Reducing Aborts in Distributed Transactional Systems through Dependency Detection*

Bo Zhang, Binoy Ravindran, Roberto Palmieri

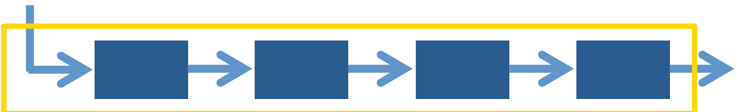
Systems Software Research Group
Virginia Tech

**Appeared as BA in PODC'10*

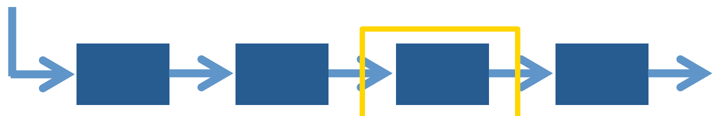
ICDCN 2015

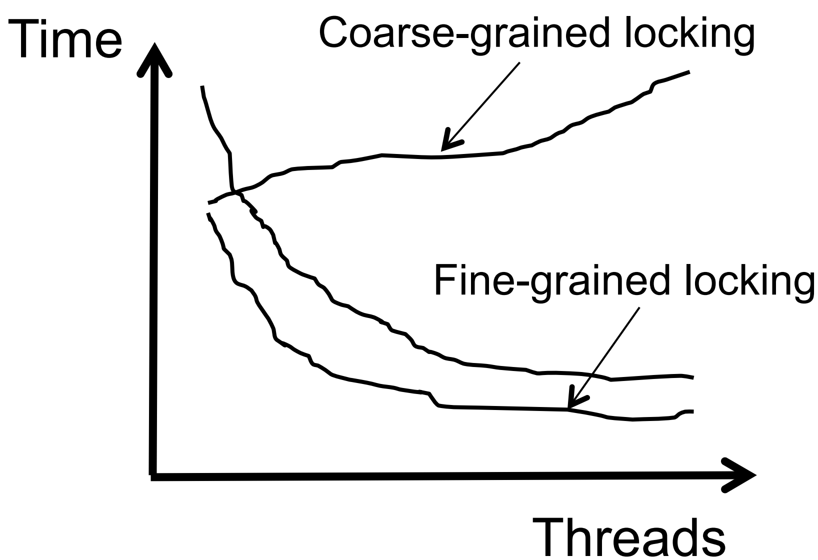
Lock-based concurrency control has serious drawbacks

- Coarse-grained locking
  - Simple
  - But no concurrency
- Fine-grained locking
  - Excellent performance
  - Poor programmability
  - Hard to compose



```
public boolean add(int item) {
  Node pred, curr;
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

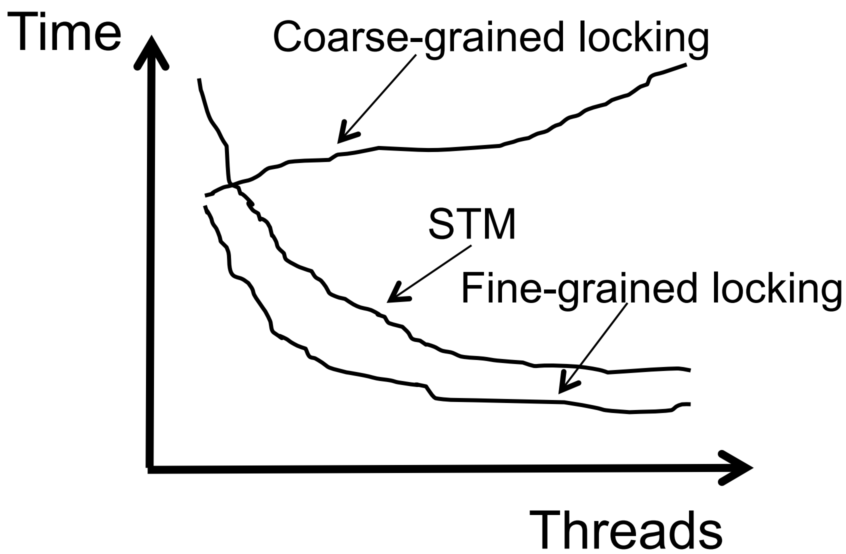





