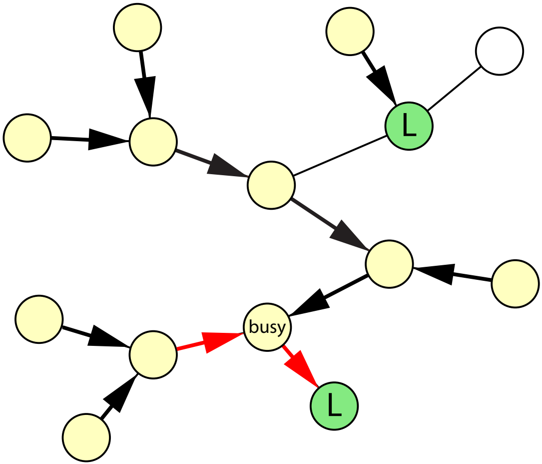
When a leader notices that one of its children is busy, it stops being a leader.

Leader Election in a Tree



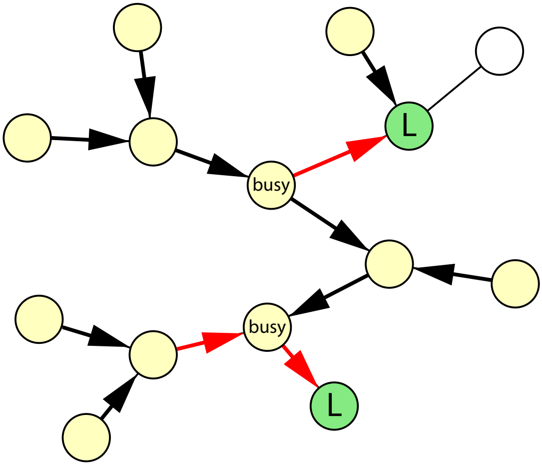
When a leader notices that one of its children is busy, it stops being a leader.

Leader Election in a Tree



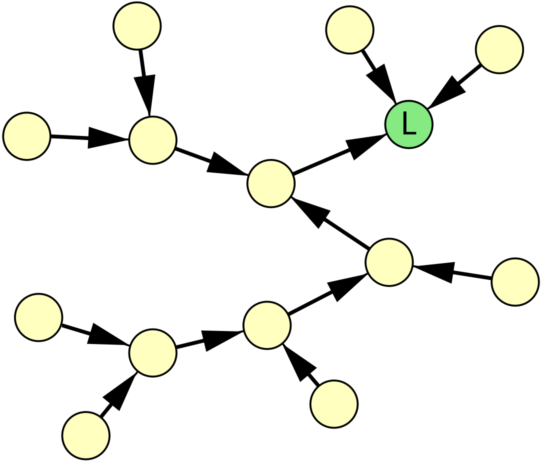
When a leader notices that one of its children is busy, it stops being a leader.

Leader Election in a Tree



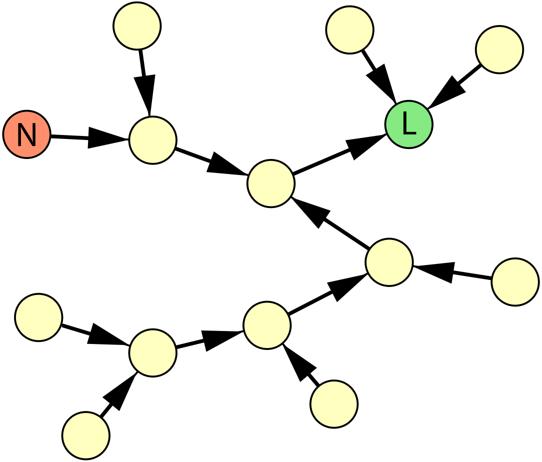
Agents that are no longer busy accept new merge requests.

Leader Election in a Tree



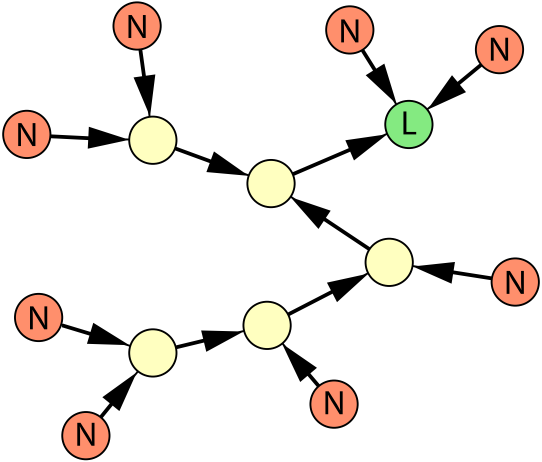
Eventually, only one leader is left, and the whole tree is oriented toward it.

Leader Election in a Tree



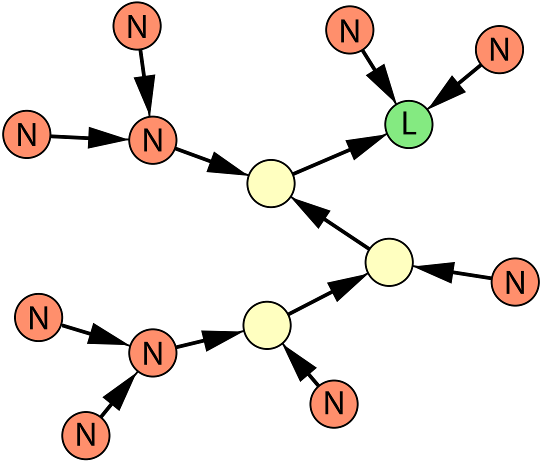
If the scheduler is k-bounded, a leaf eventually knows that it is a leaf. A non-leader leaf can safely terminate.

Leader Election in a Tree



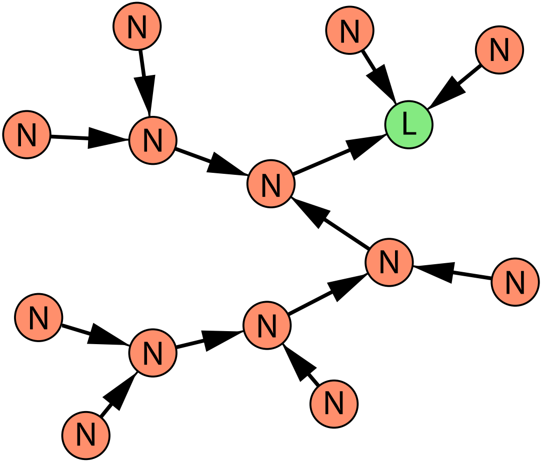
If the scheduler is k-bounded, a leaf eventually knows that it is a leaf. A non-leader leaf can safely terminate.

Leader Election in a Tree



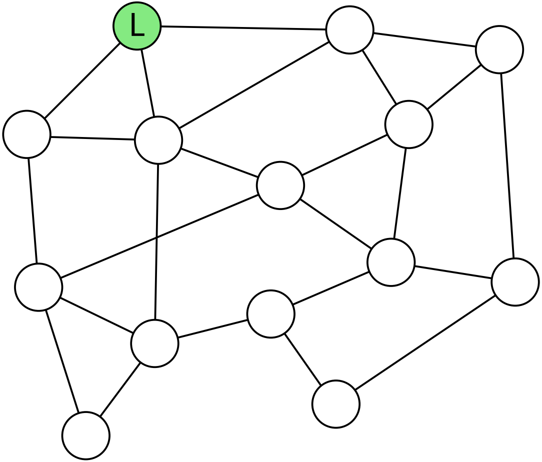
If an agent's children all have terminated, the agent eventually realizes and terminates.

Leader Election in a Tree



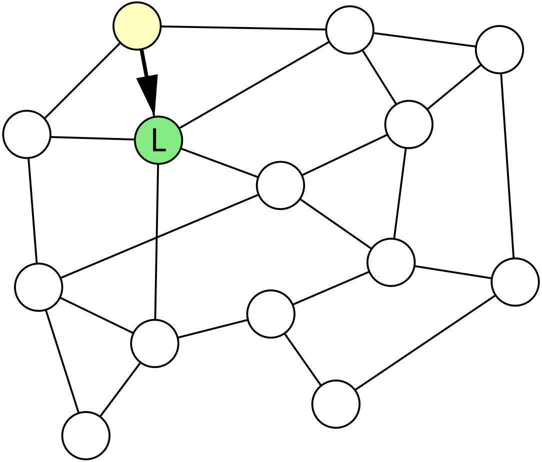
Eventually, all non-leader agents terminate, and hence the protocol is terminating.

Application: Token Circulation



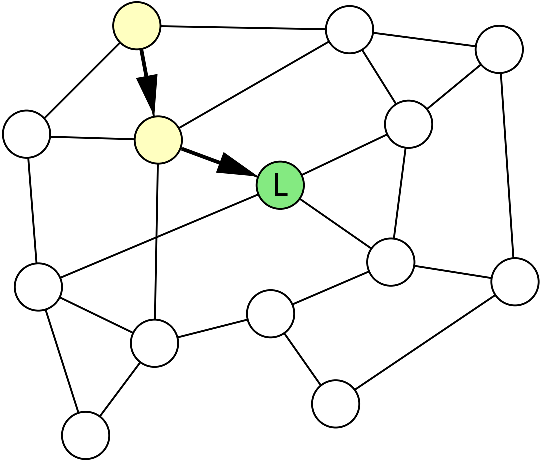
Suppose there is a unique leader and we want to make it "visit" the entire network.

Application: Token Circulation



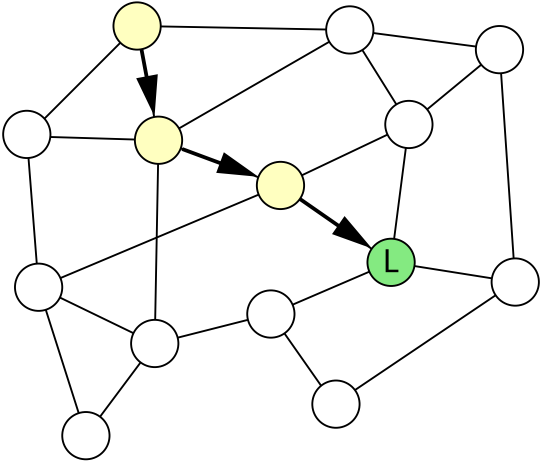
By that we mean that the leadership is “transferred” to a different agent during an interaction.

Application: Token Circulation



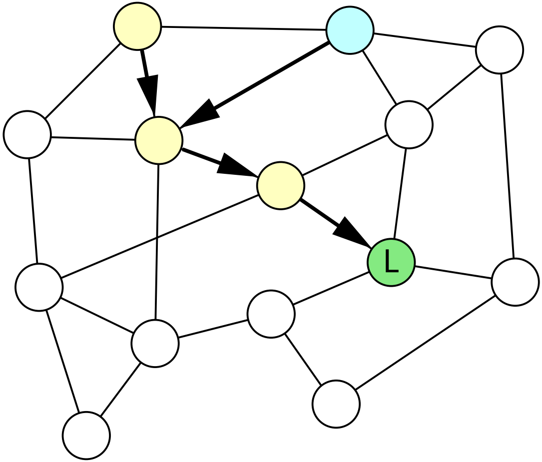
When an agent that has never been leader meets the leader, it takes the leadership.

Application: Token Circulation



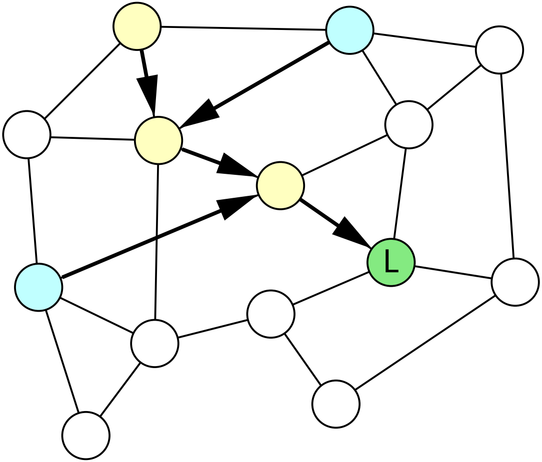
As the leader goes, it leaves a “trail” of arrows.

Application: Token Circulation



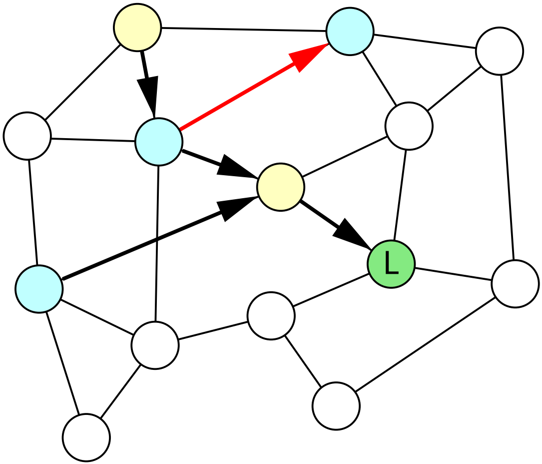
When a new agent meets an agent that has already been leader, it becomes a “summoner”.

