Population Protocols

James Aspnes
Yale University

January 29th, 2007

Acknowledgments

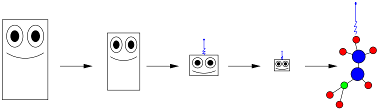
Joint work with:

- Dana Angluin (Yale)
- Melody Chan (Princeton)
- Zoë Diamadi (McKinsey & Company)
- David Eisenstat (Princeton)
- Michael J. Fischer (Yale)
- Hong Jiang (Yale)
- René Peralta (NIST)
- Eric Ruppert (York)

The past and future of computing

Economics of mass production push computer systems toward **large numbers** of **very limited** standardized components:

- Centralized systems
- Distributed systems
- Wireless distributed systems
- Sensor networks/RFID chips
- Smart molecules?

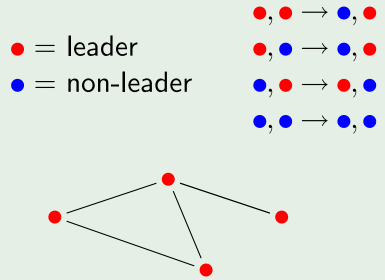


Our goal: take the limit of this process.

Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

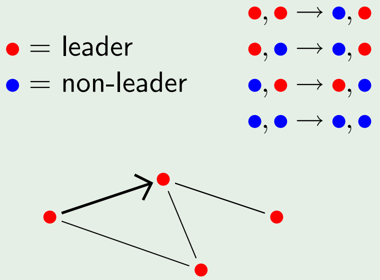
Leader Election



Population protocols

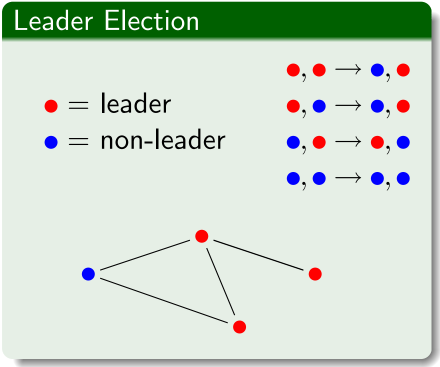
- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

Leader Election



Population protocols

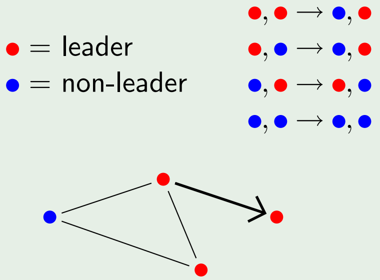
- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

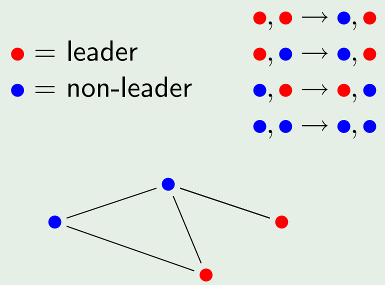
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

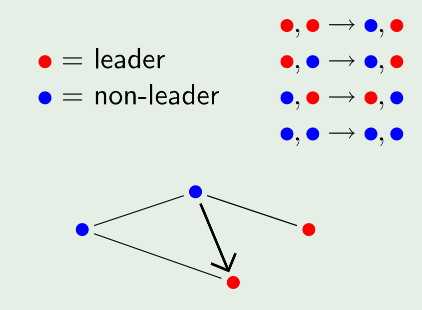
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

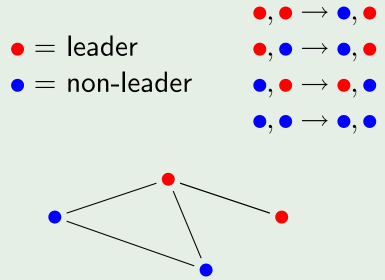
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

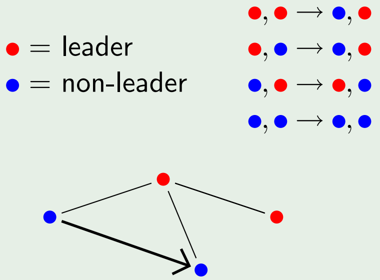
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

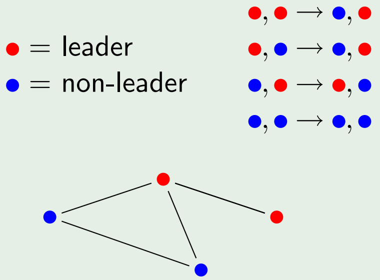
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

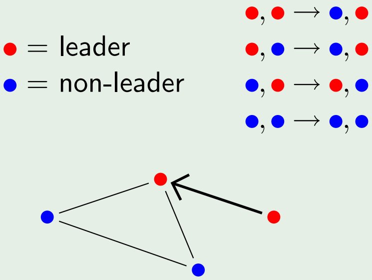
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

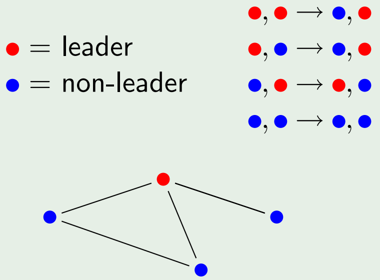
Leader Election



Population protocols

- A **population protocol** (Angluin, Aspnes, Diamadi, Fischer, and Peralta, PODC 2004) consists of a collection of **finite-state agents** organized in an **interaction graph**.
- An **interaction** between two neighbors updates the state of *both agents* according to a joint **transition function**.
- Interactions are *asymmetric*: one agent is the **initiator** and one the **responder**.

