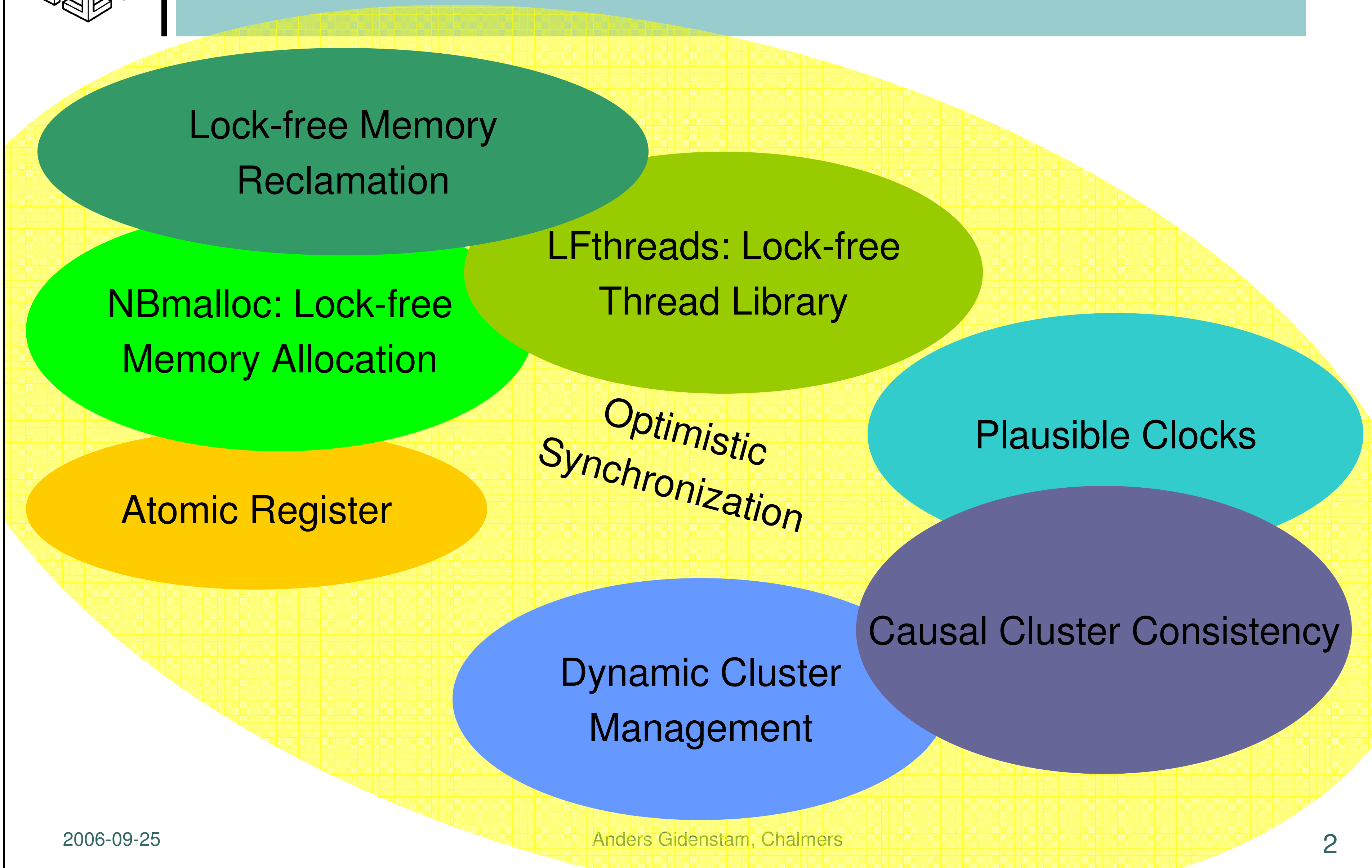
Algorithms for Synchronization and Consistency in Concurrent System Services

Anders Gidenstam

Distributed Computing and Systems group,
Department of Computer Science and Engineering,
Chalmers University of Technology

Research Overview

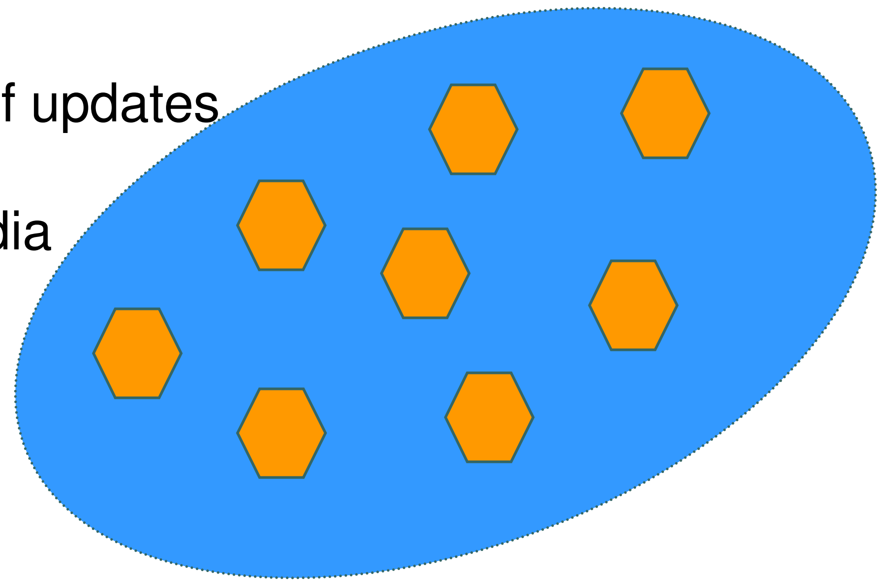


Outline

- Overview
- Sample of results
  - Scalable information dissemination
    - **Causal Cluster Consistency**
    - Plausible Clocks
  - Lock-free algorithms in system services
    - **Memory Management**
    - Threading and thread synchronization library
- Conclusions
- Future work

Causal Cluster Consistency

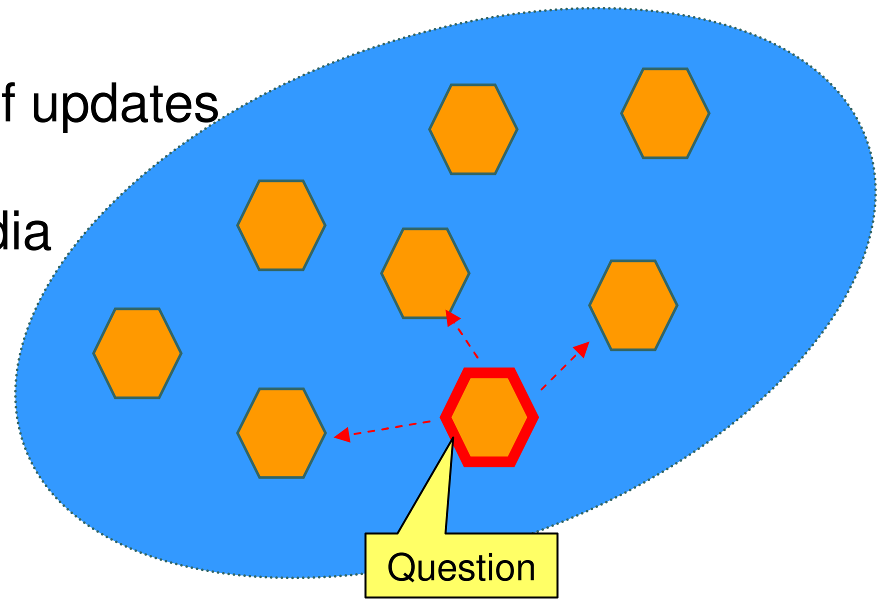
- Goal: Support Distributed Collaborative Applications
  - Provide Consistency (order of updates matter)
  - Scalable communication media
  - Dynamically changing set of participants



- Focus: Group communication
  - Propagate events (updates) to all interested processes
  - Event delivery in *causal order*

Causal Cluster Consistency

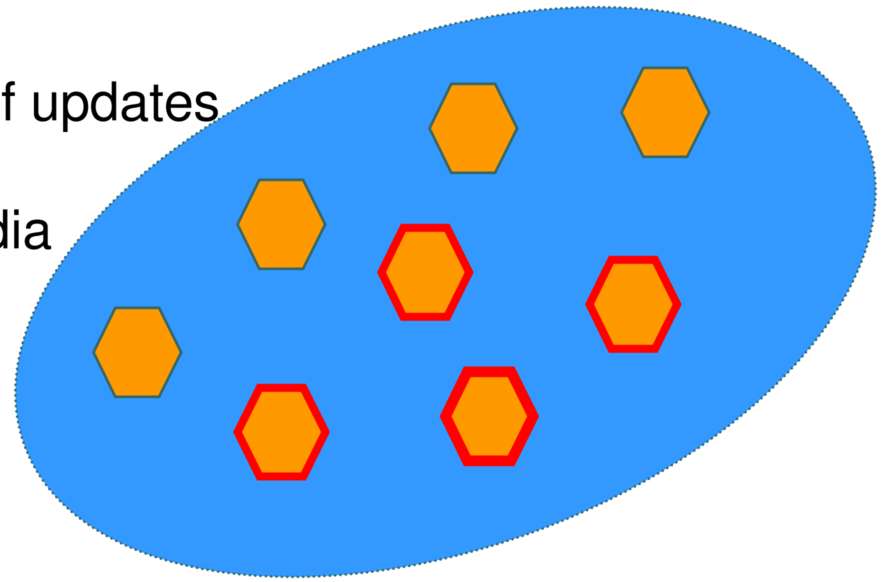
- Goal: Support Distributed Collaborative Applications
  - Provide Consistency (order of updates matter)
  - Scalable communication media
  - Dynamically changing set of participants



- Focus: Group communication
  - Propagate events (updates) to all interested processes
  - Event delivery in *causal order*

Causal Cluster Consistency

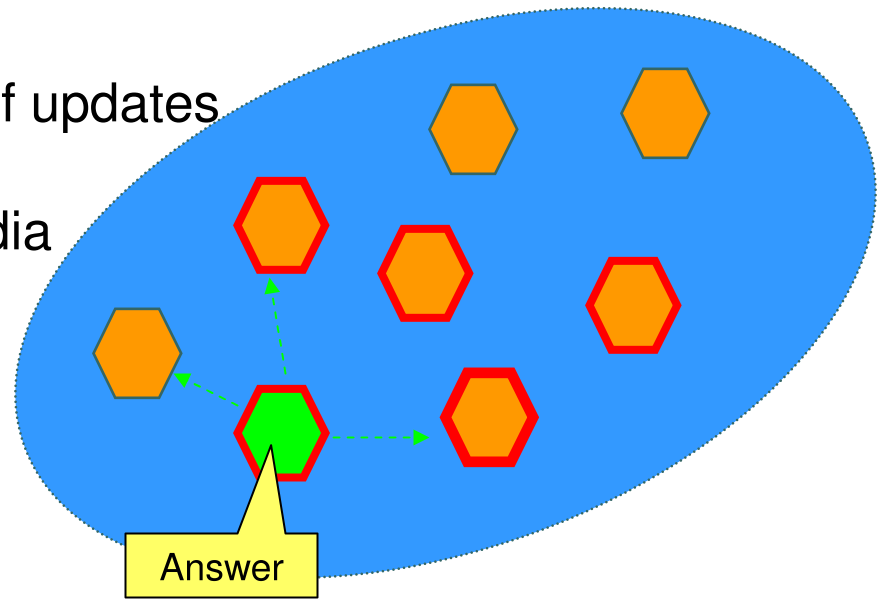
- Goal: Support Distributed Collaborative Applications
  - Provide Consistency (order of updates matter)
  - Scalable communication media
  - Dynamically changing set of participants



- Focus: Group communication
  - Propagate events (updates) to all interested processes
  - Event delivery in *causal order*

