Fast Computation by Population Protocols With a Leader

Dana Angluin (Yale)
James Aspnes (Yale)
David Eisenstat (Rochester/Princeton)

September 18th, 2006

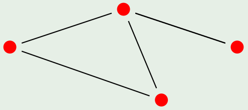
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



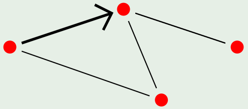
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



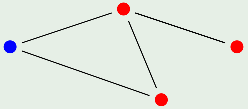
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



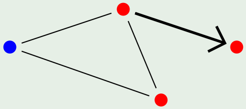
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



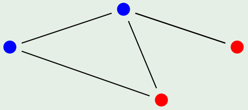
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



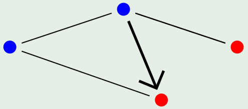
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



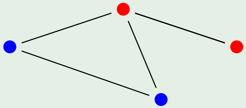
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



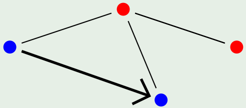
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



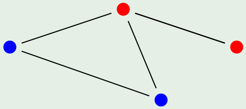
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



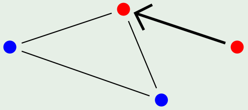
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



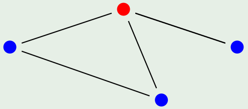
Population protocols

- 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

🔴 = leader
🔵 = non-leader

🔴, 🔴 → 🔵, 🔴
🔴, 🔵 → 🔵, 🔴
🔵, 🔴 → 🔴, 🔵
🔵, 🔵 → 🔵, 🔵



Why do we care?

- Original official motivation: Sensor networks with really dumb sensors.
- Revised official motivation: Chemical (especially biochemical) systems.
- Unofficial motivation: Cool mathematical structures that might actually be useful.

Stable computations

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

Parity

In:

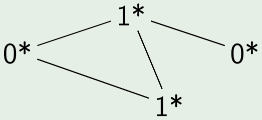
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

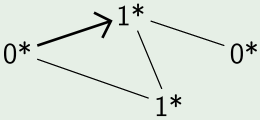
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

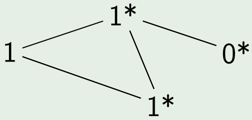
$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

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



Stable computations

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

Parity

In:

$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$0*, 0* \longrightarrow 0, 0*$

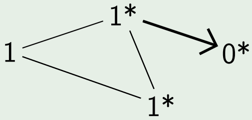
$0*, 1* \longrightarrow 1, 1*$

$1*, 0* \longrightarrow 1, 1*$

$1*, 1* \longrightarrow 0, 0*$

$x, y* \longrightarrow y*, y$

$x*, y \longrightarrow x, x*$



Stable computations

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

Parity

In:

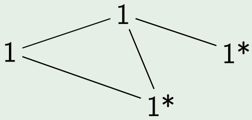
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$0*, 0* \longrightarrow 0, 0*$

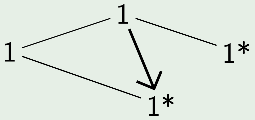
$0*, 1* \longrightarrow 1, 1*$

$1*, 0* \longrightarrow 1, 1*$

$1*, 1* \longrightarrow 0, 0*$

$x, y* \longrightarrow y*, y$

$x*, y \longrightarrow x, x*$



Stable computations

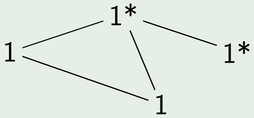
- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all agents.

Parity

In:
$x \longrightarrow x*$

Out:
$x \longrightarrow x$
$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

$x \rightarrow x*$

Out:

$x \rightarrow x$

$x* \rightarrow x$

$0*, 0* \rightarrow 0, 0*$

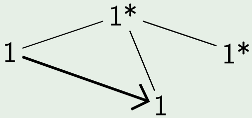
$0*, 1* \rightarrow 1, 1*$

$1*, 0* \rightarrow 1, 1*$

$1*, 1* \rightarrow 0, 0*$

$x, y* \rightarrow y*, y$

$x*, y \rightarrow x, x*$



Stable computations

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

Parity

In:

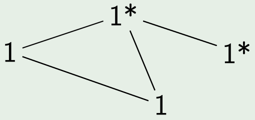
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

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



Stable computations

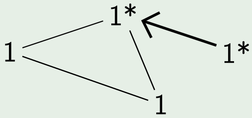
- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all agents.

Parity

In:
$x \longrightarrow x*$

Out:
$x \longrightarrow x$
$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

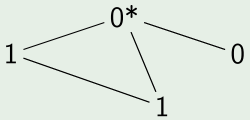
$x \to x*$

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.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all agents.

Parity

In:

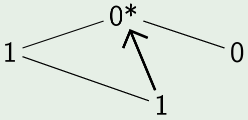
$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

$x \longrightarrow x*$

Out:

$x \longrightarrow x$

$x* \longrightarrow x$

$0*, 0* \longrightarrow 0, 0*$

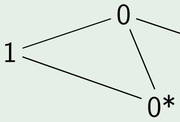
$0*, 1* \longrightarrow 1, 1*$

$1*, 0* \longrightarrow 1, 1*$

$1*, 1* \longrightarrow 0, 0*$

$x, y* \longrightarrow y*, y$

$x*, y \longrightarrow x, x*$



Stable computations

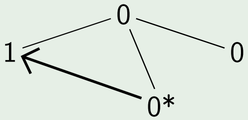
- **Input map** converts inputs (at each agent) to initial states.
- **Output map** extracts outputs from states.
- **Fairness condition** enforces that any reachable state is eventually reached.
- A **stable computation** converges to the same output at all agents.