Application: Token Circulation



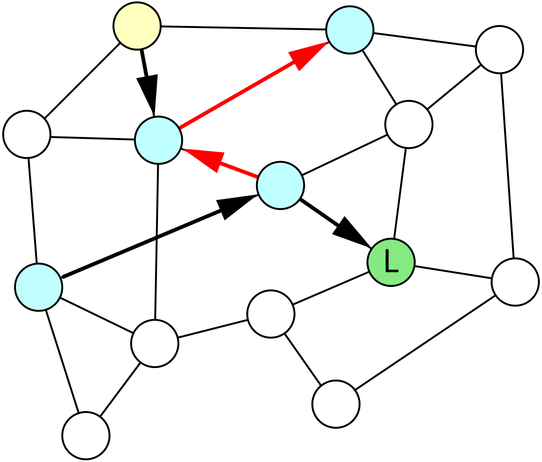
When a new agent meets an agent that has already been leader, it becomes a “summoner”.

Application: Token Circulation



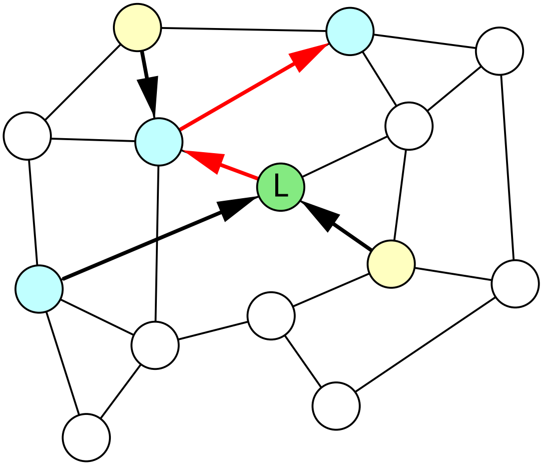
The parent of a summoner becomes a summoner as well, and reverses the corresponding arrow.

Application: Token Circulation



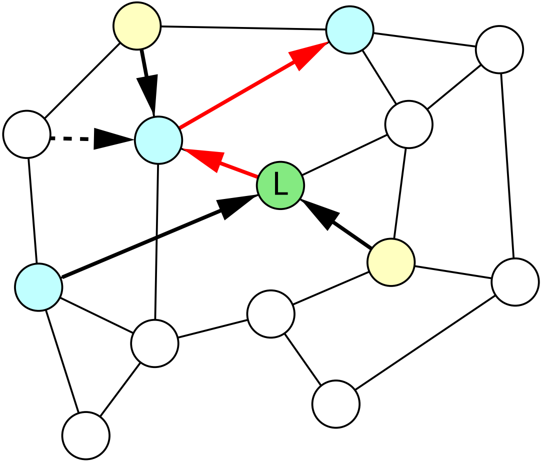
The parent of a summoner becomes a summoner as well, and reverses the corresponding arrow.

Application: Token Circulation



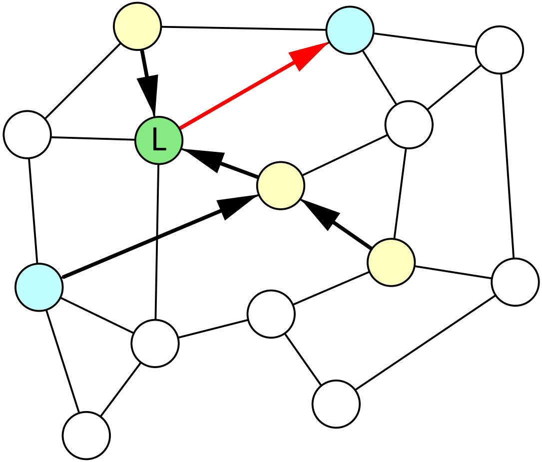
When the leader meets a summoner, it gives it the leadership.

Application: Token Circulation



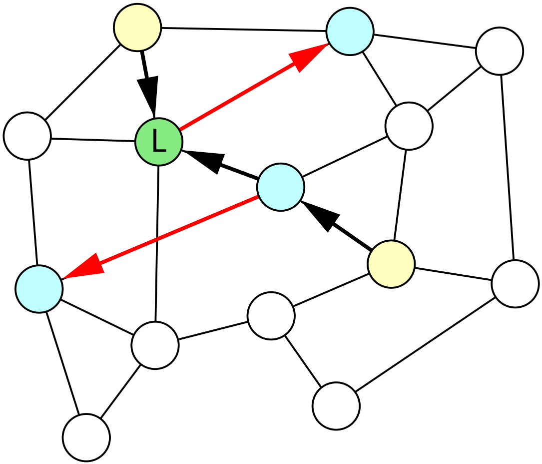
A summoner ignores all requests from new agents.

Application: Token Circulation



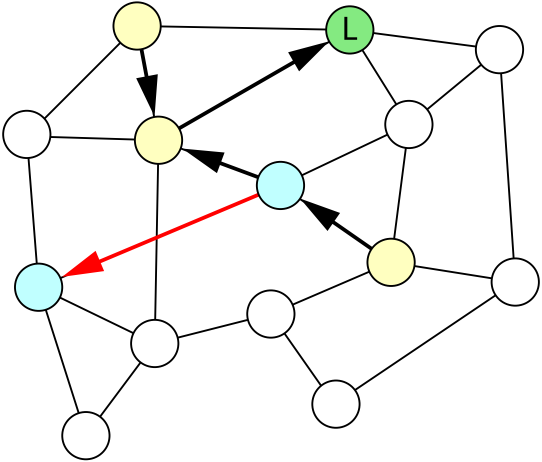
The leader keeps following the arrows through summoners, reversing them as it goes.

Application: Token Circulation



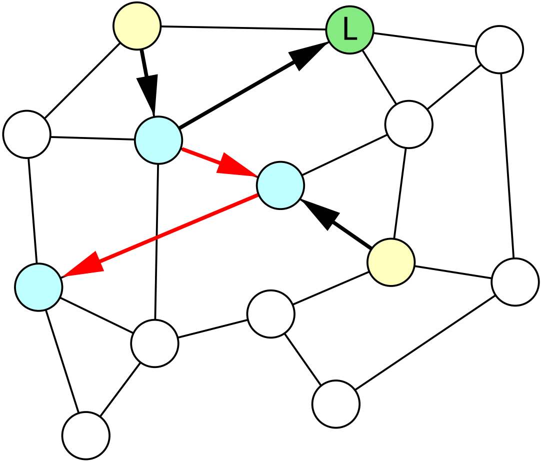
Different agents may summon the leader in parallel, but they never interfere with each other.

Application: Token Circulation



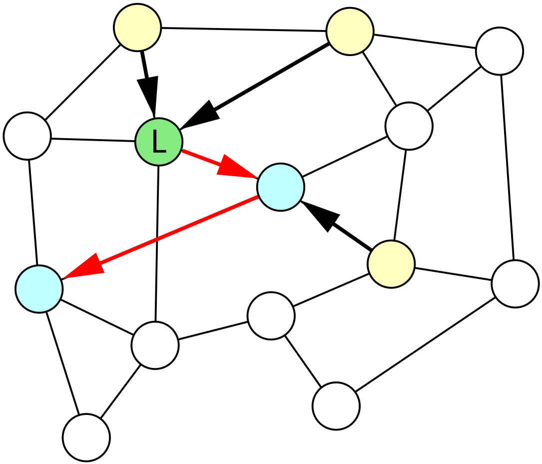
Different agents may summon the leader in parallel, but they never interfere with each other.

Application: Token Circulation



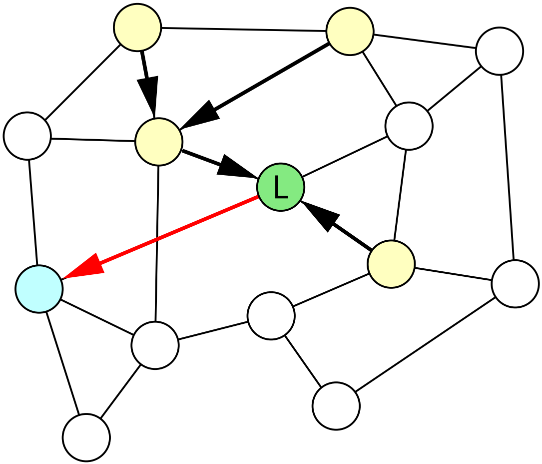
Different agents may summon the leader in parallel, but they never interfere with each other.

Application: Token Circulation



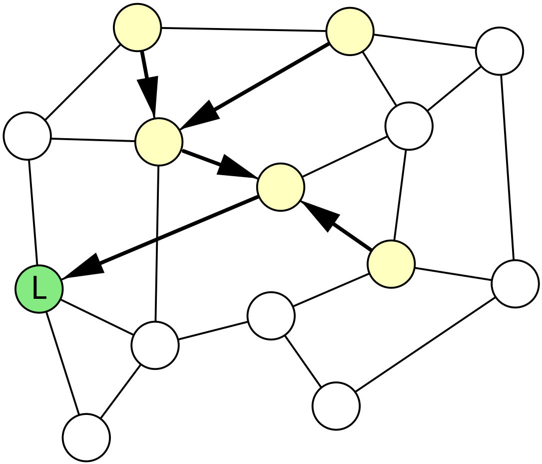
This is because all operations are performed on a subtree of the network.

Application: Token Circulation



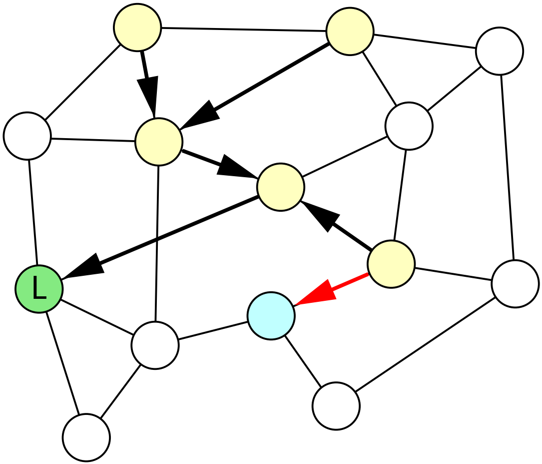
This is because all operations are performed on a subtree of the network.

Application: Token Circulation



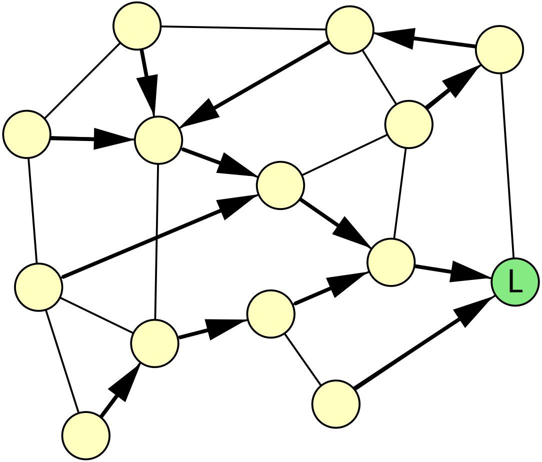
This is because all operations are performed on a subtree of the network.

Application: Token Circulation



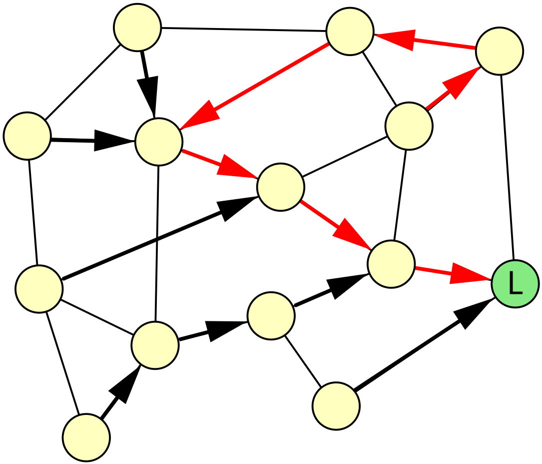
This is because all operations are performed on a subtree of the network.

Application: Token Circulation



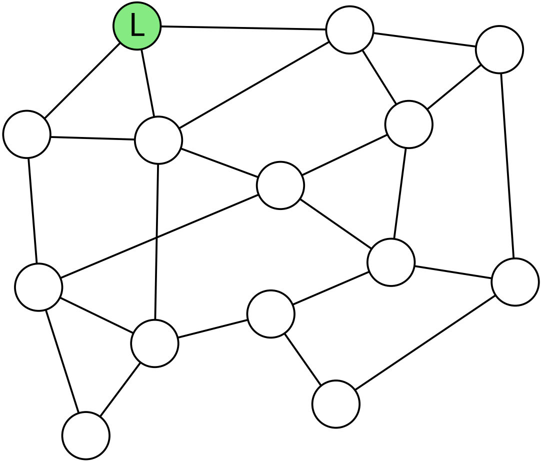
Eventually, the leader visits the entire network. As a byproduct, a rooted spanning tree has been constructed.