Causal Cluster Consistency

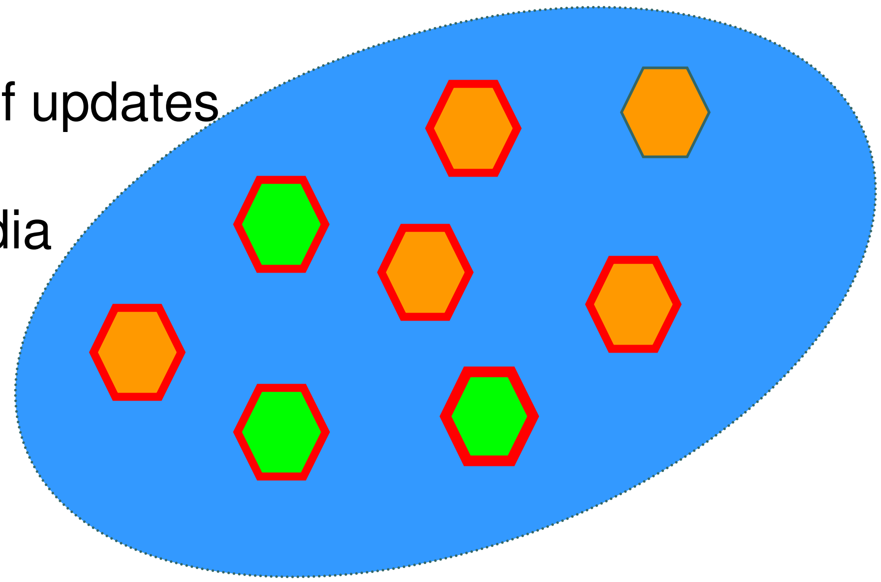
- Goal: Support Distributed Collaborative Applications
  - Provide Consistency (order of updates matter)
  - Scalable communication media
  - Dynamically changing set of participants



- Focus: Group communication
  - Propagate events (updates) to all interested processes
  - Event delivery in *causal order*

Causal Cluster Consistency

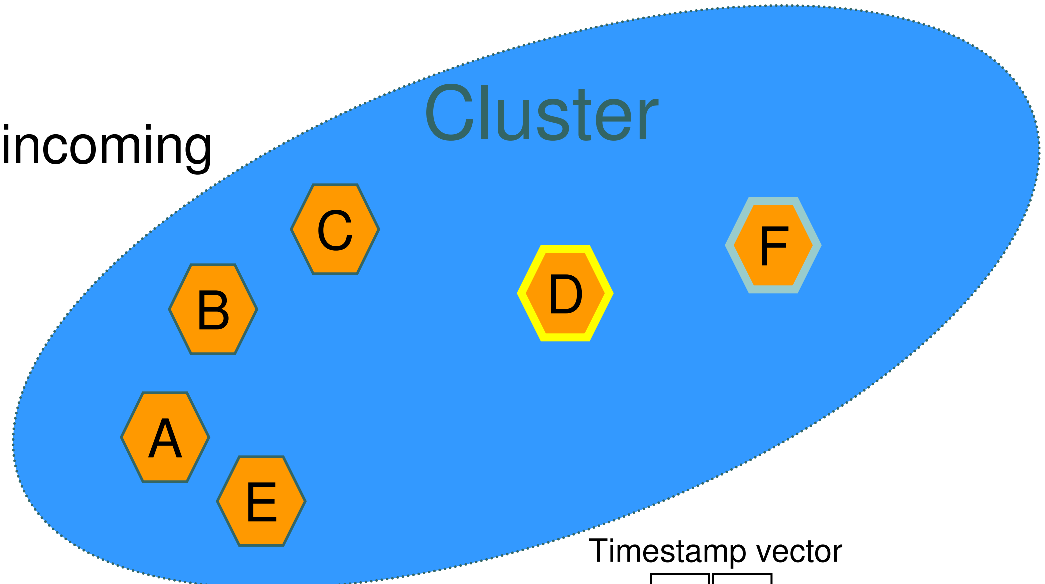
- Goal: Support Distributed Collaborative Applications
  - Provide Consistency (order of updates matter)
  - Scalable communication media
  - Dynamically changing set of participants



- Focus: Group communication
  - Propagate events (updates) to all interested processes
  - Event delivery in *causal order*

Causal Cluster Consistency: Optimistic causally ordered delivery

- Basic algorithm:
  - Vector clocks + queue of incoming events
    [Ahamad et al. 1996]
- Improvements
  - Delivery with high probability
  - Limited per-user domain of interest:
    - Nobody is interested in changing everything at once
  - Often more observers than updaters
  - Events have lifetimes/deadlines
    [Baldoni et al. 1998]

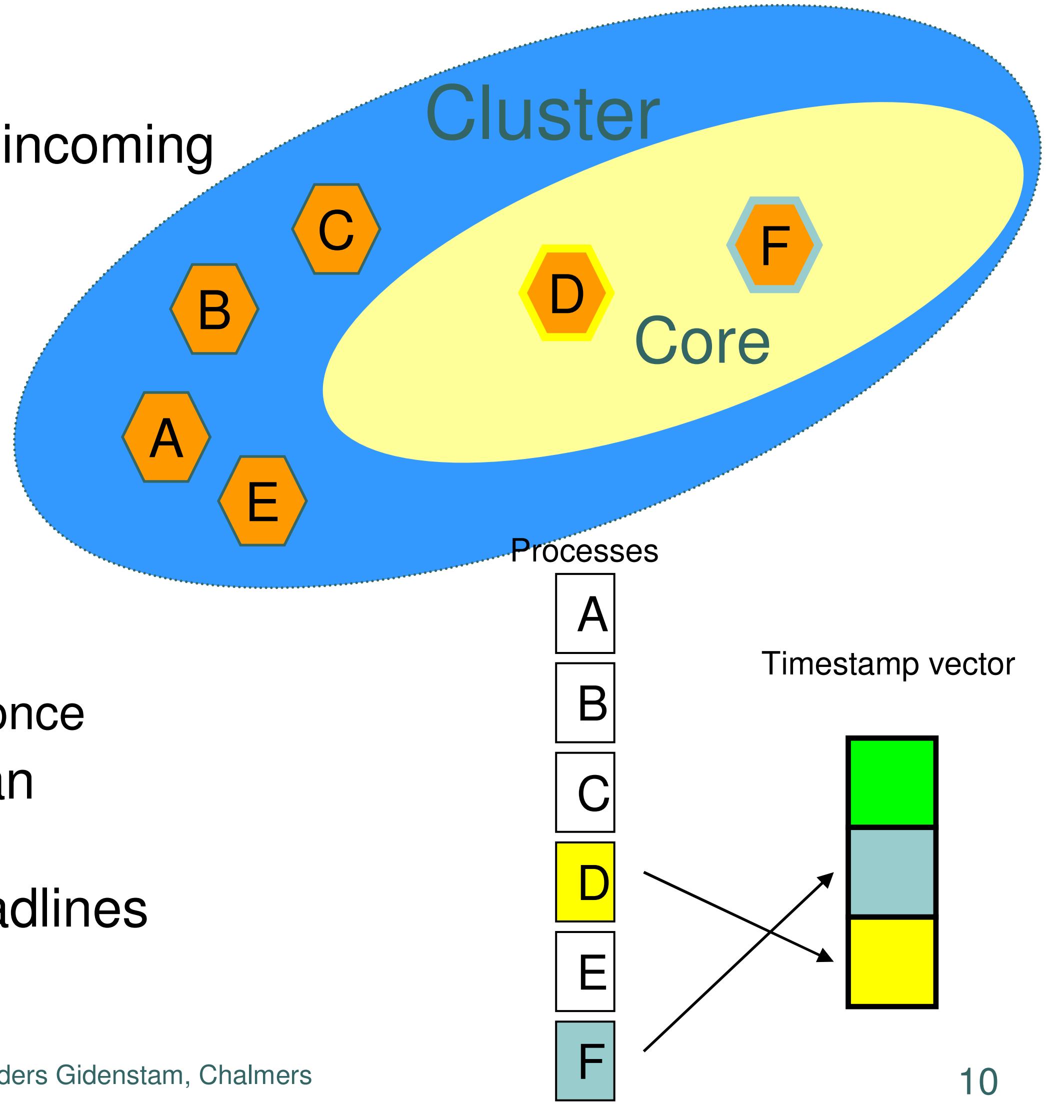


Timestamp vector

A	
B	
C	
D	3
E	
F	5

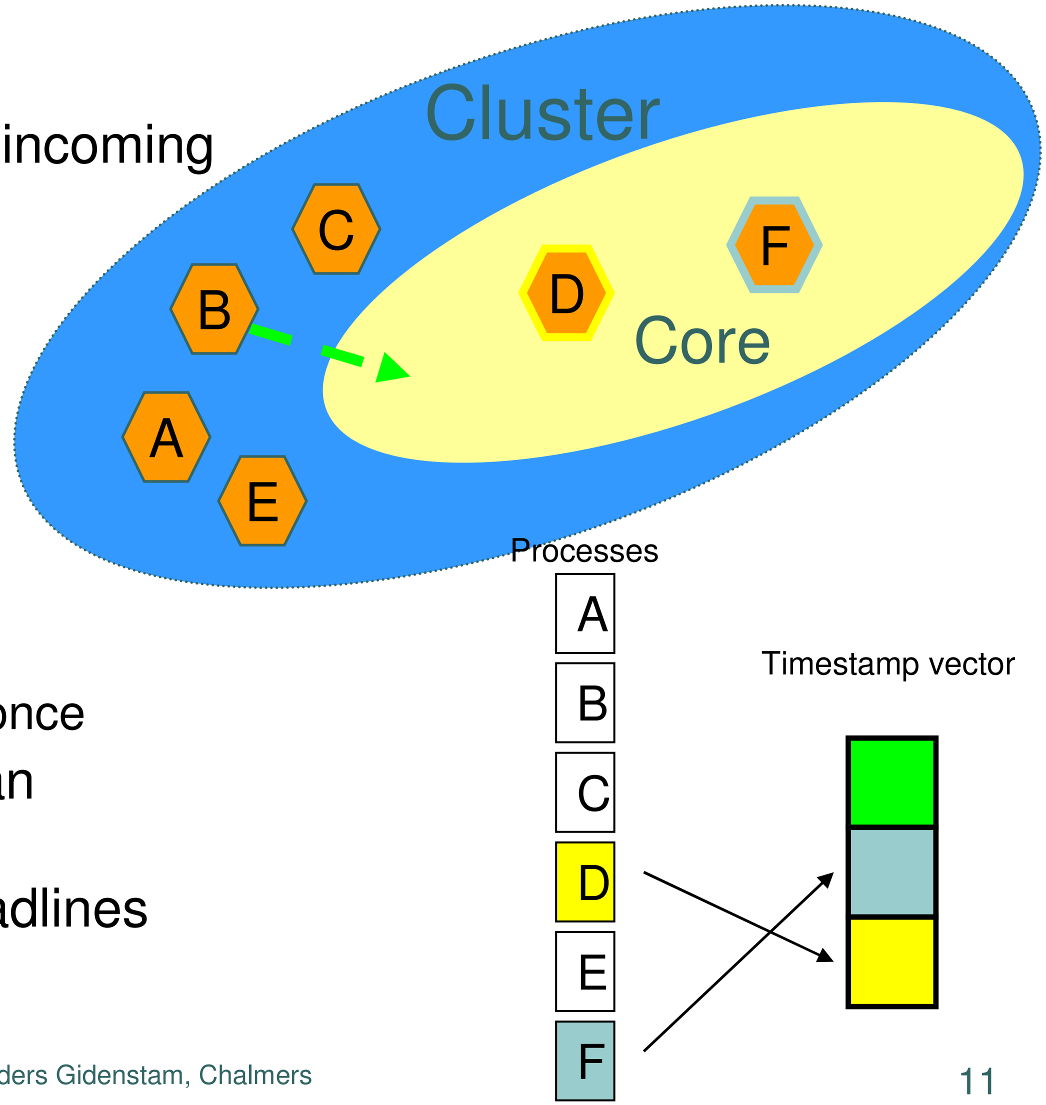
Causal Cluster Consistency: Optimistic causally ordered delivery

- Basic algorithm:
  - Vector clocks + queue of incoming events [Ahamad et al. 1996]
- Improvements
  - Delivery with high probability
  - Limited per-user domain of interest:
    - Nobody is interested in changing everything at once
  - Often more observers than updaters
  - Events have lifetimes/deadlines [Baldoni et al. 1998]