Leader Election



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

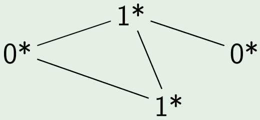
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

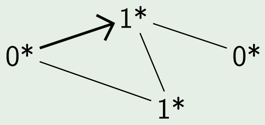
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

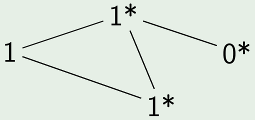
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

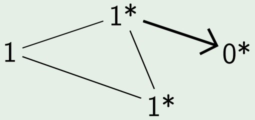
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

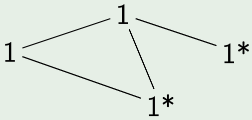
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

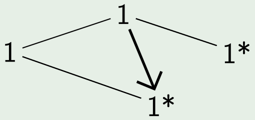
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

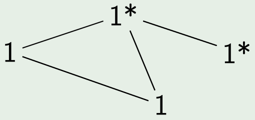
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

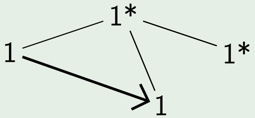
- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In: $x \to x*$

Out: $x \to x$, $x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

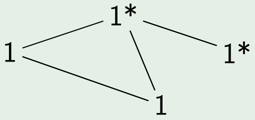
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

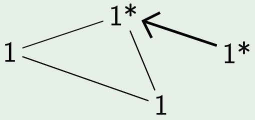
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

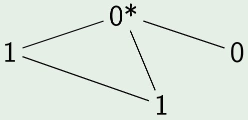
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

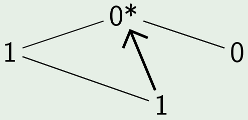
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

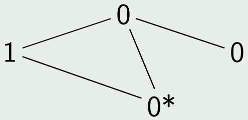
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

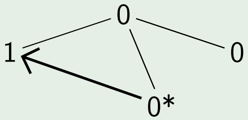
$x \to x*$

Out:

$x \to x$

$x* \to x$

$$0*, 0* \to 0, 0*$$
$$0*, 1* \to 1, 1*$$
$$1*, 0* \to 1, 1*$$
$$1*, 1* \to 0, 0*$$
$$x, y* \to y*, y$$
$$x*, y \to x, x*$$



Stable computations

- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- A **stable computation** converges to the same output at all agents.
- **Fairness condition** enforces that any reachable state is eventually reached.

Parity

In:

$x \to x*$

Out:

$x \to x$

$x* \to x$

$0*, 0* \to 0, 0*$

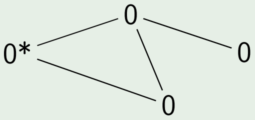
$0*, 1* \to 1, 1*$

$1*, 0* \to 1, 1*$

$1*, 1* \to 0, 0*$

$x, y* \to y*, y$

$x*, y \to x, x*$



Presburger predicates

- Trick: represent numbers by tokens scattered across the population.
- Population protocols on connected graphs can **stably compute** all of **first-order Presburger arithmetic** on counts of input tokens, including
  - Addition.
  - Subtraction.
  - Multiplication by a constant k.
  - Remainder mod k.
  - $>$, $<$, and $=$.
  - $\wedge$, $\vee$, $\neg$, $\forall x$, and $\exists x$, applied to above.
- Example: "Are there at least twice as many cold sensors as hot sensors?"

Presburger predicates (continued)

- Computable for fixed inputs (Angluin et al., PODC 2004)
- Computable if inputs converge after some finite time (Angluin, Aspnes, Chan, Fischer, Jiang, and Peralta, DCOSS 2005).
- Computable with one-way communication (Angluin, Aspnes, Eisenstat, Ruppert, OPODIS 2005).
- Computable if a small number of agents fail (Delporte-Gallet, Fauconnier, Guerraoui, Ruppert, DCOSS 2006).
- Nothing else is computable on a **complete interaction graph**, i.e. if any agent can interact with any other (Angluin, Aspnes, Eisenstat, PODC 2006).
  - Example: can't compute "Is the number of cold sensors the square of the number of hot sensors?"

What population protocols can't do

- Complete interaction graph gives the *weakest* model.
- Conjectured in PODC 2004 paper that this model can only compute the Presburger predicates.
- Proved in PODC 2006 paper.
- We'll describe a simplified version of this result now, then switch to what we *can* do in stronger models.

Presburger predicates in disguise

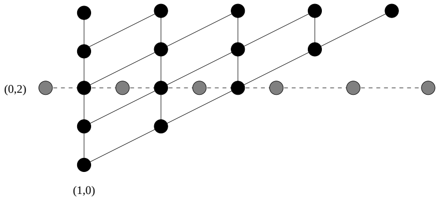
Other ways to define a Presburger predicate:

- Take a regular language L and forget about the order of symbols in each word.
  - Resulting **Parikh map of a regular set** is Presburger-definable.
  - All Presburger-definable sets can be constructed this way.
  - Cute fact: going to context-free languages doesn't change anything.
- Take a finite union of **linear sets** of the form

$$\{\vec{b} + k_1\vec{x}_1 + k_2\vec{x}_2 + \cdots + k_m\vec{x}_m\}.$$

  - Resulting **semilinear set** is Presburger-definable.
  - All Presburger-definable sets can be constructed this way.

Example



A semilinear set S, equal to the union of

- $\{(1,0) + k_1(1,0) + k_2(2,1)\}$ (dark circles), and
- $\{(0,2) + k_3(2,0)\}$ (shaded circles).

Plan

- Want to show that every predicate stably computed by a population protocol is a finite union of sets of the form

$$\{\vec{b} + k_1\vec{x}_1 + k_2\vec{x}_2 + \cdots + k_m\vec{x}_m\}.$$

- But the proof is too big to fit in the next eight slides.
- So instead we'll prove a weaker **Pumping Lemma**: Every predicate stably computed by a population protocol is a finite union of **monoids**: sets of the form

$$\{\vec{b} + k_1\vec{x}_1 + k_2\vec{x}_2 + \dots\},$$

where the number of terms may be infinite!

