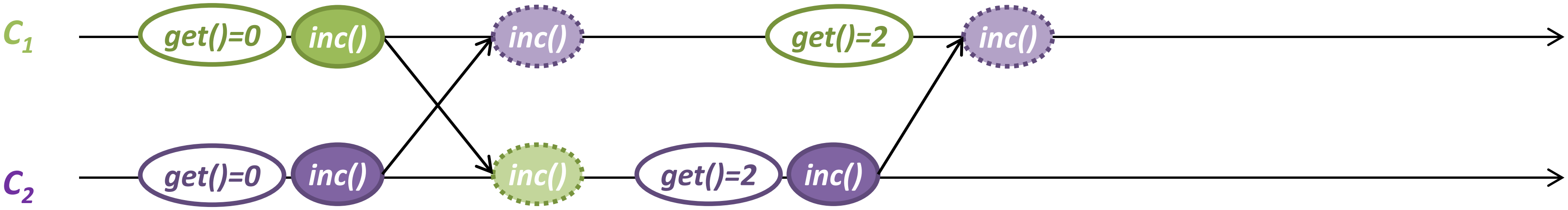
ConcoRDanT+STREAMS M24 meeting, Nantes, November 2012

CRDT transactions in a scalable way* with SwiftCloud

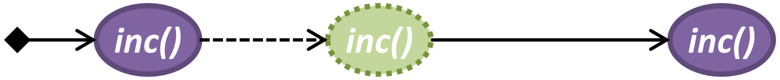
Marek Zawirski	UPMC-LIP6 & INRIA
Annette Bieniusa	U. Kaiserslautern
Valter Balegas	UNL
Nuno Preguiça	UNL
Sérgio Duarte	UNL
Marc Shapiro	INRIA & UPMC-LIP6
Carlos Baquero	U. Minho

Object model and transactional guarantees

Example: a replicated counter object (CRDT)



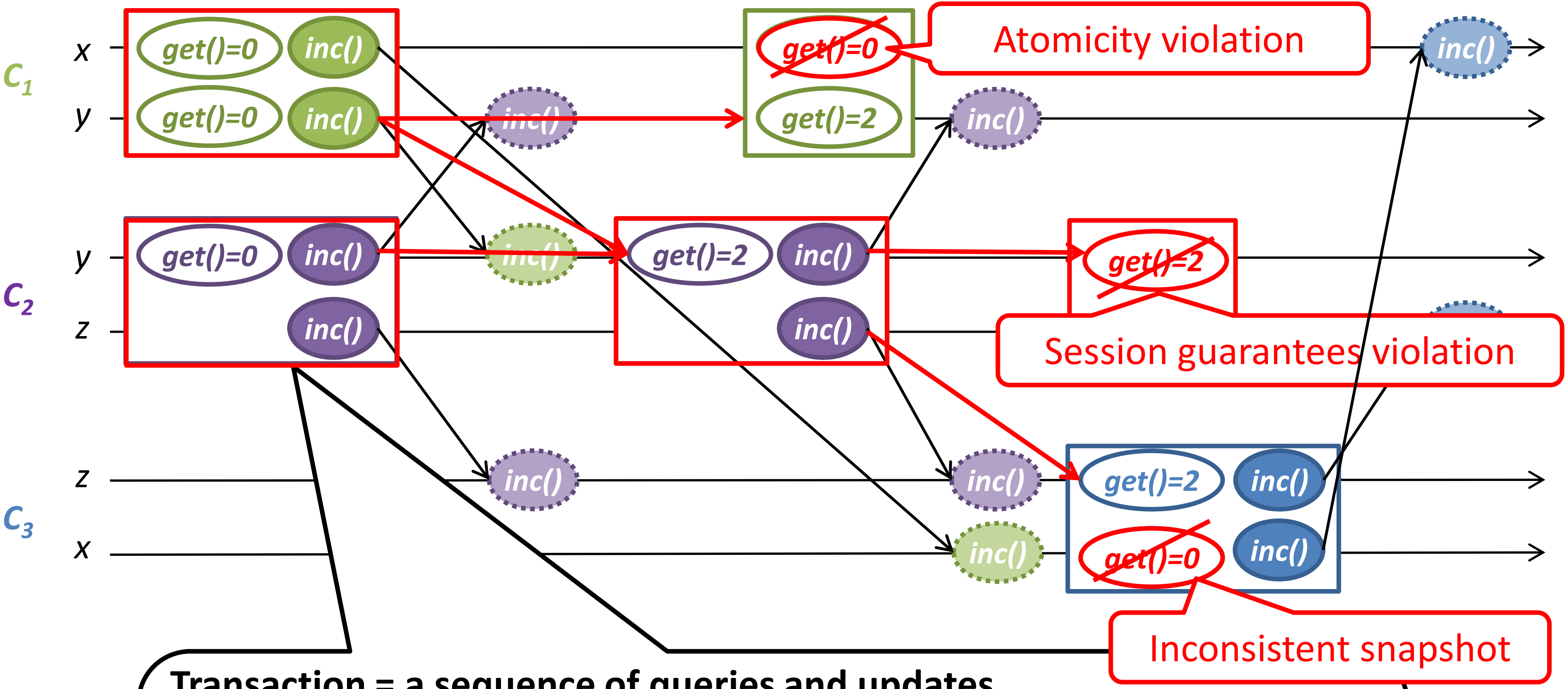
A simplified model of object replica



- Log of updates, a linear extension of ”happened-before” order
- Object version = initial state + application of a valid subsequence
- Suboptimal, later extended with pruning