Application: Token Circulation

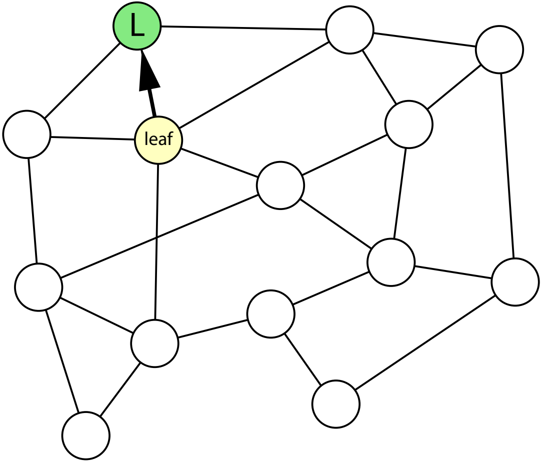


Note that this spanning tree may not be balanced.

Application: Shortest-Path Spanning Tree Construction

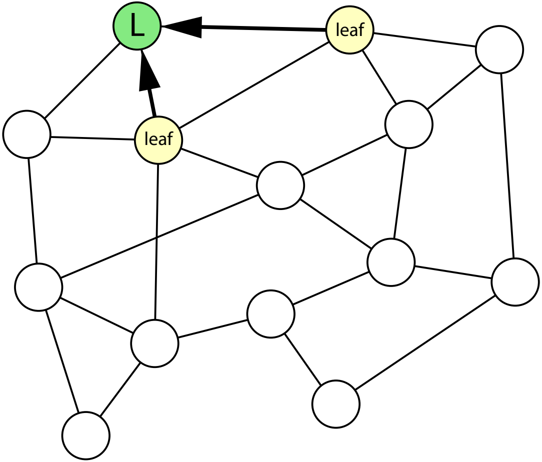


Say we want to construct a better spanning tree rooted at the leader, under the <u>k-bounded</u> scheduler.



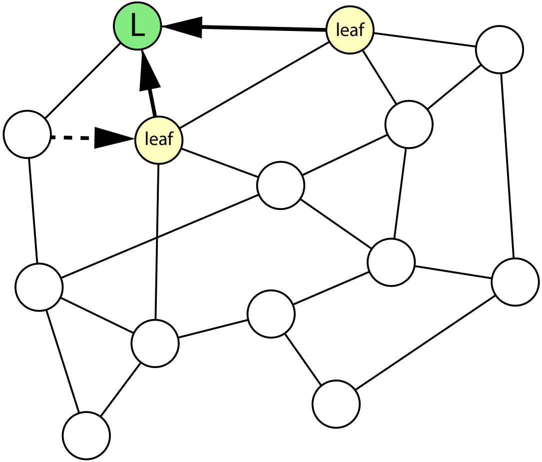
When a new agent interacts with the leader, it becomes a “leaf”.

Application: Shortest-Path Spanning Tree Construction



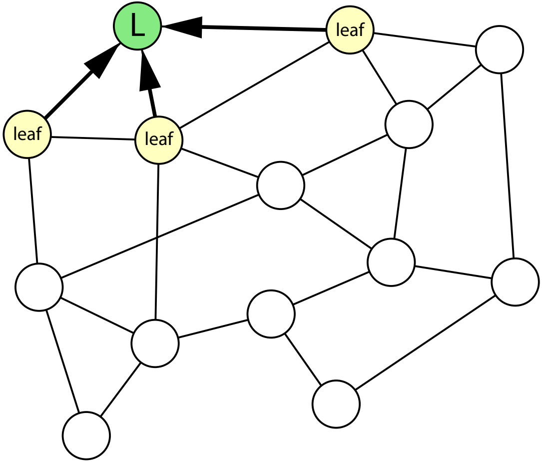
When a new agent interacts with the leader, it becomes a "leaf".

Application: Shortest-Path Spanning Tree Construction



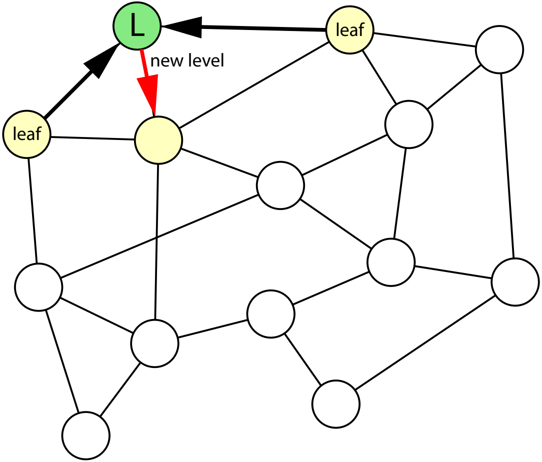
When a new agent interacts with the leader, it becomes a "leaf".

Application: Shortest-Path Spanning Tree Construction



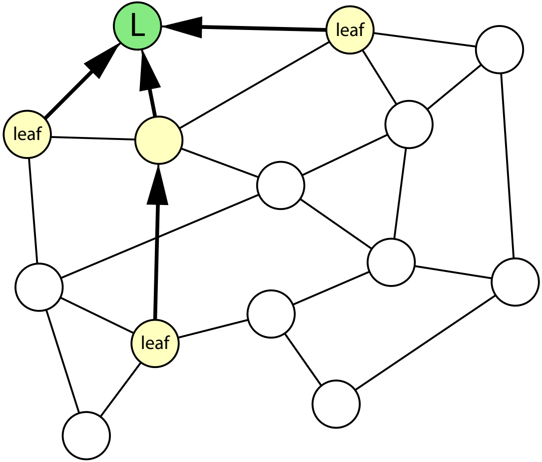
Since the scheduler is k-bounded, the leader knows when all its neighbors are leaves.

Application: Shortest-Path Spanning Tree Construction



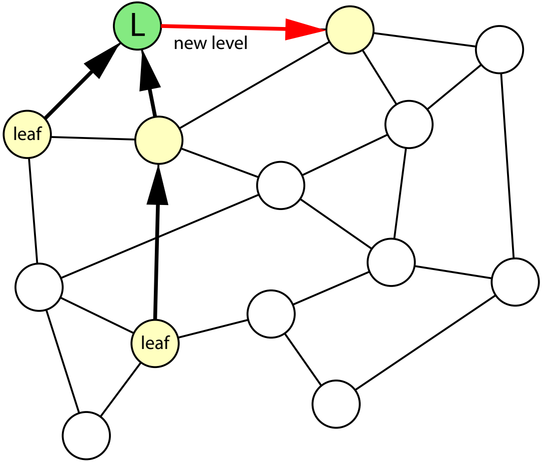
The leader issues a “new level” command along the tree.

Application: Shortest-Path Spanning Tree Construction



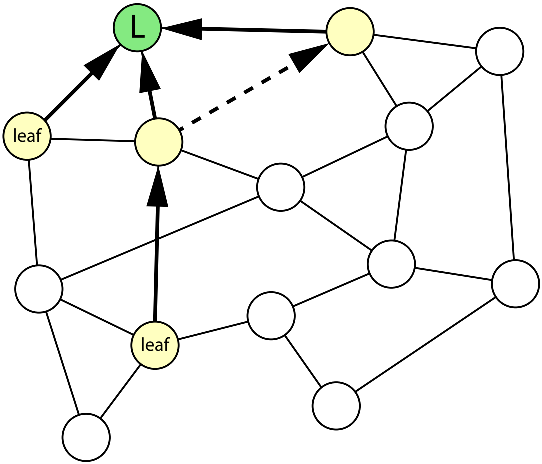
The leaves that receive a "new level" message start a new level of the spanning tree.

Application: Shortest-Path Spanning Tree Construction



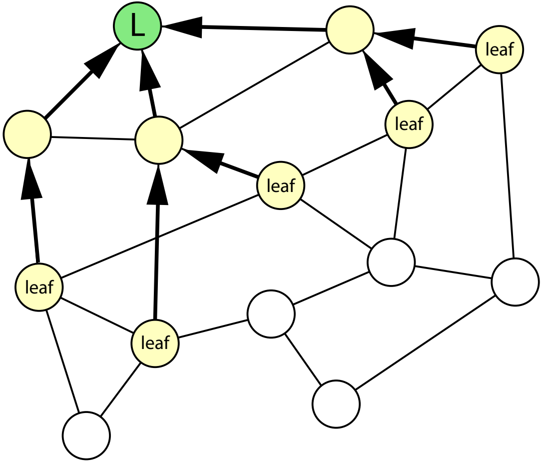
The leaves that receive a “new level” message start a new level of the spanning tree.

Application: Shortest-Path Spanning Tree Construction



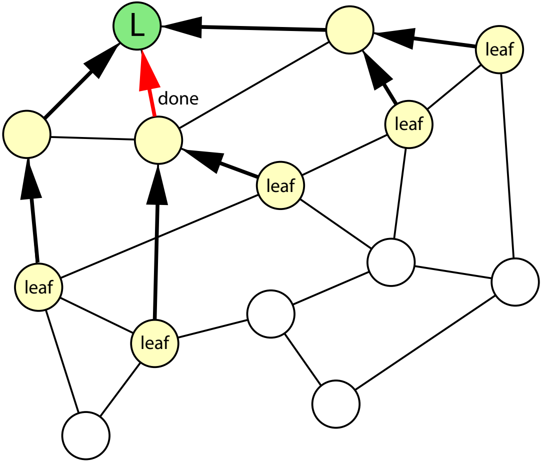
The leaves that receive a “new level” message start a new level of the spanning tree.

Application: Shortest-Path Spanning Tree Construction



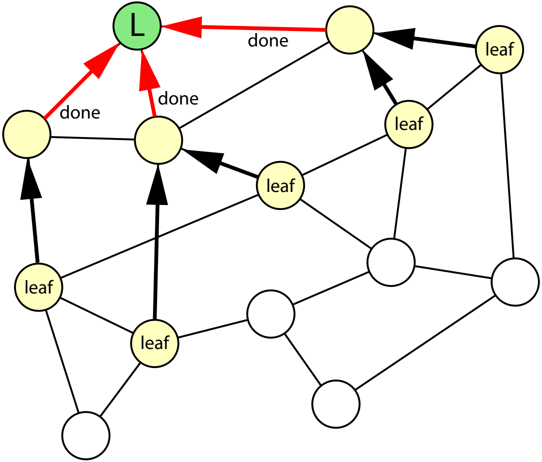
The leaves that receive a “new level” message start a new level of the spanning tree.

Application: Shortest-Path Spanning Tree Construction



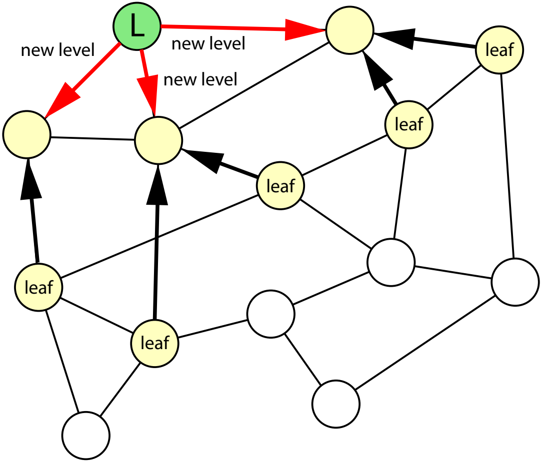
When they realize that all their neighbors have been included in the spanning tree, they send a “done” message to the leader.

Application: Shortest-Path Spanning Tree Construction



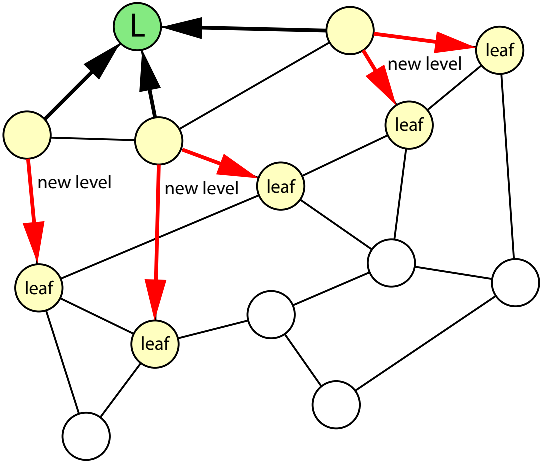
When they realize that all their neighbors have been included in the spanning tree, they send a “done” message to the leader.