Parity

In:
$x \longrightarrow x*$

Out:
$x \longrightarrow x$
$x* \longrightarrow x$

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



Stable computations

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

Parity

In:

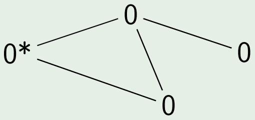
$x \to x*$

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?"

Hooray! No more population protocol papers!

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

Hooray! No more population protocol papers!

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

Hooray! No more population protocol papers!

- 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 (Angluin et al., PODC 2004).
  - These results involve *very slow* Turing machine simulations.

Hooray! No more population protocol papers!

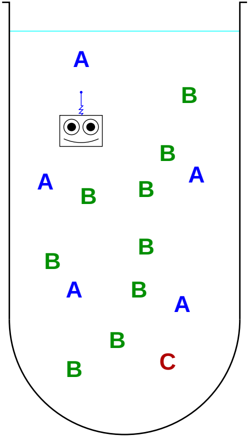
- Question: If we have an exact characterization of what population protocols can do, aren't we done?
- Answer: No.
  - Bounded-degree interaction graph gives all of LINSPACE (Angluin et al., DCOSS 2005).
  - Random scheduling in a complete graph gives all of LOGSPACE (Angluin et al., PODC 2004).
  - These results involve *very slow* Turing machine simulations.
- Today: Fast simulations of register machines, assuming random scheduling.

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.

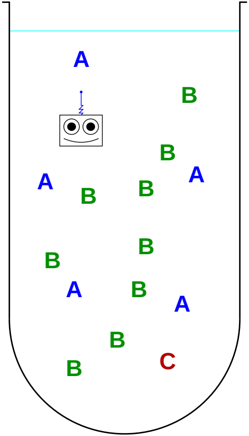


Epidemics

- 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.

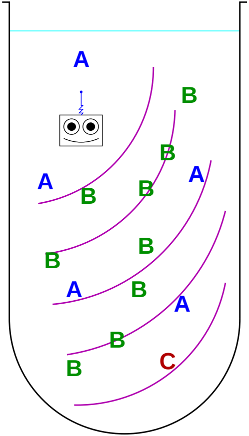
Instruction cycle

- 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.
  - $A \overset{?}{=} 0$: 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.



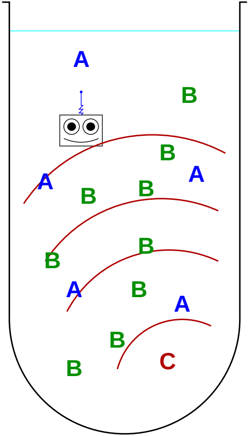
Instruction cycle

- 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.
  - $A \overset{?}{=} 0$: 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.



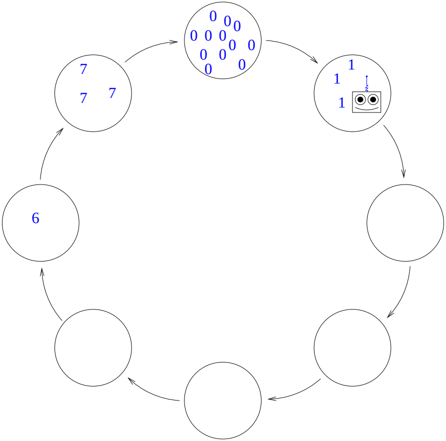
Instruction cycle

- 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.
  - $A \overset{?}{=} 0$: 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.

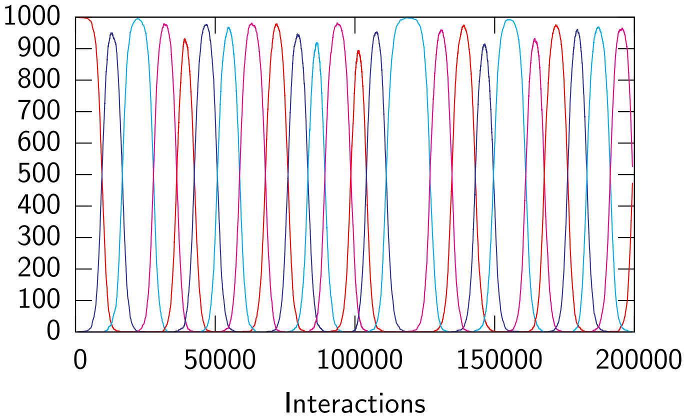


Phase clock

- 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.

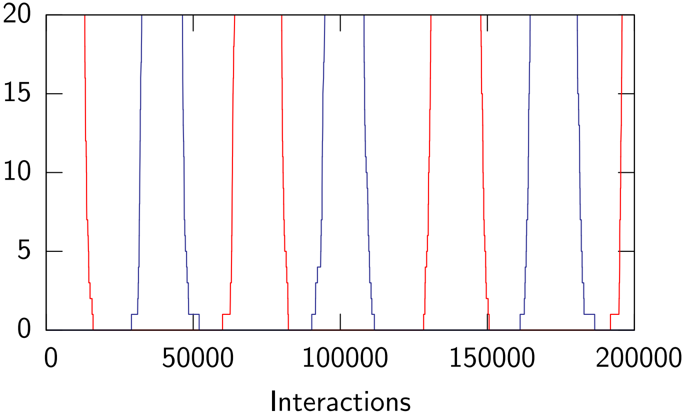


Phase clock: simulation results



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 sufficiently large m, 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 $A \overset{?}{>} B$ 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?

We have some promising simulation results, but better analytical tools may be needed.