Object model and transactional guarantees

Example: a database of replicated counters



Atomicity violation

Session guarantees violation

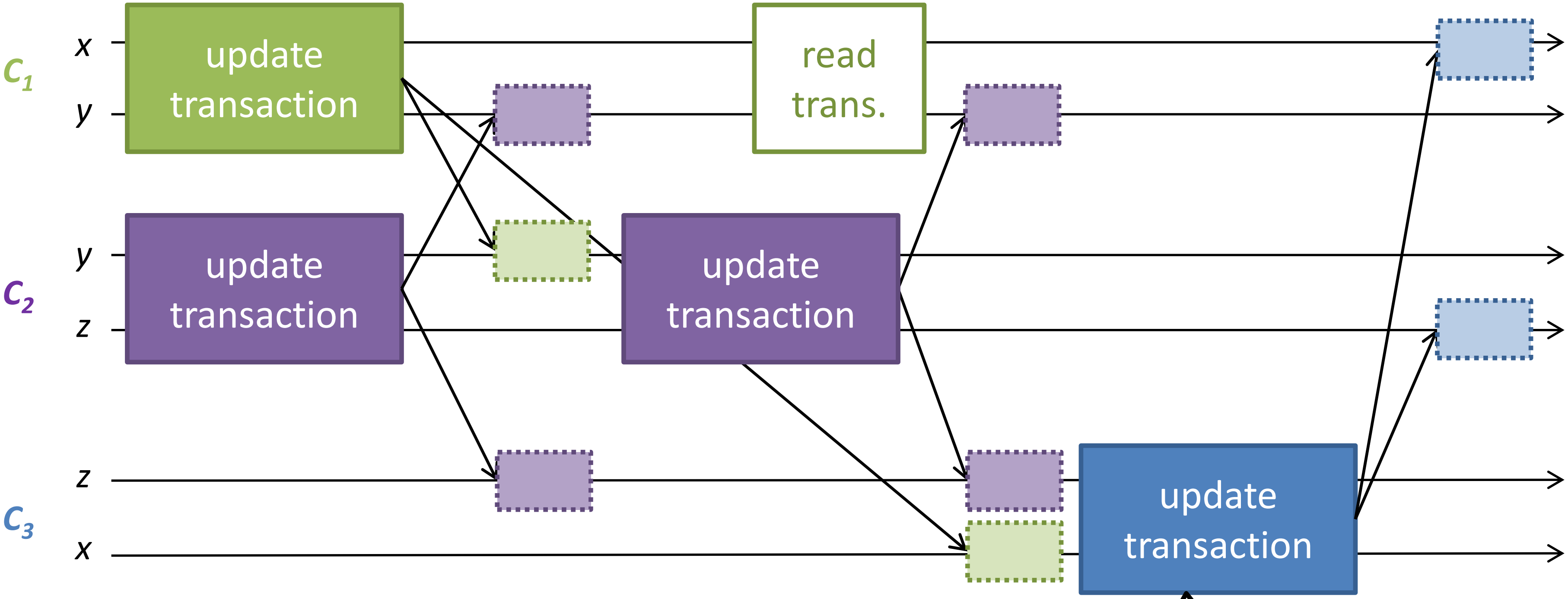
Inconsistent snapshot

Transaction = a sequence of queries and updates

- *Atomicity* => all-or-none updates visible
- *Consients snapshot for reads* => consistent cut based on causality
  - Causality extended across objects through each transaction
- *Session guarantees* => growing snapshot including prior updates
- *No aborts* => asynchronous implementation possible

Object model and transactional guarantees

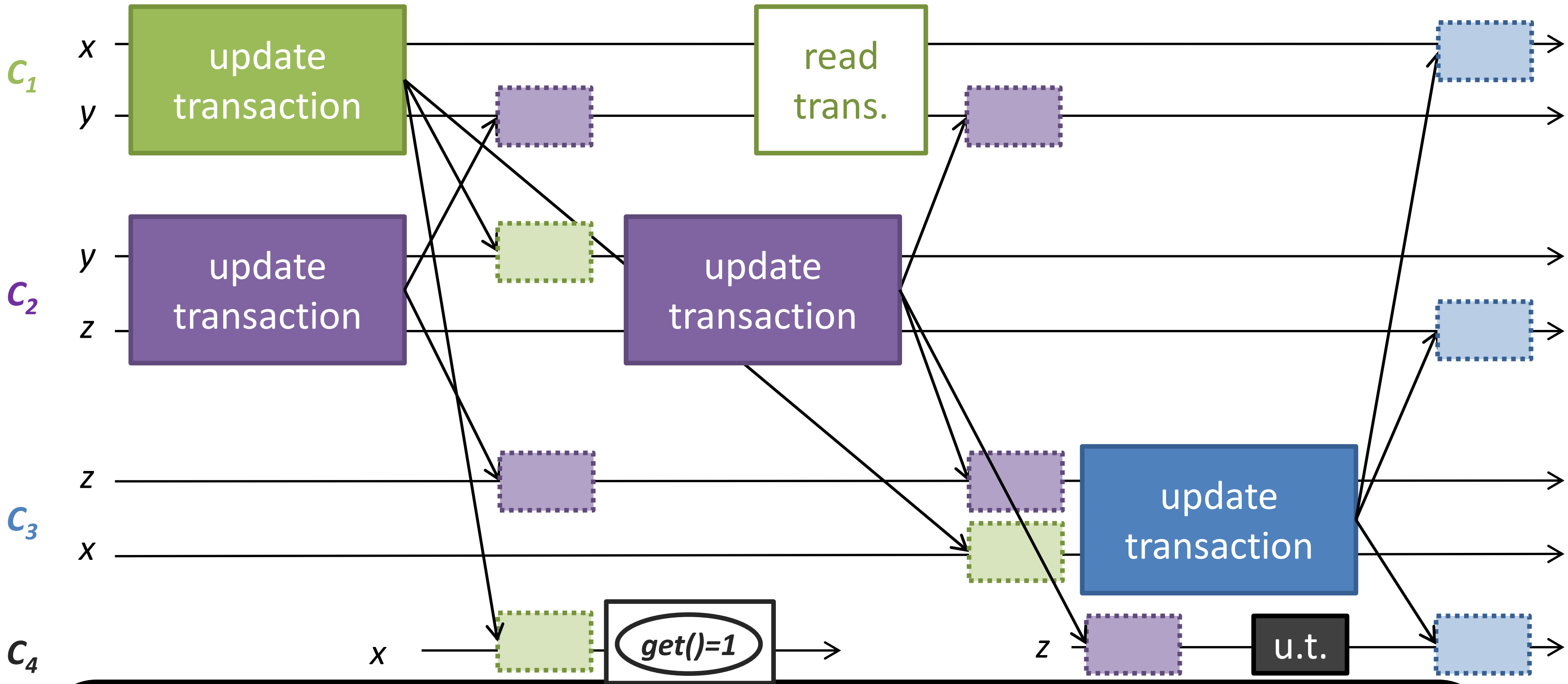
Example: a database of replicated objects



Simplified update transaction record

- Read set = write set

What makes the problem hard? Workaround!