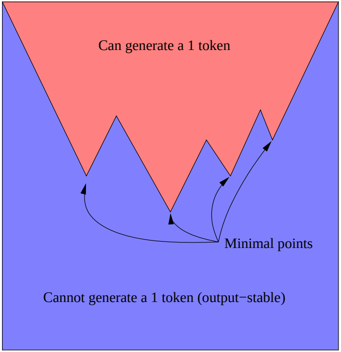
- This is the first step in the real proof, which then shows that finitely many terms are enough using large doses of algebra and geometry.

Higman's Lemma

- **Higman's Lemma**:
  - Any infinite sequence $a_1, a_2, \ldots$ in $\mathbb{N}^d$ has elements a_i, a_j with $a_i \leq a_j$ and $i < j$.
- Corollaries:
  - Every subset of $\mathbb{N}^d$ has finitely many minimal elements. (**Dickson's Lemma**.)
  - Every infinite subset of $\mathbb{N}^d$ contains an infinite ascending sequence $a_1 < a_2 < a_3 \ldots$.

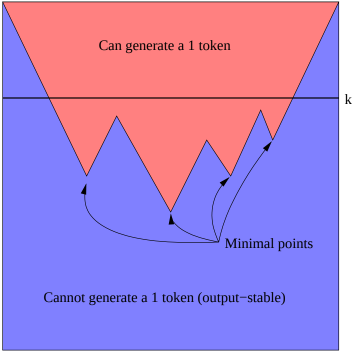
Output-stable configurations are semilinear

- Recall a configuration is **output-stable** if you can't generate a token with a different output.
- If x can generate a 1, so can any configuration $y \geq x$.
- Non-stable configurations are closed upwards $\Rightarrow$ are union of cones over finitely many minimal points (by Dickson's Lemma).
- Non-stable configurations are semilinear $\Rightarrow$ so are stable configurations.



Truncation

- From preceding slide, x is output-stable iff $x \not\geq b_i$ for some finite list of b_i.
- Let $k > \max b_i(j)$ and define the **truncation** $\tau_k(x(j)) = \min(k, x(j))$.
- We can detect from $\tau_k(x)$ if $x \geq b_i$ or not.
- $\Rightarrow$ x is output-stable iff its truncation is.



Extensions

- Want to recognize when adding more tokens to an input doesn't change its behavior.
- Define the set $X(c)$ of **extensions** of c by

$$X(c) = \{x \mid \exists d : c + x \rightarrow d \text{ and } \tau_k(d) = \tau_k(c)\}.$$

- Intuition is that x is in $X(c)$ if c can be "pumped" by x.
- Not hard to show that extensions are composable: if x, y are in $X(c)$, then so is $x + y$. (This shows $\{c + X(c)\}$ is a monoid.)

Monoid covers

- Now we will hunt for a finite monoid cover of some stably computable Y using extensions. The method is to build up a family of sets $x + X(c)$ where x is an input and c is an output-stable configuration reachable from that input.
- Order Y so that $y_i \leq y_j$ implies $i < j$. Let $B_0 = \emptyset$. Compute B_i as follows:
  - If $y_i \in x + X(c)$ for some $(x, c) \in B_{i-1}$, let $B_i = B_{i-1}$.
  - Otherwise, construct B_i by adding to B_{i-1} the pairs
    - $(y_i, s(y_i))$, and
    - $(y_i, s(c + y_i - x))$ for all $(x, c) \in B_{i-1}$ with $x \leq y_i$,

  where $s(z)$ is any stable configuration reachable from z.
- Finally, let $B = \bigcup B_i$.

B covers Y

Definition of B_i

- $B_i = B_{i-1}$ if $y_i \in x + X(c)$, $(x, c) \in B_{i-1}$; else
- $B_i = B_{i-1}$ plus
  - $(y_i, s(y_i))$, and
  - $(y_i, s(c + y_i - x))$ for all $(x, c) \in B_{i-1}$ with $x \leq y_i$,

1. $\{x + X(c)\}$ for $(x, c) \in B$ covers Y. (We add y_i to B_i if it doesn't.)
2. $\{x + X(c)\}$ doesn't contain anything outside Y. (Proof: $z \in x + X(c)$ implies $z \to z' \in c + X(c)$ and so z converges to same output as c by definition of $X(c)$.)
3. If we can show B is finite, we are done.

B is finite

Definition of B_i

- $B_i = B_{i-1}$ if $y_i \in x + X(c)$, $(x, c) \in B_{i-1}$; else
- $B_i = B_{i-1}$ plus
  - $(y_i, s(y_i))$, and
  - $(y_i, s(c + y_i - x))$ for all $(x, c) \in B_{i-1}$ with $x \leq y_i$,

1. Suppose B is infinite.
2. Use Higman's Lemma to get an increasing sequence $z_1 < z_2 < \ldots$ such that $(z_i, c_i) \in B$ for some c_i.
3. Use Higman's Lemma *again* to get an infinite subsequence (z_{i_j}, c_{i_j}) where both z and c components are increasing.
4. Eventually, truncated $\tau_k(c_{i_{(j+1)}}) = \tau_k(c_{i_j})$.
5. But then $z_{i_{(j+1)}} - z_{i_j}$ is in $X(c_{i_j})$, so $z_{i_{(j+1)}}$ can't be in B.

A pumping lemma

- We just showed that any stably computable set has a finite cover by monoids:

$$\{\vec{b} + k_1\vec{x}_1 + k_2\vec{x}_2 + \dots\},$$

- Corollary: Any infinite stably computable set S can be pumped: there is some $\vec{b}$ and $\vec{x}$ such that $\vec{b} + k\vec{x}$ is in S for all $k \in \mathbb{N}$.
- Sadly, this is not enough to exclude some non-semilinear sets like $\{(x, y) \mid x < y\sqrt{2}\}$.