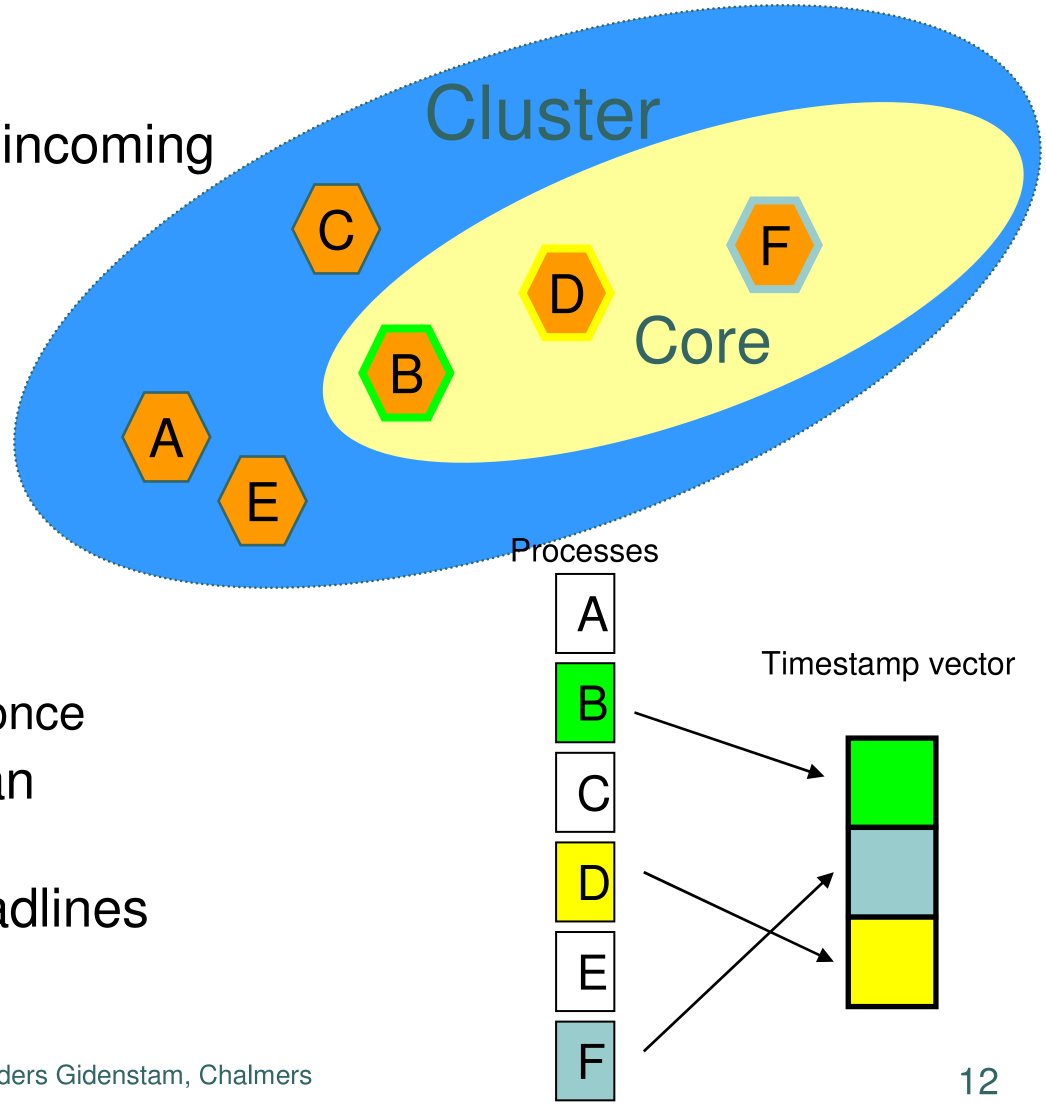
Causal Cluster Consistency: Optimistic causally ordered delivery

- Basic algorithm:
  - Vector clocks + queue of incoming events [Ahamad et al. 1996]
- Improvements
  - Delivery with high probability
  - Limited per-user domain of interest:
    - Nobody is interested in changing everything at once
  - Often more observers than updaters
  - Events have lifetimes/deadlines [Baldoni et al. 1998]



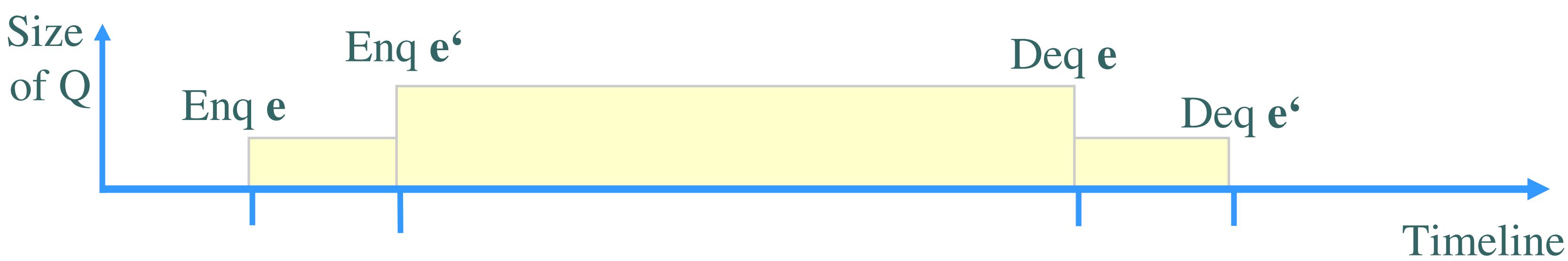
Causal Cluster Consistency: Optimistic causally ordered delivery

- Basic algorithm:
  - Vector clocks + queue of incoming events
    [Ahamad et al. 1996]
- Improvements
  - Delivery with high probability
  - Limited per-user domain of interest:
    - Nobody is interested in changing everything at once
  - Often more observers than updaters
  - Events have lifetimes/deadlines
    [Baldoni et al. 1998]



Event recovery & bounds

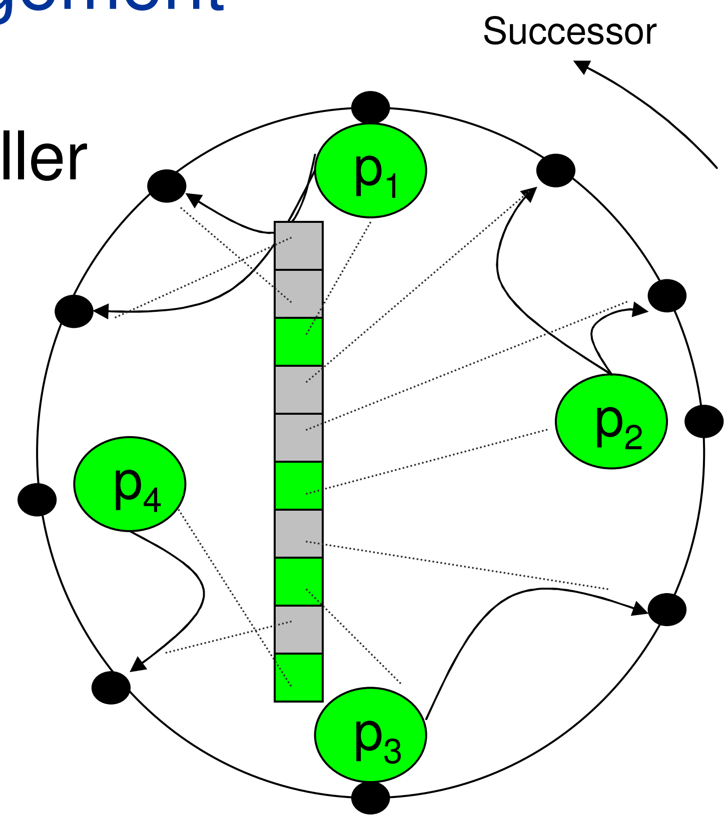
- Recovery buffer of observed events
- "Pull" missing events: request from *k* processes



- analyse relation between adversary's Q and buffer size:
  - **| Buffer | > 2npT** => availability of an event w.h.p. > $\mathbf{1-(e/4)^{npT}}$
- processes fail independently with probability q<k/(2n)
  - **request recovery from *k*** => w.h.p. (>$\mathbf{1-(e/4)^{nq}}$) response

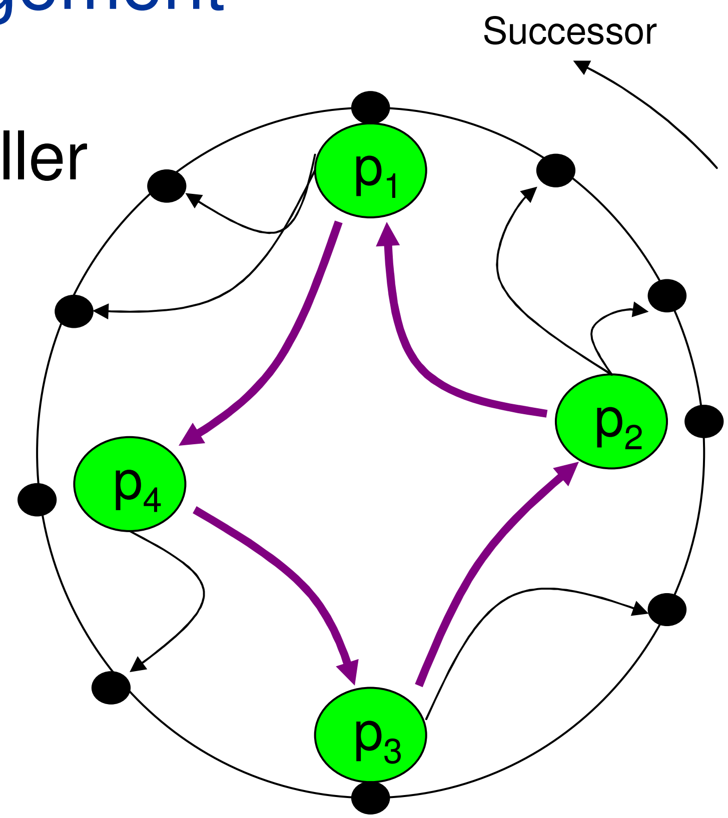
Cluster Core Management

- Fault-tolerant cluster management algorithm:
  - Dynamic who-is-who in smaller group (combined with dissemination)
  - Assigns unique id/ticket/ entry to each core process.
  - Inspired by algorithms for distributed hash tables
  - Tolerates bounded # failures
    - Process stop failures
    - Communication failures
  - Reclaims tickets (ids) from crashed processes



Cluster Core Management

- Fault-tolerant cluster management algorithm:
  - Dynamic who-is-who in smaller group (combined with dissemination)
  - Assigns unique id/ticket/ entry to each core process.
  - Inspired by algorithms for distributed hash tables
  - Tolerates bounded # failures
    - Process stop failures
    - Communication failures
  - Reclaims tickets (ids) from crashed processes



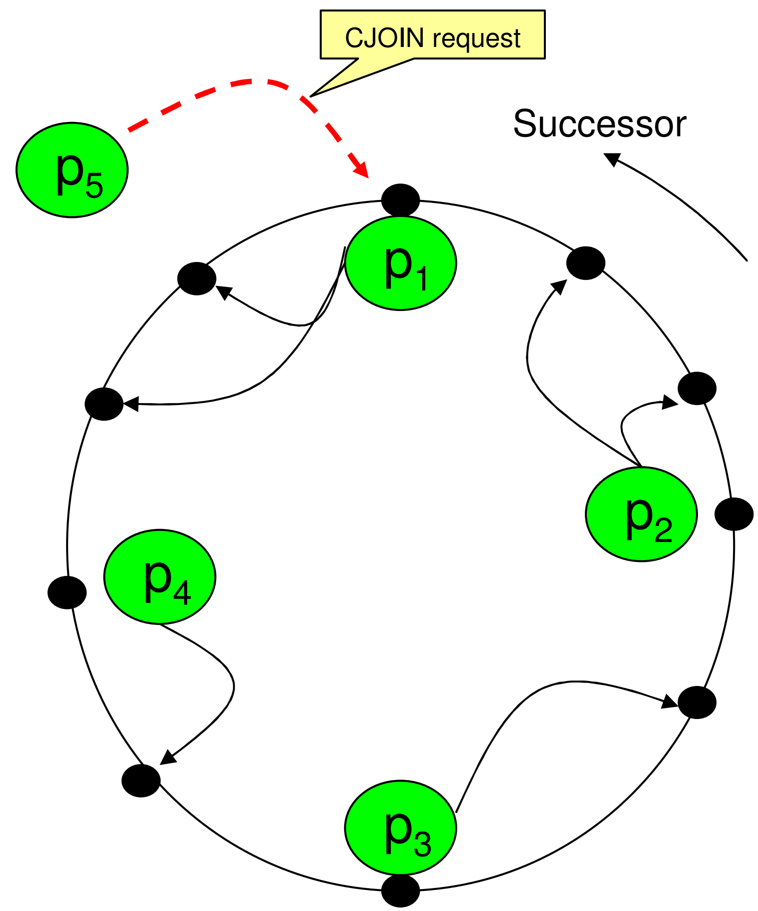
Cluster Management Algorithm

Recall:

- Each coordinator manages the tickets between itself and its successor

Joining the core

- Contact any coordinator
- Coordinator assigns ticket
- Inform the other processes about the new core member



