Fully Concurrent Garbage Collection of Actors

Sylvan Clebsch, Sophia Drossopoulou

Imperial College

OOPSLA13

Motivation

- Actor collection is equivalently useful to object collection.
- We can do it manually.
  - But this is a problem for correctness, performance and productivity.
  - It also leads to *less dynamic* actor topologies.
- We can use a tracing collector (Agha).
  - But this is a problem for performance.
  - Actor execution must be halted during tracing.

Goals

- Fully concurrent actor collection.
  - No execution halting at any time.
  - No read or write barriers.
- Use message passing only.
  - No other synchronisation primitives.
  - No reliance on shared memory.

Message-based collection

- We can use reference counting...
- And use messages for increments and decrements...
- And use a form of deferred reference counting for performance...
- ...and we still have cyclic graphs of actors that don't get collected.

Cycle detection

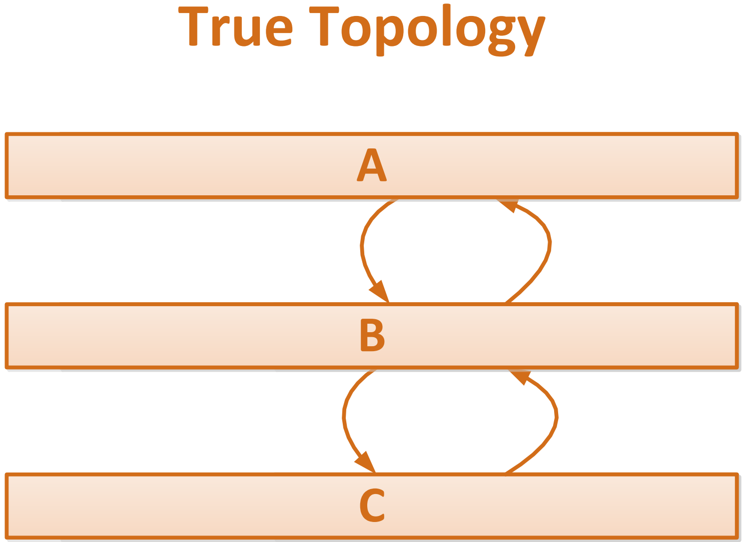
- We'll introduce a dedicated cycle detector actor...
- And actors will tell it when they *block* and *unblock*...
- And when they block, they will tell the CD:
  - Their own reference count.
  - The set of other actors referenced in their working set.
- The CD can then detect cycles and collect them.

This doesn't work

- In-flight (sent but not yet received) messages may alter the topology of the system.
- An actor's view of it's own topology may be *out of date.*
  - Pending reference count increment and decrement messages.
- The CD's view of the global topology may be *out of date.*
  - An actor that reported itself blocked may no longer be blocked.
- The challenge is to correctly determine when the true topology is the same as the perceived topology...
- ...and will stay that way forever.

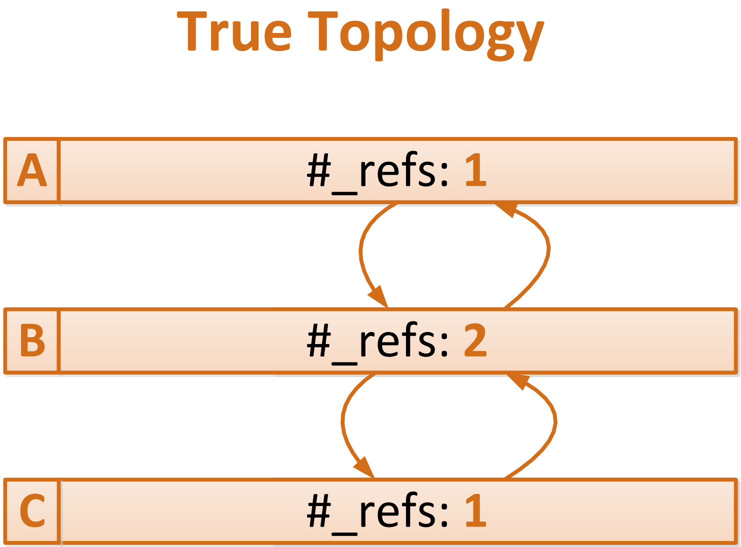
Actors, topology, CD, ext ref cnt -1

Start with three active actors: **A**, **B** and **C**, linked in a chain:



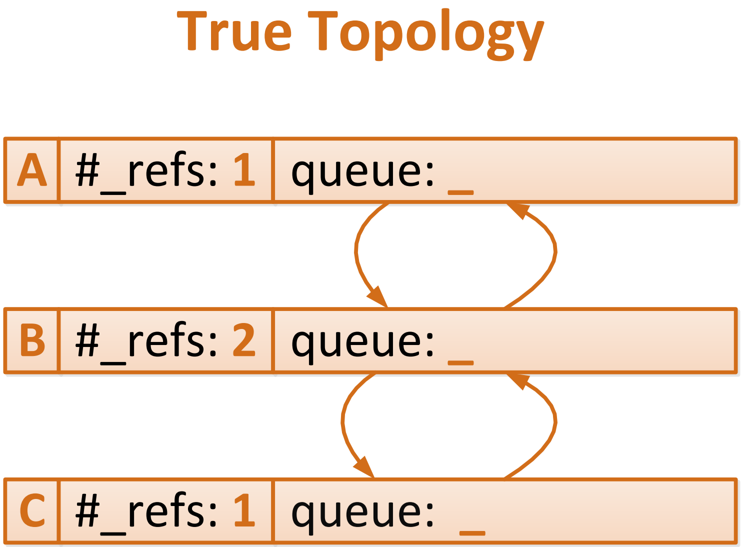
Actors, topology, CD, ext ref cnt - 2

External reference count = # actors referring to current actor:



Actors, topology, CD, ext ref cnt - 3

Moreover, message queues are empty:

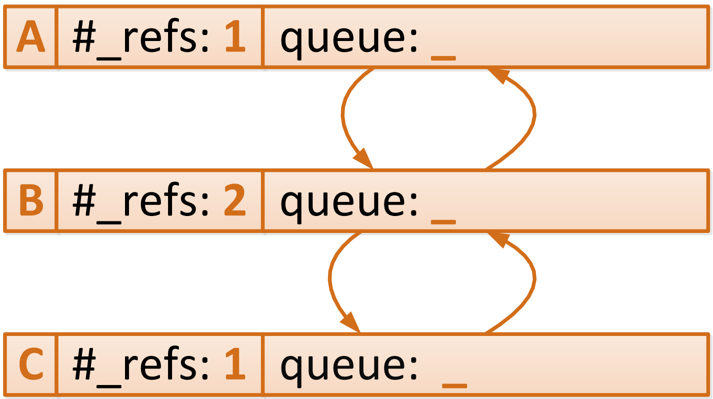


Actors, topology, CD, ext ref cnt - 4

Cycle Detector has not yet received information about **A**, **B**, **C**:

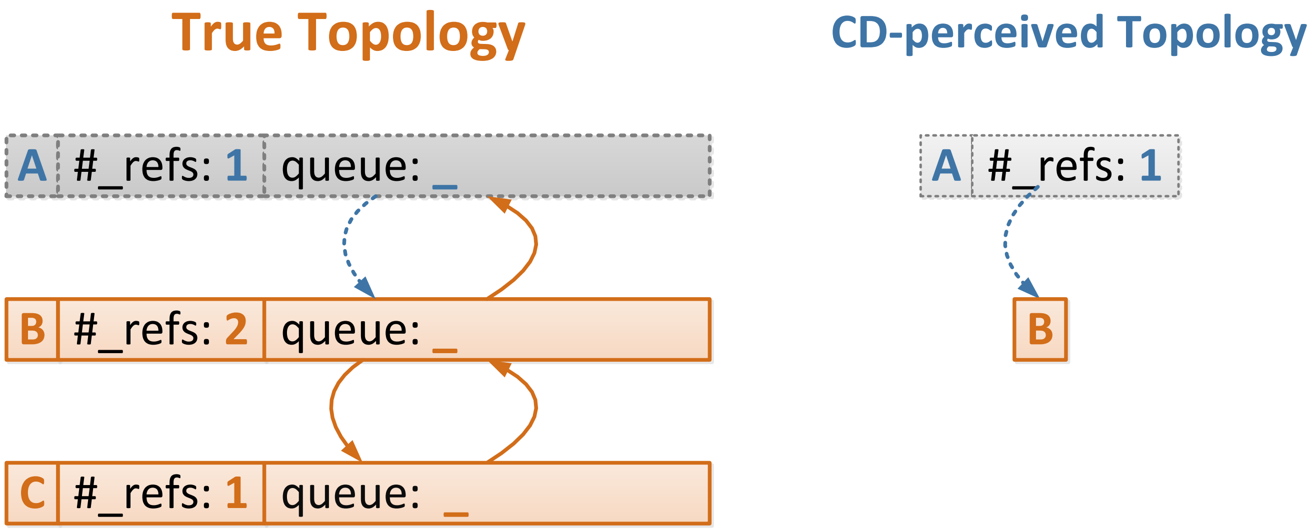
True Topology

CD-perceived Topology



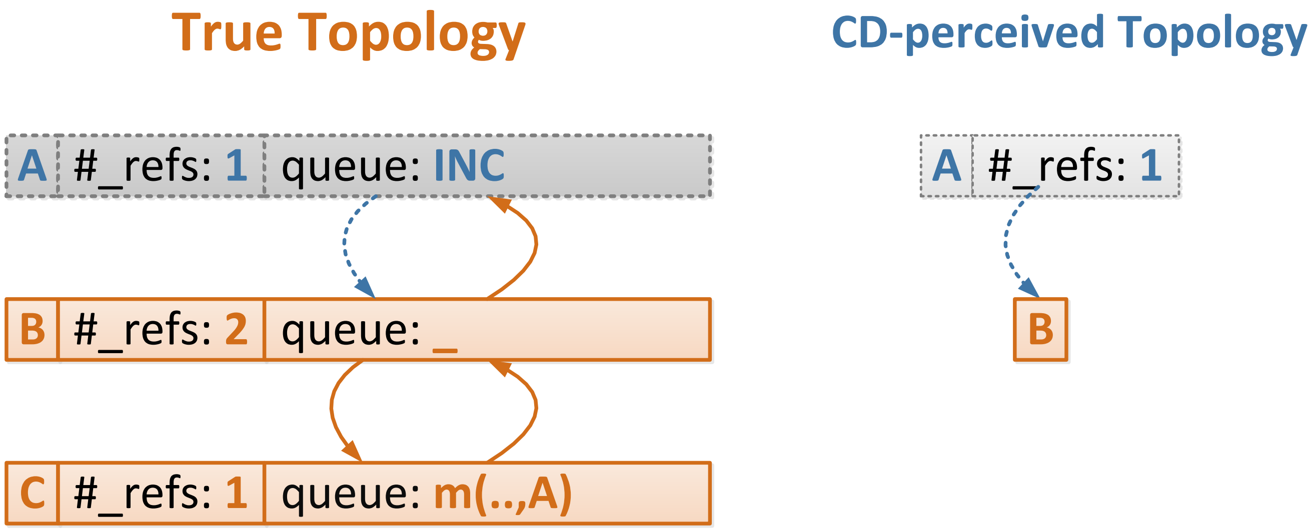
Out of date view - 1

Actor A blocks and informs cycle detector CD:



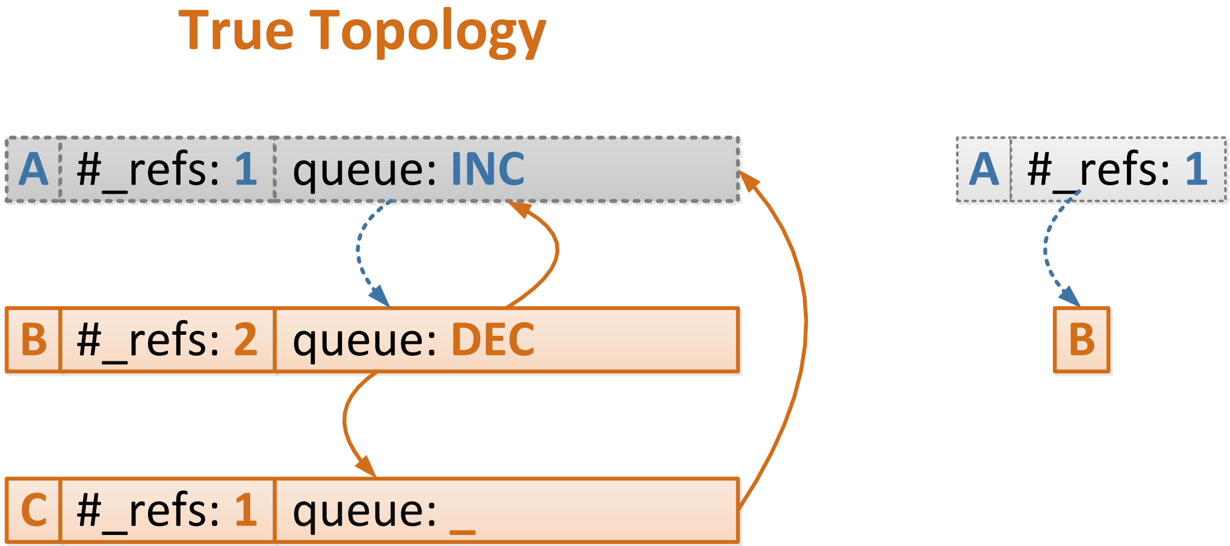
Out of date view - 2

B sends to C some message m containing reference to A:



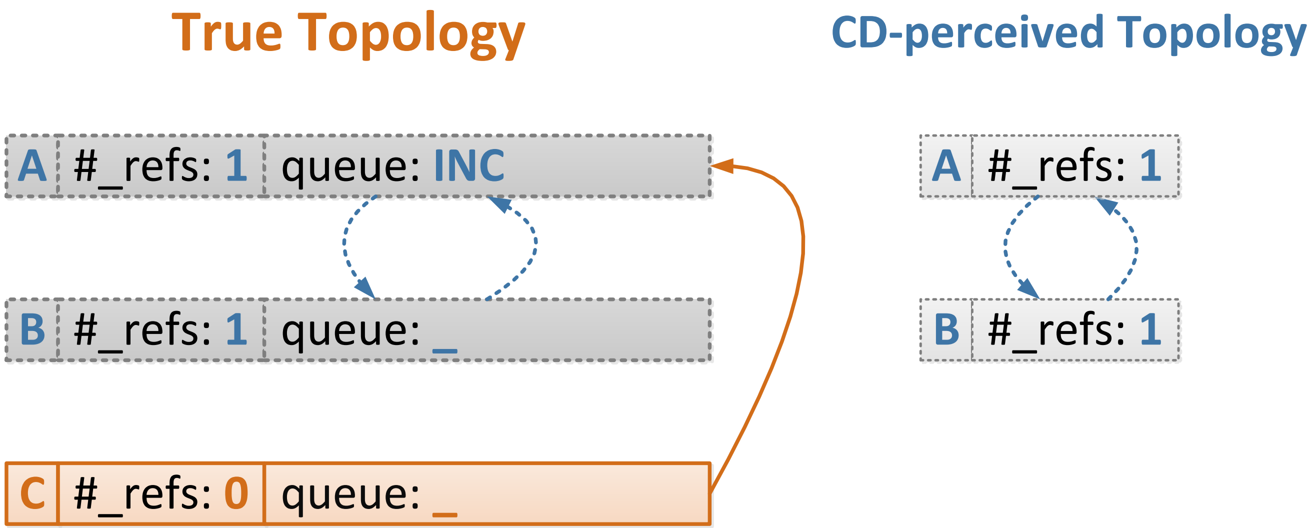
Out of date view - 3

C drops reference to B, and stores reference to A:



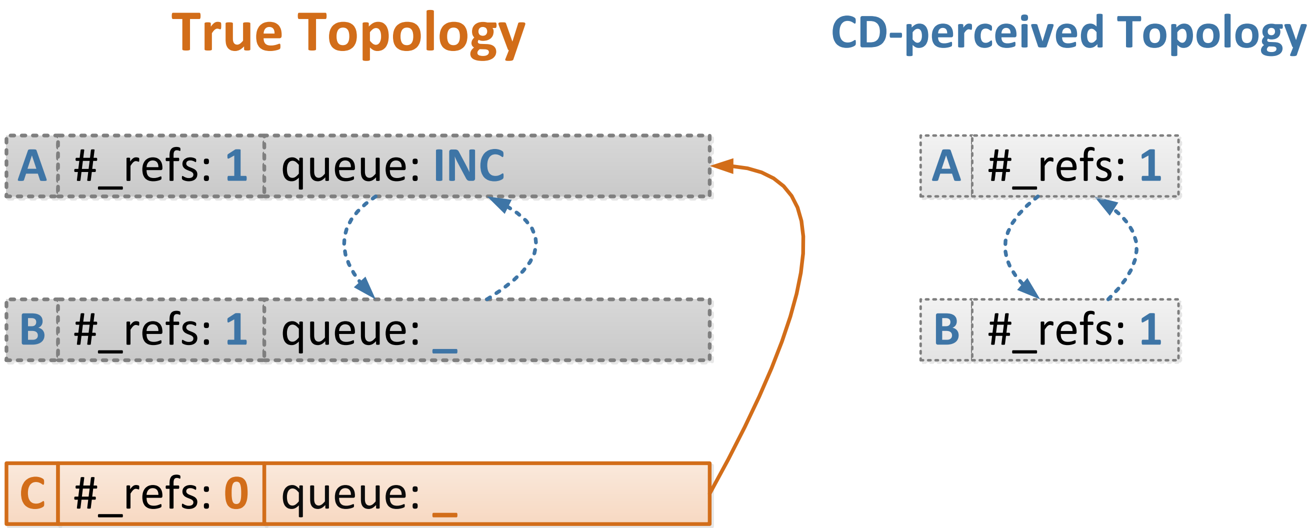
Out of date view - 4

B drops reference to C, processes the DEC message, and blocks:



ERROR!

CD now thinks that **A** and **B** form a cycle and removes them!

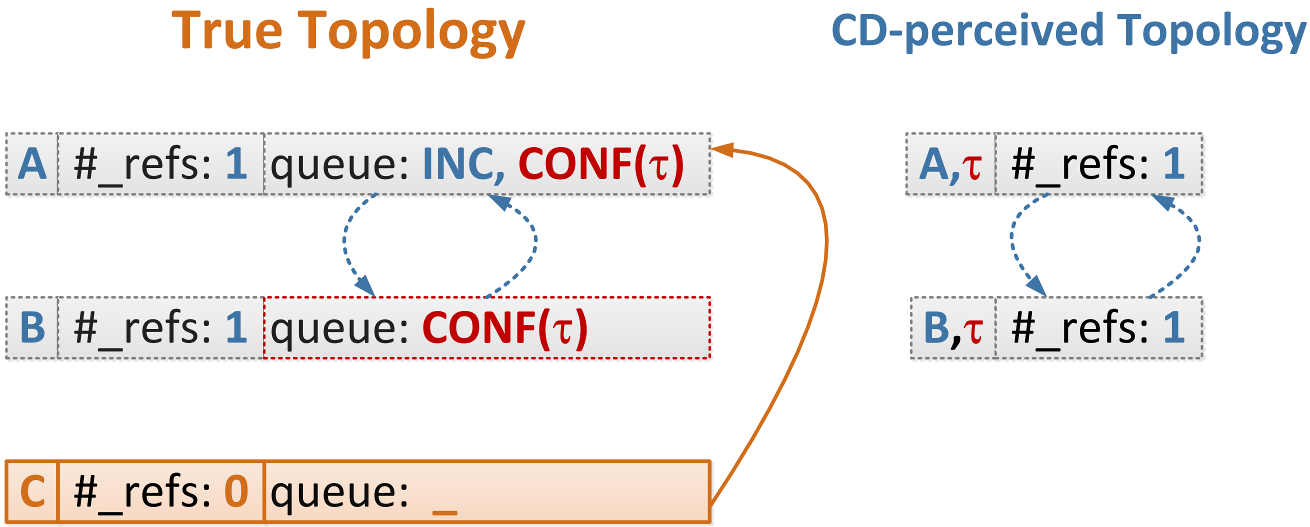


The conf-ack protocol

- We introduce a conf-ack protocol to reconcile out of date views of the topology.
- When the CD detects a *perceived cycle* it does not collect it...
  - It sends a confirmation message to each actor in the cycle.
  - The actors respond with an acknowledge message - always.
- If the CD gets acknowledge messages from all actors without any actor in the cycle unblocking, then the perceived cycle is a *true cycle* and can be collected.
- If any actor in a perceived cycle unblocks before acknowledging, the perceived cycle is discarded.
- This really works! There's a formal proof in the paper.