The full result

- Stably computable sets are semilinear, i.e. finite unions of sets of the form

$$\{\vec{b} + k_1\vec{x}_1 + k_2\vec{x}_2 + \cdots + k_m\vec{x}_m\}.$$

- This excludes pretty much anything that requires multiplication, irrational constants, or nested loops to define.
- Proof: see PODC 2006 paper.

Hooray! We're done!

- Question: If we have an exact characterization of what population protocols can do, aren't we done?

Hooray! We're done!

- Question: If we have an exact characterization of what population protocols can do, aren't we done?
- Answer: No.

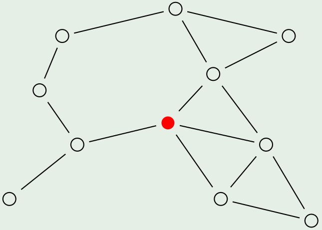
Hooray! We're done!

- Question: If we have an exact characterization of what population protocols can do, aren't we done?
- Answer: No.
  - Bounded-degree interaction graph gives all of LINSPACE (Angluin et al., DCOSS 2005).
  - Random scheduling in a complete graph gives all of LOGSPACE with exponential slowdown using simple techniques (Angluin et al., PODC 2004), or *polylogarithmic* slowdown using more sophisticated techniques (Angluin et al., DISC 2006).
- Rest of talk: bounded-degree graphs, then random scheduling.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

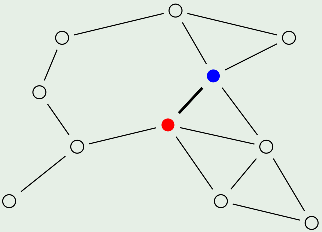


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

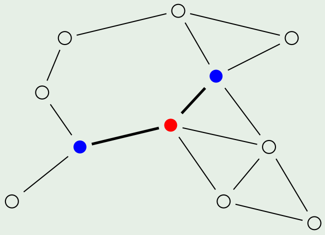


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

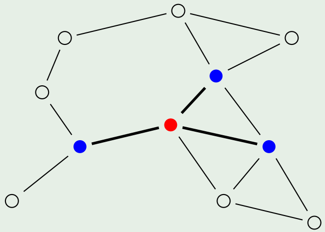


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

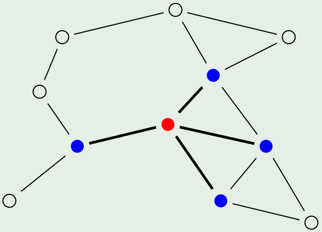


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

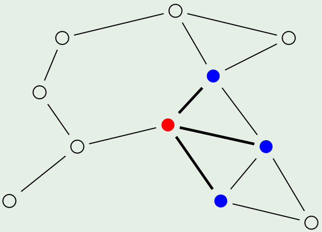


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

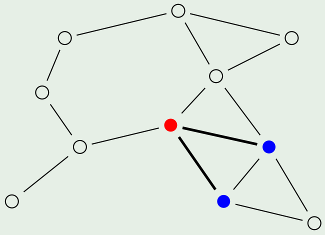


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

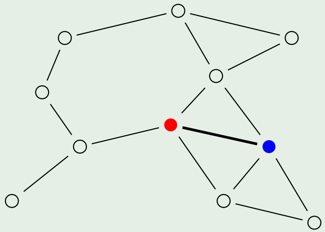


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

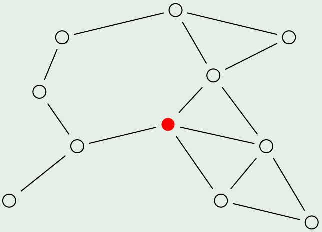


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

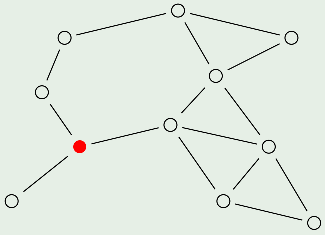


Leader (●) obtains lower bound on degree by placing followers (●) on adjacent nodes.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

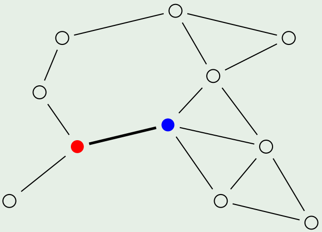


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

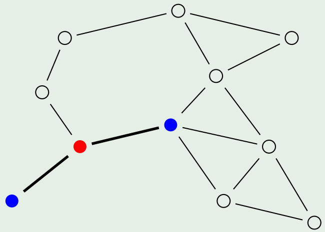


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

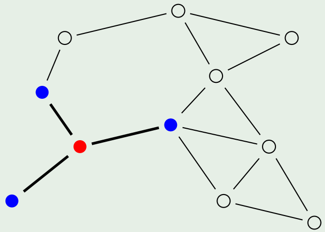


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

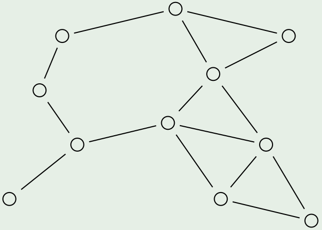


Leader (●) moves to new node and repeats the experiment.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