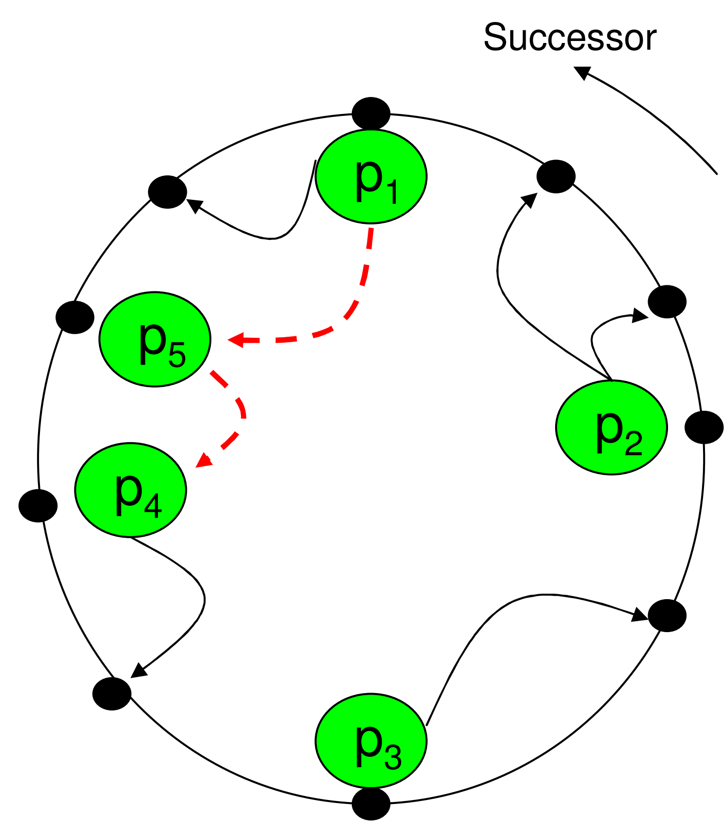
Cluster Management Algorithm

Recall:

- Each coordinator manages the tickets between itself and its successor

Joining the core

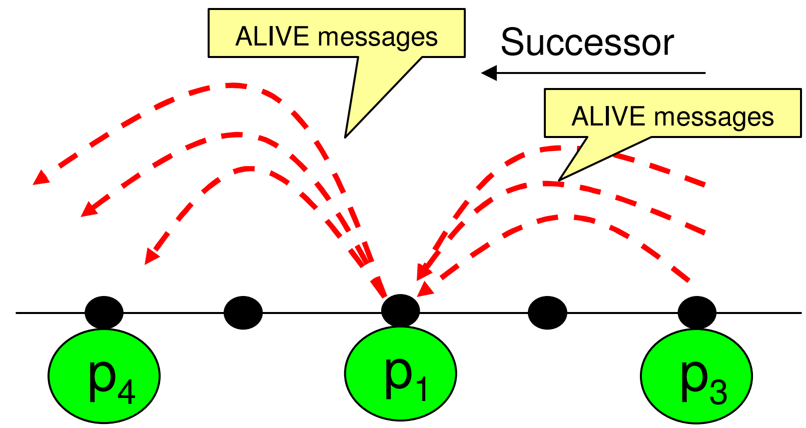
- Contact any coordinator
- Coordinator assigns ticket
- Inform the other processes about the new core member



Cluster Management Algorithm: Dealing with Failures

Failure detection

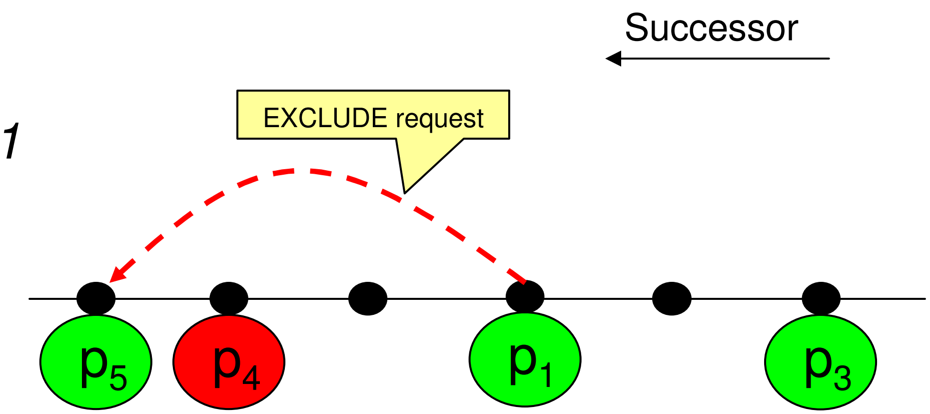
- Each process in each round
  - Sends ALIVE messages to *2k+1* closest successors
  - Needs to receive *k+1* ALIVE messages from known predecessors to stay alive



Cluster Management Algorithm: Dealing with Failures

Failure detection

- Each process in each round
  - Sends ALIVE messages to *2k+1* closest successors
  - Needs to receive *k+1* ALIVE messages from known predecessors to stay alive



Failure handling

- Suspects successor to have failed:
  - Exclusion algorithm contacts the next closest successor

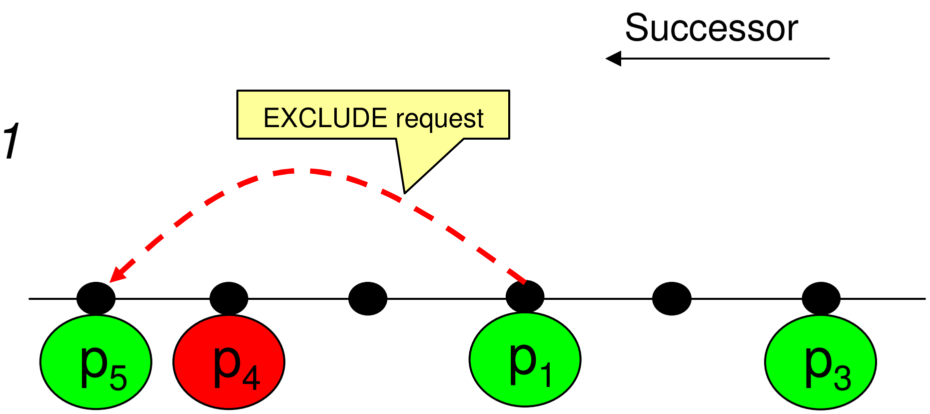
Cluster Management Algorithm:
Dealing with Failures

Failure detection

- Each process in each round
  - Sends ALIVE messages to *2k+1* closest successors
  - Needs to receive *k+1* ALIVE messages from known predecessors to stay alive

Failure handling

- Suspects successor to have failed:
  - Exclusion algorithm contacts the next closest successor



Tolerates *k* failures in a *2k +1* neighbourhood.

Summary cluster consistency

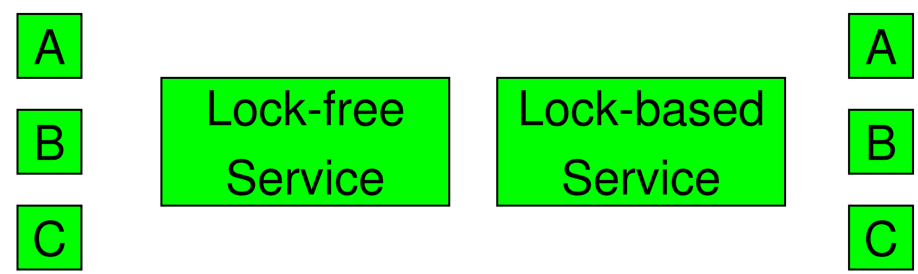
- Causal Cluster Consistency
  - Interesting for collaborative applications and environments
  - Preserves optimistic causal order relations
  - Good predictable delivery guarantees
  - Scalability
- Event Recovery Algorithm
  - Can decrease delivery latency
  - Good match with protocols providing delivery w.h.p.
- Fault-tolerant cluster management algorithm
  - Can support scalable and reliable peer-to-peer services:
    - Resource and parallelism control
  - Good availability of tickets in the occurrence of failures
  - Low message overhead

Outline

- Overview
- Sample of results
  - Scalable information dissemination
    - **Causal Cluster Consistency**
    - Plausible Clocks
  - Lock-free algorithms in system services
    - **Memory Management**
    - Threading and thread synchronization library
- Conclusions
- Future work

Memory Management for Concurrent Applications

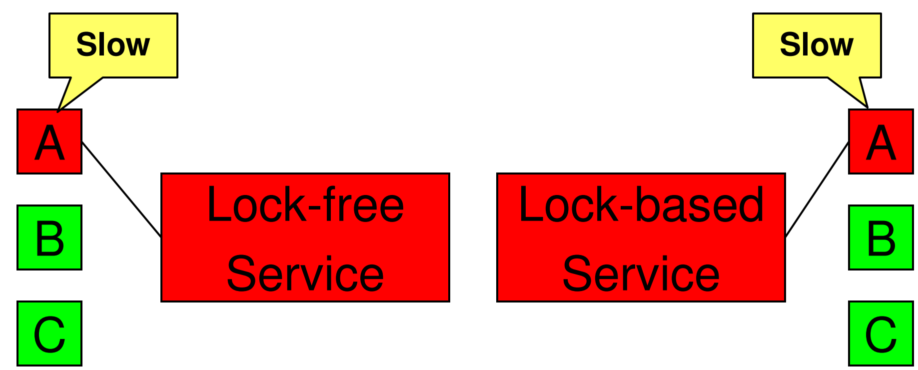
- Applications want to
  - Allocate memory dynamically
  - Use
  - Return/deallocate
- Concurrent applications
  - Memory is a shared resource
  - Concurrent memory requests
  - Potential problems: contention, blocking, lock convoys etc



Memory Management for Concurrent Applications

Why lock-free system services?

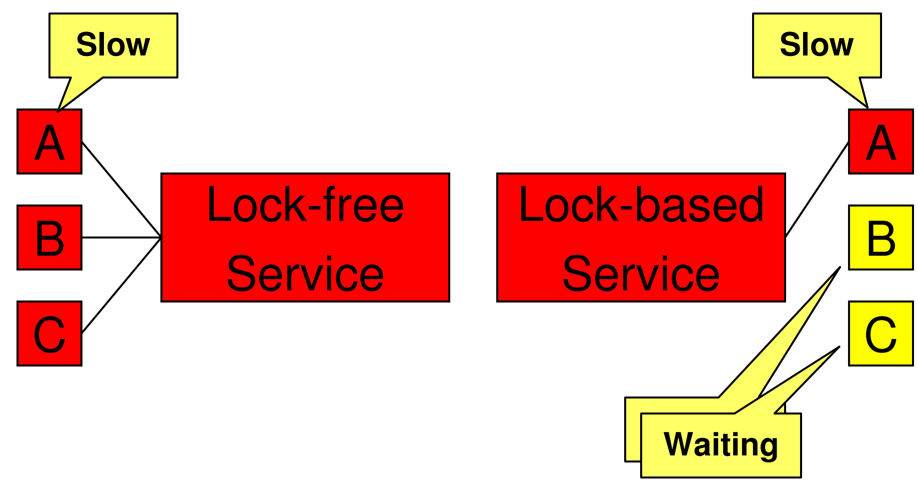
- Scalability/fault-tolerance potential
  - Prevents a delayed thread from blocking other threads
    - Scheduler decisions
    - Page faults etc
- Many non-blocking algorithms uses dynamic memory allocation
  - => non-blocking memory management needed



Memory Management for Concurrent Applications

Why lock-free services?

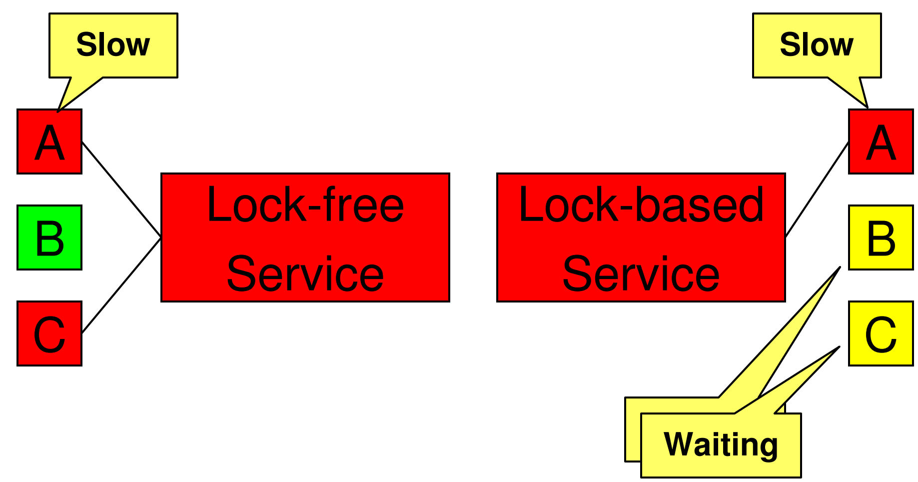
- Scalability/fault-tolerance potential
  - Prevents a delayed thread from blocking other threads
    - Scheduler decisions
    - Page faults etc
- Many non-blocking algorithms uses dynamic memory allocation
  - => non-blocking memory management needed



Memory Management for Concurrent Applications

Why lock-free services?

