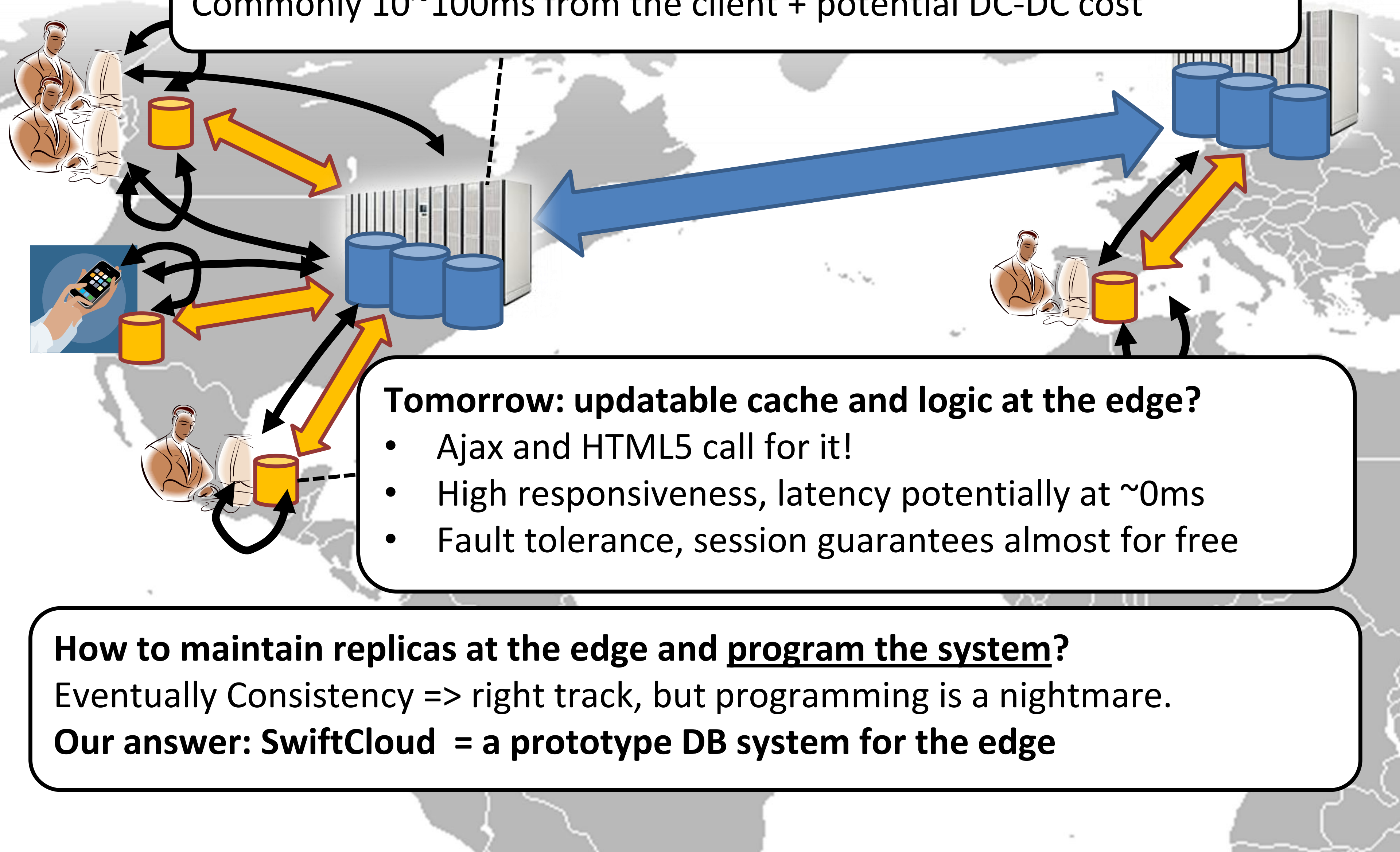
SwiftCloud
It's Time to Move the Data to the Edge

Annette Bieniusa Nuno Preguiça Carlos Baquero
Marc Shapiro Sérgio Duarte
Marek Zawirski Valter Balegas

INRIA / LIP6 CITI, UNL HASLab,
UM Paris Lisbon Minho

Can we extend geo-replicated DBs to the edge?