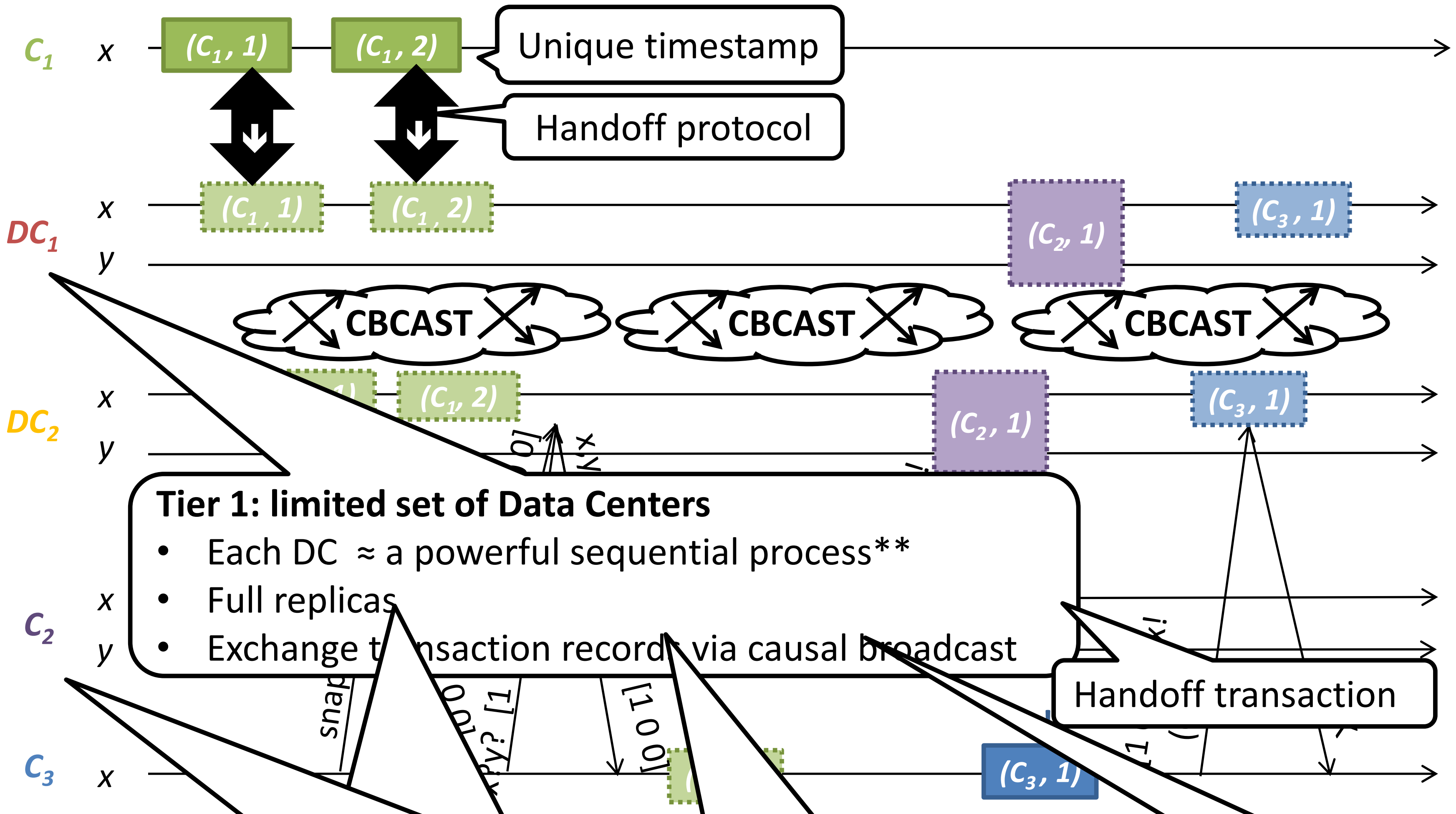


Versioning and tracking causality is difficult under our assumptions:

Partial replication	=> direct dependency (causality) tracking hard
Dynamic replica set	=> locating objects and updates dissemination hard
High client churn	=> very big vector clocks

Workaround: use Data Centers as full reference replicas processing transactions!

The 2-tier architecture with (naïve) handoff



Unique timestamp

Handoff protocol

Tier 1: limited set of Data Centers

- Each DC ≈ a powerful sequential process**
- Full replicas
- Exchange t nsaction record via causal broadcast