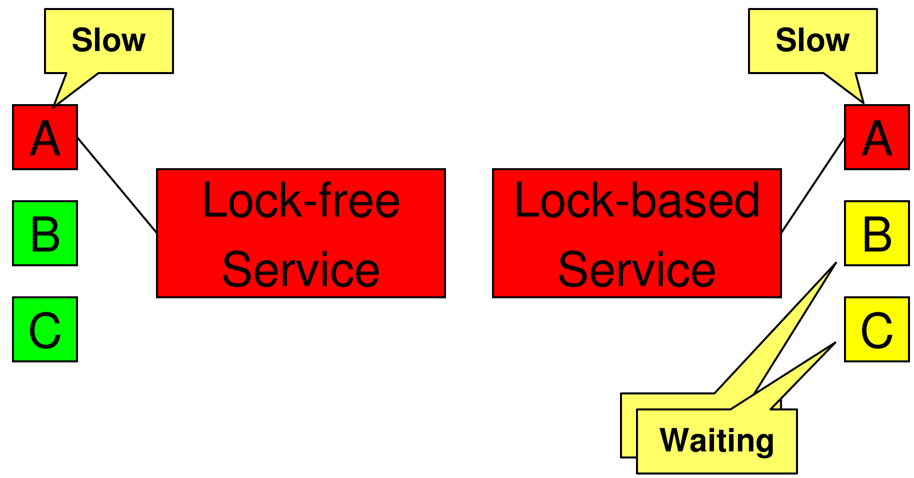
- Scalability/fault-tolerance potential
  - Prevents a delayed thread from blocking other threads
    - Scheduler decisions
    - Page faults etc
- Many non-blocking algorithms uses dynamic memory allocation
  - => non-blocking memory management needed



Memory Management for Concurrent Applications

Why lock-free services?

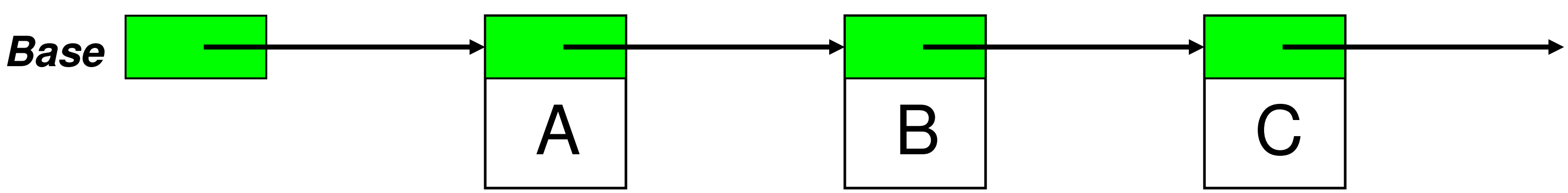
- Scalability/fault-tolerance potential
  - Prevents a delayed thread from blocking other threads
    - Scheduler decisions
    - Page faults etc
- Many non-blocking algorithms use dynamic memory allocation
  - => non-blocking memory management needed



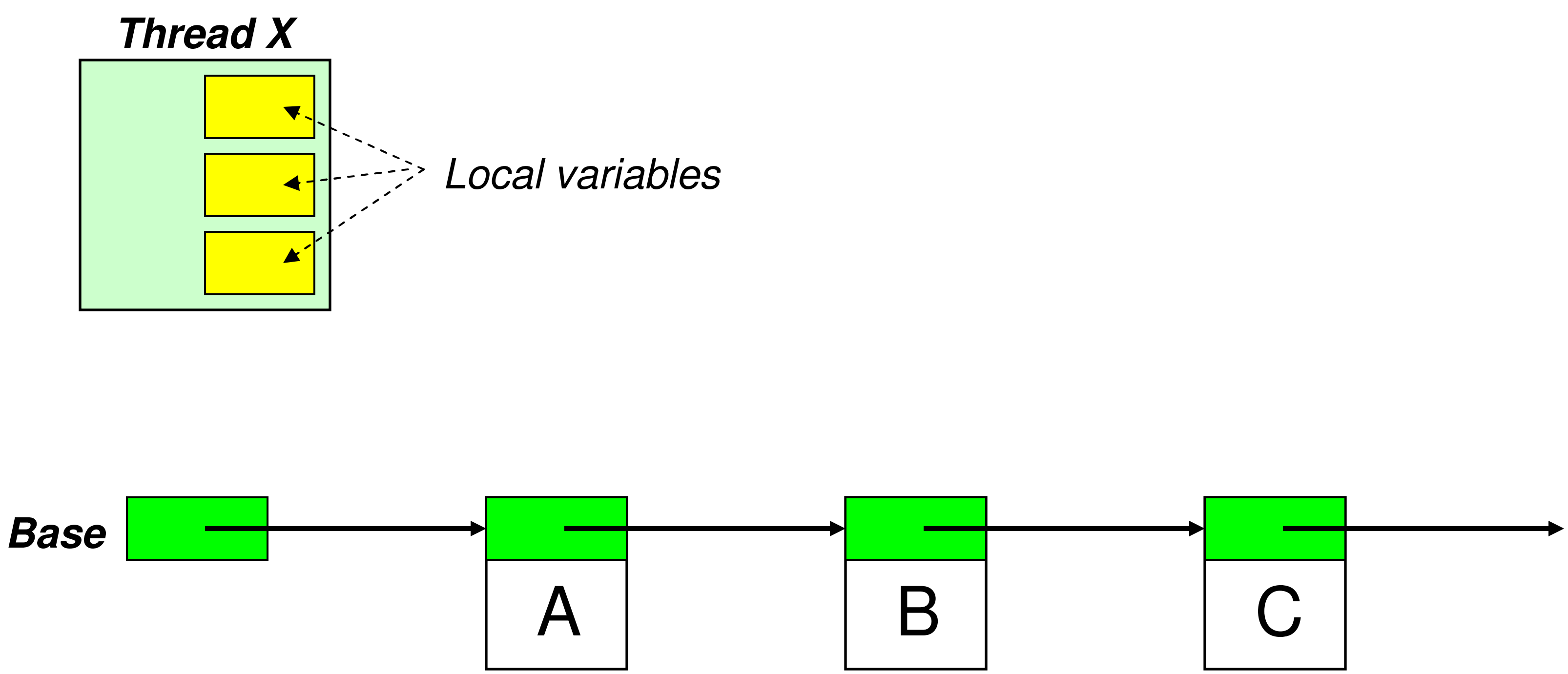
The Lock-Free Memory Reclamation Problem

- Concurrent shared data structures with
  - Dynamic use of shared memory
  - Concurrent and overlapping operations by threads or processes

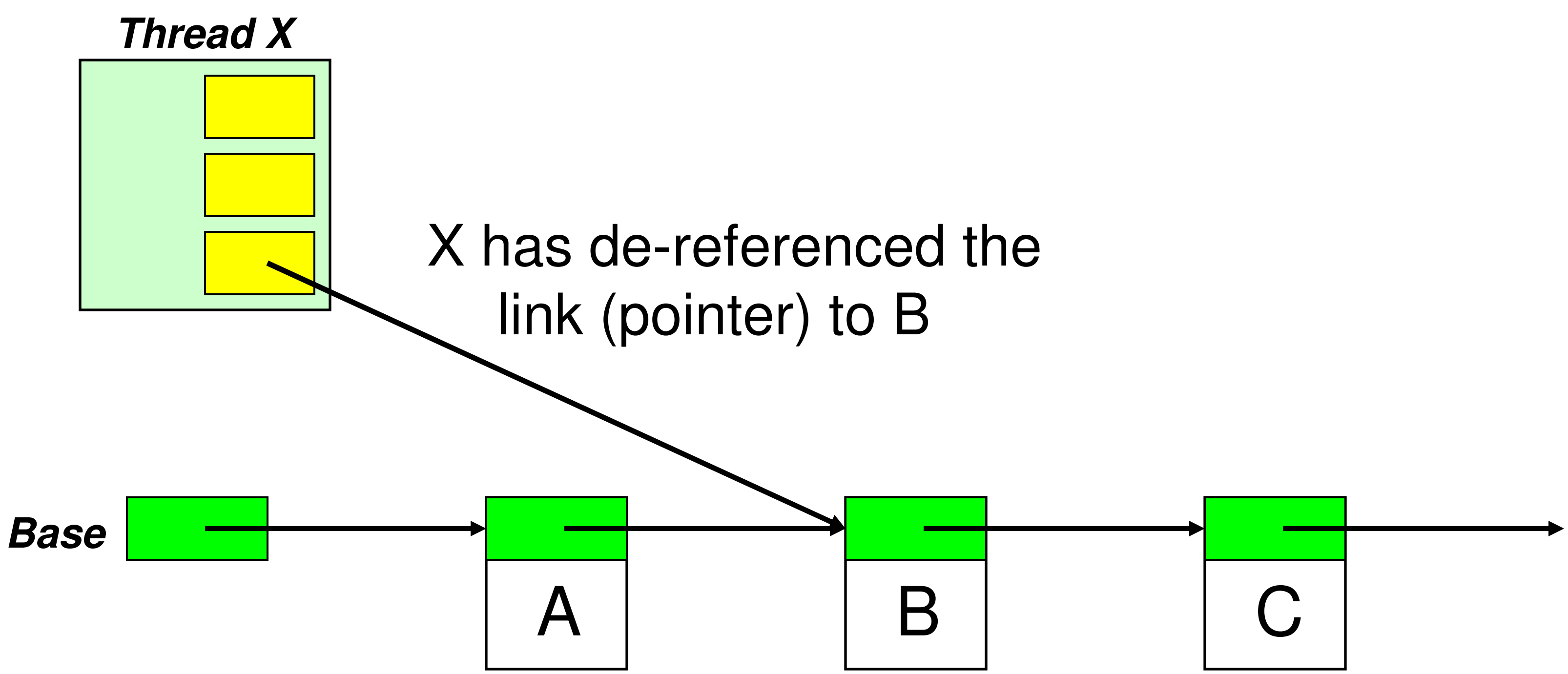
Can nodes be deleted and reused safely?