Application: Shortest-Path Spanning Tree Construction



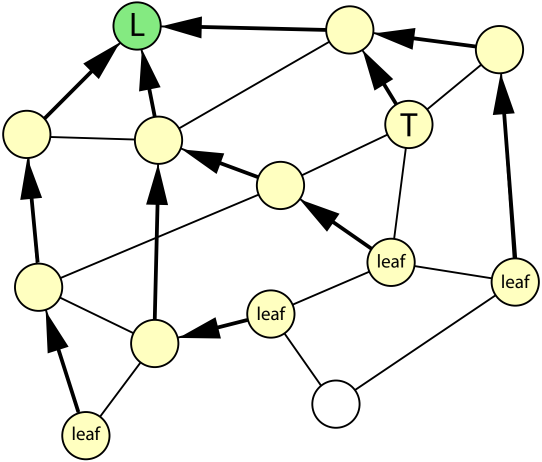
The leader issues another “new level” command, which is forwarded along the spanning tree.

Application: Shortest-Path Spanning Tree Construction



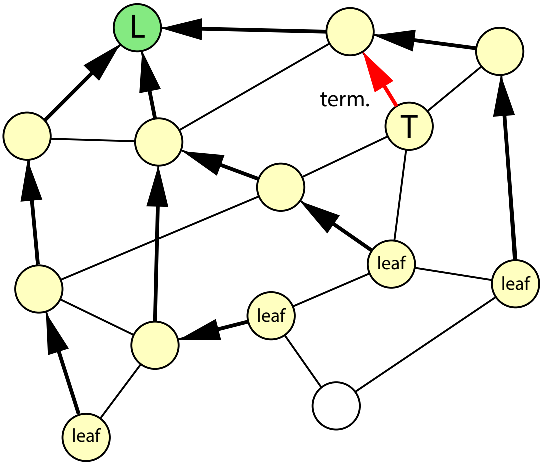
The leader issues another “new level” command, which is forwarded along the spanning tree.

Application: Shortest-Path Spanning Tree Construction



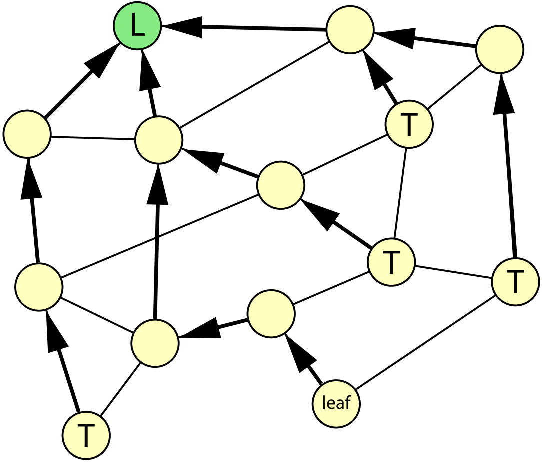
A new level of the spanning tree is constructed.

Application: Shortest-Path Spanning Tree Construction



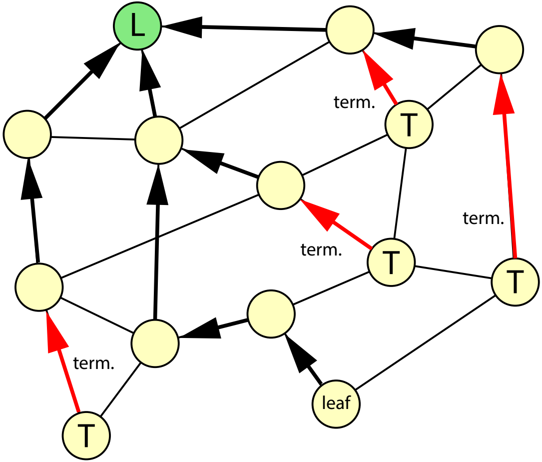
The leaves that are unable to expand assume a terminal state and send a "terminated" message toward the leader.

Application: Shortest-Path Spanning Tree Construction



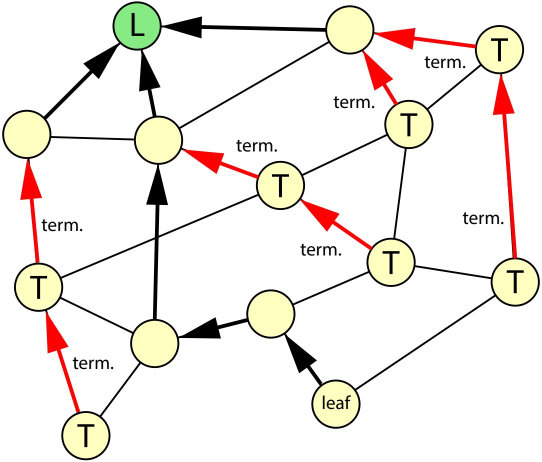
The leaves that are unable to expand assume a terminal state and send a "terminated" message toward the leader.

Application: Shortest-Path Spanning Tree Construction



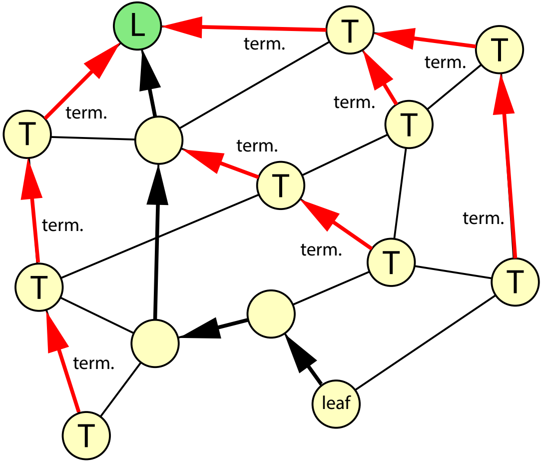
The leaves that are unable to expand assume a terminal state and send a "terminated" message toward the leader.

Application: Shortest-Path Spanning Tree Construction



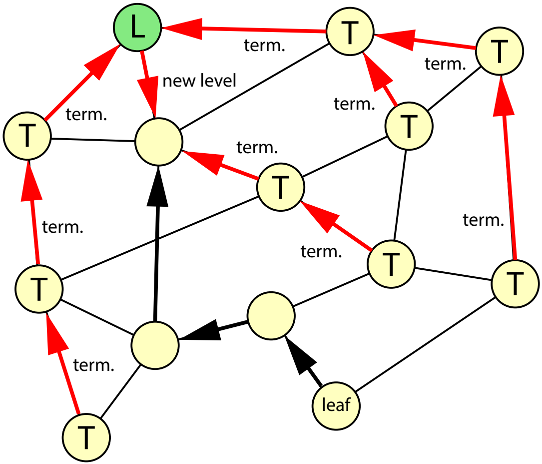
When an agent's children are all sending a "terminated" message, the agent forwards it and terminates as well.

Application: Shortest-Path Spanning Tree Construction



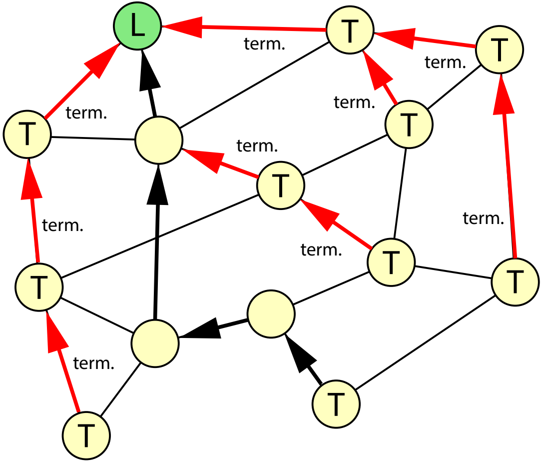
When an agent's children are all sending a "terminated" message, the agent forwards it and terminates as well.

Application: Shortest-Path Spanning Tree Construction



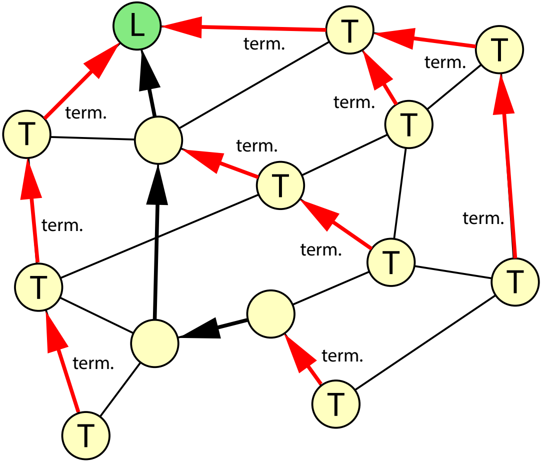
When an agent's children are all sending a "terminated" message, the agent forwards it and terminates as well.

Application: Shortest-Path Spanning Tree Construction



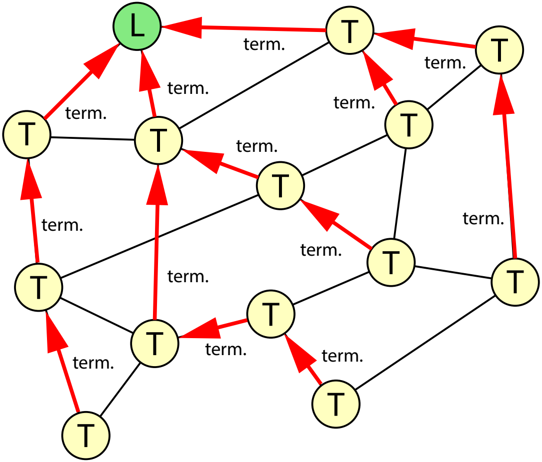
When an agent's children are all sending a “terminated” message, the agent forwards it and terminates as well.

Application: Shortest-Path Spanning Tree Construction



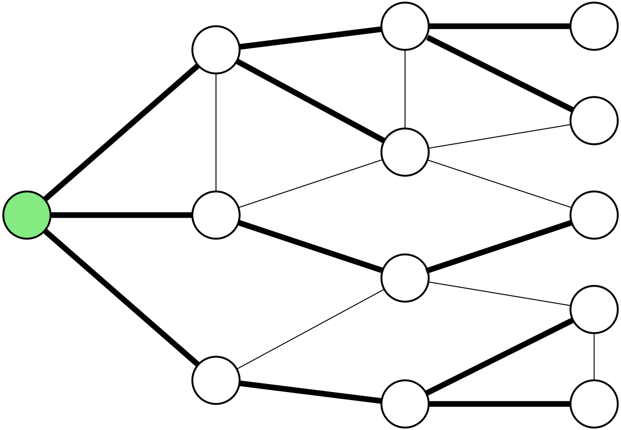
When an agent's children are all sending a "terminated" message, the agent forwards it and terminates as well.

Application: Shortest-Path Spanning Tree Construction



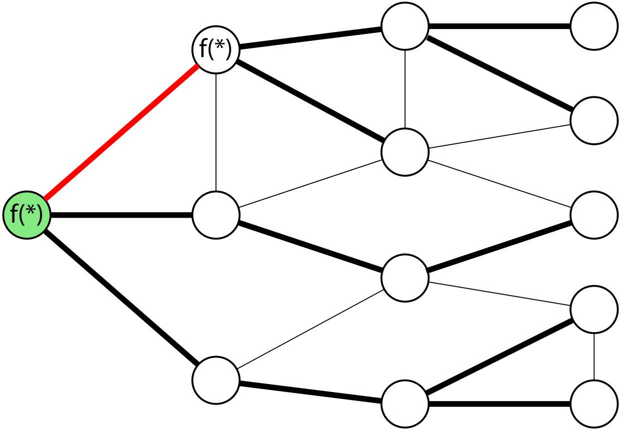
When the leader receives a "terminated" message from all its children, it terminates.

Application: Stability Detection



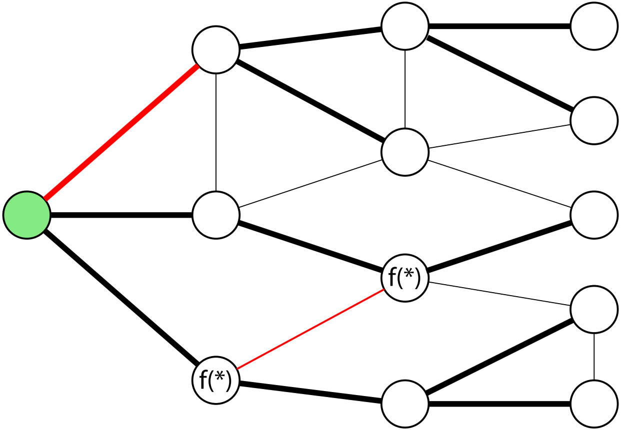
Say we have a leader and a spanning tree, and we want to detect (under the k-bounded scheduler) when a protocol P stabilizes.