DBs today: within DC boundaries
Commonly 10~100ms from the client + potential DC-DC cost

Tomorrow: updatable cache and logic at the edge?

- Ajax and HTML5 call for it!
- High responsiveness, latency potentially at ~0ms
- Fault tolerance, session guarantees almost for free

How to maintain replicas at the edge and <u>program the system</u>?
Eventually Consistency => right track, but programming is a nightmare.
Our answer: SwiftCloud = a prototype DB system for the edge

Outline

Motivation

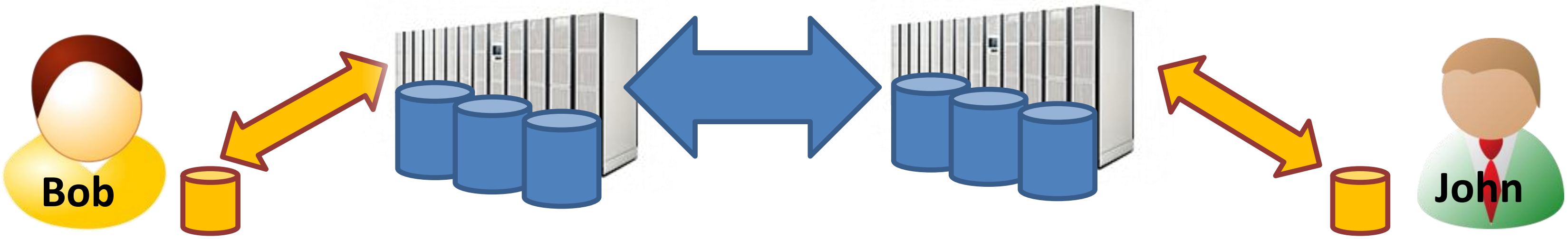
Programming model

Architecture

Evaluation

Challenges of programming EC system at the edge

Problem: programming a key-value store is notoriously hard
With updates at the edge it can only get worse (high concurrency, stale data)




```
Get("bob_posts") => ["X"]
Put("bob_posts", [])
Get("bob_posts") => ???
```

```
Get("bob_posts", ["X"])
Put("bob_posts", ["X", "Y"])
Get("bob_posts") => ???
```

Step 1: use high-level conflict-free replicated objects (CRDTs)

- CRDT offers a predefined deterministic outcome on concurrent updates
- Pick a data type from catalogue (sets, counters, lists…) or define one!

```
bob_posts.get() => ["X"]
bob_posts.remove("X")
bob_posts.get() => ["Y"]
```

```
bob_posts.get() =>["X"]
bob_posts.append("Y")
bob_posts.get() => ["Y"]
```

- Bonus: switch between operation- or state-propagation for performance

Programming EC system at the edge

Step 2: asynchronous transactions for multi-object access

- A useful abstraction that hides DC<->edge replication

```
Begin()
bob_notifications_counter.get() => 4
bob_friend_requests.get() => {“anna”}
bob_friends.add(“anna”)
ana_friends.add(“bob”)
bob_notifications_counter.inc(-1)
Commit()
```

Queries operate on a consistent snapshot

Updates on different objects visible atomically

Asynchronous commit by default + session guarantees

Atomicity: updates visible atomically

Queries execute in a consistent snapshot

Programming EC system at the edge

Step 3: give control over data freshness and other guarantees

- Unit of control: session / transaction / object access

```
Begin(SNAPSHOT_ISOLATION, CACHED)
bob_friends.get(SUBSCRIBE_UPDATES)
Commit()
```

Isolation levels

- Snapshot isolation
- Repeatable reads

Freshness levels

- Cached
- Most recent

Fault-tolerant

- DC failure tolerant

Efficiency of commit

- Sync/async commit

Programming EC system at the edge

Step 4: being notified of other users' updates

```
Begin(SNAPSHOT_ISOLATION, CACHED)
Subscribe( bob_wall, Listener( Update u) {
                                    //bob wall modified
                              });
Commit()
```

