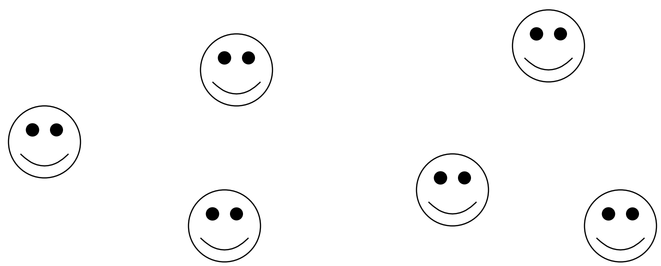
Relationships Between Broadcast and Shared Memory in Reliable Anonymous Distributed Systems

James Aspnes, Yale University
Faith Ellen Fich, University of Toronto
Eric Ruppert, York University

Anonymity

Processes do not have identifiers and execute identical programmes.



Advantages

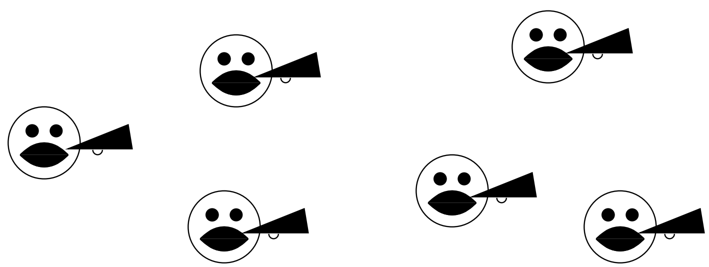
- cheaper to mass-produce
- less testing required
- can enhance privacy

Disadvantage

Problems that require symmetry-breaking become impossible.

Anonymous Broadcast

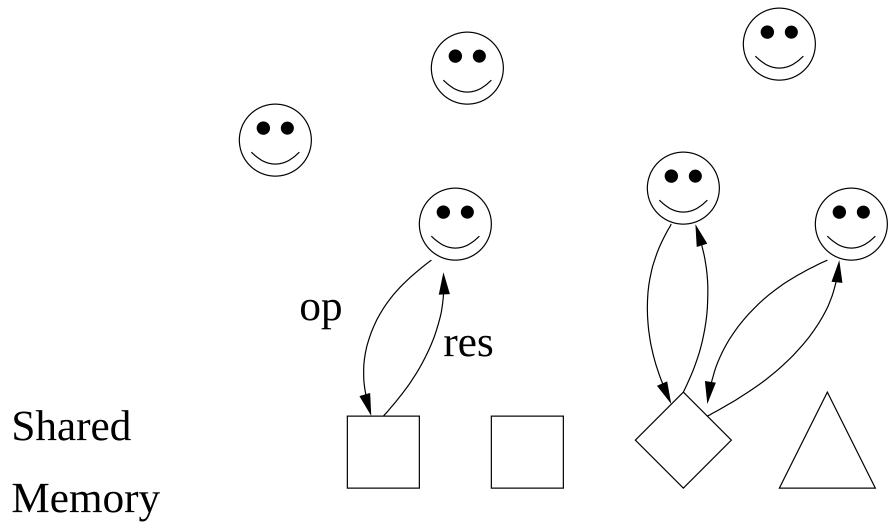
The processes communicate by broadcasting messages to all processes.



No failures, n known to all processes.

Anonymous Shared Memory

Processes communicate via shared objects of various types.



- No failures, n known to all processes.
- Linearizable objects, initialized as programmer wishes.
- Provides abstractions that are useful for programmer.

Question: How do shared-memory models relate to broadcast model (and one another) in anonymous systems?

Previous Work on Anonymity

• Some research on point-to-point message-passing systems.
Impossibility of leader election in a ring [Angluin, 1980].
Using asymmetry of network to solve problems [Boldi, Vigna, 1999–2001].

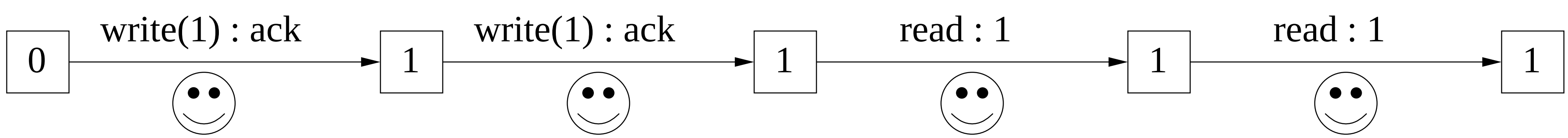
• Agreement tasks solvable using registers (no failures) [Attiya, Gorbach, Moran, 2002].