Application: Stability Detection



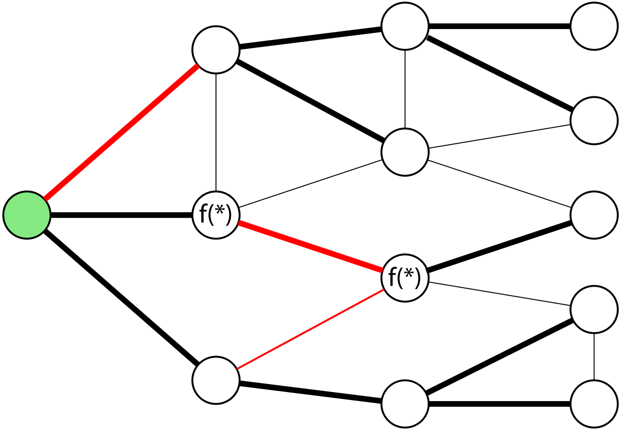
Whenever a new edge is activated, its endpoints "simulate" a transition according to P.

Application: Stability Detection



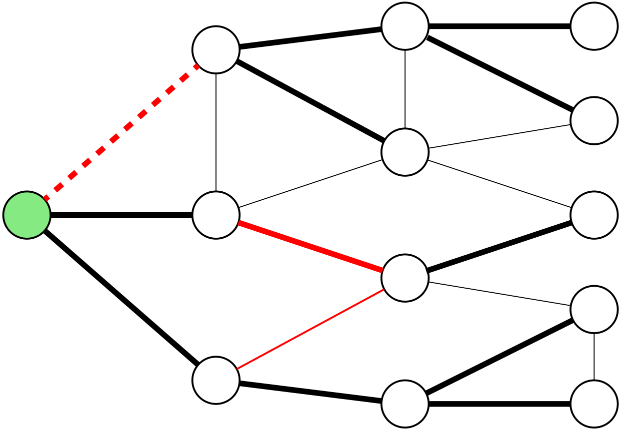
Whenever a new edge is activated, its endpoints "simulate" a transition according to P.

Application: Stability Detection



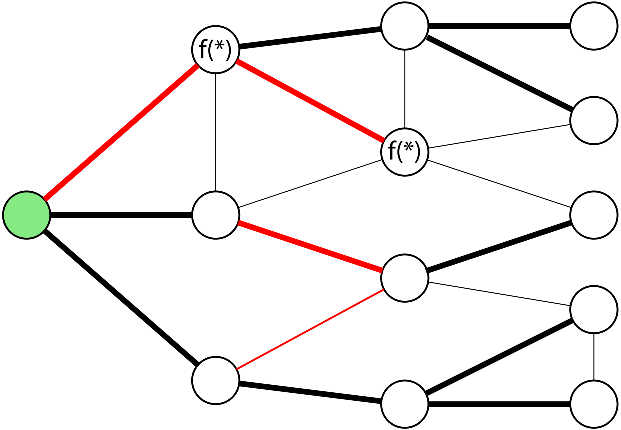
Whenever a new edge is activated, its endpoints “simulate” a transition according to P.

Application: Stability Detection



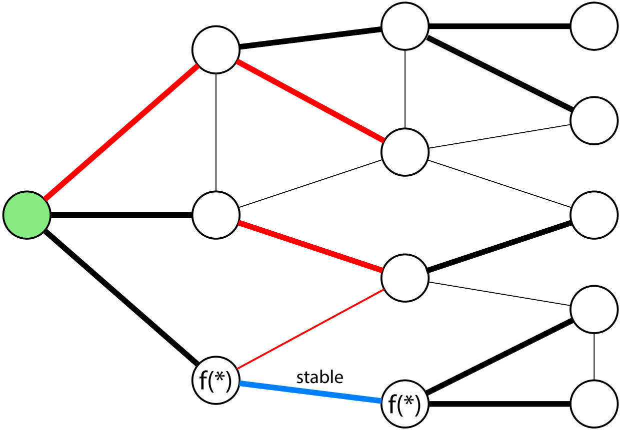
These edges are marked, so the corresponding simulated interaction does not occur twice.

Application: Stability Detection



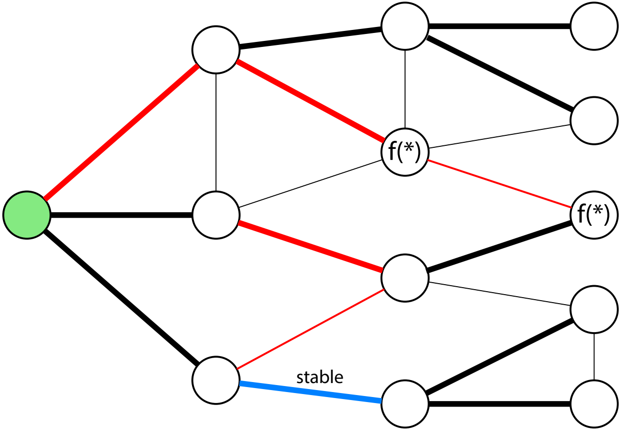
These edges are marked, so the corresponding simulated interaction does not occur twice.

Application: Stability Detection



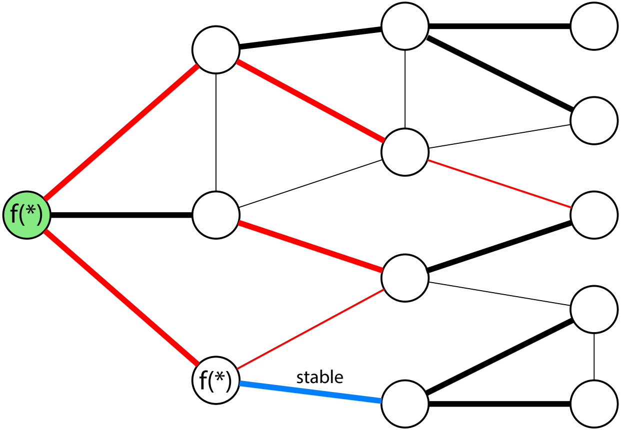
If a simulated interaction over an edge leaves the simulated states unchanged, the edge is marked as “stable”.

Application: Stability Detection



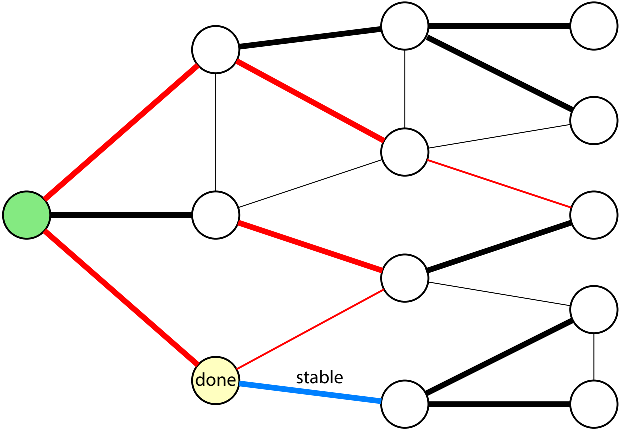
If a simulated interaction over an edge leaves the simulated states unchanged, the edge is marked as “stable”.

Application: Stability Detection



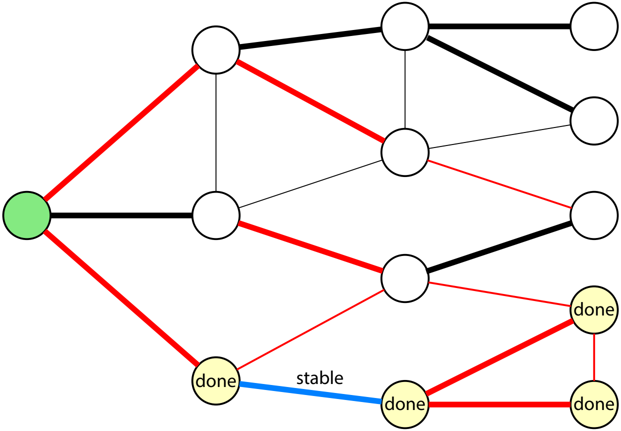
Since the scheduler is $\underline{k\text{-bounded}}$, an agent eventually realizes that it has interacted with all its neighbors.

Application: Stability Detection



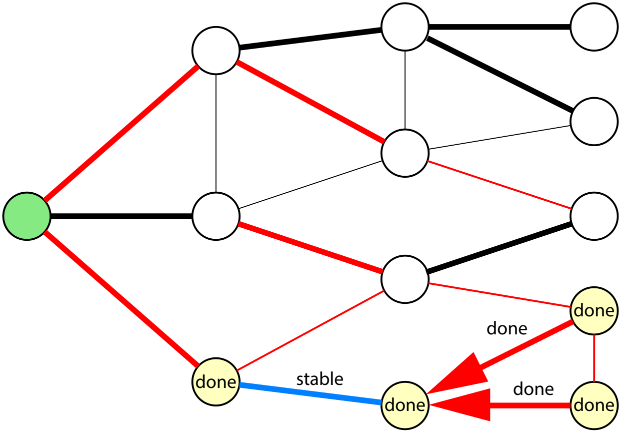
When this happens, the agent becomes “done”.

Application: Stability Detection



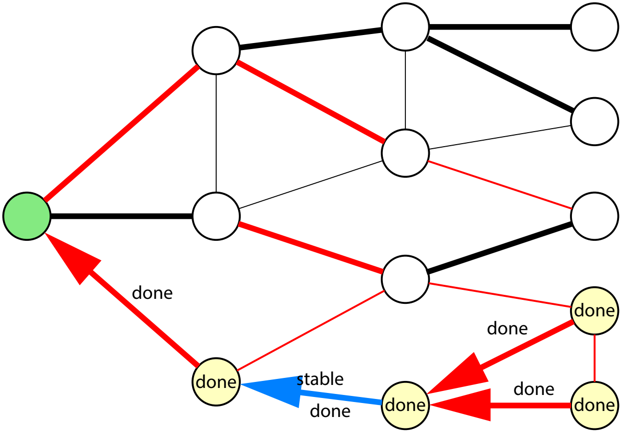
If all the children of a “done” agent (in the spanning tree) are “done”, the agent forwards a “done” message to its parent.

Application: Stability Detection



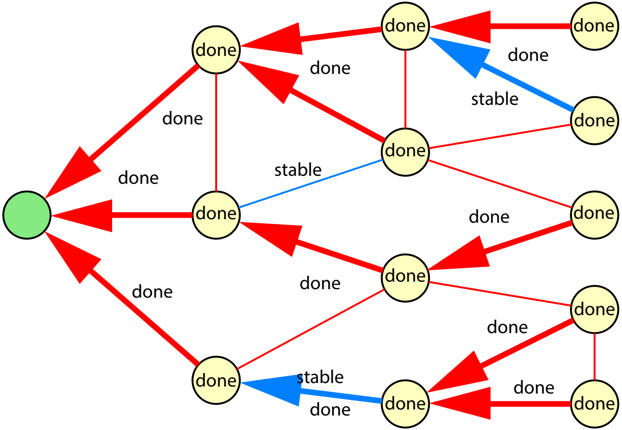
If all the children of a "done" agent (in the spanning tree) are "done", the agent forwards a "done" message to its parent.

Application: Stability Detection



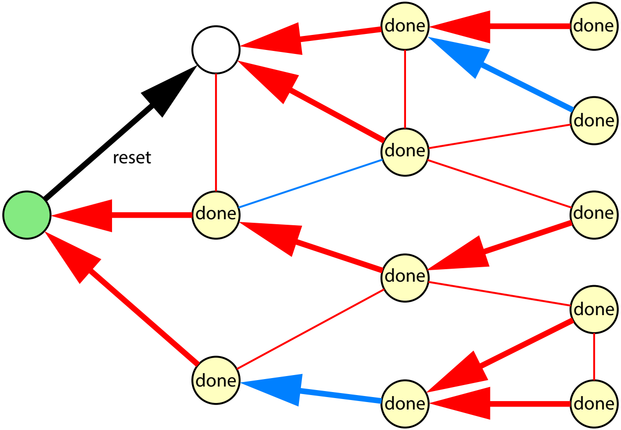
If all the children of a “done” agent (in the spanning tree) are “done”, the agent forwards a “done” message to its parent.

Application: Stability Detection



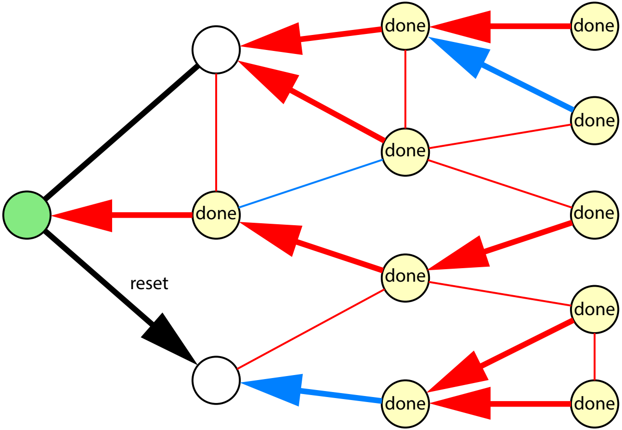
Eventually, the leader receives “done” messages from all its children.