Best-effort

Information for establishing FIFO

Quick for supporting realtime applications

Outline

Motivation

Programming model

Architecture

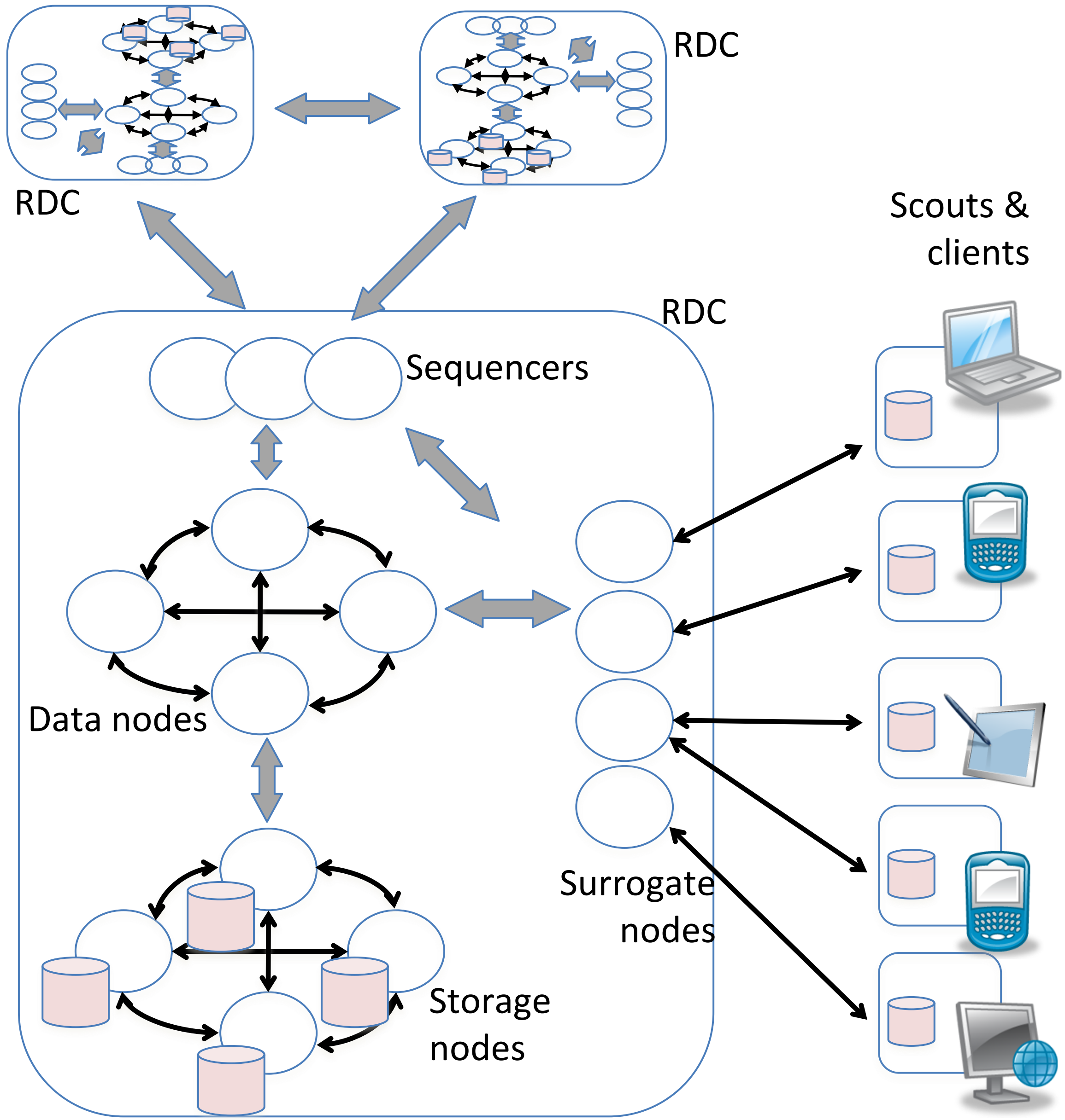
Evaluation

SwiftCloud: architecture

Clients
- Run applications

Scouts
- Cache mutable data
- @clients or @CDN

Data center
- Surrogate
  - Client proxy in DC
- Sequencer
  - Orders transactions
  - Keep info on DC state
- Data nodes
  - Maintains data copies: kind of memcached



Transaction execution

For each CRDT, the system maintains a list of versions

- To be more precise: keep a versioned CRDT

Transactions access a given CRDT version depending on the isolation level

- More on that on Marek's talk

Transaction Commit

Surrogate receives a transaction and replies it in the DC, as follows:

1. get tx identifier from sequencer

   sequencer can start replicating transaction

2. execute updates on CRDTs, by contacting data servers – new version is generated
3. makes the transaction visible, by updating the vector that summarizes DC state in sequencer

   this makes sure that a new transactions only sees a complete transaction

Outline

Motivation

Programming model

Architecture

Evaluation

Evaluation environments

DC with single node running all components

DC @ Amazon EC2 (Europe and US West)

CDN + clients @ planetlab

Configuration:

- latency(client-CDN) + latency(CDN-DC) <= latency(client-DC)
- Rationale: CDNs are at client ISP and may have privileged connectivity to the DCs

Evaluation environment

Scenarios

- scout@client (no CDN nodes in this case)
- scout@CDN
- scout@DC (current web systems configuration)

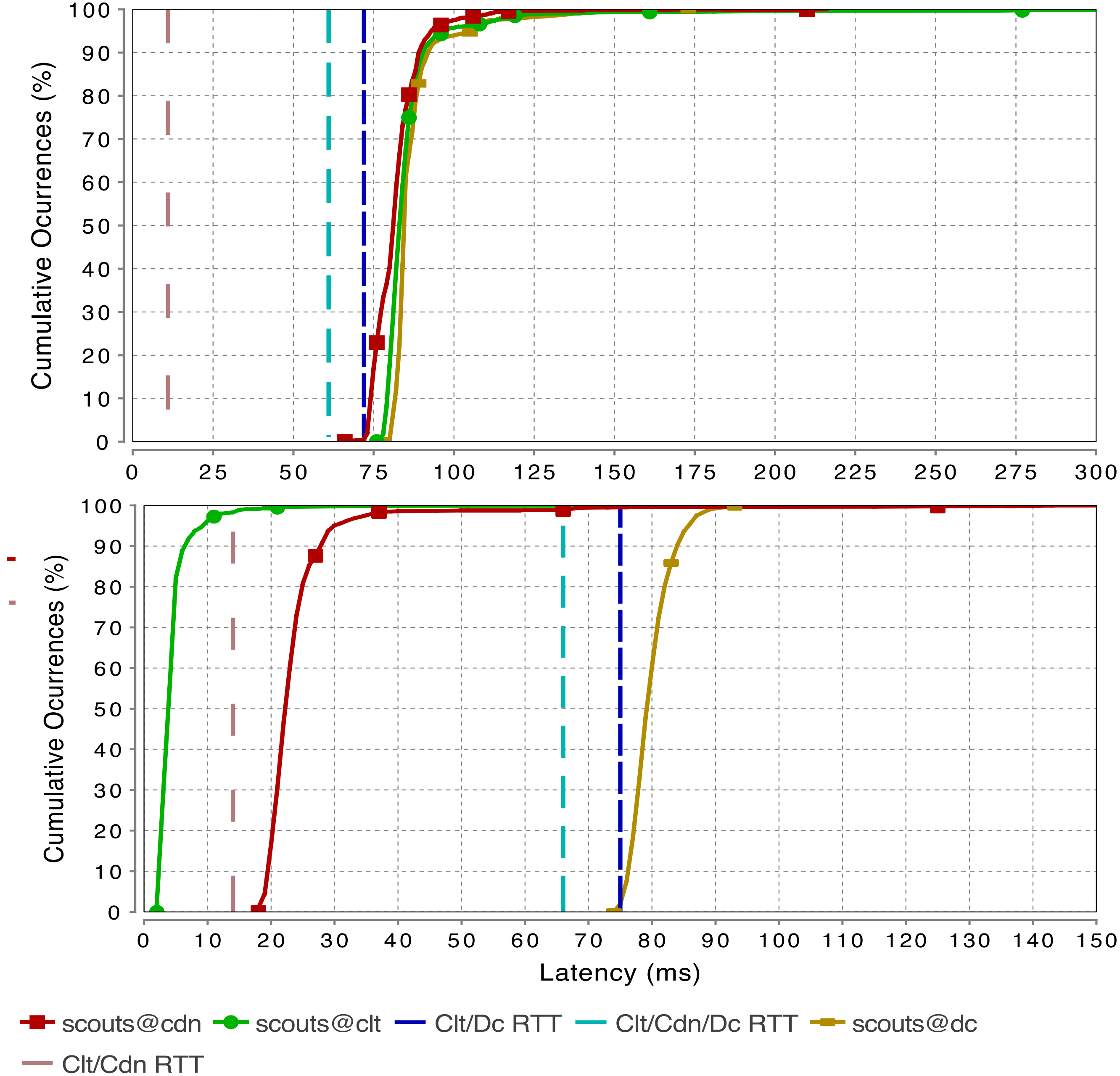
Applications

- Swiftdoc – two clients replay wikipedia traces; ping-pong for measuring latency
- SwiftSocial – social networking application (Facebook-like)