• Naming possible, but not consensus, using randomization and registers (halting failures) [Buhrman *et al.* 2000].

• Topological approach used to characterize tasks with wait-free solutions from registers [Herlihy, Shavit 1999].

Idemdicence

A shared object is idemdicent ("same-saying") if two consecutive invocations of the same operation (with the same arguments) always return identical responses.



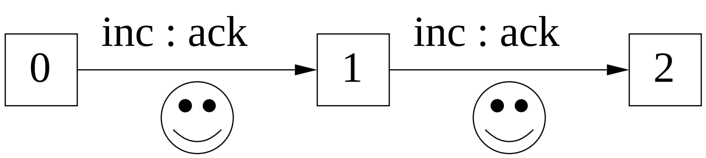
Idemdicent objects: registers, snapshots, consensus objects.

Non-idemdicent objects: queues, compare&swap.

More Examples

Counter

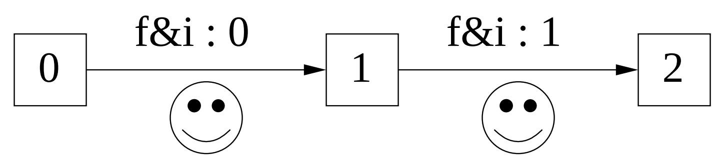
- stores an integer
- increment operation
  (returns ack)
- read operation



Counters are idemdicent.

Fetch&Increment

- stores an integer
- single fetch&inc operation
  (returns value)



Fetch&Inc objects are not.

What Can Broadcasts Implement?

Theorem Broadcasts can simulate shared objects iff the objects are idemdicent.

($\Rightarrow$): We show that asynchronous broadcasts can simulate a synchronous system that contains any idemdicent objects.

Store a local copy of shared memory at each process.

To simulate round r:
A process that wants to perform op on object X broadcasts (r, op, X).
Each process collects all n broadcasts for round r
and simulates all operations on the shared objects locally.

If several processes access same object in the round, order operations lexicographically.
Idemdicence $\Rightarrow$ no need to break ties consistently.

The Converse

Theorem Broadcasts can simulate shared objects iff the objects are idemdicent.

($\Leftarrow$): We show that even synchronous broadcast cannot implement an asynchronous system with a non-idemdicent object.

Consider a non-idemdicent object.
If all processes perform same operation on it, at least two will get different results.
Synchronous broadcasts cannot break symmetry in this way.

Broadcast $\equiv$ Counters

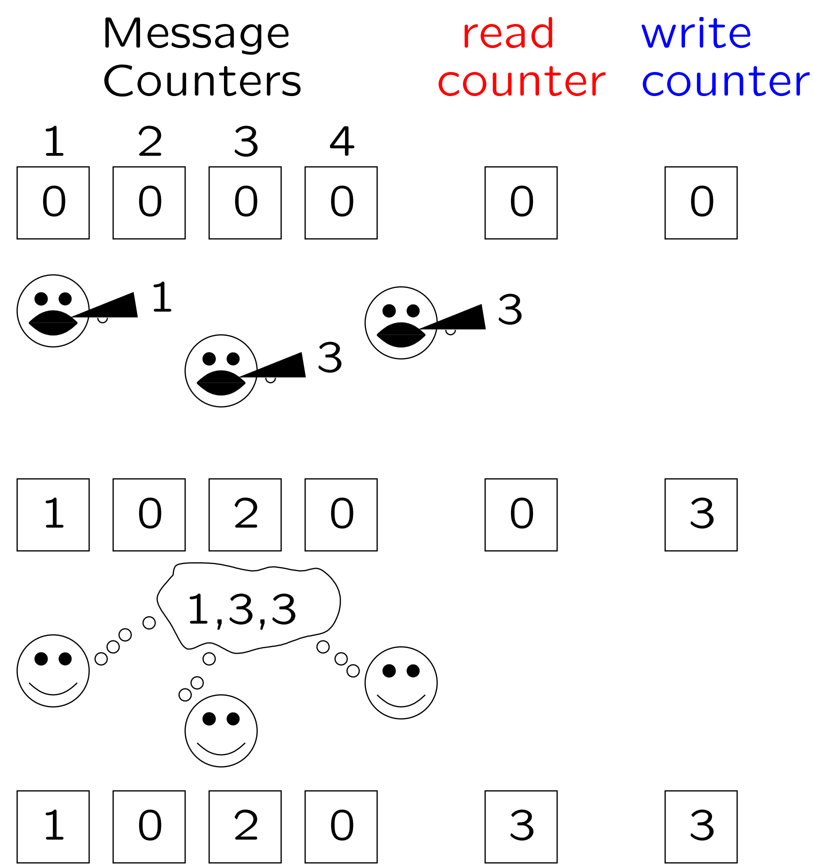
Counters are idemdicent, so broadcasts can simulate them.