Handoff transaction

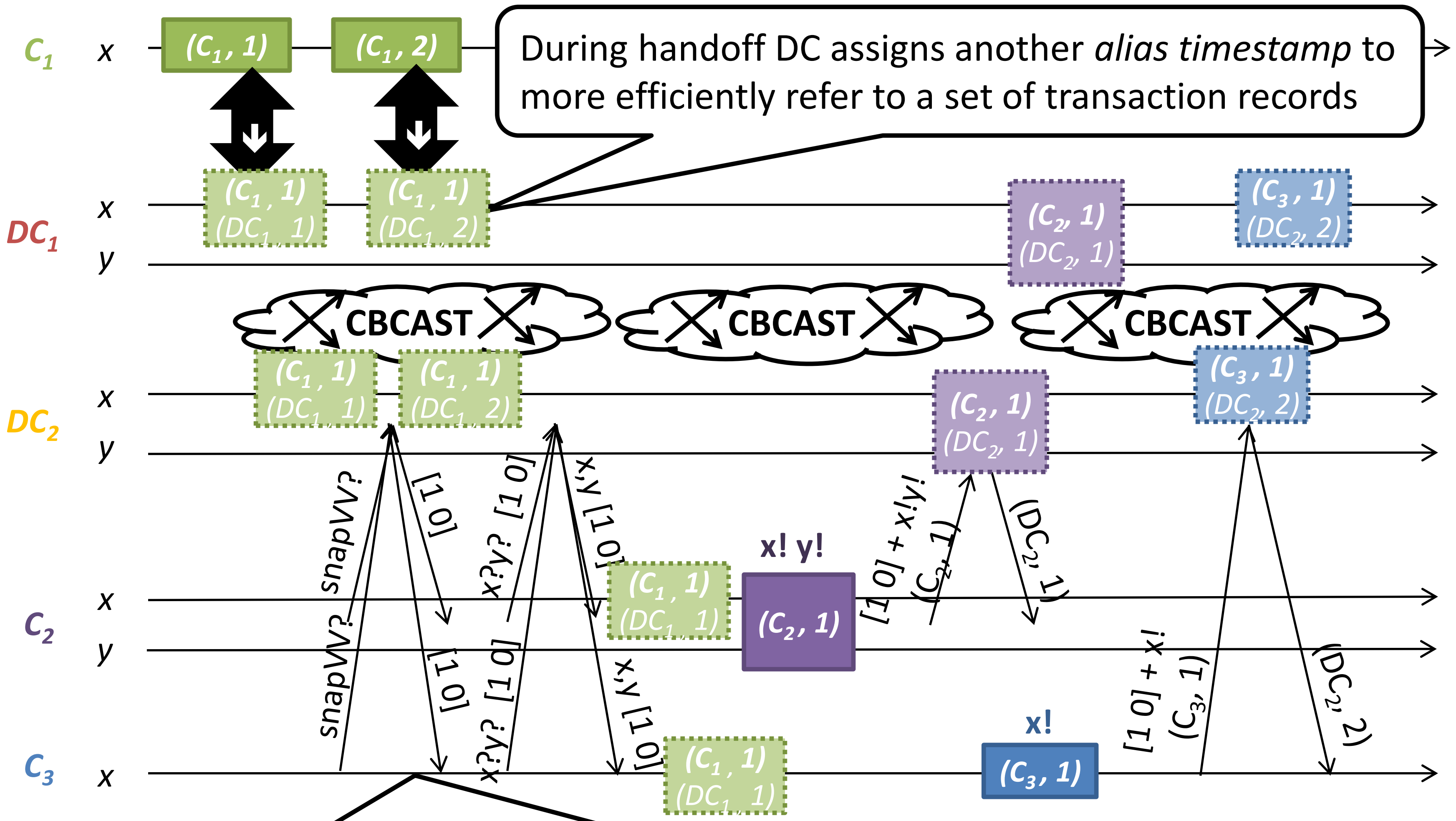
New tra
- Defi
- Effici

ction
ates)

✓ Dissemination and snapshot problems solved: DC-CBCAST is affordable! (records delivery order is *stronger* than causality)

— Version Vectors (e.g. snapVV) unacceptably large: O(|Clients|)

Handoff with DC-assigned alias timestamp

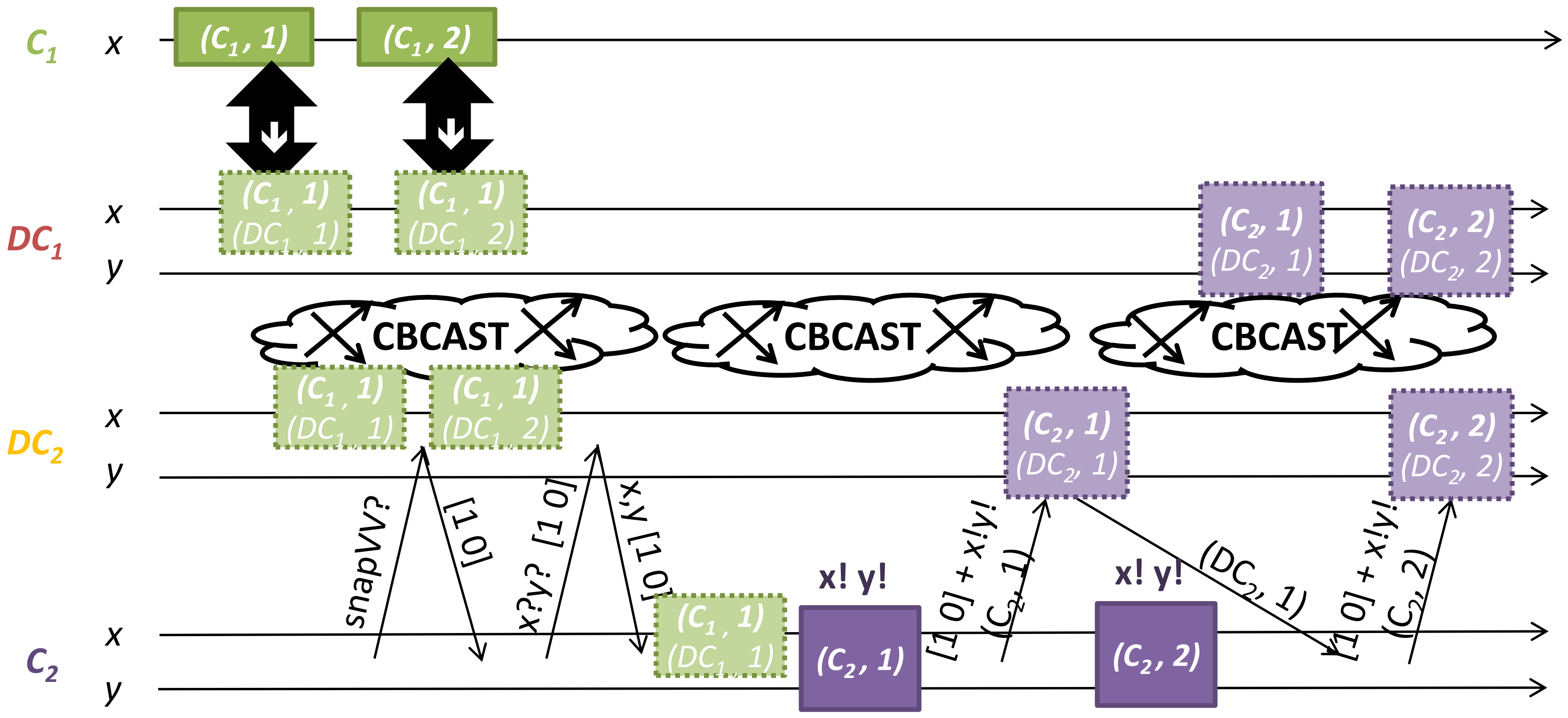


During handoff DC assigns another *alias timestamp* to more efficiently refer to a set of transaction records

✓ DC-assigned aliases allow us to use managable vectors: O(|DCs|)

— But… is handoff asynchronous? Is it fault-tolerant and wait-free?