SwiftDoc: propagation latency

Different scout

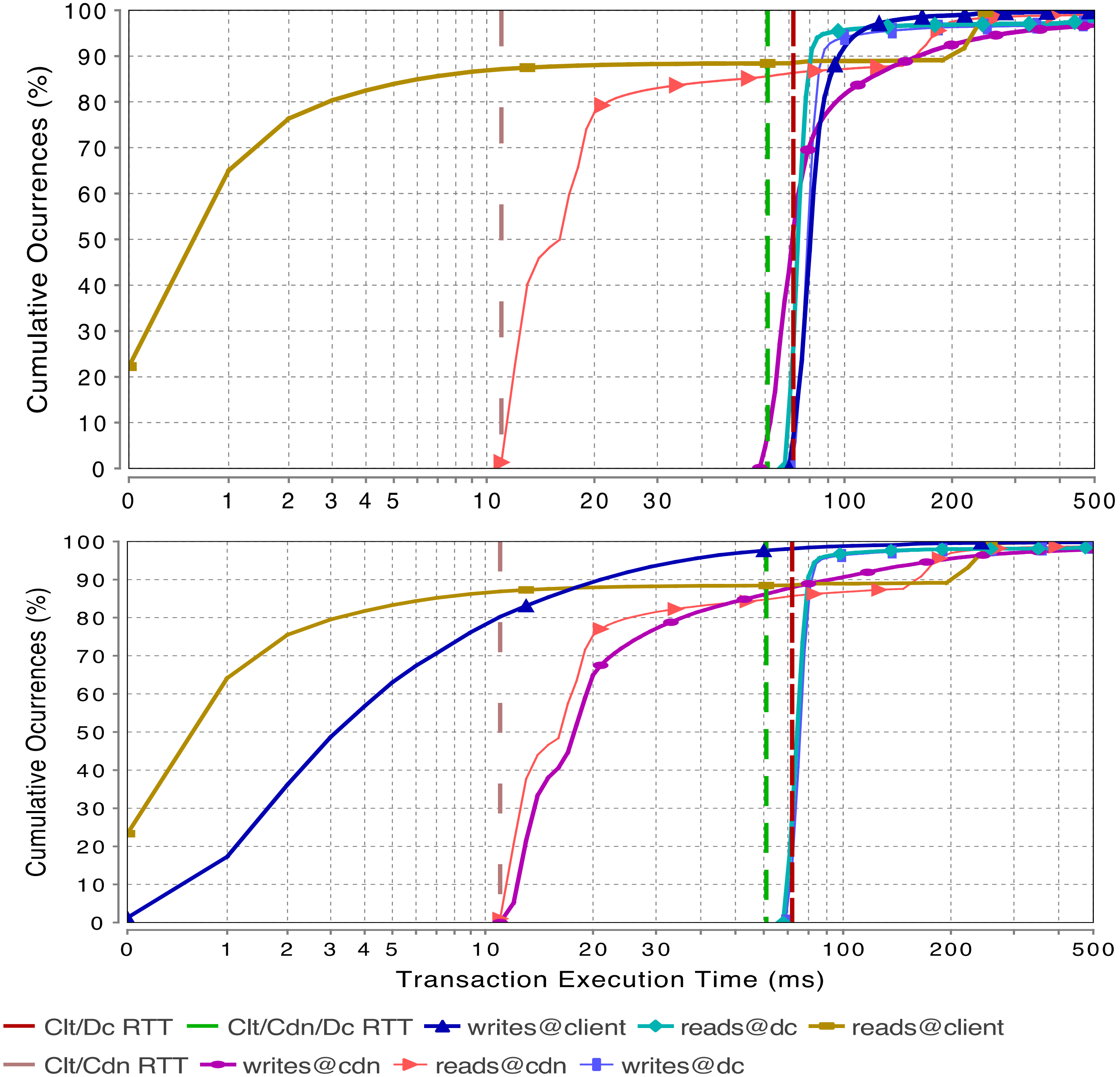
Shared scout



SwiftSocial: sync vs. async commit

Sync
RR cached

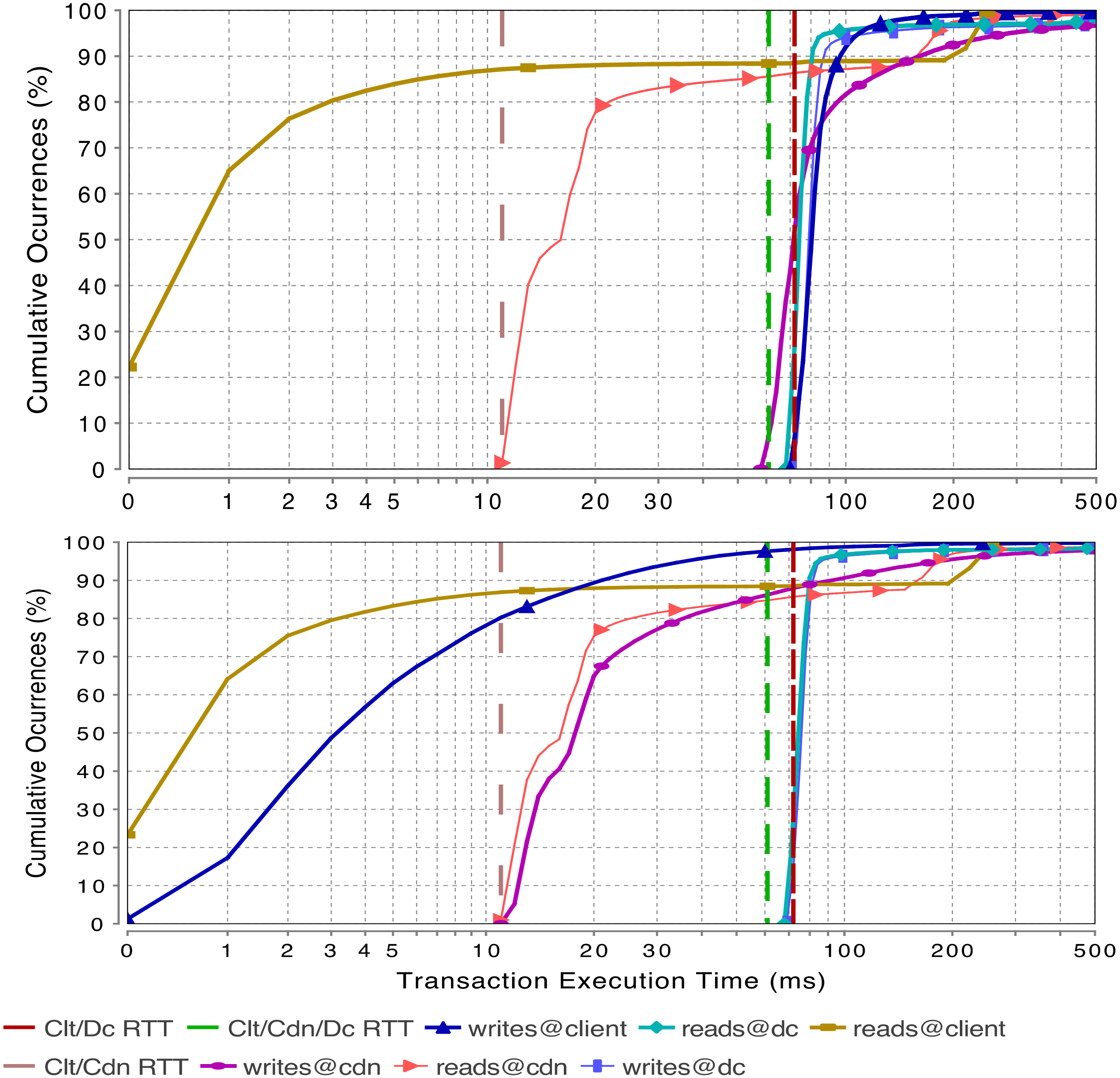