```
public boolean add(int item) {
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.val < item) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

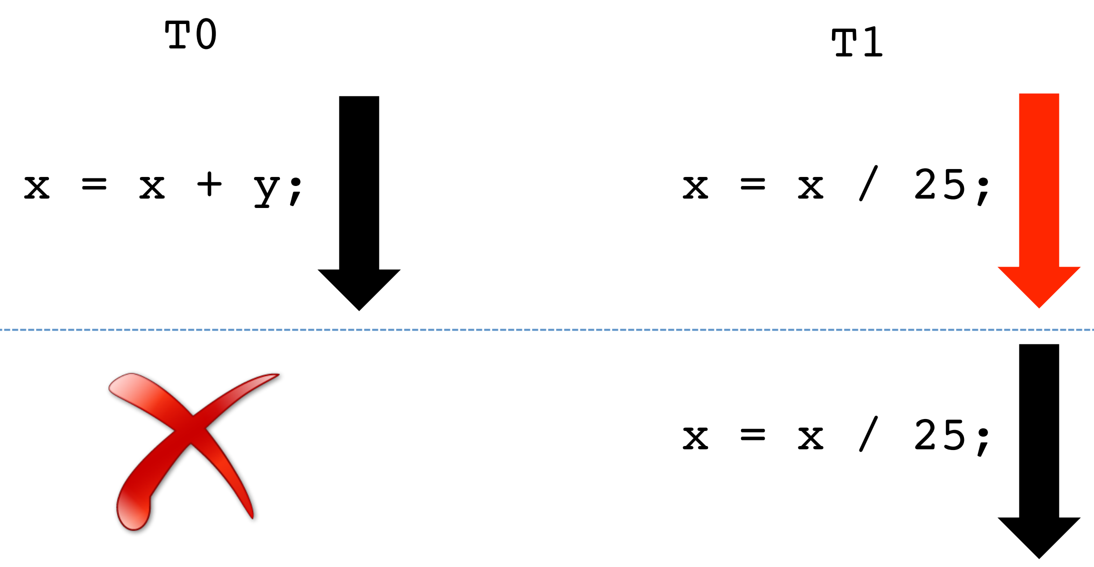
Transactional memory promises to alleviate these difficulties

- Similar to database transactions
- Easier to program
- Composable




```
public boolean add(int item) {
  Node pred, curr;
  atomic {
   pred = head;
   curr = pred.next;
   while (curr.val < item) {
     pred = curr;
     curr = curr.next;
   }
   if (item == curr.val) {
     return false;
   } else {
     Node node = new Node(item);
     node.next = curr;
     pred.next = node;
     return true;
   }
}
```

TM manages contention using a contention manager



- Decides which transaction must abort
- Can cause too many aborts, e.g., when a long running transaction conflicts with shorter transactions
- An aborted transaction may wait too long