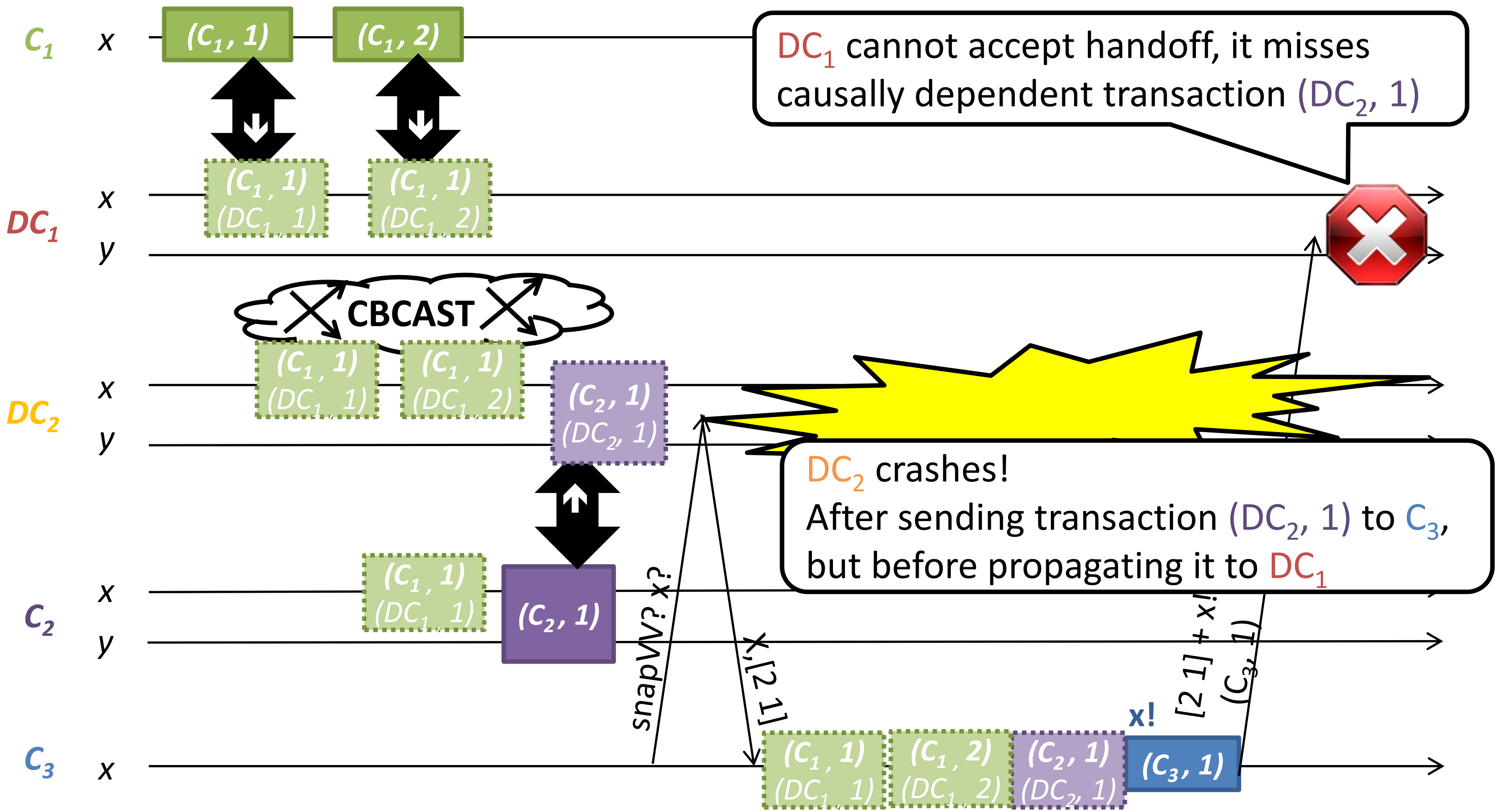
Problem 1: making handoff asynchronous



C_2 awaits handoff ack+alias, does it prevent executing transaction concurrently?

✓ No! Include previous local transactions in snapshot by referring to their client timestamp rather than alias (session guarantees), e.g. snapVV(C_2, 2) = [1 0 C_2=1]

Problem 2: reading partially disseminated trans.