The Lock-Free Memory Reclamation Problem

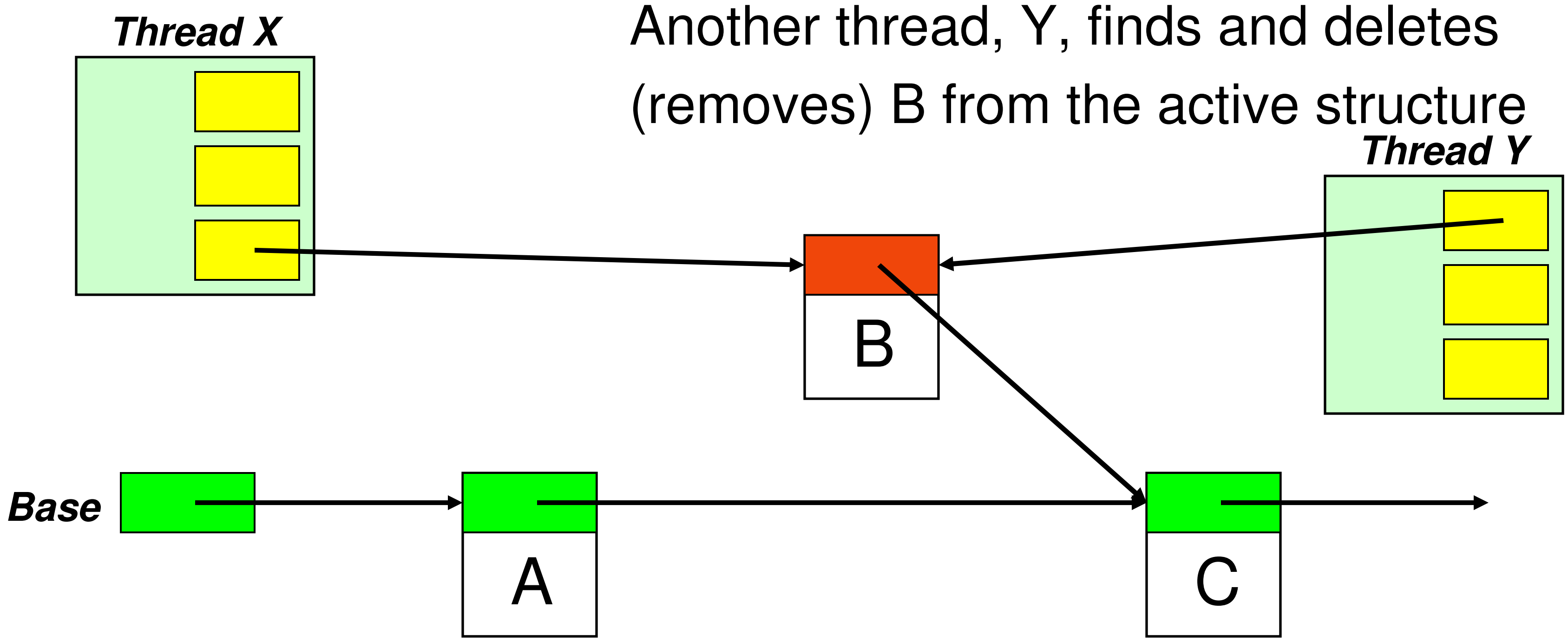


The Lock-Free Memory Reclamation Problem



The Lock-Free Memory Reclamation Problem

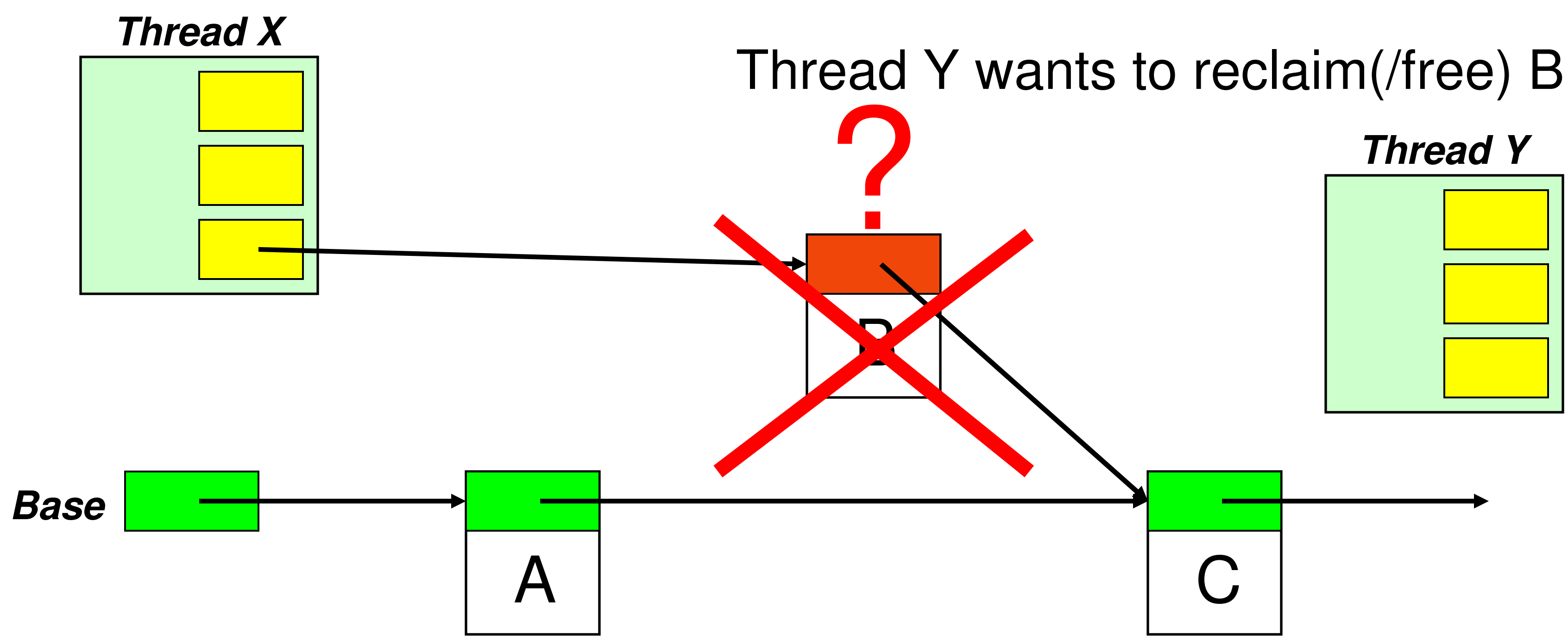
Another thread, Y, finds and deletes (removes) B from the active structure



The Lock-Free Memory Reclamation Problem

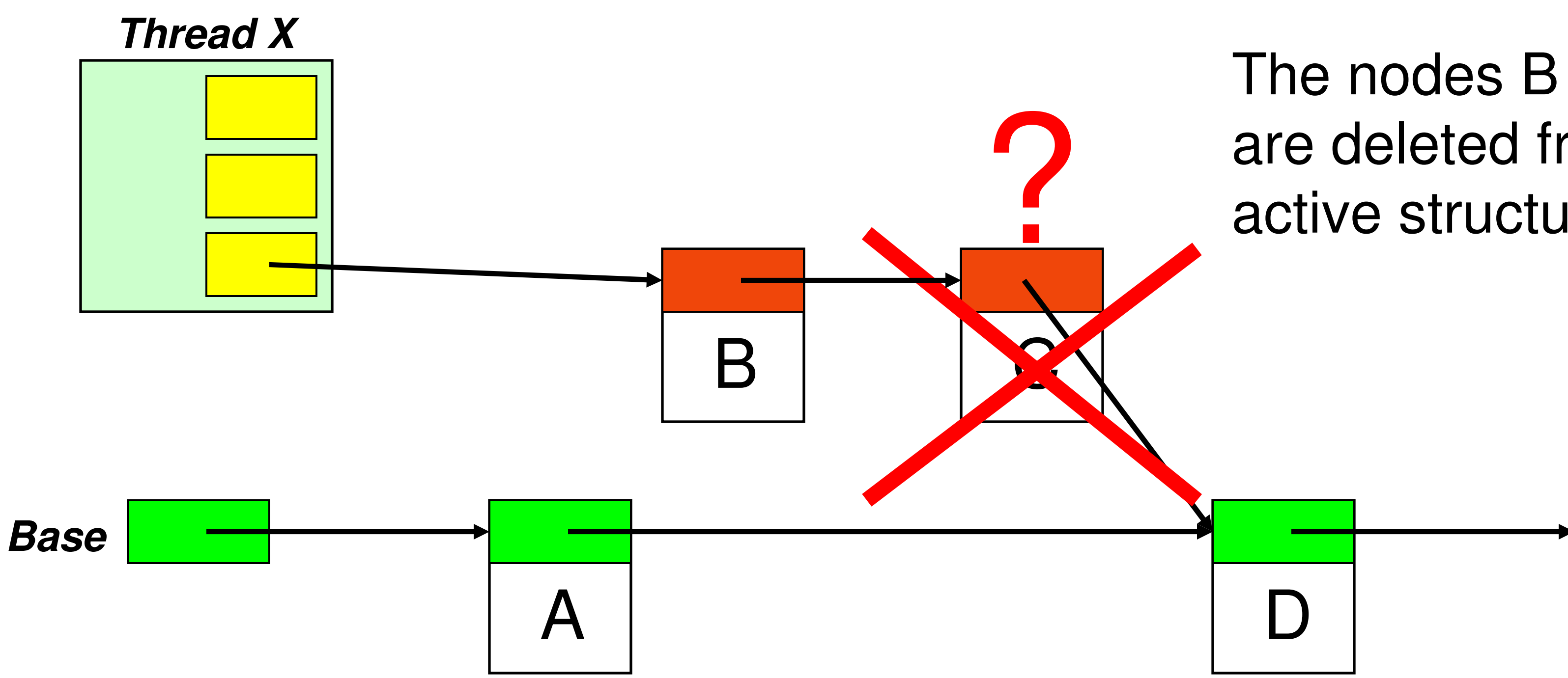
Property I: A (de-)referenced node is not reclaimed

Thread Y wants to reclaim(/free) B



The Lock-Free Memory Reclamation Problem

Property II: Links in a (de-)referenced node should always be de-referencable.



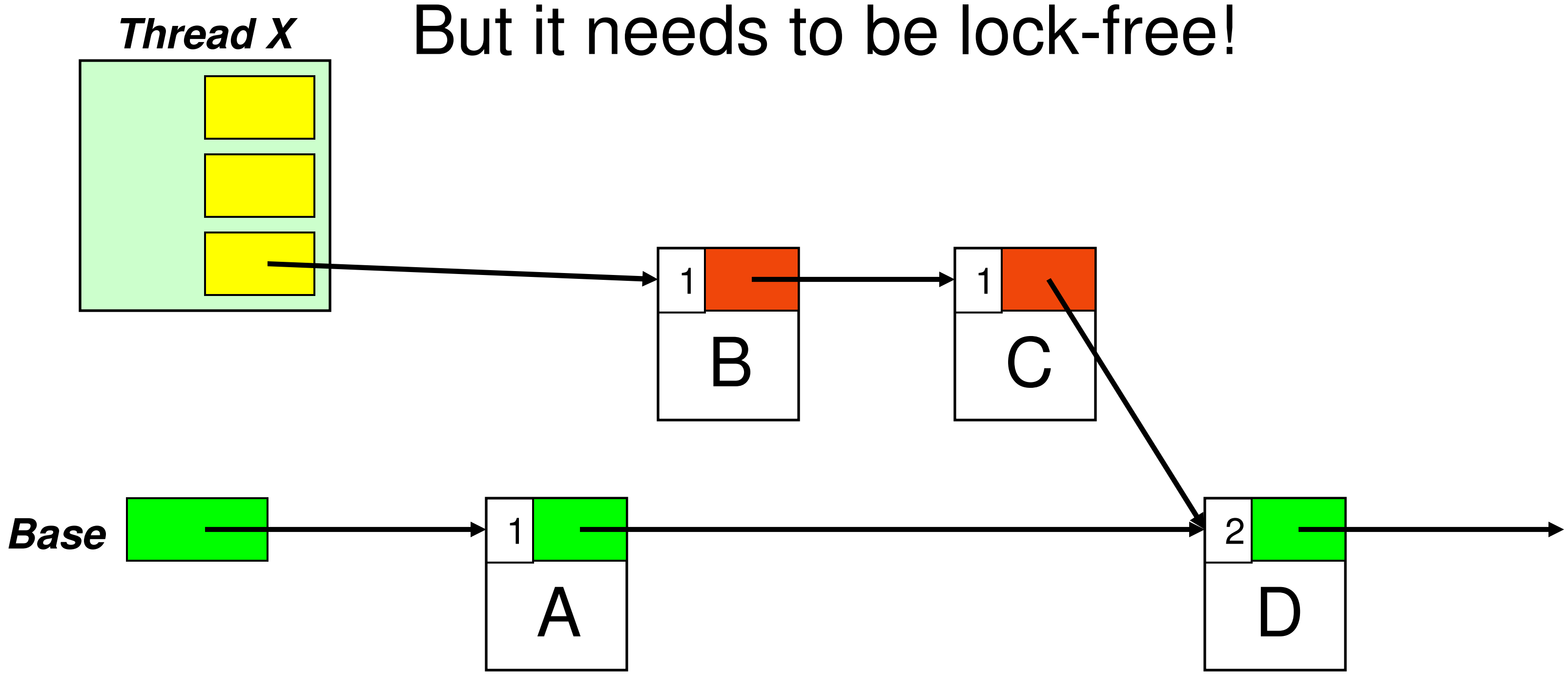
The nodes B and C are deleted from the active structure.

The Lock-Free Memory Reclamation Problem

Reference counting can guarantee

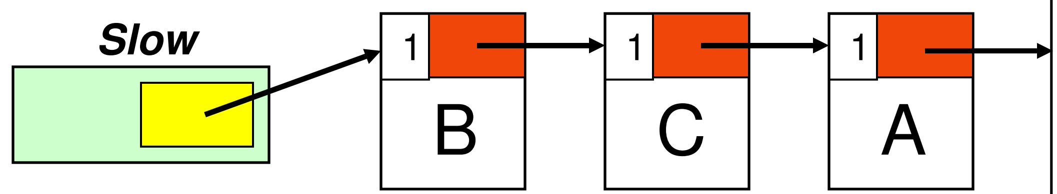
- Property I
- Property II

But it needs to be lock-free!