Conversely, we show how the asynchronous counter model can simulate synchronous broadcasts.

WLOG, assume bounded-length messages.

Use one counter for every possible message that can be sent plus a read counter and a write counter.

Counters Simulate Broadcasts



To send a message, increment the corresponding message counter and then the write counter.

Wait until write counter mod $n = 0$.
Read all message counters.
Increment read counter.

Wait until read counter mod $n = 0$.
Start next round.

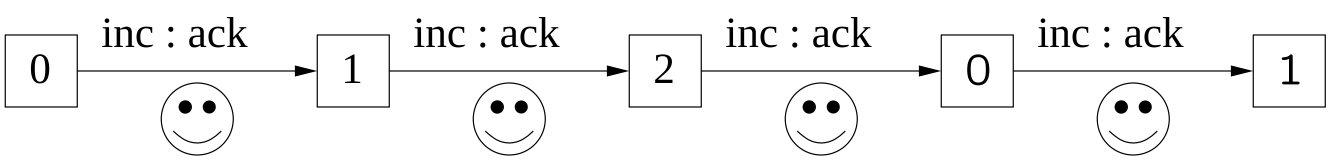
Idempotence

An object is m-idempotent if

- it is idemdicent, and
- doing an operation $m+1$ times has same effect as doing it once.

Examples A register is 1-idempotent.

A mod-3 counter is 3-idempotent:



Clones

A collection of processes behave as clones in an execution if

- they have the same input,
- they run in lock-step, and
- all perform the same step in each round.

If objects accessed by P in an execution are m-idempotent, we can add m clones of P to the execution, and nobody will notice their presence.

Configurations C', C'' are indistinguishable (except to the two clones) if objects accessed are 2-idempotent.

Broadcast is Stronger than Registers

Registers are idemdicent, so broadcasts can implement them.

Threshhold-2 function:

- binary function of n variables
- output is 1 iff at least 2 inputs are 1.

Easy to compute using broadcast.

Impossible to compute (even synchronously) using registers (when $n > 2$) since no register-based algorithm can distinguish

2 clones with input 0,
1 process with input 1

from

1 process with input 0,
2 clones with input 1

Robustness

Robustness is a desirable property of shared-memory models.

It says objects that are weak when used individually are no stronger when used together.

I.e. types A and B can implement type C only if
A alone can implement C or
B alone can implement C.

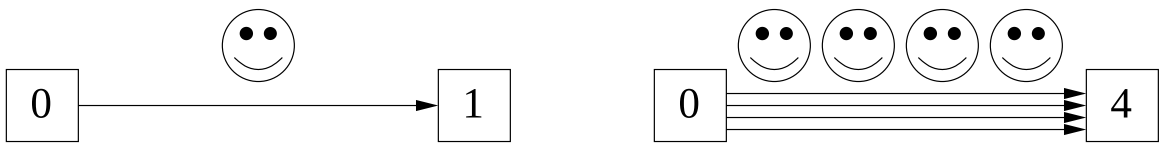
Lots of research on robustness in asynchronous, wait-free models.
Robustness violated by somewhat strange objects.

Here we have a natural counter example to robustness:
mod-2 counters and mod-3 counters can be used together to count up to 5.

mod-3 Counters Cannot Count up to 5

Consider any algorithm constructed using mod-3 counters.

Since mod-3 counters are 3-idempotent,
4 clones are indistinguishable from a single process.