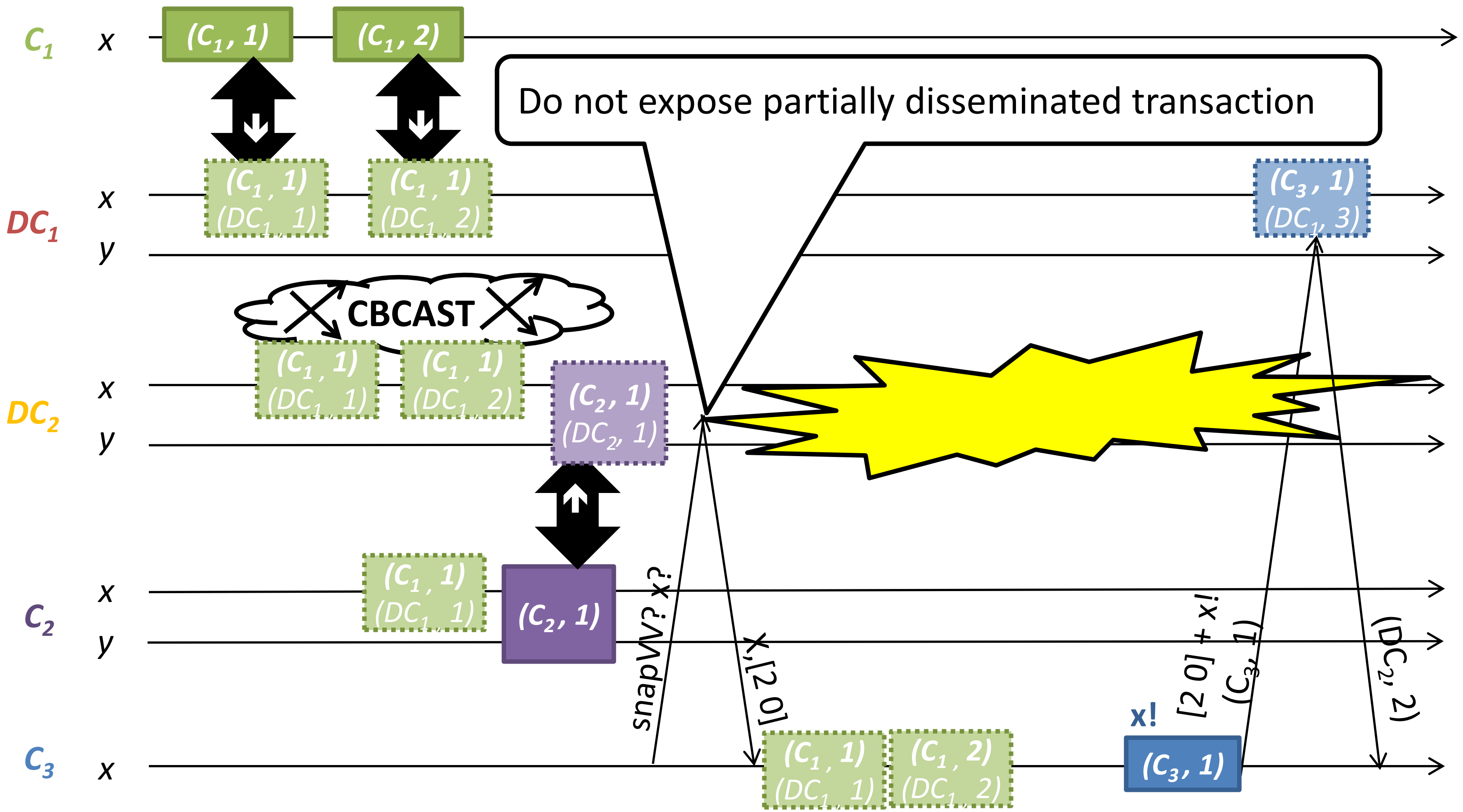
DC_1 cannot accept handoff, it misses causally dependent transaction $(DC_2, 1)$

DC_2 crashes!
After sending transaction $(DC_2, 1)$ to C_3,
but before propagating it to DC_1

Liveness issue: updates of client C_3 are invisible to other clients until DC_2 recovers

- Side-effect of depending on *partially disseminated* transaction
- Client C_3 cannot recover missing transaction, it does not replicate y!

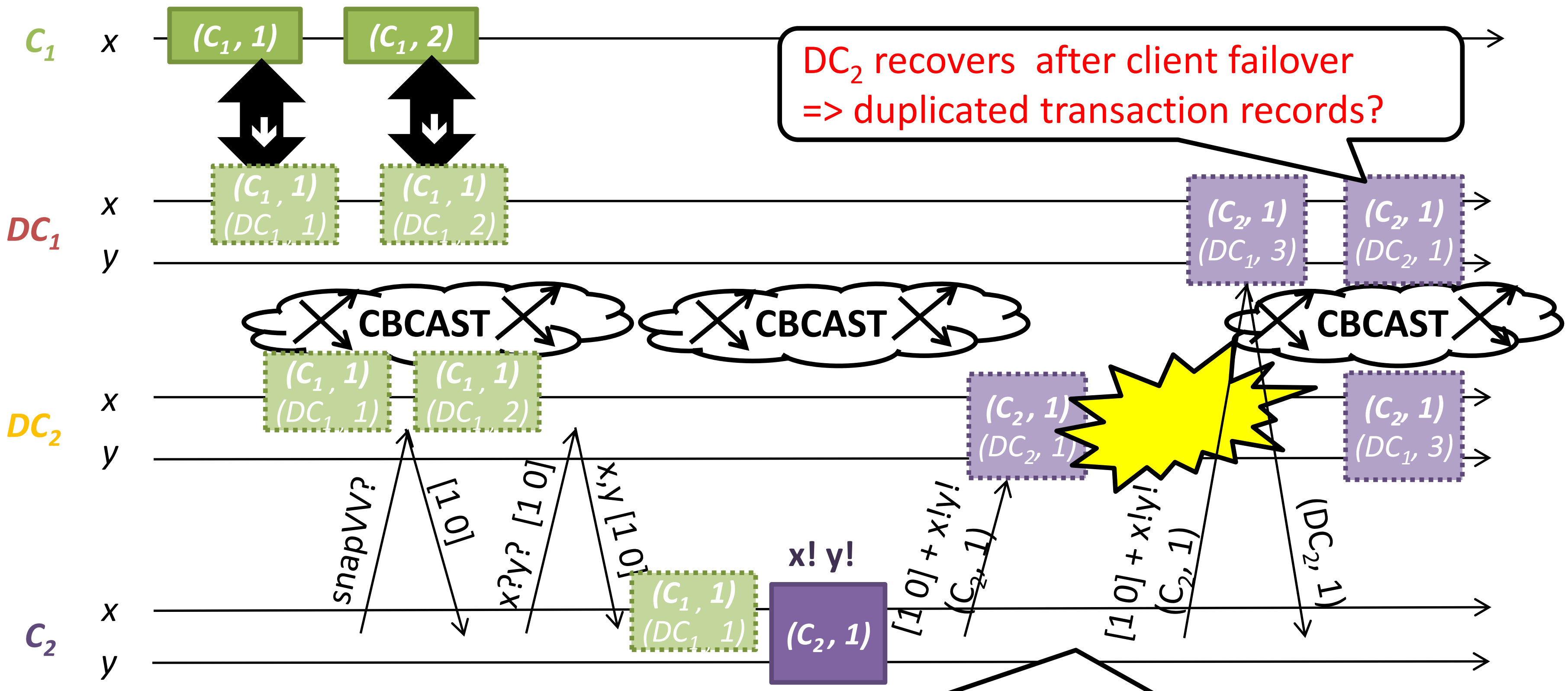
Solution: DC offers only stable transactions