Paper's focus is on *distributed* transactional memory

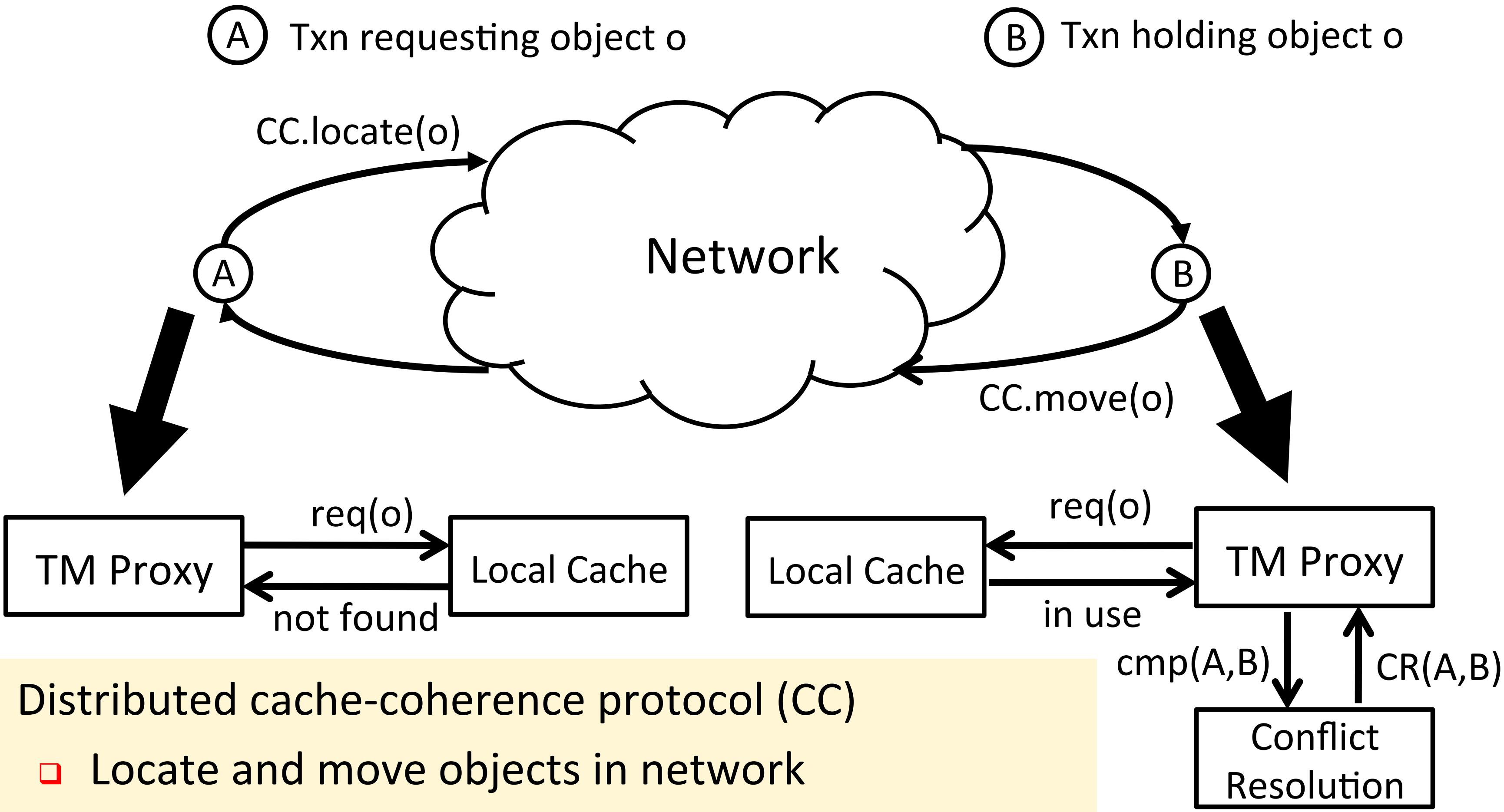
- Nodes interconnected with message passing links
- Similar advantages as that for multicores
  - No manual implementation of distributed synchronization
  - No code translation required (e.g., no SQL)
  - Transactions written in same app programming language
  - Data do not need relational organization
  - Distribution is programmer-transparent

Transaction execution models in DTM can be classified

- Control flow [Waldo and Arnold, ‘00]
  - Transactions migrate; objects do not
  - Synchronization: distributed commit (e.g., 2PC)
  - Inherit traditional database synchronization techniques

- Data flow [Herlihy and Sun, ‘07]
  - Objects migrate (to invoking transactions); transactions do not
  - Synchronization: optimistic
    - Conflicts are resolved by conflict resolution strategy
    - No need for distributed commit
  - Easier to exploit locality

Dataflow DTM mechanics



- Distributed cache-coherence protocol (CC)
  - Locate and move objects in network
  - Ensures consistency among multiple object copies
- Conflict resolution module (CR)
  - Resolve conflicts among transactions