Problems not fully addressed by previous solutions

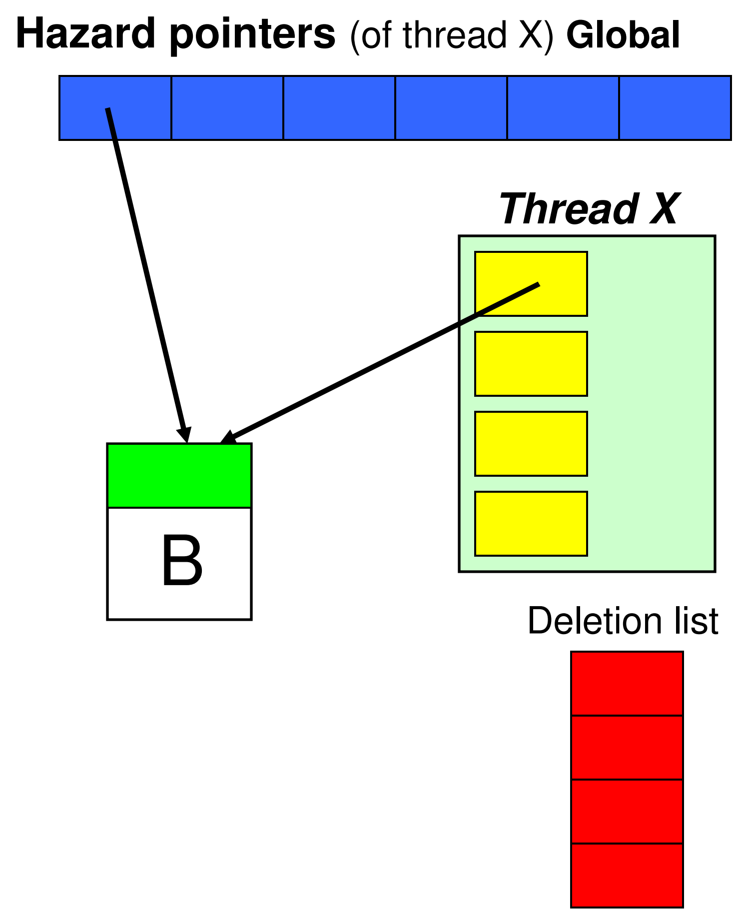
- Reference counting issues
  - A slow thread might prevent reclamation
  - Cyclic garbage



- Implementation practicality issues
  - Reference-count field MUST remain writable forever [Valois, Michael & Scott 1995]
  - Needs double word CAS [Detlefs et al. 2001]
  - Needs double width CAS [Herlihy et al. 2002]
  - Large overhead

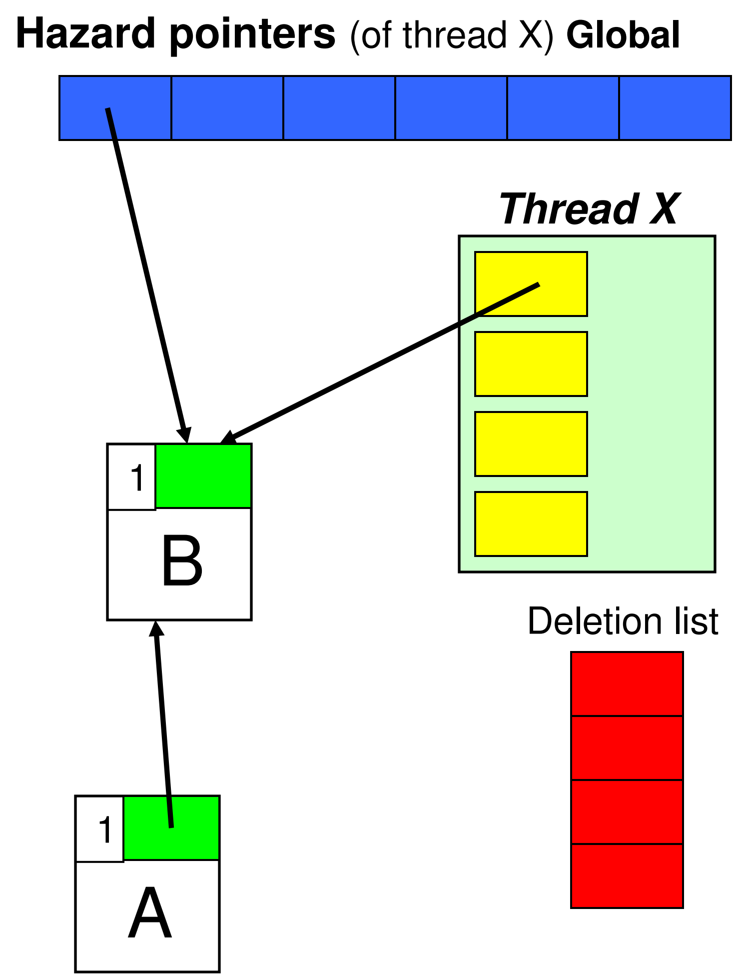
Our approach – The basic idea

- Combine the best of
  - Hazard pointers [Michael 2002]
    - Tracks references from threads
    - Fast de-reference
    - Upper bound on the number of unreclaimed deleted nodes
    - Compatible with standard memory allocators



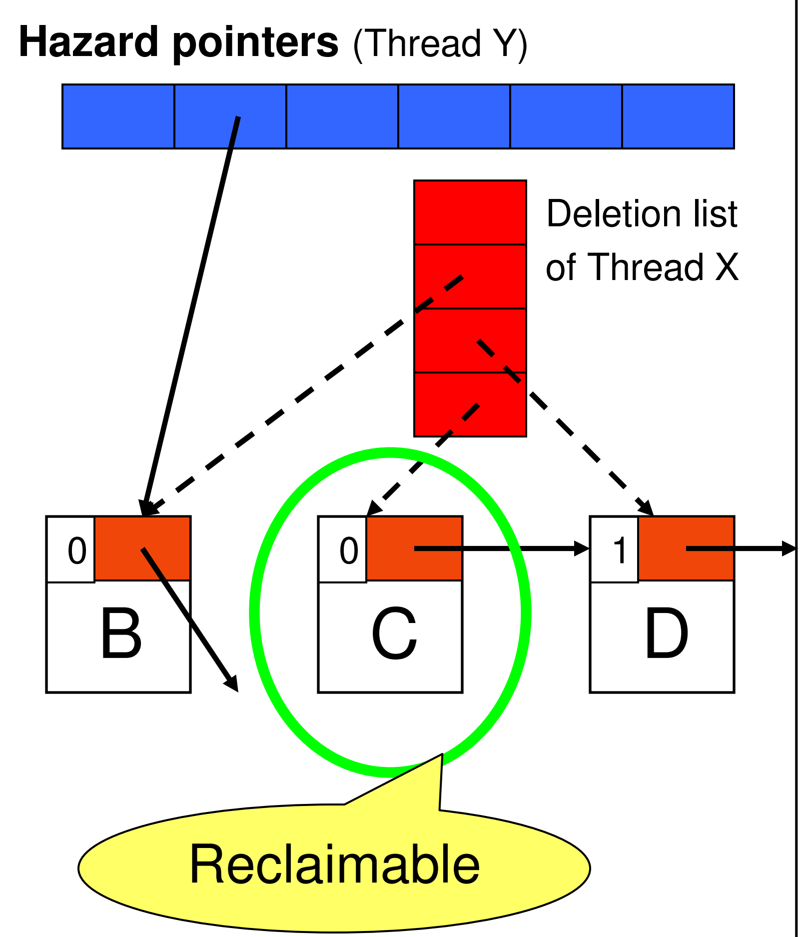
Our approach – The basic idea

- Combine the best of
  - Hazard pointers [Michael 2002]
    - Tracks references from threads
    - Fast de-reference
    - Upper bound on the number of unreclaimed deleted nodes
    - Compatible with standard memory allocators
  - Reference counting
    - Tracks references from links in shared memory
    - Manages links within dynamic nodes
    - Safe to traverse links (also) in deleted nodes



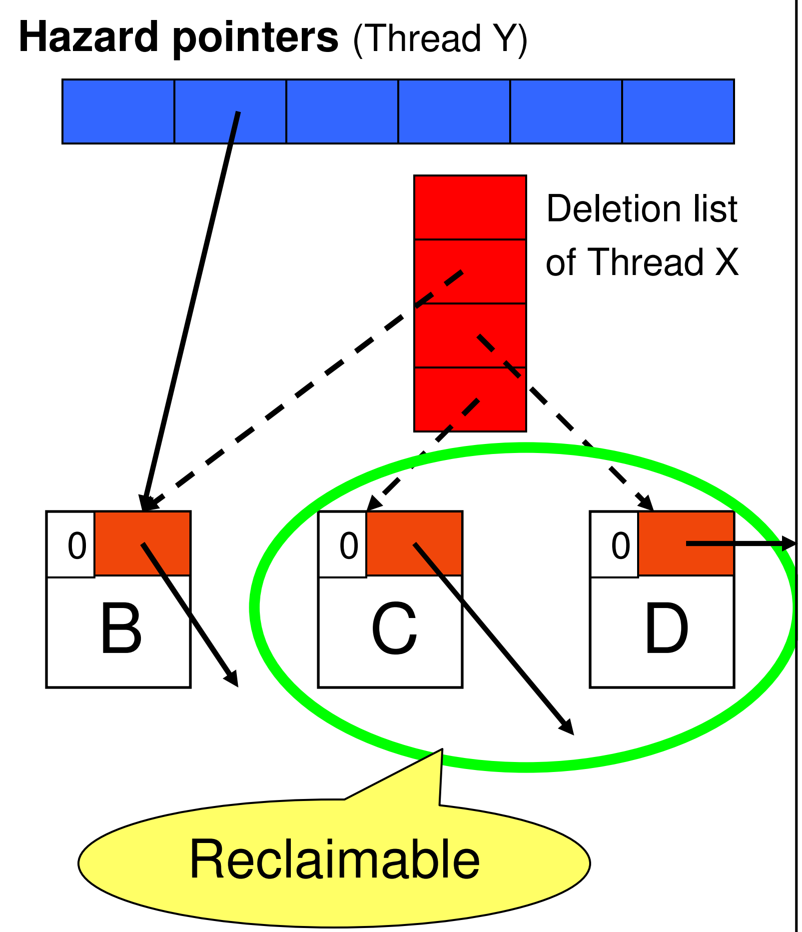
Bound on #unreclaimed nodes

- The total number of unreclaimable deleted nodes is bounded



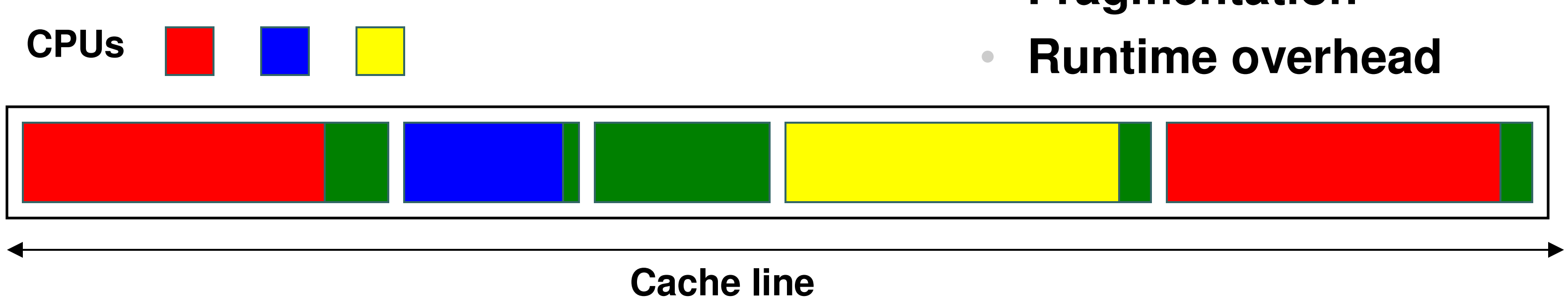
Bound on #unreclaimed nodes

- The total number of unreclaimable deleted nodes is bounded



Concurrent Memory Allocators

- Provide dynamic memory to the application
  - Allocate / Deallocate interface
- Maintains a pool of memory (a.k.a. heap)
- Online problem – requests are handled at once and in order