Do not expose partially disseminated transaction

- ✓ Keep track of *stable transactions*, e.g. disseminated to majority of DCs
- ✓ Offer only stable transactions to the client (modulo his own transactions)
- — Delays visibility of recent transactions

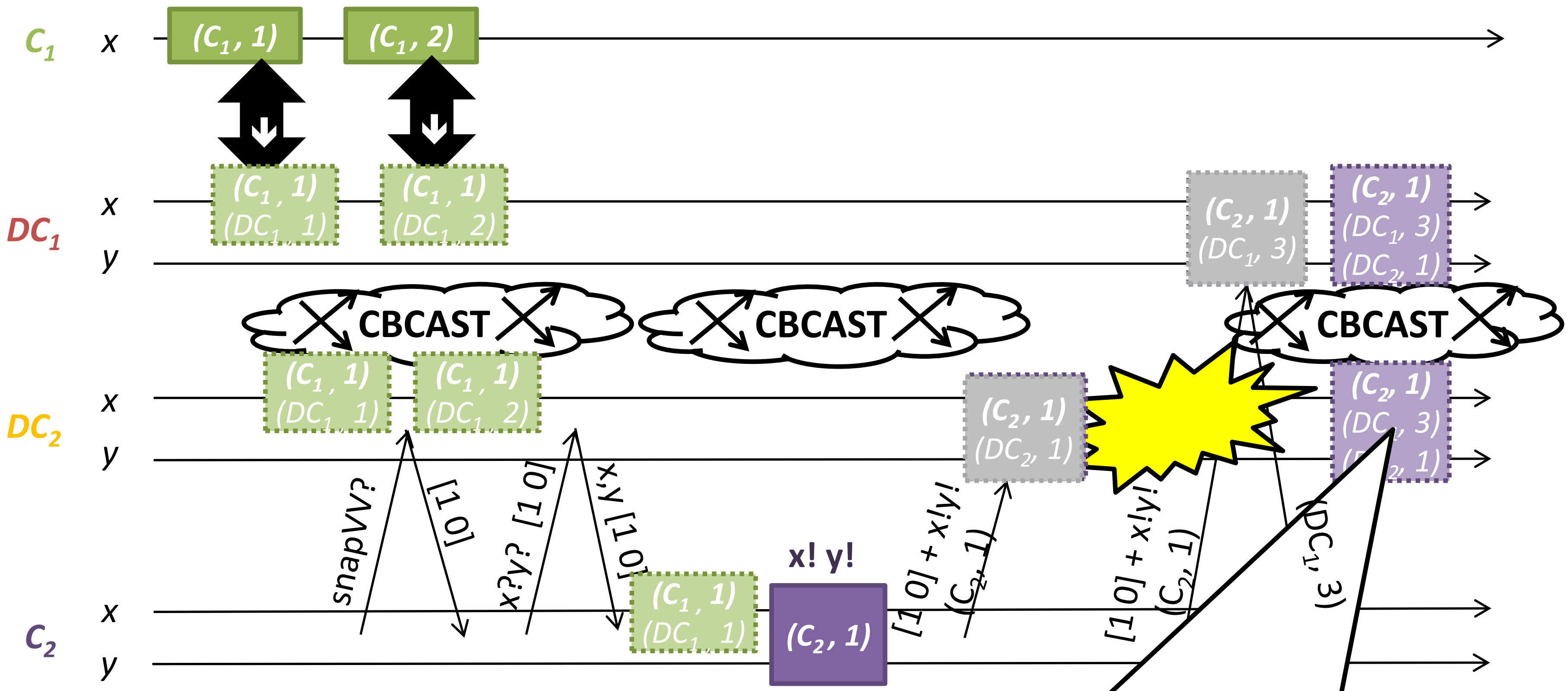
Problem 3: retrying handoff request



DC_2 recovers after client failover
=> duplicated transaction records?