Contention management in data flow DTM can cause too many aborts

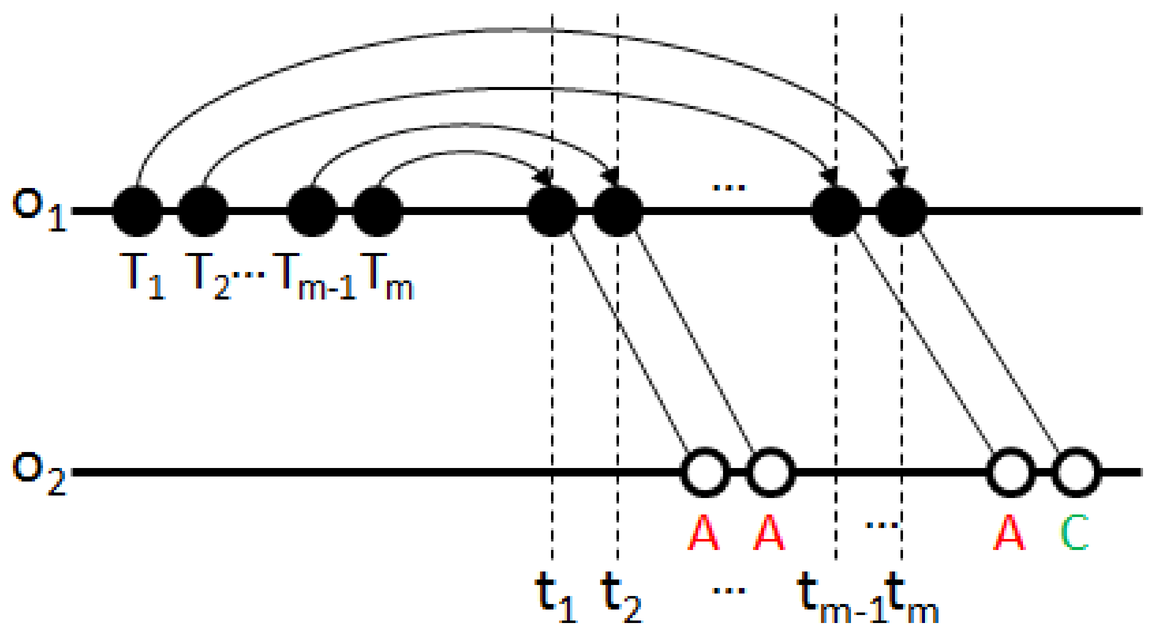
- Uses globally-consistent contention management (GCCM)
  - A running transaction can only be aborted by another transaction (even if still in-flight) with a higher priority
  - E.g., Greedy contention manager (Guerraoui, ‘08)

- Generally too conservative
  - No concurrency among conflicting transactions
  - Only one writable copy available at-a-time per object
- Excessive degree of aborts
  - Even if correctness is <u>not</u> violated
  - “Poor” permissiveness (with respect to opacity)

GCCM example (contrived)

- T1,…, Tm transactions
- Each transaction writes o1 and reads o2
- All transactions concurrently access o1 for writing
- T1 has highest priority, but Tm requests o1 first
- m-1 transactions aborted; only T1 commits

Empty circle = read
Solid circle = write



GCCM is not effective

Objects are scattered in network

Transaction's starting node is unpredictable

Poor performance because objects move repeatedly and abort rate is high

Can we avoid these aborts?

Objective: increase concurrency in data-flow model

- Increase “degree” of permissiveness
  - Accept more schedules than GCCM

- When two transactions conflict over an object, allow to them proceed concurrently
  - Both get an object copy
- If their other operations do not conflict, possible to serialize them in object access order
  - Determine transaction precedence graph and ensure its acyclicity

- Inspired by [Perelman, ’09, Ramadan, ’09] for multiprocessors
  - Cannot copy and paste!
- *Key challenge: how to compute/maintain (acyclic) graph in a decentralized way, without additional communications?*

Paper's contribution: Distributed Dependency-Aware (DDA) model

- Uses multi-versioning
  - Each node stores a version data structure for each object
  - Objects have pending list and committed list
- Read-only transactions always commit by reading latest committed versions
- Write-only transactions always commit by serializing themselves before (or after) conflicting read-write transactions

DDA computes precedence graph without a centralized coordinator

- Objects store important events (read, write)
  - Implicitly through pending list, committed list, transaction IDs, timestamps, etc.