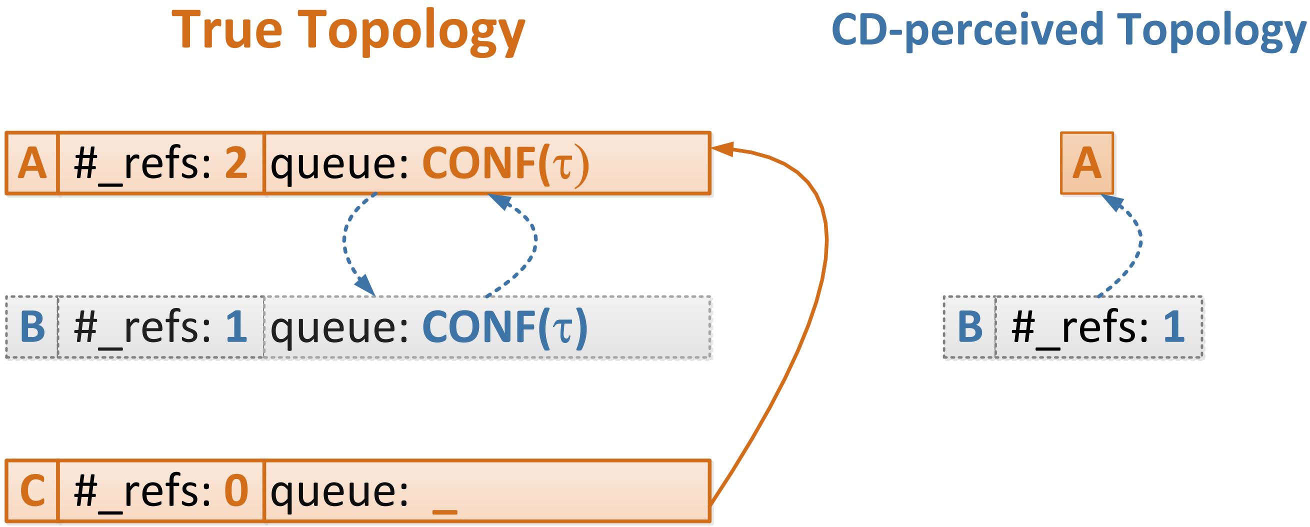
Reconciling the out of date view - 1

- Each perceived cycle is uniquely identified with a token τ
- Instead of collecting, CD sends **confirmation requests**:



Reconciling the out of date view - 2

A unblocks before confirming; **CD** updates perceived topology:



What does acknowledgement mean?

- When the CD receives an acknowledgement message from an actor...
- If that actor is still blocked...
- The CD knows its view of that actor's topology was the true topology at the moment that perceived cycle was detected.

Initial benchmarks

Language	Time (s)	Throughput (msg/s)
Erlang OTP	~9	~333,333
Erlang	~7	~428,571
Scala (react)	~9	~333,333
libcppa	~5.5	~545,454
MAC, disable CD	0.24	12,500,000
MAC, normal CD	0.24	12,500,000
MAC, force CD	0.24	12,500,000

Table : Message handling: 3 million messages, 2 cores

Benchmarking

- There are more benchmarks in the paper, all taken from the excellent benchmarking work done by the libcppa project.
- But we need even more.
- A standard benchmark suite for actor-model languages would be a helpful research tool.

Why did we build this?

- I have a day job at a large financial institution.
  - We build high performance time-dependent event-stream processors.
  - ...just like everyone else.
- Why not use threads, or thread pools, or TBB, or OpenMP, or...?
  - We do! C/C++ with all of the above.
  - Too many programmer errors, too inflexible, too slow.
- Why not just use an existing actor-model language?
  - We do! Erlang, Scala/Java with Akka.
  - Improvement in robustness and flexibility, but not speed.

Production use case

- We create tens of thousands of new actors per second...
  - Each may have a complex relationship with existing actors.
  - Lifetime depends not just on I/O related to that actor...
  - ...but on future I/O with an unknown set of future actors.
- Fast, lightweight actors give us maximum parallelism.
  - But it also means a single process often has millions of actors.
  - And they form many unrelated cyclic graphs.
  - And they have unpredictable lifetimes.
  - Manual lifetime management is much more difficult than manual memory management.

Future work

- We are extending this work to the distributed setting.
    - Distributed causal messaging.
    - Hierarchical cycle detection.
    - Using the conf-ack protocol to solve other issues.
    - ...such as distributed termination detection.
    - ...and transparent actor migration.
- We have developed a capabilities-based type system for data race freedom.
    - We use this to extend this work to cover passive object collection.