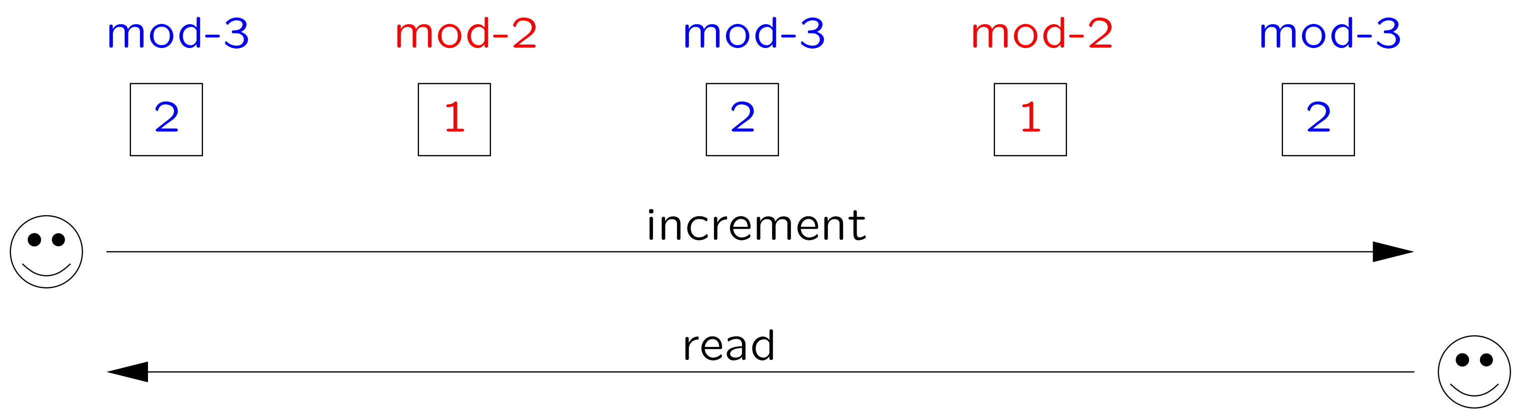
Cannot tell if the value of an up-to-5 counter should be 1 or 4.

Similarly, mod-2 counters cannot count up to 5.

mod-3 + mod-2 Counters Can Count to 5

Our definition of an "up-to-5 counter" says correct responses required only when fewer than 6 increments occur.

Use 3 mod-3 counters and 2 mod-2 counters arranged in a row.

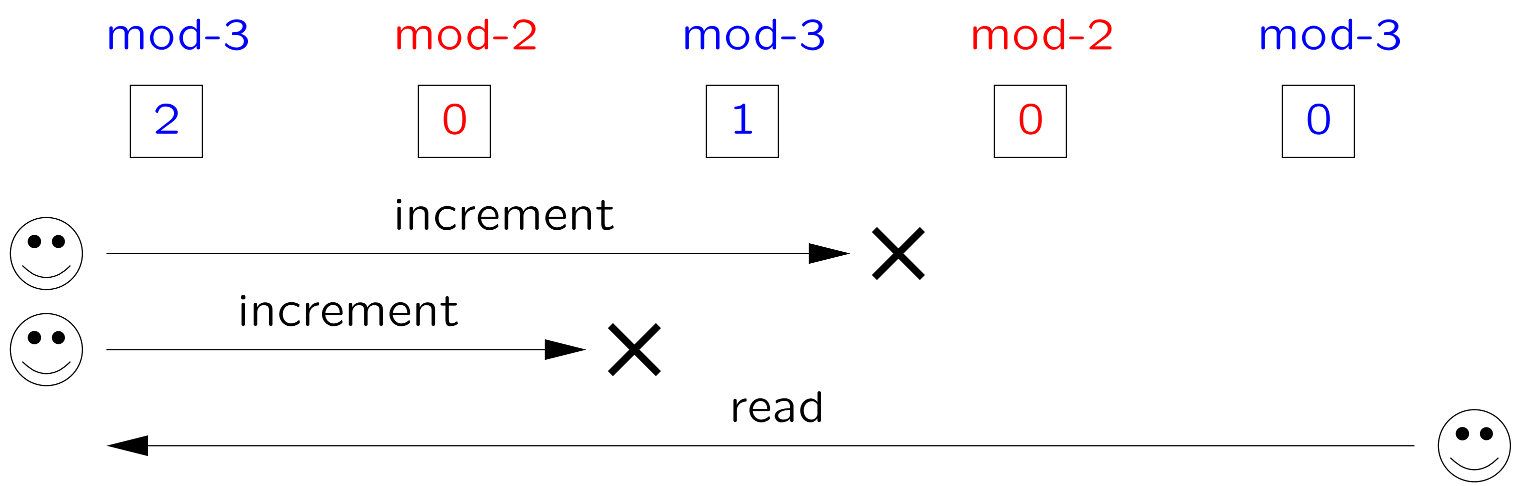


A set of reads is consistent if all mod-3 counters return equal responses and both mod-2 counters return equal responses.

Read repeatedly until you get a consistent set.
Return unique value in $\{0, 1, .., 5\}$ that could give these values.

Termination

Inconsistent sets of reads are caused when a read "crosses" an increment.



No process will do more than 5 increments.
Eventually all increments will terminate, and consistent set will be obtained.

