What Can Be Implemented Anonymously?

Rachid Guerraoui
EPFL
Lausanne, Switzerland

Eric Ruppert
York University
Toronto, Canada

Anonymity

- No process ids
- Identical code

Why?

- Privacy
- Tiny, cheap, mass-produced devices
- Intellectual curiosity

How?

- Trusted third party (anonymizer)
- Onion routing
- Anonymous radio broadcast

Previous work

What can(not) be done anonymously & asynchronously?

No Failures:

✔ Consensus using registers [Attiya Gorbach Moran 02].
✔ ✘ What can be done with registers [Attiya Gorbach Moran 02].
✔ ✘ What can be done with broadcasts (= counters)
[Aspnes Fich Ruppert 04].
✔ Randomized naming using registers [many papers].

Crash Failures:

✔ Randomized naming using single-writer registers
[Panconesi *et al.* 98].
✘ Randomized naming using registers [several papers].
✔ Randomized consensus using registers [Buhrman *et al.* 00].

What Makes Anonymity Hard?

- Cannot break symmetry.
- Cannot count the number of processes that have the same input.
- There are no “safe” places to write information:
  other processes can write to any location that you write to.
- Hard to help other processes if you do not know who needs help.

Our Results

Model: shared registers, crash failures, deterministic algorithms.

✔ Wait-free timestamping.

✔ Wait-free snapshots.

✔ Obstruction-free consensus.

✔ ✘ Characterization of what objects have obstruction-free implementations.

Progress Conditions

We consider three kinds of progress.

• Wait-free: each operation by process P finishes after finite number of P's own steps.

• Non-blocking: some operation finishes after finite number of steps.

• Obstruction-free: an operation must finish if it takes sufficiently many steps without interruption by other processes.

Timestamps

Timestamps are a fundamental tool for synchronization.

Provide distinct values that increase over time.

Can they be implemented from registers?

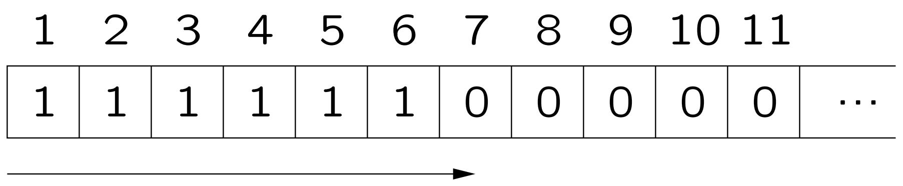
✔ Easy to implement in non-anonymous systems.
✘ Impossible to implement anonymously: they break symmetry.

We define a weaker version of timestamps:

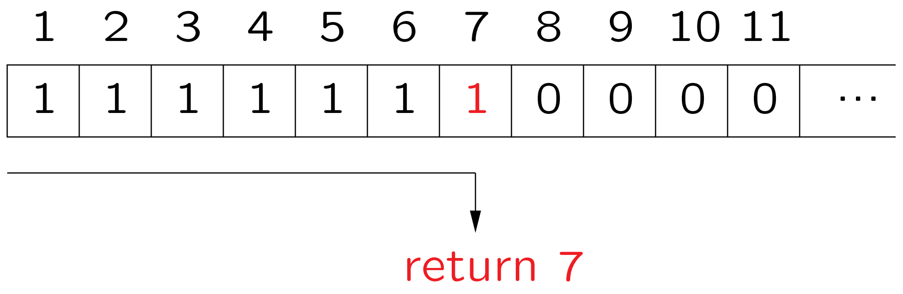
- concurrent operations may get the same value,
- for non-concurrent operations, later one must return larger value

A Non-Blocking Timestamp Implementation

- Use an array of single-bit registers initialized to 0.
- Search for leftmost 0 entry.



- Change it to 1 and return the index of that entry.



Note: A process can starve if other processes write 1's faster than it can search.

A Wait-Free Timestamp Implementation

To make algorithm wait-free, processes must help one another.

- Use one additional register.
- Each operation writes its output there.
- Read register periodically. If it changes n times, return the largest value seen there.

Complexity

If n processes do k accesses to timestamp object,
- k single-bit registers and one $(\log k)$-bit register are used,
- worst-case time per operation is $O(\log k)$, and
- average time per operation is $O(\log n)$.