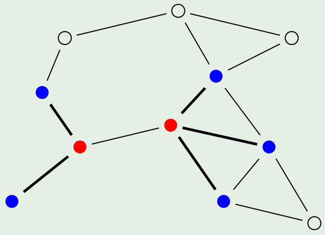
Computing degrees



Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

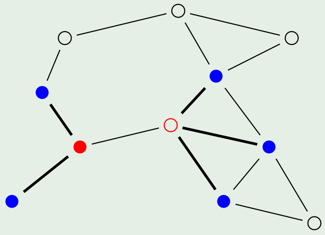


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

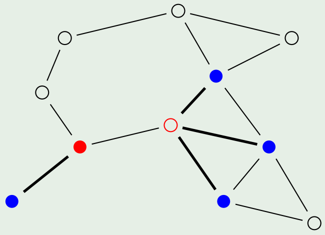


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

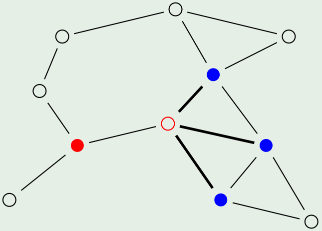


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

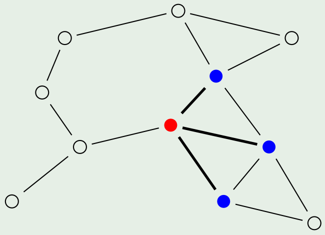


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

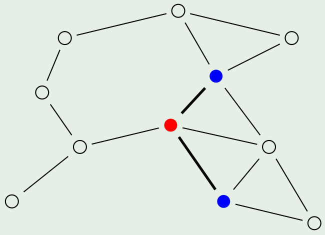


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

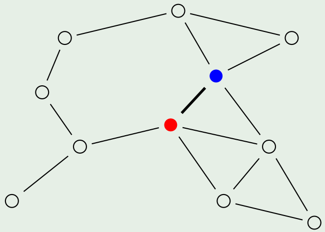


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

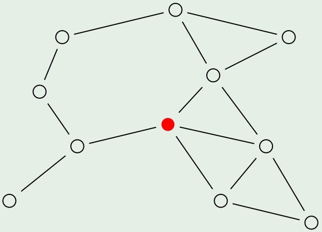


With two leaders, one consumes the other and then cleans up all followers.

Leaders and followers

- Generate a single wandering **leader** token as in parity protocol.
- Leader deploys **followers** to mark out subgraphs.
- When two leaders collide, survivor cleans up extra followers.

Computing degrees

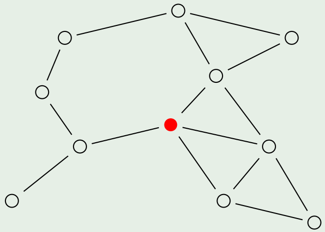


With two leaders, one consumes the other and then cleans up all followers.

Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.

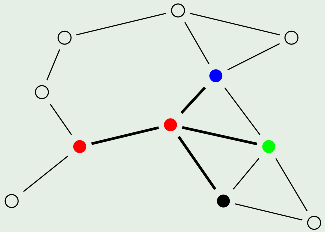
Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.

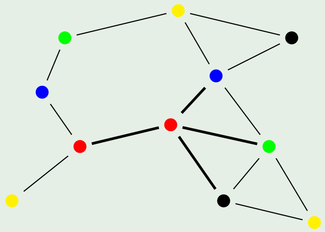
Distance-2 coloring



Distance-2 colorings

- In **bounded-degree** graphs, we can color the nodes so that all neighbors of any given node have different colors, giving a **distance-2 coloring**.
- Colors act as **local identifiers**, allowing a node to point to particular neighbors.

Distance-2 coloring

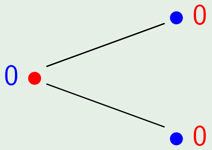


Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



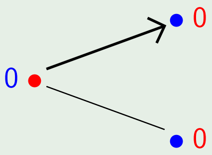
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



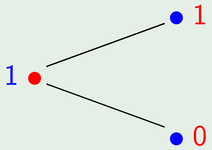
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



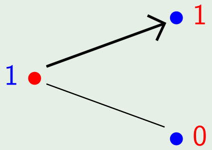
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



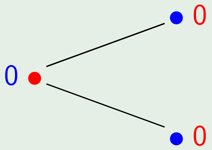
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



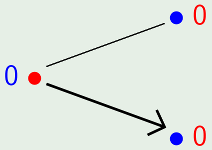
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



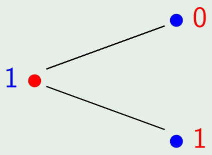
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



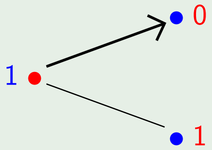
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates



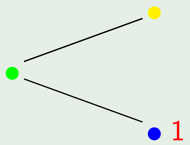
(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Distance-2 coloring

- Main problem is to detect duplicate colors among neighbors.
- Each node records how many times it has interacted with each neighbor (mod 2).
- On a mismatch, both nodes pick a new color nondeterministically.[a]

[a] Can be reduced to a deterministic protocol by exploiting nondeterministic scheduling—see OPODIS 2005 paper.

Detecting duplicates

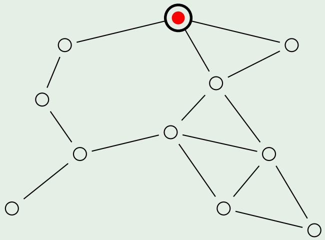


(Numbers are mod-2 interaction counts, indexed by neighbor color.)

Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

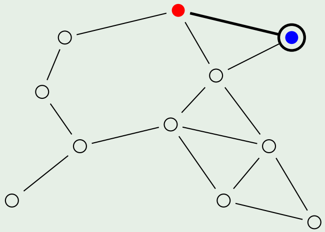
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

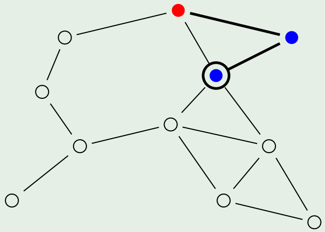
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

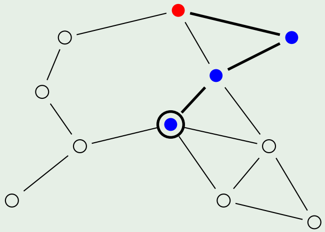
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

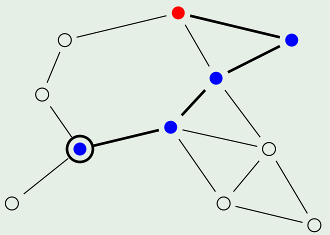
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