Application: Stability Detection



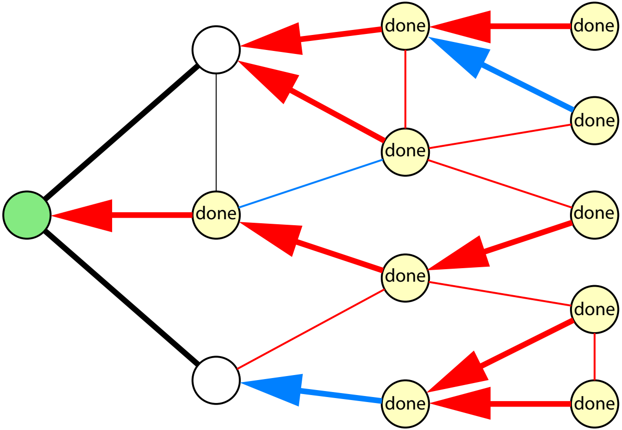
At this point, the leader broadcasts a "reset" message.

Application: Stability Detection



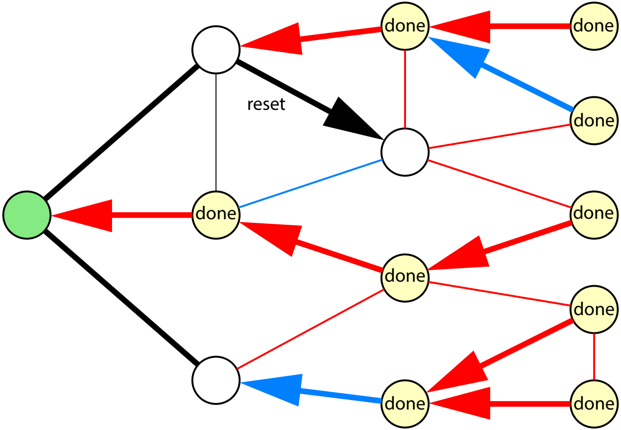
At this point, the leader broadcasts a “reset” message.

Application: Stability Detection



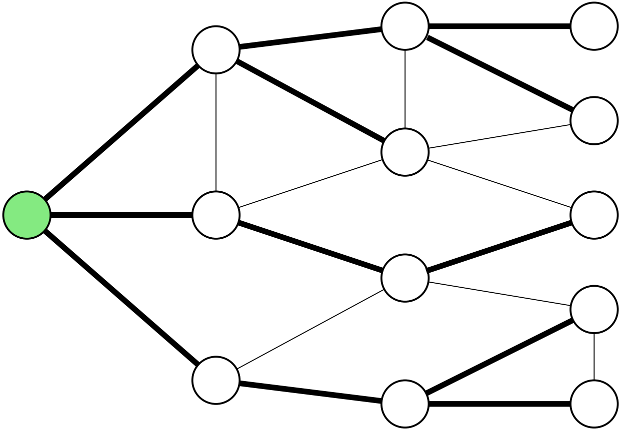
All edges that are incident to a “reset” agent become unmarked.

Application: Stability Detection



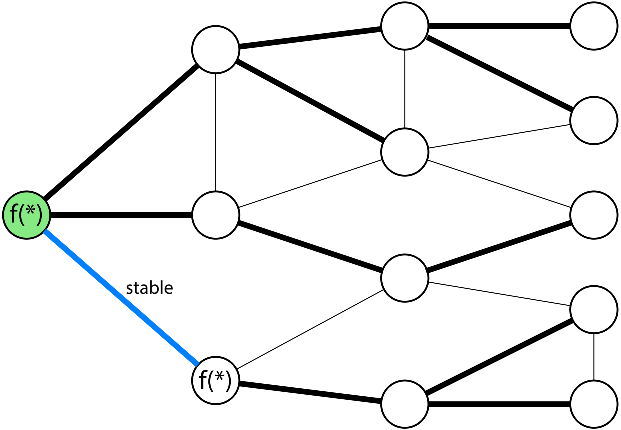
The “reset” message is forwarded along the spanning tree.

Application: Stability Detection



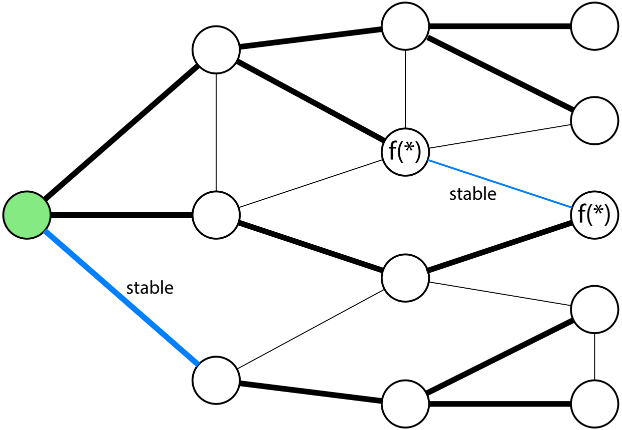
Eventually, the whole network is reset. The leader is notified, and starts a new simulation phase.

Application: Stability Detection



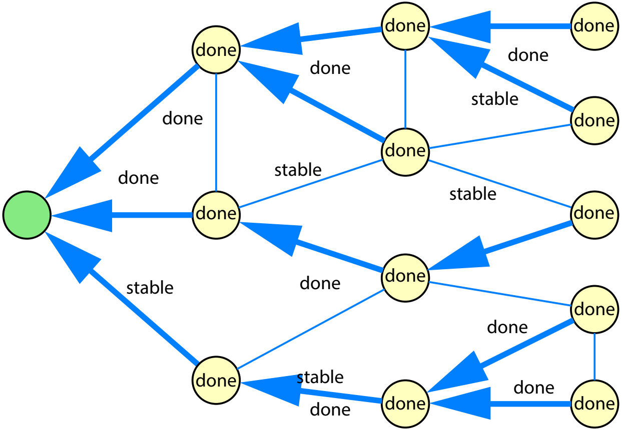
If P is stable, eventually all edges will be marked as “stable”.

Application: Stability Detection



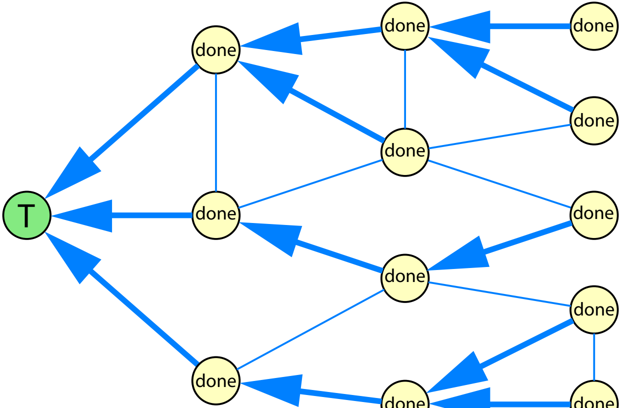
If P is stable, eventually all edges will be marked as "stable".

Application: Stability Detection



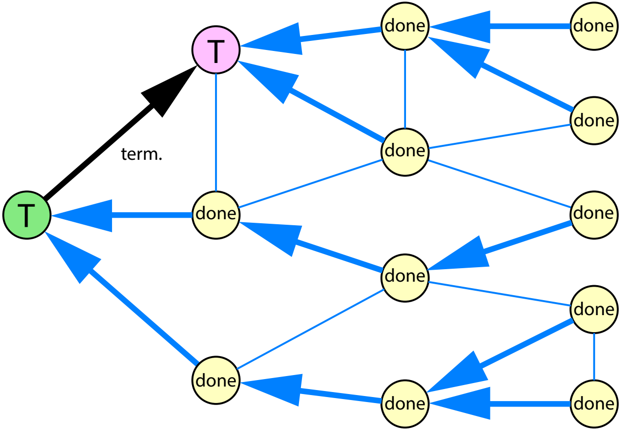
All agents send “done” and “stable” messages to their parents.

Application: Stability Detection



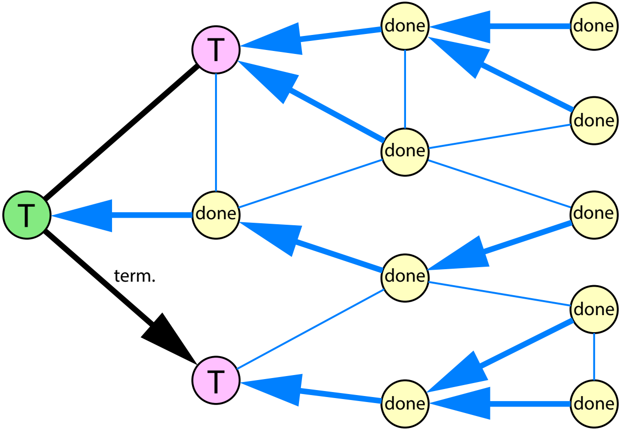
Eventually, the leader receives “done” and “stable” messages from all its children.

Application: Stability Detection



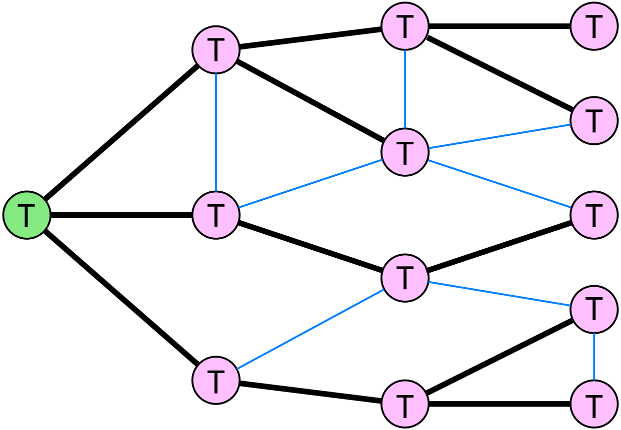
The leader then terminates and broadcasts a "terminate" message, which is forwarded along the spanning tree.

Application: Stability Detection



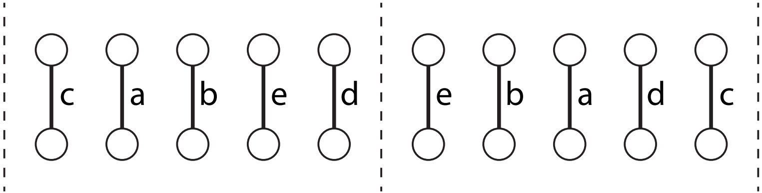
The leader then terminates and broadcasts a “terminate” message, which is forwarded along the spanning tree.

Application: Stability Detection



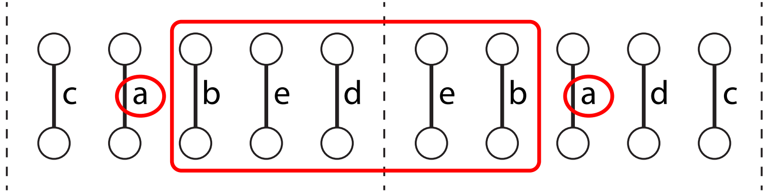
This converts the stable protocol P into a terminating one.

Application: Simulation of 2-Bounded Schedulers



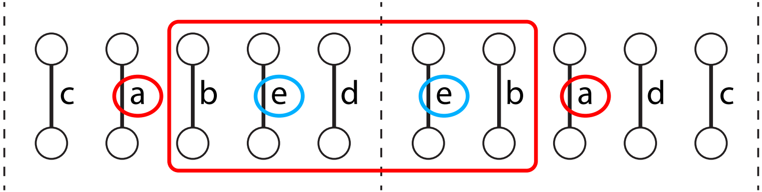
Note: The simulated schedule activates all edges of the network in some order, then it activates them again in some other order, etc.

Application: Simulation of 2-Bounded Schedulers



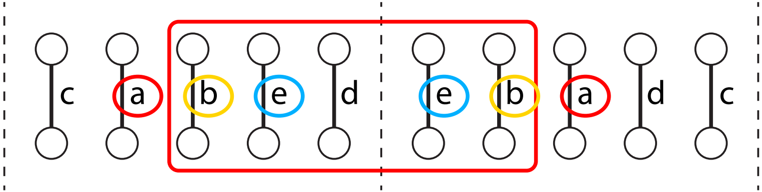
So, between two activations of an edge (say, a), each other edge is activated at most twice.