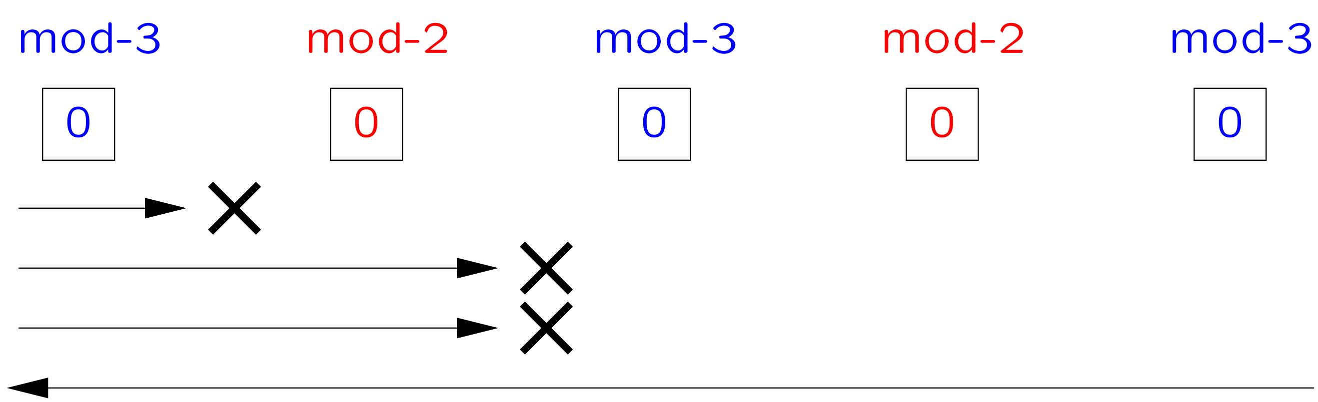
Correctness

Prove correctness only when fewer than 6 increments are done.
Linearize all operations when they (last) access middle object.

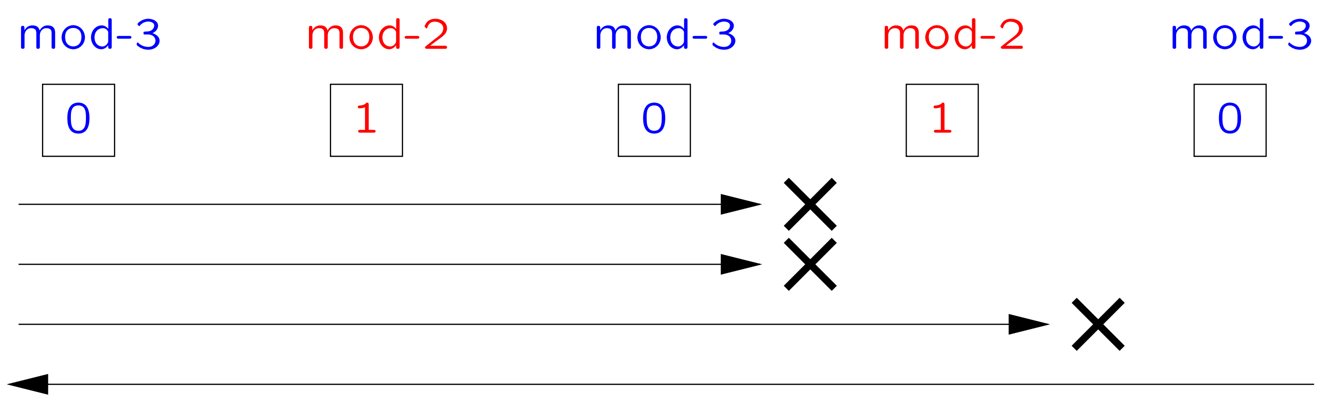
If a set of reads is consistent, there are three cases.

Case 1: No increments in progress.

Case 2: 3 increments are crossed in left half.



Case 3: 3 increments are crossed in right half.



In each case, the values that were read tell us exactly how many increments accessed the middle counter.
$\Rightarrow$ Linearization is correct.

Generalizing the Construction

Theorem Let $m = \text{lcm}(m_1, \ldots, m_r)$.
There is an implementation of an m-valued counter from the set $\{\text{mod} - m_1 \text{ counter}, \ldots, \text{mod} - m_1 \text{ counter}\}$.

Proof uses a larger array of various counters, and the Generalized Chinese Remainder Theorem.

Open Questions

What is computable when n is unknown to processes?

What about models with failures?

How can anonymity be used in a practical way to help protect privacy?