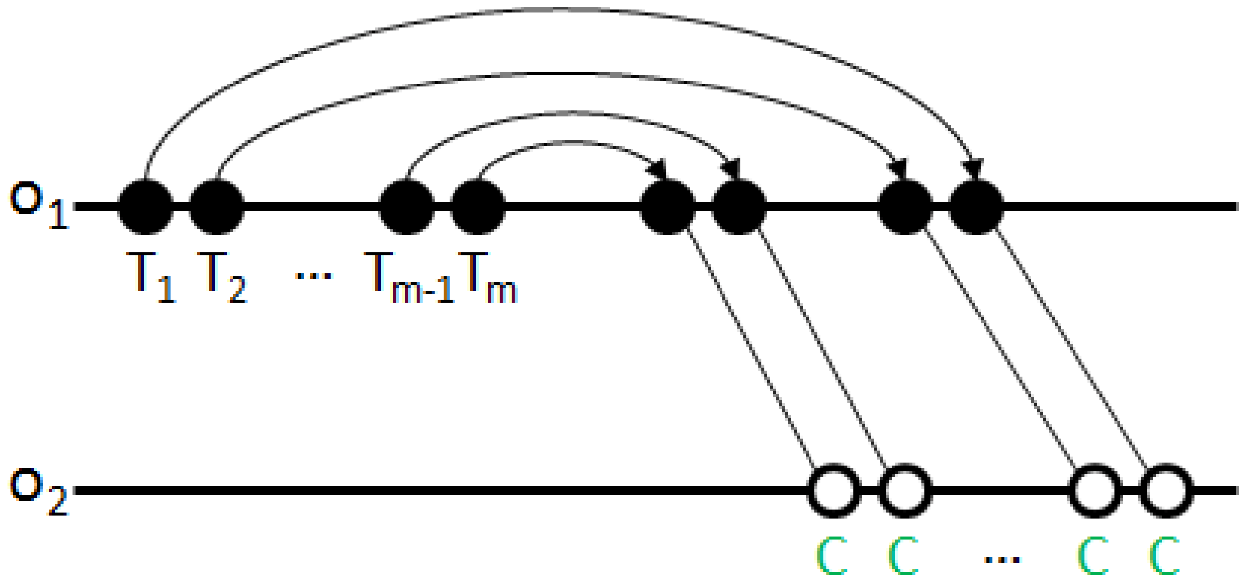
- When a transaction fetches an object, stored events are retrieved to determine real-time order and conflicts:
  - If operation violates correctness, aborted

    (Otherwise, will introduce cycle in precedence graph)
  - If safe to execute, proceeds

- Graph kept acyclic without additional communication steps

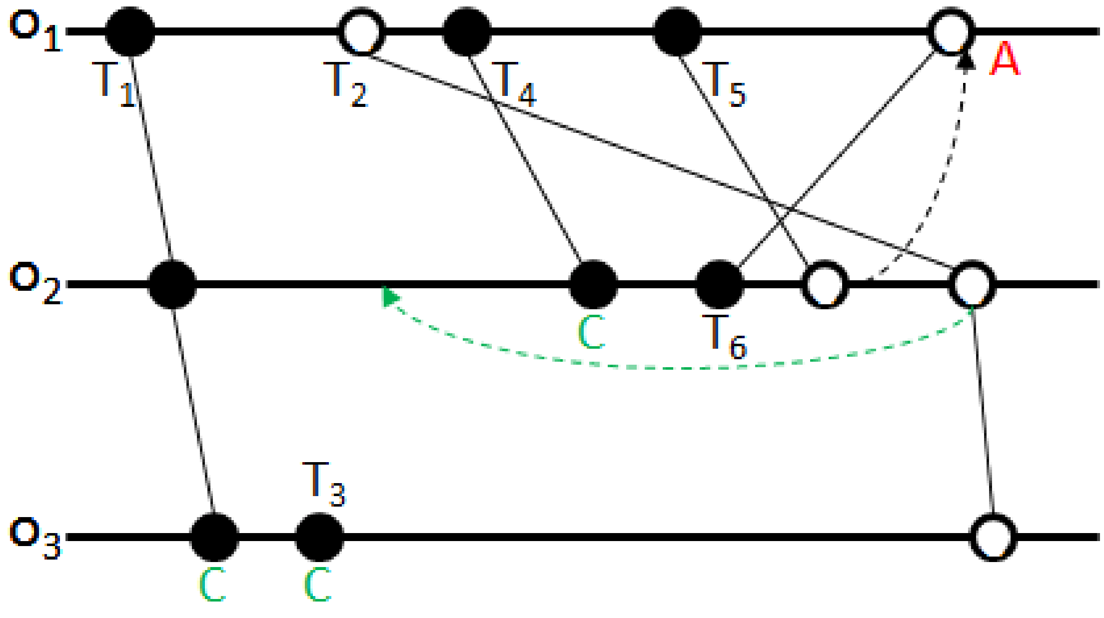
DDA example

- All write operations on o1 are conflicting with each other, but they can be serialized in any order
- Read operation on o2 are not conflicting
- Final serialization order is the access order on o1