Async
RR cached



SwiftSocial: sync vs. async commit

Sync
RR cached

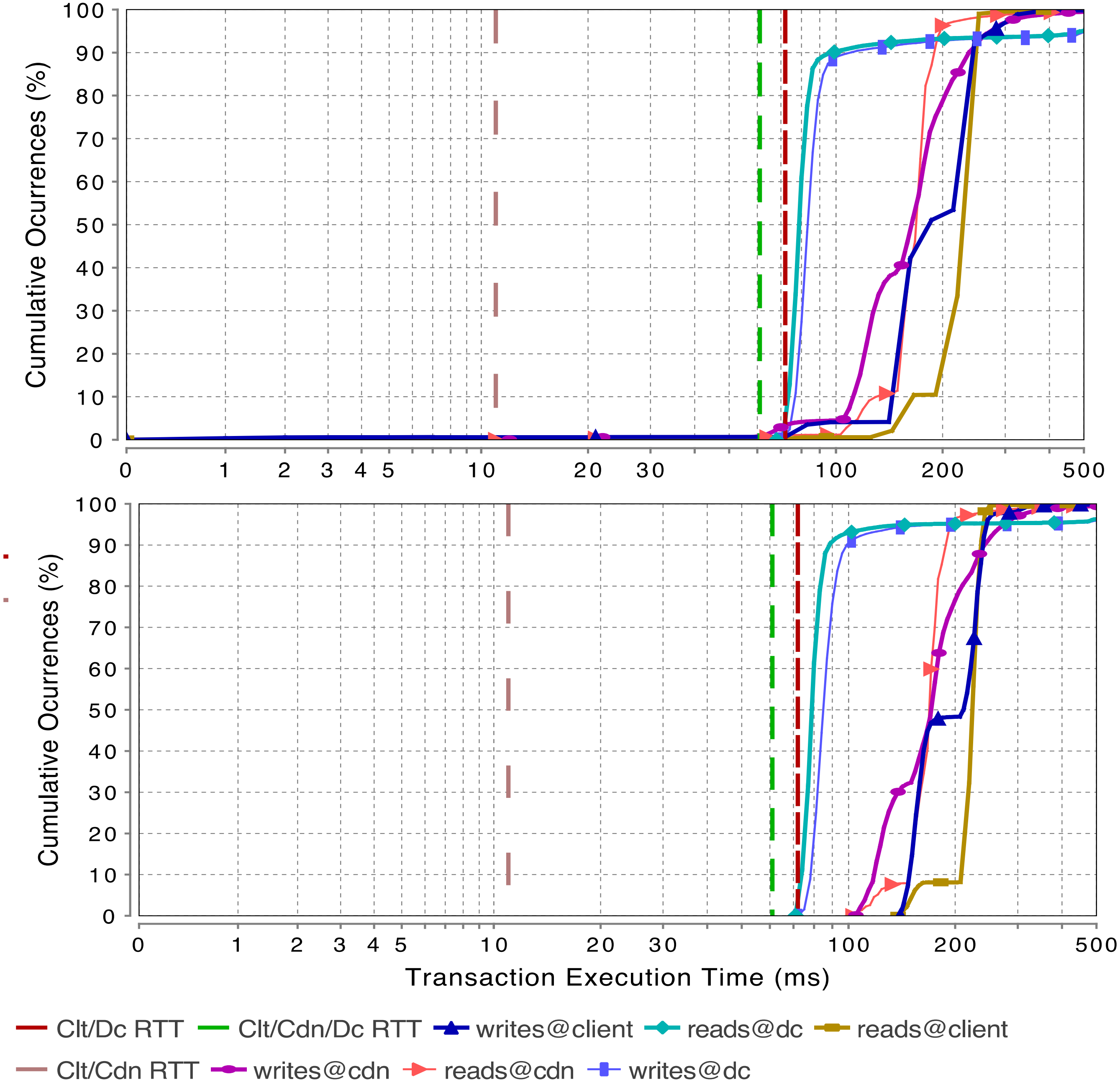
Async
RR cached



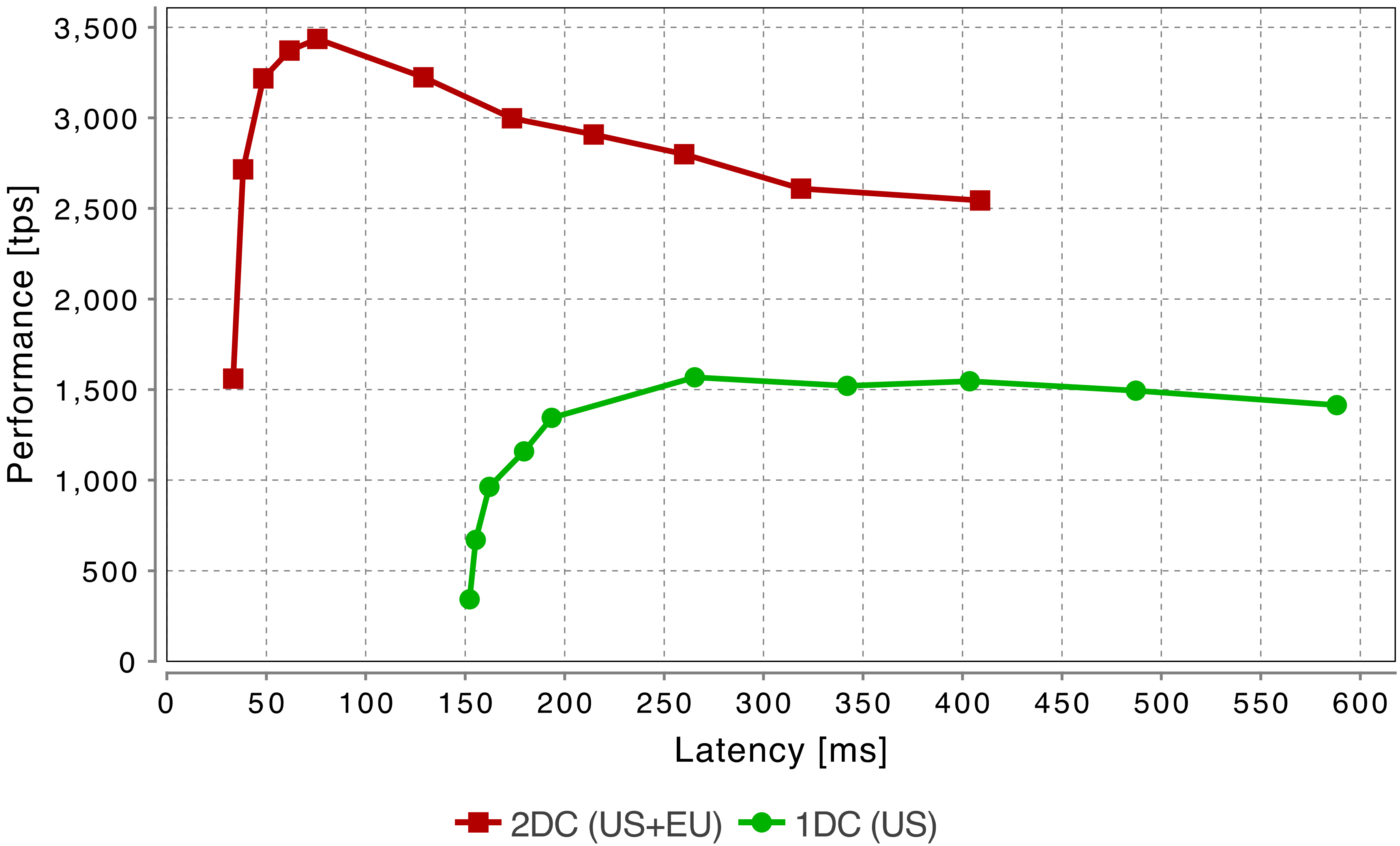
SwiftSocial: SI & most recent

Async
SI cached

Async
RR
most recent



Swift social: scalability



Final remarks

- Key-CRDT store supporting
  - Geo-replication among data centers
  - Geo-replication to the client nodes
- Key features
  - Efficient causality tracking
  - Asynchronous transactions with multiple semantics, session guarantees

Future work

Interface for notifications

Cache coherence/invalidation mechanism