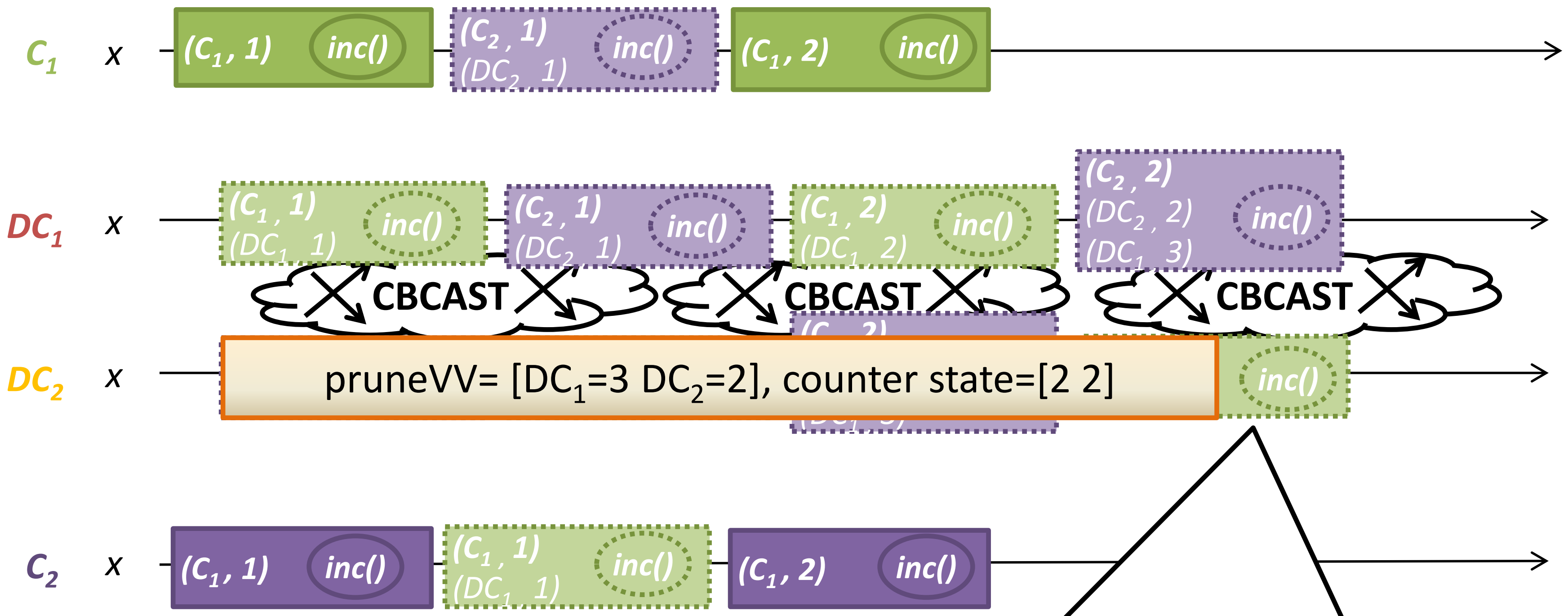
No reply for C_2 handoff request, status and alias unknown! DC_2 crash?
Can C_2 retry handoff with DC_1 or DC_2 or should it block?

Solution: make handoff idempotent



- ✓ Merge transaction records with identical client timestamp (allow multiple aliases)
- ✓ This makes handoff idempotent, allows retries or failover
- — Handoff idempotence more difficult with object pruning

Problem 4: pruning object updates log safely

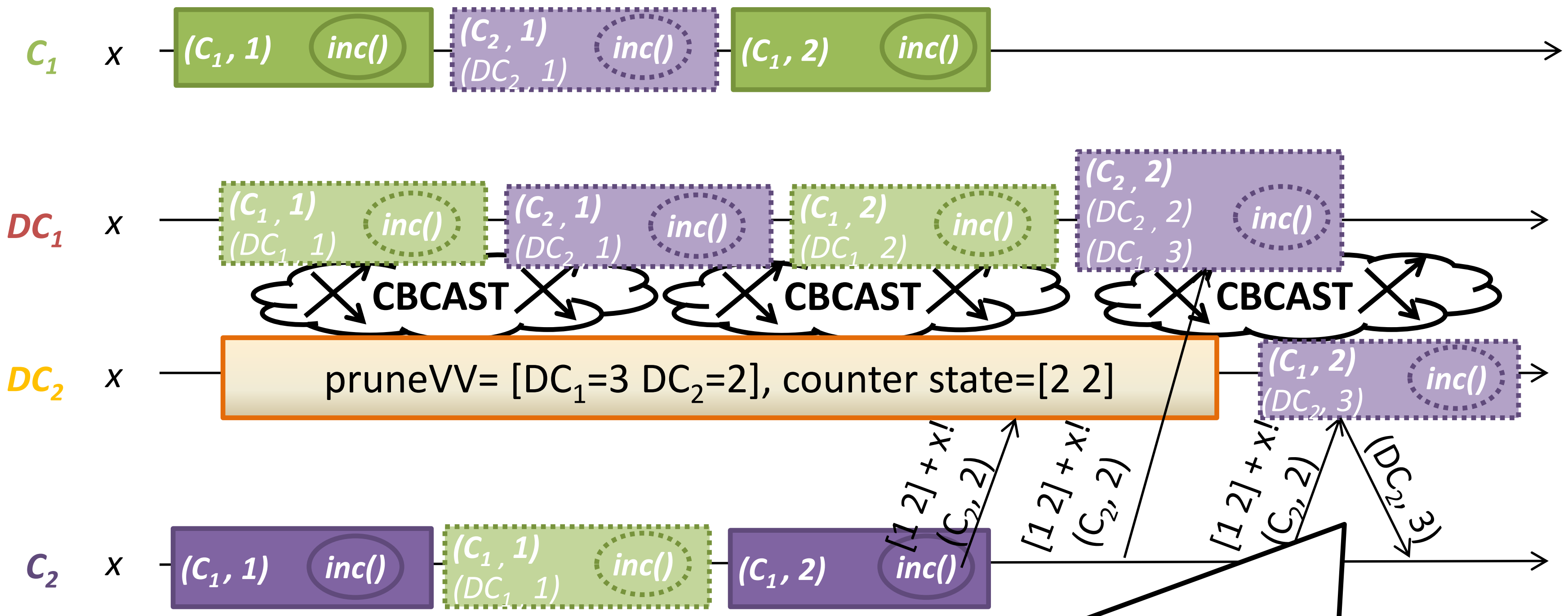


– Log-based object representation very costly: O(|updates|)

Replace old log of operation with state-based CRDT and pruneVV

- For more efficient representation utilize timestamp aliases*
- Update representation must be uniform across DCs
  => pruning when set of aliases is stable (if many, select one)
- Current implementation: every DC prunes independently
  => client may need to merge states from different DCs

Pruning and handoff idempotence



— Retrying handoff of pruned transaction => violated idempotence!

✓ Maintain single clientVV per DC – “idempotence guard”

✓ Never transmitted, only single entry

Lessons learnt

- Implementing CRDT transactions ≈ implementing a huge “database“ semi-lattice
  - Difference w.r.t. ordinary object: (dynamic) fragmentation*
  - Use different techniques inside and across objects

- Causality tracking is difficult at scale, both inside/across CRDTs
  - Limit communication topology, here: 2-tier architecture
  - Use handoff protocol with timestamp aliasing and

- Making handoff live and correct despite Tier 1 failures
  - Reading stable versions helps failover
  - Timestamp aliasing helps too
  - When forced to store a big VV, share it across DB