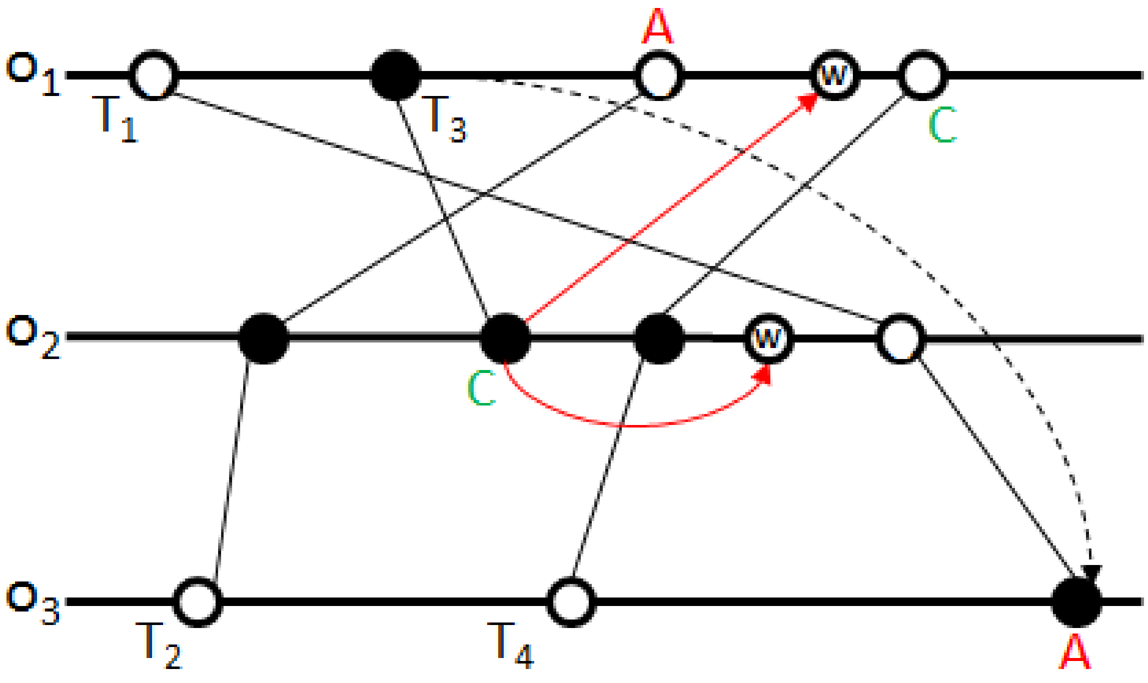
Example 2: the case of irreconcilable histories

- T5 = {write o1; read o2}
- T6 = {write o2; read o1}
- T5 and T6 cannot execute concurrently, so T6 is aborted
- T2 is read-only and always commits by reading previous versions



Example 3: write-only transactions never abort

- T3 aborts T1 and T2
  - T1 because T3 is write-only and cannot abort
  - T2 because T2 wants to read o1, and T2 is serialized after T3
- T4 can commit because its read operations do not conflict



DDA has desirable properties

- Precedence graph is always acyclic
- Opacity
- Strong MV-permissiveness
  - Read-only and write-only transactions never abort
  - Read-only transactions never cause other transactions' abort
- Invisible reads
- Real-time prefix garbage collection
- Proofs in paper

Conclusions

- Dataflow DTM model can exploit locality

- GCCM is easy to implement, but has high aborts
- Can use a coordinator to compute and maintain acyclic precedence graph, but high communication cost

- DDA is somewhere in between:
  - Stores events in migrating objects to compute precedences
  - Allows maximum concurrency for some
  - Contention management for others to ensure acyclicity