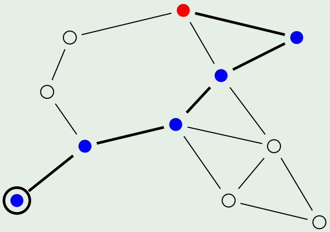
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

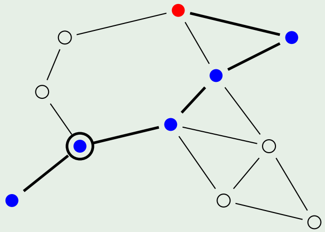
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

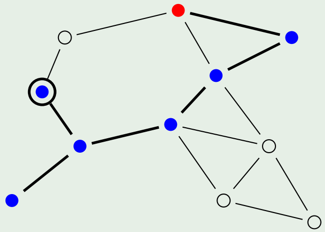
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

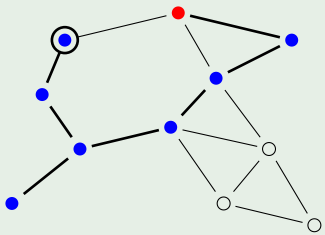
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

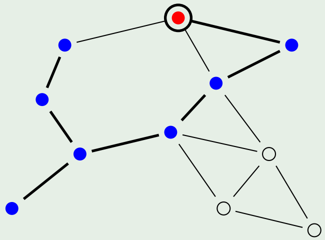
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

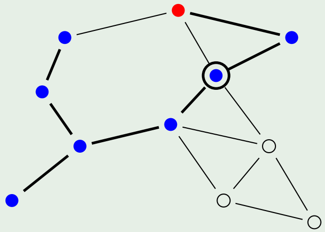
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

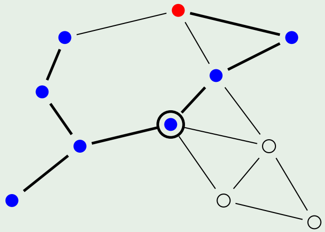
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

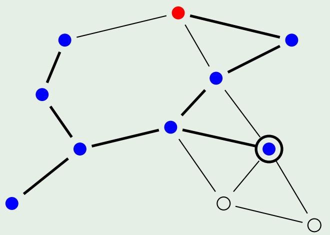
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

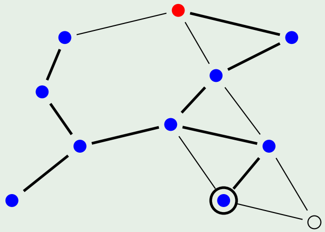
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

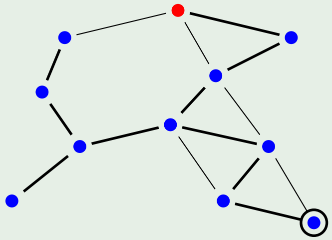
Spanning tree



Spanning trees

- Can build a spanning tree starting at some unique root.
- Assumes we already have a distance-2 coloring.
- Solution: build tree in parallel with coloring, reset tree builder whenever a node changes color.

Spanning tree



Distributed computation

- Unroll DFS traversal of spanning tree to get a linear-size Turing machine tape (Itkis and Levin, FOCS 1994).
- $\Rightarrow$ bounded-degree graph can compute all of LINSPACE.

Is it practical?

Algorithms have poor performance even if we assume non-adversarial interaction pattern.

- Wandering leaders may require $\Theta(N^3)$ cover time to visit all nodes.
- Unique leader/colorizer/walker agents are bottlenecks.

More work is needed to get efficient algorithms and good programming tools.

Computation by epidemic

Last part:

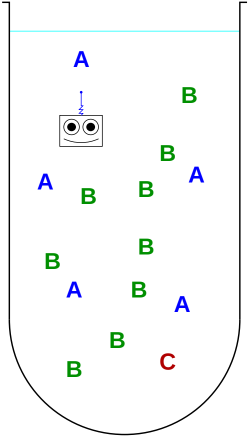
- Back to complete interaction graph.
- But assume **random scheduling**.
- Goal: efficient computation in a test tube.

Randomized population protocols

- Assume next pair of agents to interact is chosen uniformly (i.e. with probability $\frac{1}{N(N-1)}$).
- This gives the **randomized population protocol** model from (Angluin et al., PODC 2004).
- It also is the uniform-rate case of the standard model for well-mixed chemical systems (e.g. (Gillespie 1977)).
- Expected **time** is obtained by dividing expected interactions by N—each agent interacts at a fixed rate regardless of size of the population.

A test-tube computer

- **Register values** (up to $O(N)$) are stored as tokens distributed across the population.
- A unique **leader agent** acts as the (finite-state) CPU.
- We want to support the usual operations of addition, subtraction, comparison, multiplication, division, etc.
- We want to do them all in polylogarithmic time ($O(N \log^{O(1)} N)$ interactions).
- We'll accept a small ($O(N^{-\Theta(1)})$) probability of error.

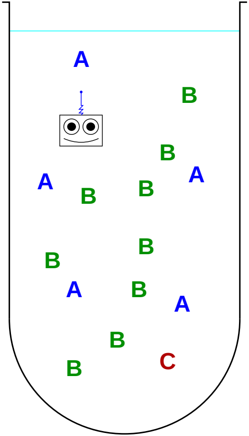


Epidemics

- Key fact: An epidemic starting from one infected agent spreads to all agents in $\Theta(\log N)$ time with high probability.
- This gives us a broadcast primitive.