Application: Simulation of 2-Bounded Schedulers



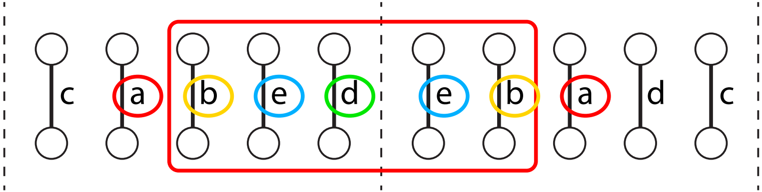
So, between two activations of an edge (say, a), each other edge is activated at most twice.

Application: Simulation of 2-Bounded Schedulers



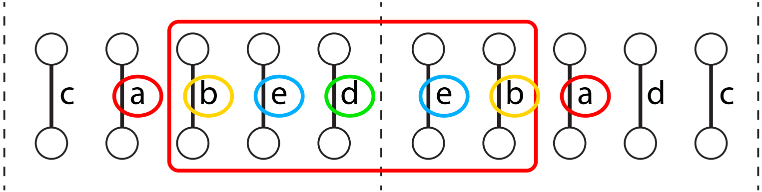
So, between two activations of an edge (say, a), each other edge is activated at most twice.

Application: Simulation of 2-Bounded Schedulers



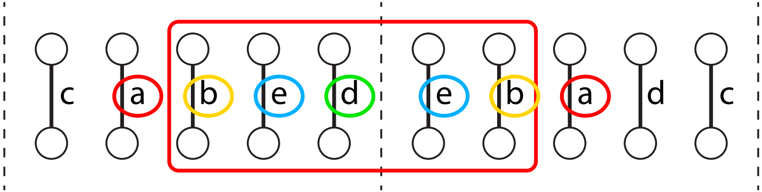
It follows that the simulated schedule is $\underline{2\text{-bounded}}$.

Application: Simulation of 2-Bounded Schedulers



So, the protocols that work under the 2-bounded scheduler also work under all k-bounded schedulers, for all $k > 2$.

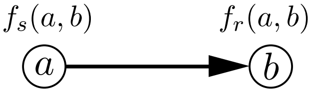
Application: Simulation of 2-Bounded Schedulers

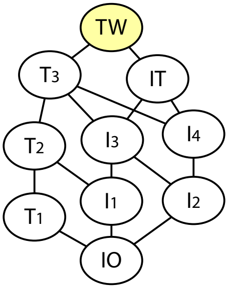


Theorem: in every network where a leader can be elected, the k-bounded schedulers are all equivalent, for $k > 1$.

One-Way and Faulty Models

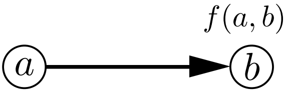
One-way models and omission faults

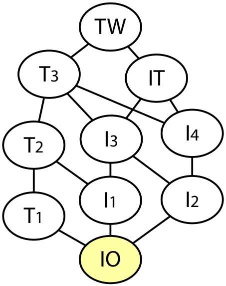




The traditional interaction model is called **Two-Way**.

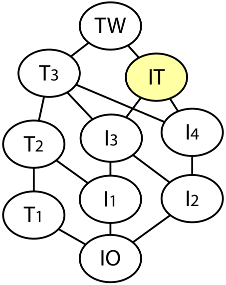
One-way models and omission faults





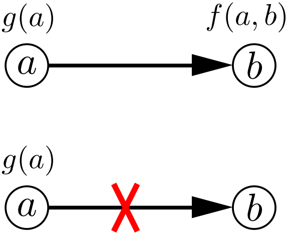
Immediate Observation: only the second agent transitions.

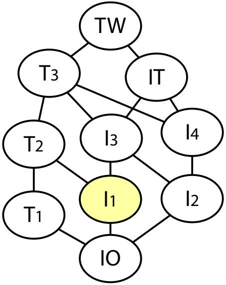
One-way models and omission faults



Immediate Transmission: the first agent detects *proximity*.

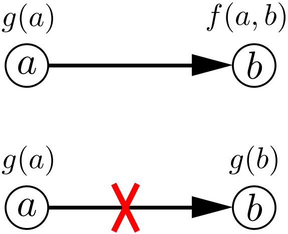
One-way models and omission faults

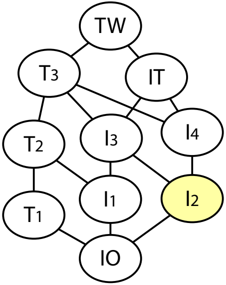




I_1: IT with omission faults, no detection.

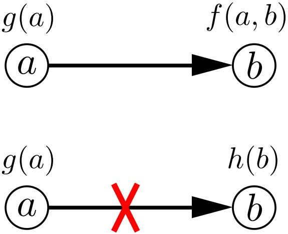
One-way models and omission faults

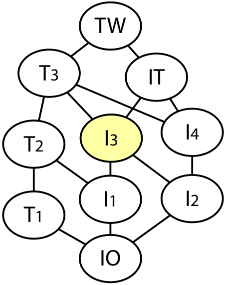




I$_2$: IT with omission faults, proximity detection.

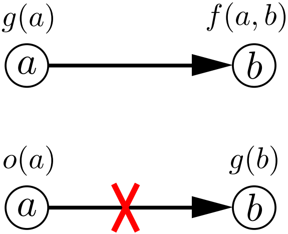
One-way models and omission faults

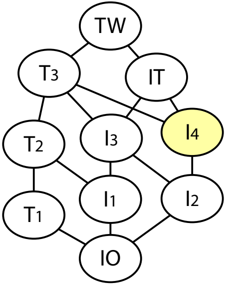




I_3: IT with omission faults, reactor-side omission detection.

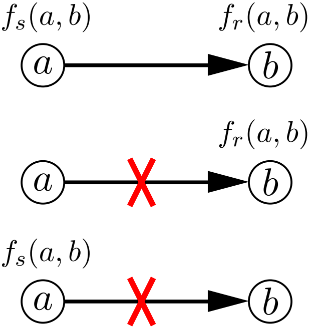
One-way models and omission faults

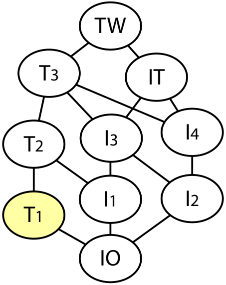




I_4: IT with omission faults, starter-side omission detection.

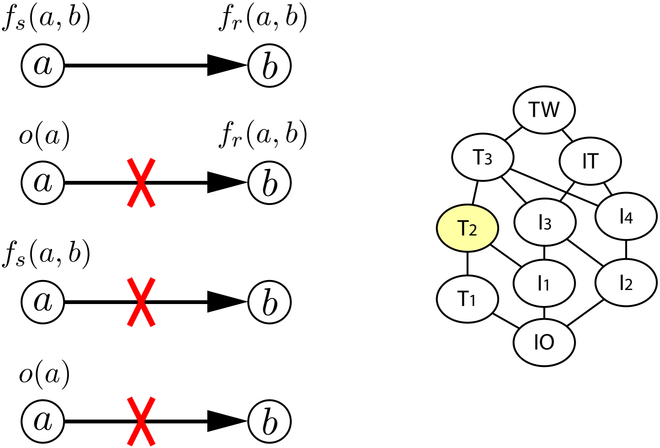
One-way models and omission faults





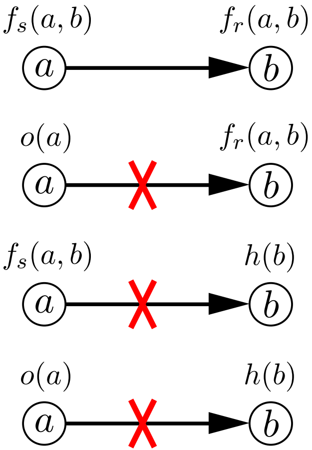
T_1: TW with omission faults, no detection.

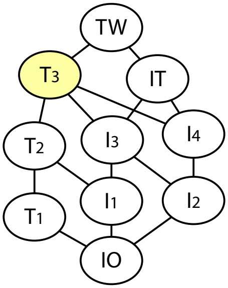
One-way models and omission faults



T_2: TW with omission faults, starter-side omission detection.

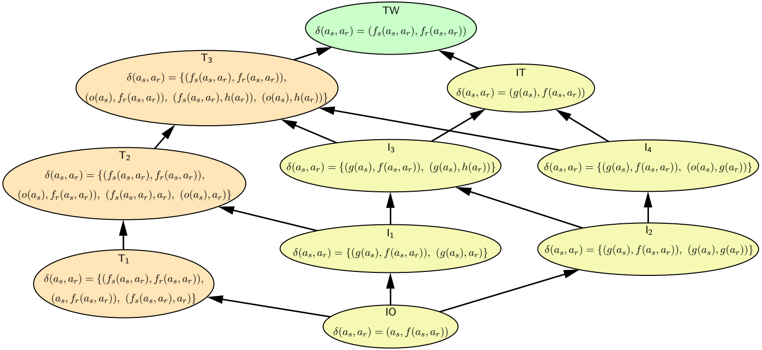
One-way models and omission faults





T_3: TW with omission faults, omission detection by both sides.

One-way models and omission faults

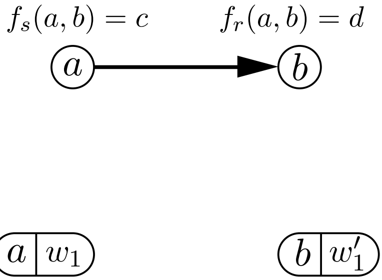


Theorem: all possible models obtained by combining one-way and two-way interactions with omission detection and proximity detection, starter-side or reactor-side, fall into one of these classes.

Simulating TW protocols with weaker ones

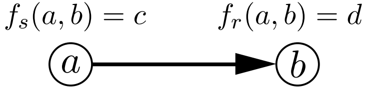
We seek to *simulate* two-way interactions in weaker models.

Simulating TW protocols with weaker ones



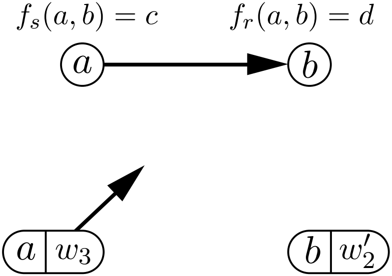
The simulating agents have a *simulated state* and a *work state*.

Simulating TW protocols with weaker ones



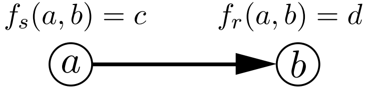
Typically, an interaction determines a change in the work state.

Simulating TW protocols with weaker ones



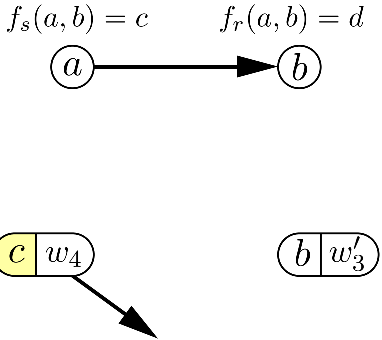
Typically, an interaction determines a change in the work state.

Simulating TW protocols with weaker ones



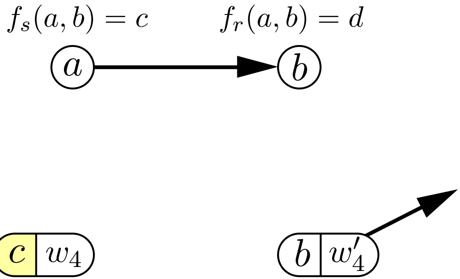
Typically, an interaction determines a change in the work state.

Simulating TW protocols with weaker ones



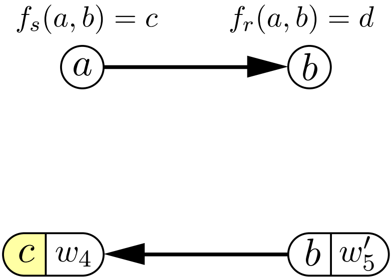
Occasionally, changes in the simulated state may occur.

Simulating TW protocols with weaker ones



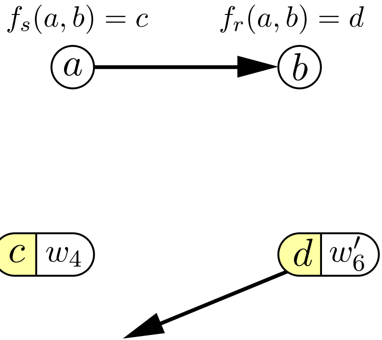
Occasionally, changes in the simulated state may occur.

Simulating TW protocols with weaker ones



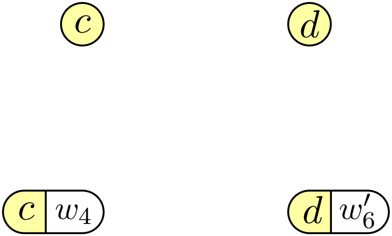
Occasionally, changes in the simulated state may occur.