Snapshots

A snapshot object stores an array of m components and provides two operations:

- UPDATE component i with value v, and
- SCAN, which returns vector of values in all components.

Snapshots are an important building block of many algorithms.

All previous implementations use process ids.

Classic wait-free algorithm of Afek *et al.* can be modified to work anonymously, using our anonymous timestamps.

Consensus

- Each process begins with an input value.
- All must output same value, which is one of the input values.

✗ Wait-free (or non-blocking) consensus is impossible [FLP 84].

✔ We give obstruction-free algorithms.

For the talk, we focus on binary consensus. (Multi-valued consensus can be done by agreeing on the output bit by bit.)

Obstruction-free Consensus

Observation: randomized wait-free consensus algorithms can be "derandomized" to yield obstruction-free algorithms [Herlihy Luchangco Moir 03].

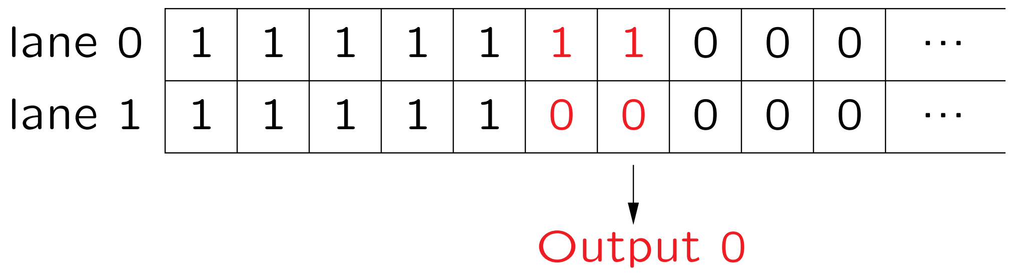
Applying this to Chandra's anonymous randomized algorithm gives us a deterministic anonymous obstruction-free algorithm.

But this algorithm uses an unbounded number of registers.

Unbounded Space Algorithm

Use a $2 \times \infty$ array of bits (initialized to 0) as a race track.

Processes are divided into two teams according to input values.
Each team uses one lane of the race track.
Mark your progress along the track by writing 1's.

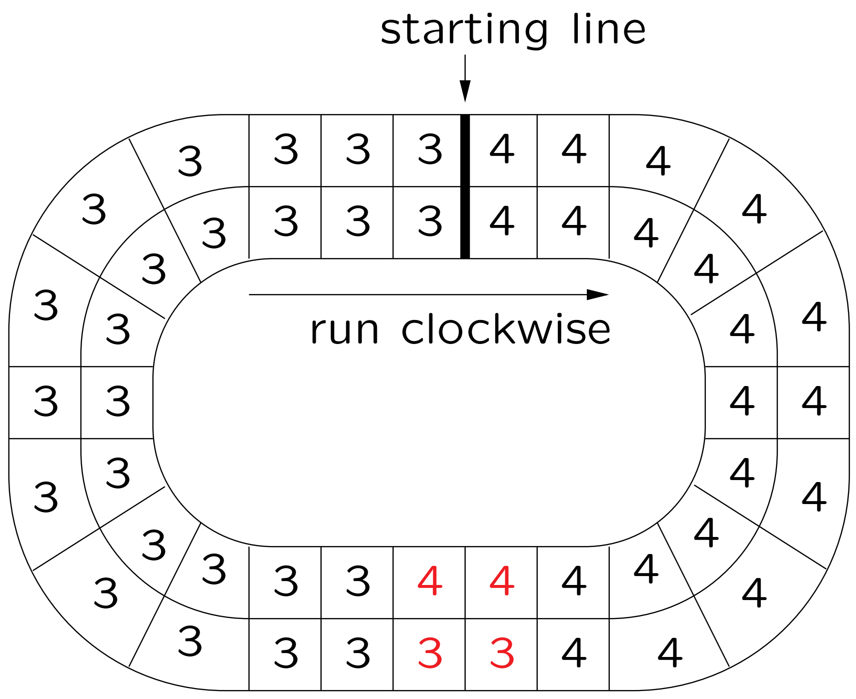

lane 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | ···
lane 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ···
Output 0

If you see the opposite team is ahead of you, switch teams.
If your team is safely ahead of the other, stop and output your team's value.