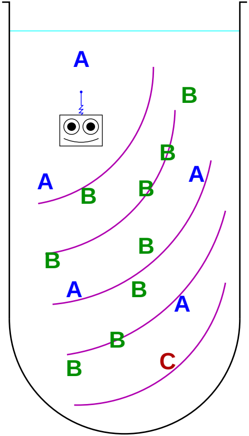
Instruction cycle

- Leader propagates a new opcode via epidemic.
- Followers carry out chosen operation:
  - $A \leftarrow 0$: Erase your A token upon receipt of opcode.
  - $A \leftarrow A + B$: Make a new A token for each B token.
  - $A \stackrel{?}{=} 0$: Start a counter-epidemic if you have an A.
  - $A > B$, $A \leftarrow A - B$, etc.: more complicated.
- Leader collects response (if any) from counter-epidemic, updates its state, and starts a new cycle.



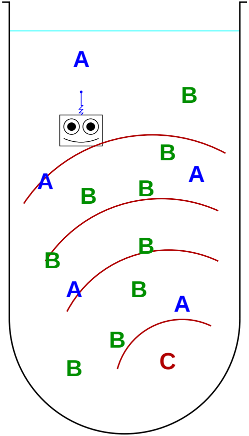
Instruction cycle

- Leader propagates a new opcode via epidemic.
- Followers carry out chosen operation:
  - $A \leftarrow 0$: Erase your A token upon receipt of opcode.
  - $A \leftarrow A + B$: Make a new A token for each B token.
  - $A \stackrel{?}{=} 0$: Start a counter-epidemic if you have an A.
  - $A > B$, $A \leftarrow A - B$, etc.: more complicated.
- Leader collects response (if any) from counter-epidemic, updates its state, and starts a new cycle.



Instruction cycle

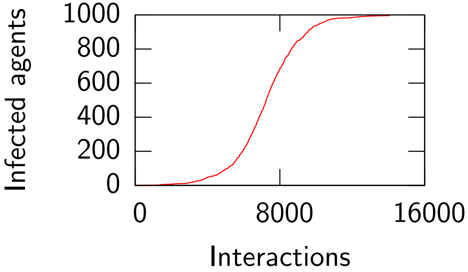
- Leader propagates a new opcode via epidemic.
- Followers carry out chosen operation:
  - $A \leftarrow 0$: Erase your A token upon receipt of opcode.
  - $A \leftarrow A + B$: Make a new A token for each B token.
  - $A \stackrel{?}{=} 0$: Start a counter-epidemic if you have an A.
  - $A > B$, $A \leftarrow A - B$, etc.: more complicated.
- Leader collects response (if any) from counter-epidemic, updates its state, and starts a new cycle.



What's missing?

Problem: How does the leader know when to start the next instruction cycle?

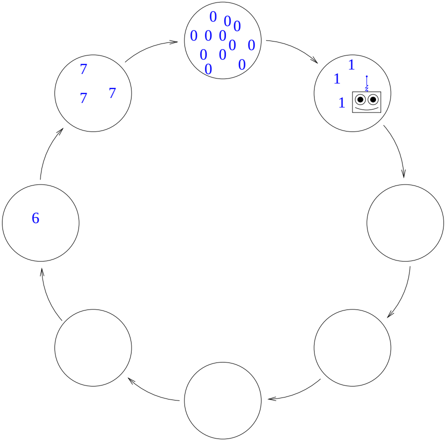
Bounding the time for epidemics



- Average interactions to infect next victim is $\frac{N(N-1)}{i(N-i)}$.
- For $i > N/2$, this is $\Theta(N/i)$, the waiting time for coupon collector.
- $\Rightarrow$ Known coupon collector concentration results (Kamath et al., 1995) bound $i > N/2$ case: $\Theta(N \log N)$ w.h.p.
- Symmetry bounds $i > N/2$ case.

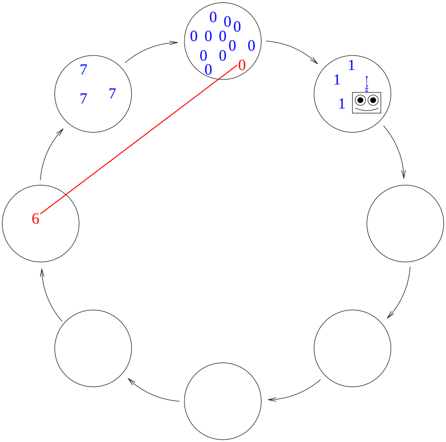
Phase clock

- Each agent is in a **phase** in the range 0 to $m-1$.
- An initiator in a later phase $\bmod m$ recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.



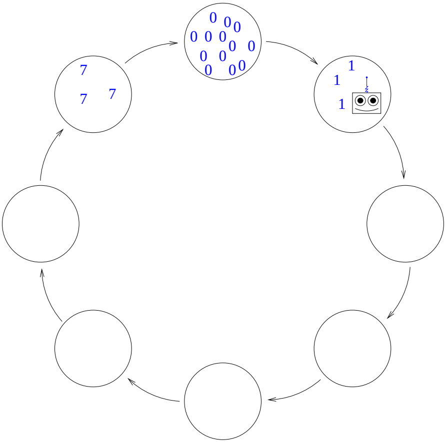
Phase clock

- Each agent is in a **phase** in the range 0 to $m-1$.
- An initiator in a later phase $\bmod m$ recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.



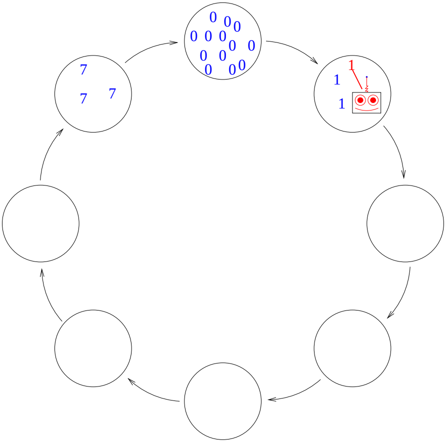
Phase clock

- Each agent is in a **phase** in the range 0 to $m-1$.
- An initiator in a later phase $\bmod m$ recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.