Simulating TW protocols with weaker ones



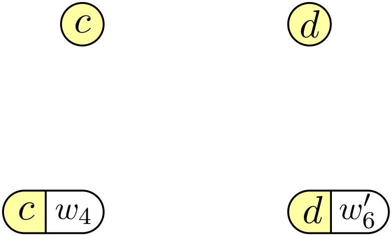
These have to mimic transitions in the simulated TW protocol.

Simulating TW protocols with weaker ones



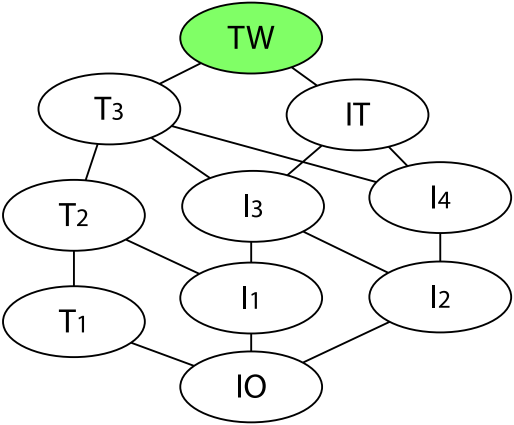
Globally, we want to pair up simulated states transitions...

Simulating TW protocols with weaker ones



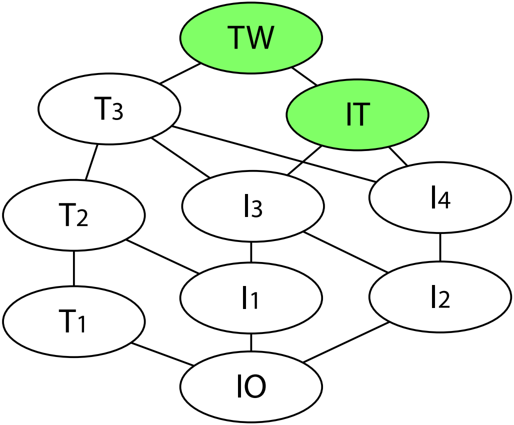
...in a way that is compatible with the simulated TW protocol.

Results: infinite memory



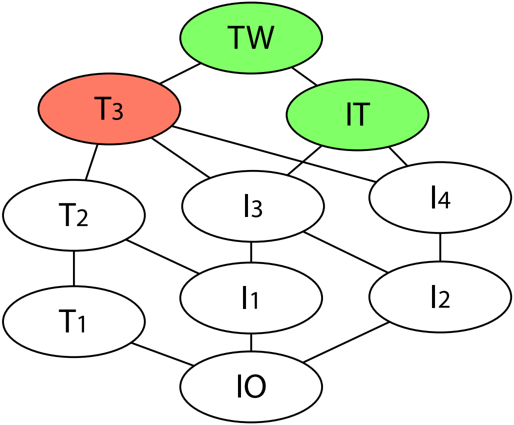
Suppose the simulating agents have **infinite memory**: what models can simulate *all* TW population protocols?

Results: infinite memory



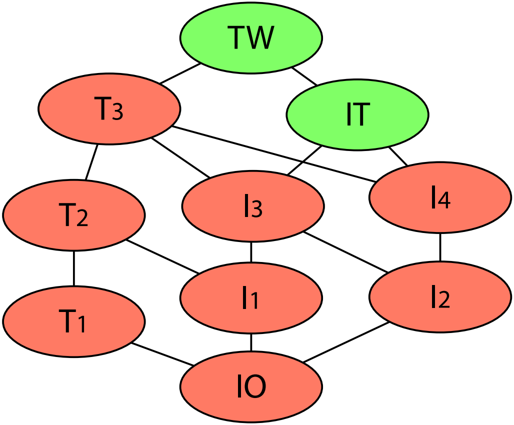
In **IT**, we can implement a *token-passing* technique that can be used to simulate two-way interactions.

Results: infinite memory



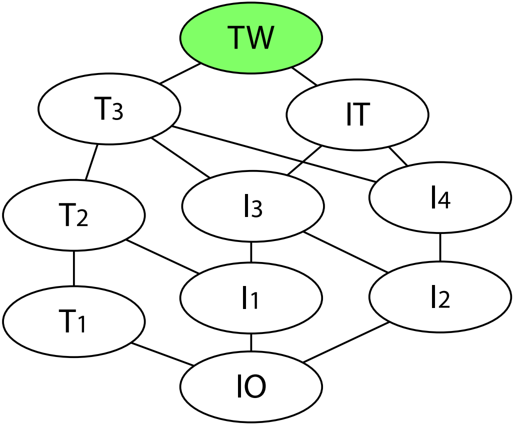
In $\mathbf{T_3}$, it is impossible to simulate a two-way protocol for the *pairing problem*.

Results: infinite memory



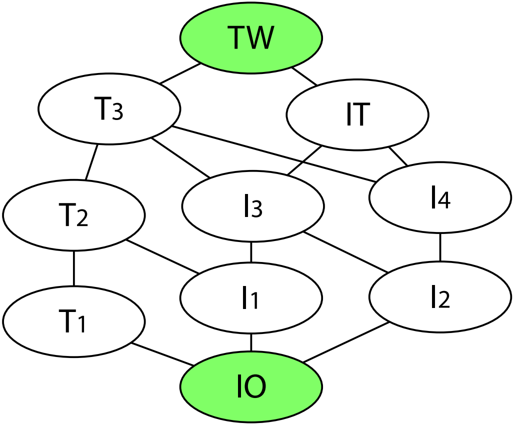
As a consequence, simulation is impossible also in the weaker interaction models.

Results: unique IDs



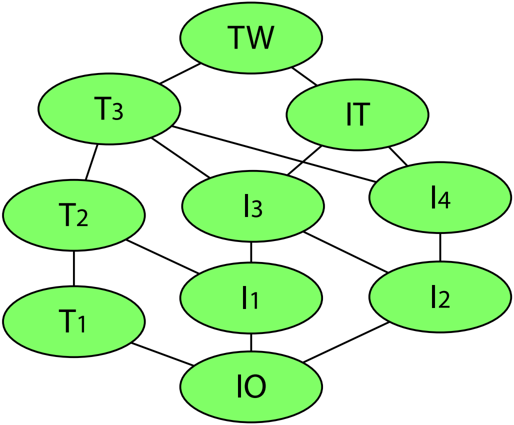
Suppose the simulating agents have *unique IDs* as part of their initial state.

Results: unique IDs



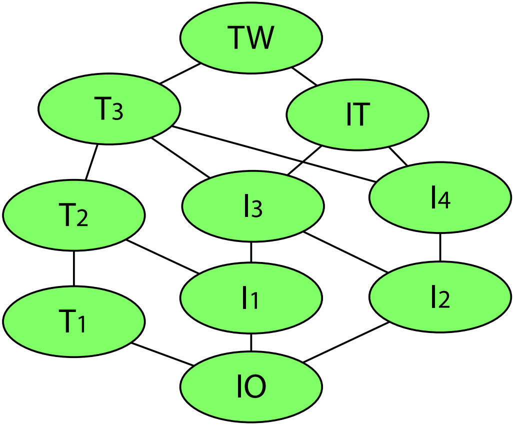
In **IO**, we can implement a *locking mechanism*, along with a *rollback process* to avoid deadlocks.

Results: unique IDs



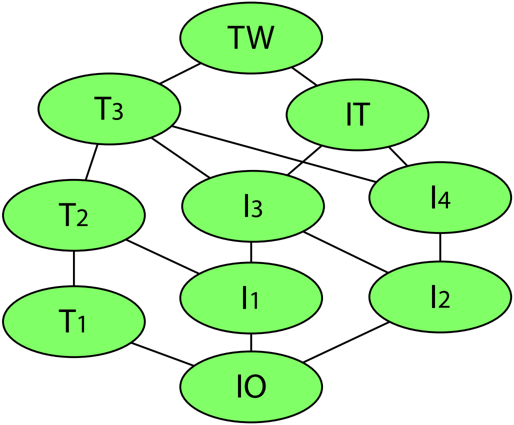
As a consequence, simulation is possible also in the stronger interaction models.

Results: knowledge of n



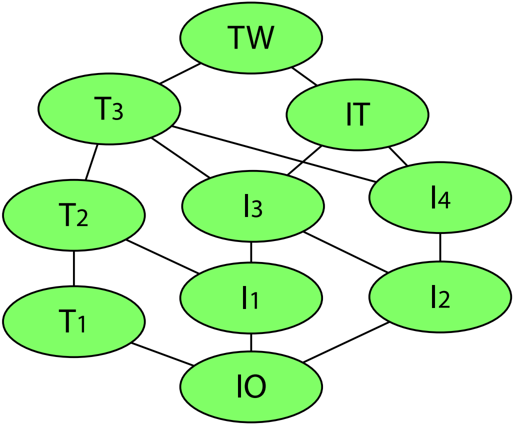
Suppose the simulating agents know the size of the system, n, and have $O(\log n)$ bits of internal memory.

Results: knowledge of n



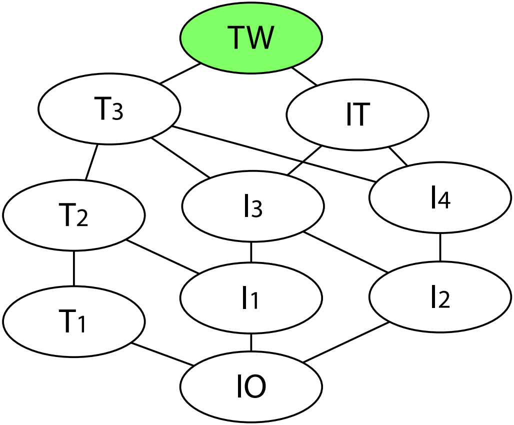
In **IO**, we can implement a *naming algorithm* that eventually gives each agent a unique ID.

Results: knowledge of n



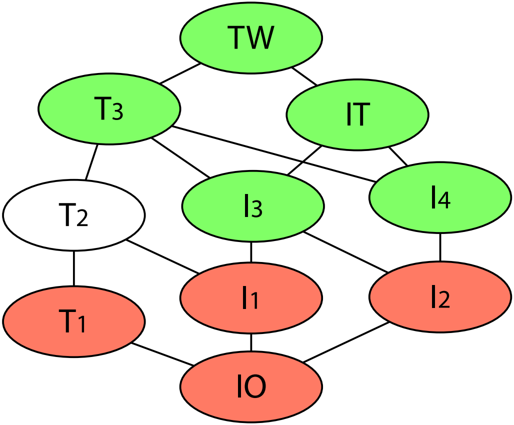
When an agent has ID n, the system starts executing the previous unique-ID simulation protocol.

Results: knowledge on omissions



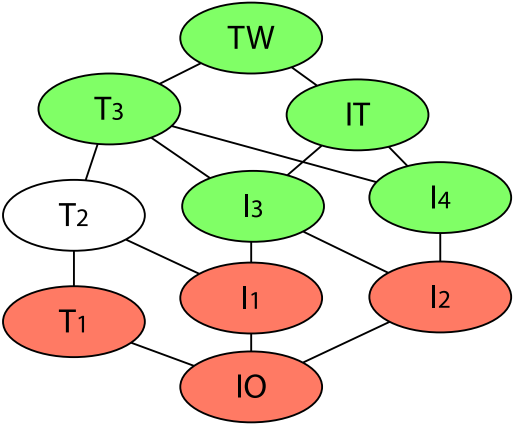
Suppose that the simulating agents are given an upper bound b on the number of faulty interactions in the system.

Results: knowledge on omissions



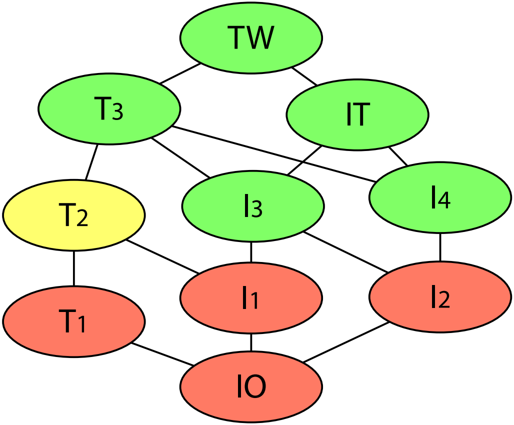
In $\mathbf{I_3}$ and $\mathbf{I_4}$, we can extend the token-passing technique by splitting each token into $b+1$ parts.

Results: knowledge on omissions



In $\mathbf{T_1}$, $\mathbf{I_1}$, and $\mathbf{I_2}$, it is impossible to simulate the pairing protocol, even for $b = 1$.

Results: knowledge on omissions



Open problem: is it possible to simulate all TW protocols in $\mathbf{T_2}$, given an upper bound on the number of faulty interactions?