If one process runs by itself long enough, it will win.
All others will switch to its team and output the same value.

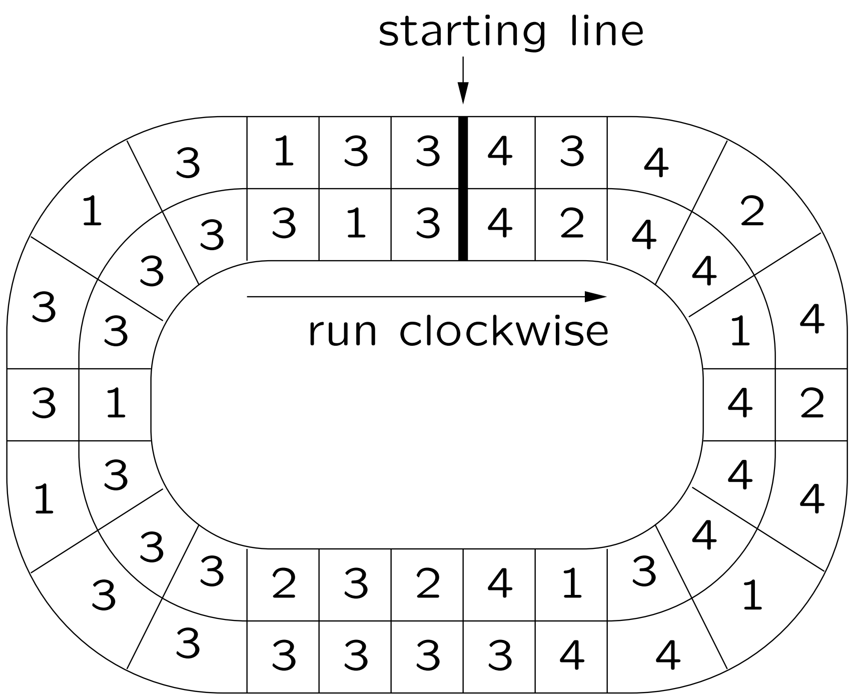
$O(n)$ Space Algorithm

Idea: Instead of an infinite race track, use a circular one.
Write numbers to indicate what lap you are running.



Actual Picture

But things will actually look much messier:



Difficulties of a Circular Track

Complication 1:
After a fast process writes an entry, a slow teammate may overwrite it with a smaller number.

Partial Solution:
Implement the array with a snapshot object.
Between each step, do a SCAN of the entire race track.
If you observe another process working on a higher lap than you, jump to the start of that lap.
$\Rightarrow$ Limits the number of out-of-date values that get written.

Complication 2:
Because of overwrites, you may think the wrong team is winning.

Solution:
Switch teams only when you are really sure the team is winning.
Stop only when your team is winning by a wide margin.

Randomized Wait-Free Anonymous Consensus

Recall: randomized wait-free algorithms can be derandomized to yield obstruction-free algorithms.

Conversely, our obstruction-free algorithm can be “randomized” to yield an obstruction-free anonymous randomized algorithm that uses $O(n)$ registers.

Characterization

Theorem An object has an anonymous obstruction-free implementation from registers iff it is idempotent.

Idempotence means that applying any particular operation twice in a row has the same effect (and response) as doing it once.

Examples of idempotent objects:
registers, sticky bits, snapshot objects, resettable consensus.

Proof (implementable $\Rightarrow$ idempotent): A non-idempotent object can be used to break symmetry,
which is impossible in an anonymous system equipped with registers.

Characterization

Proof (idempotent $\Rightarrow$ implementable):

Use a modified version of Herlihy's universal construction:
- Store a list of operations that have been performed.
- Use consensus to thread new operations on to end of list.

Get a timestamp.
Use your operation and timestamp as input to consensus object.
If successful, compute response.

Idempotence is crucial: several (concurrent) identical operations can all get added to the list as a group, at the same time, and all get the same response.
Nobody can tell how many identical operations are in each group.

Open Problems

- Improving space and time complexity of algorithms.
- Complexity lower bounds.
- Efficient anonymous implementations of strong registers from weak registers.
- Characterization of what can be done wait-free.