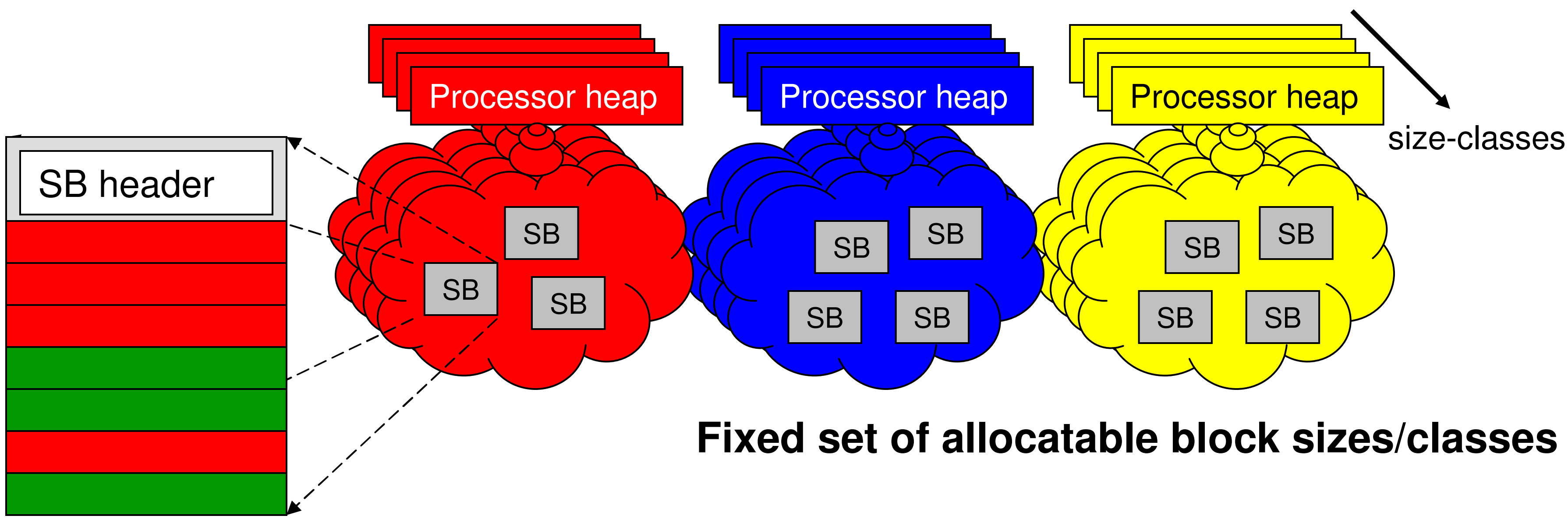
- Performance Goals
  - **Scalability**
  - Avoiding
    - **False-sharing**
      - Threads use data in the same cache-line
    - **Heap blowup**
      - Memory freed on one CPU is not made available to the others
    - **Fragmentation**
    - **Runtime overhead**

NBmalloc architecture

Based on the Hoard architecture [Berger et al, 2000]

Avoiding heap blowup: Superblocks

Avoiding false-sharing: Per-processor heaps



Fixed set of allocatable block sizes/classes

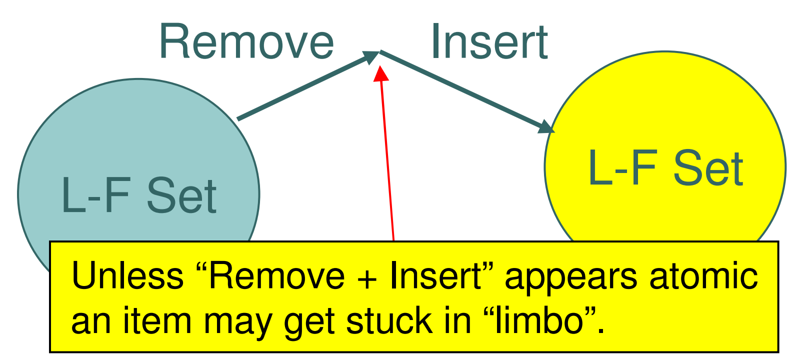
The lock-free challenge

Finding and moving superblocks

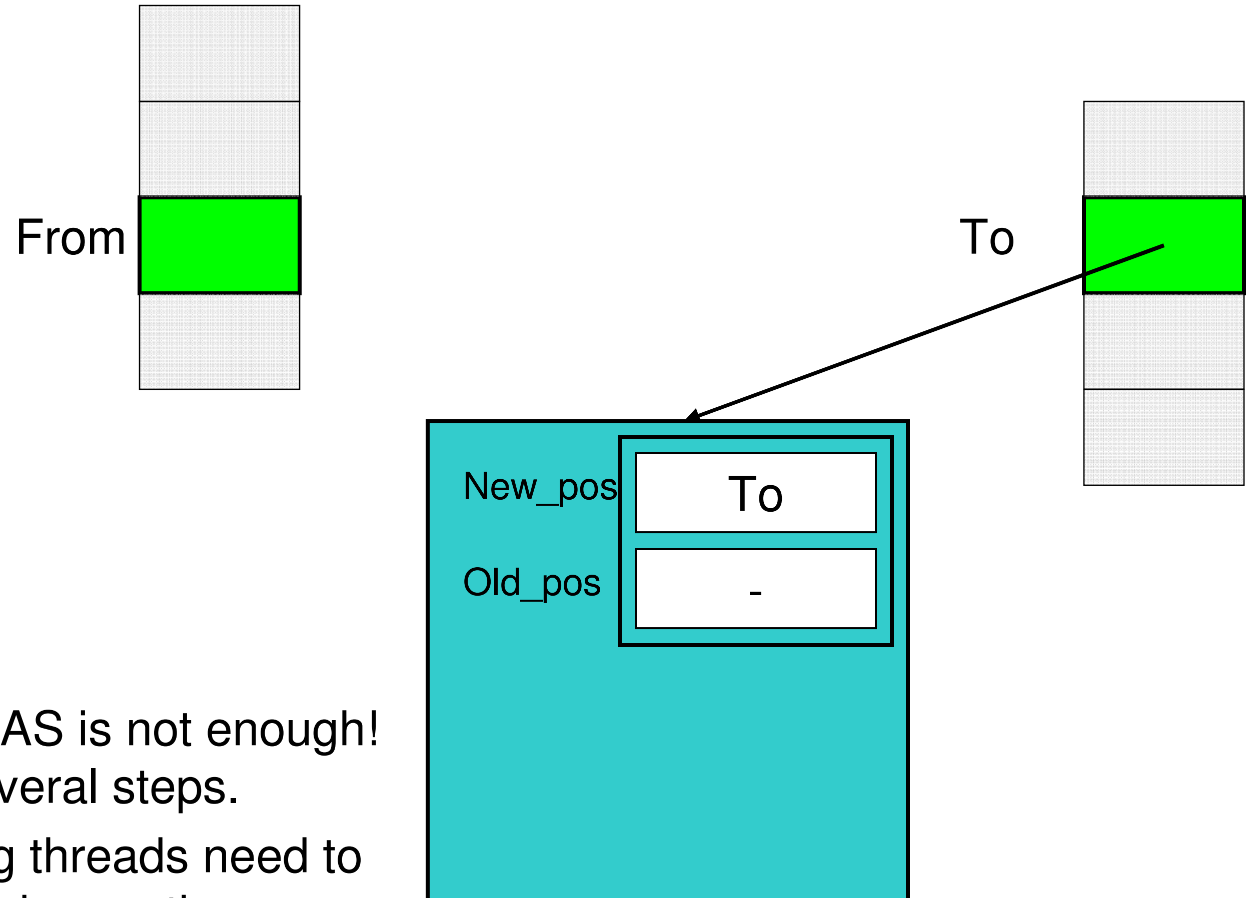
- Within a per-processor heap
- Between per-processor heaps

New lock-free data structure: The flat-set.

- Stores superblocks
- Operations
  - Finding an item in a set
  - Moving an item from one set to another atomically

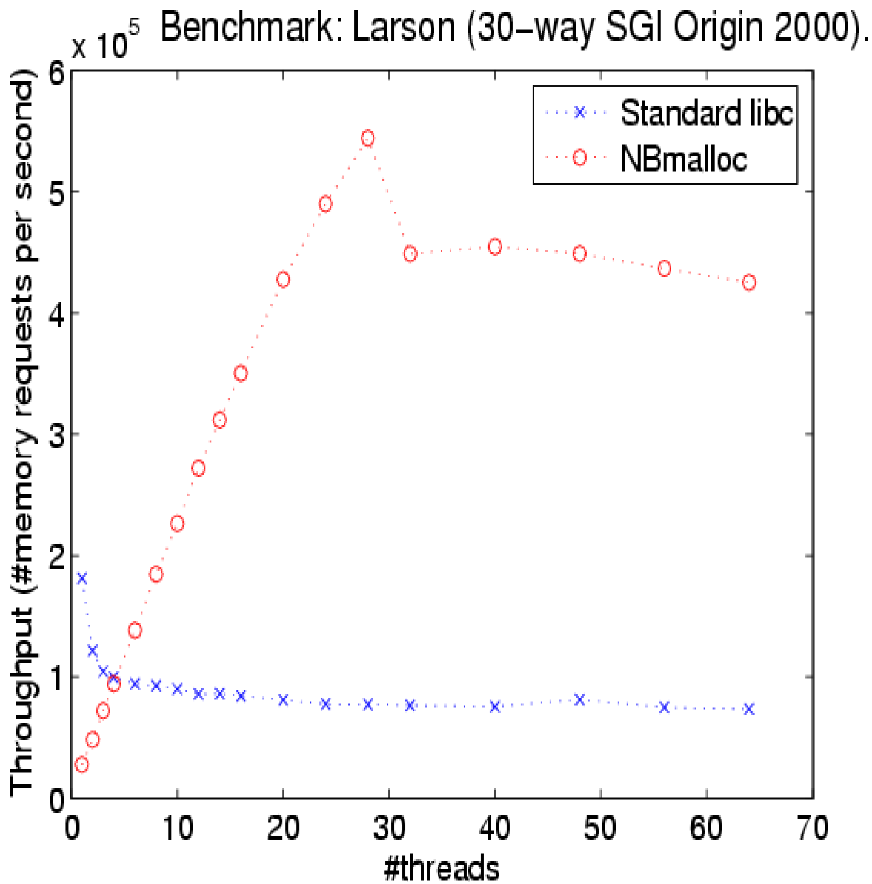


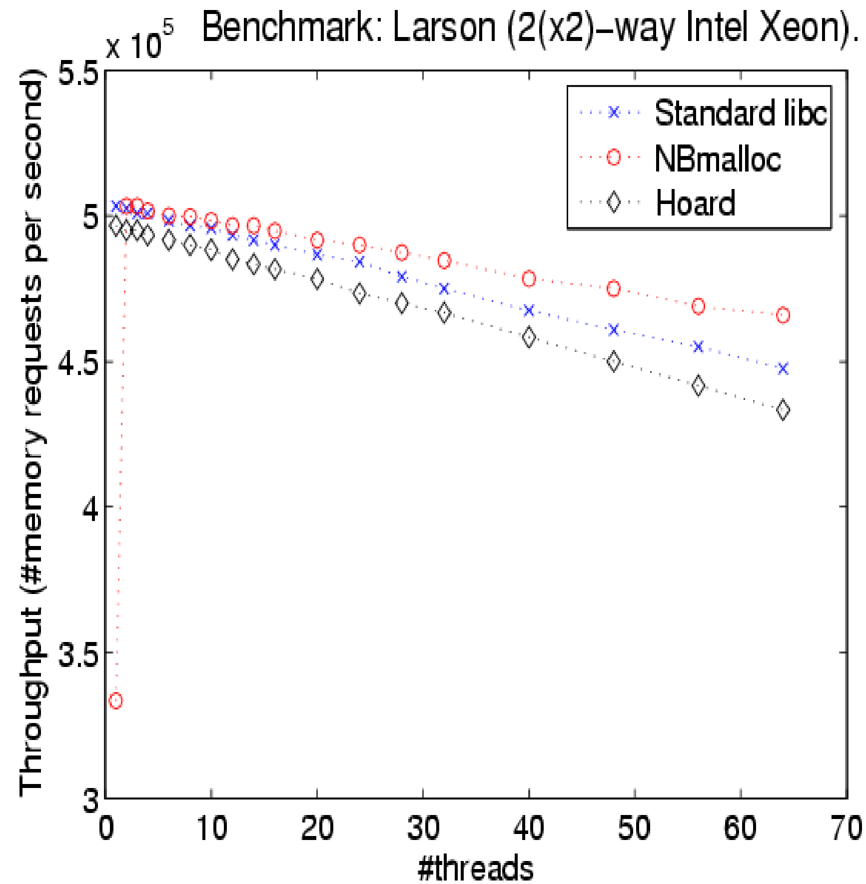
Moving a shared pointer



- Issues
  - One atomic CAS is not enough! We'll need several steps.
  - Any interfering threads need to *help* unfinished operations

Experimental results





Larson benchmark (emulates a multithreaded server application).

Summary memory management

- **1st lock-free memory reclamation algorithm ensuring**
  - Safety of local and global references
  - An upper bound on the number of deleted but unreclaimed nodes
  - Safe arbitrary reuse of reclaimed memory
- **Lock-free memory allocator**
  - Scalable
  - Behaves well on both UMA and NUMA architectures
- **Lock-free flat-sets**
  - New lock-free data structure
  - Allows lock-free inter-object operations
- **Implementation**
  - Freely available (GPL)

Conclusions and future work

- Presented algorithms for consistency in information dissemination services and scalable memory management and synchronization
- Optimistic synchronization is feasible and aids scalability

- Some new research problems opened by this thesis:
  - Use of plausible clocks in cluster consistency
  - Other consistency models in cluster consistency framework
  - Generalized method for lock-free composite objects from “smaller” lock-free objects
  - Making other systems services lock-free