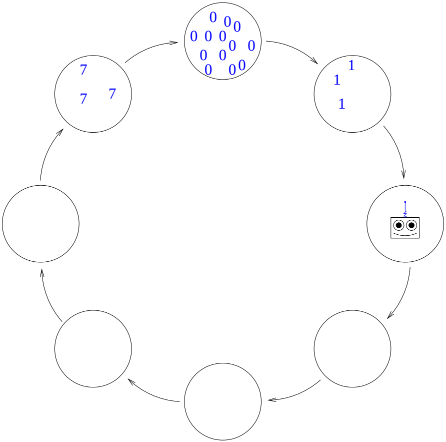
Phase clock

- Each agent is in a **phase** in the range 0 to $m-1$.
- An initiator in a later phase $\bmod m$ recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.

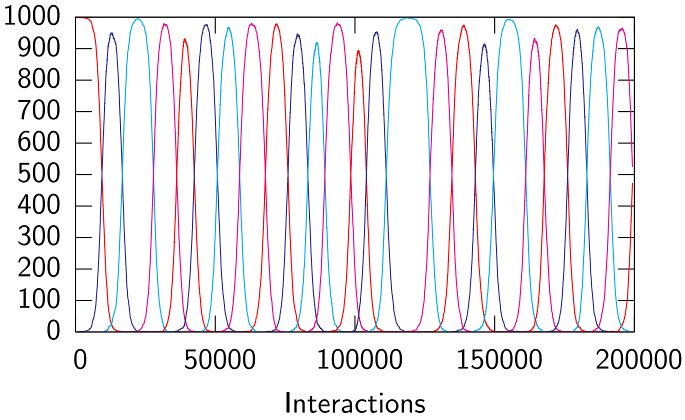


Phase clock

- Each agent is in a **phase** in the range 0 to $m - 1$.
- An initiator in a later phase mod m recruits agents in earlier phases.
- The leader advances if it sees an initiator in its own phase.
- Result: Leader goes all the way around every $\Theta(\log N)$ time units.

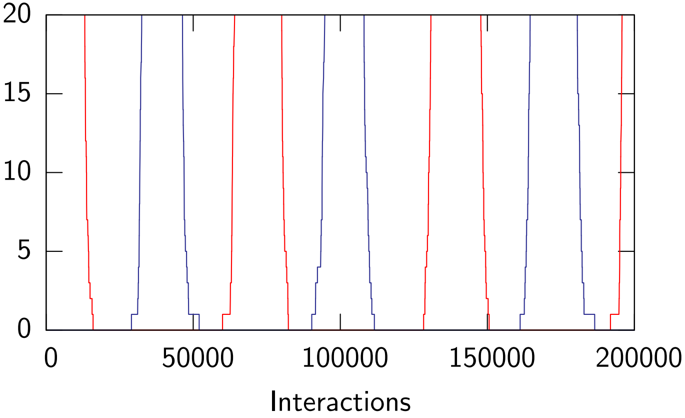


Phase clock: simulation results



Phase clock with $N = 1000$ and $m = 8$.

Phase clock: simulation results



Zoomed view of phase 0 and phase 4.

Why it works

- Phases i and higher act as an epidemic wiping out phases $i-1$ and lower.
- This epidemic finishes in $a \log N$ time (with high probability).
- When the leader advances, it takes at least $b \log N$ time (w.h.p.) to generate at least N^{ϵ} agents in the same phase $\Rightarrow$ leader advances before $b \log N$ time (a **short phase**) with probability $N^{O(\epsilon)-1}$.
- For a sufficiently large number of phases m, the chance of too many short phases in a row is $O(N^{-c})$.
- Amazing fact: m depends on c but not N.

Other operations

- Operations like assignment and addition that don't require tokens to interact can be done in one instruction cycle ($O(\log N)$ time).
- Operations that do require interaction may take longer.
  - Naive $A \stackrel{?}{>} B$ algorithm: Have A and B tokens cancel until only one kind is left.
  - This takes $\Omega(N^2)$ interactions if there are few A's and B's.
- How can we do cancellation faster?

Cancellation by amplification

- Cancellation is fast if there are many tokens to cancel.
- Solution: Alternate between canceling and doubling.
- Invariant $A_k - B_k = 2^k(A_0 - B_0)$ after k rounds.
- If no winner in $2 \log N$ rounds, $A_0 = B_0$.
- This gives $A \overset{?}{<} B$ in $O(\log^2 N)$ time.

Subtraction and division by binary search

- To compute $C \leftarrow A - B$, do binary search for C such that $A = B + C$.
- This takes $O(\log N)$ rounds of binary search at $O(\log^2 N)$ time each $\Rightarrow O(\log^3 N)$ time.
- Similar approach for division gives $O(\log^4 N)$ time. (This is our most expensive operation.)

Results

For a randomized population protocol with a unique initial leader, we have:

- Register machine simulation:
  - $\Theta(\log N)$-bit registers.
  - $O(\log^4 N)$ expected time per operation.
  - $O(N^{-c})$ probability of failure.
- Presburger predicate computation:
  - $O(\log^4 N)$ expected time. (Cf. $O(N)$ for previous protocols.)
  - Zero probability of failure.
  - Trick: Combine fast fallible protocol with slow robust one.

What's left?

- What happens if we don't have a leader to start with?
  - Election by fratricide takes $\Theta(N^2)$ interactions.
  - Phase clock is irretrievably corrupted during election process.
- Can we elect a leader faster?
- Can we build a more robust phase clock?
- Can we cut down the polylog overhead?

Summary

What we have:

- Clean model of large-scale small-scale systems.
- Complete characterization of complete-graph case with adversarial scheduling.
- Powerful (but slow) Turing machine simulator for bounded-degree case with adversarial scheduling.
- Fast register-machine simulator for complete-graph case with random scheduling and an initial leader.

Summary

What we still want:

- Better algorithms and programming tools for bounded-degree case.
- Better understanding of intermediate large-degree cases.
- Better performance (and assumptions) for random-scheduling case.