Towards New Systems for Mobile/Cloud Applications

Irene Zhang

UNIVERSITY of WASHINGTON

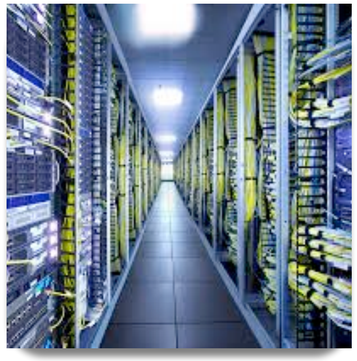
User devices have undergone a fundamental change

Mainframe

1960s

Desktop PC

1980s



Cloud

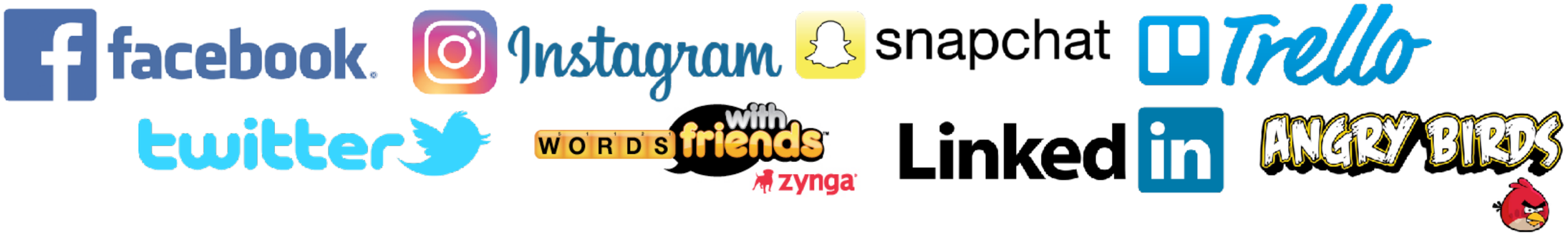
2000s

Mobile

2010s

from **centralized** to **distributed**.

Today's popular apps no longer run on a single desktop but span cloud servers and mobile devices.



And today's important apps …

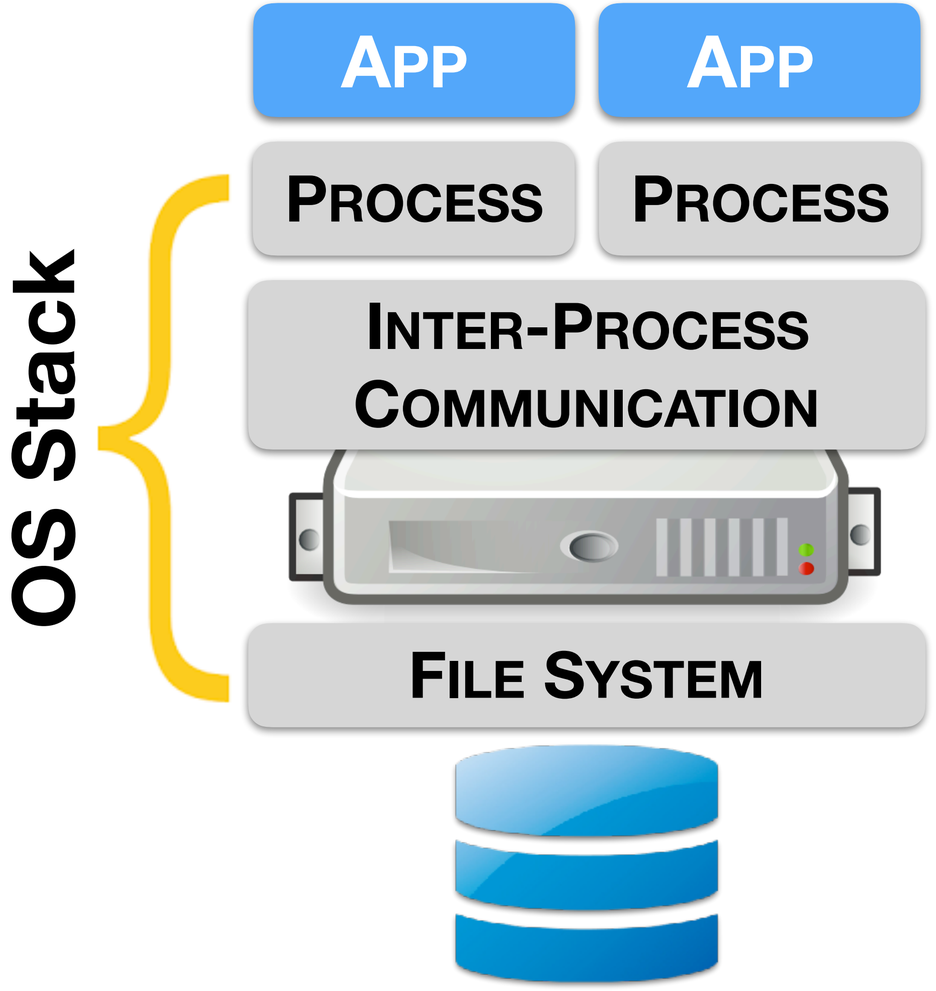
And even today's desktop apps …

iWork for iCloud

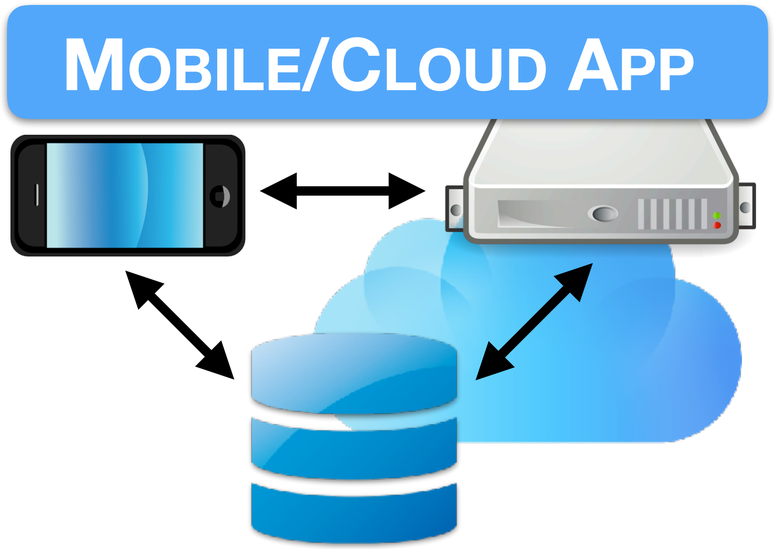
Desktop apps rely on operating systems

to help them meet important design challenges.

- How to deploy multiple apps on a single computer?
- How to share data and coordinate between processes?
- How to persistently store data?



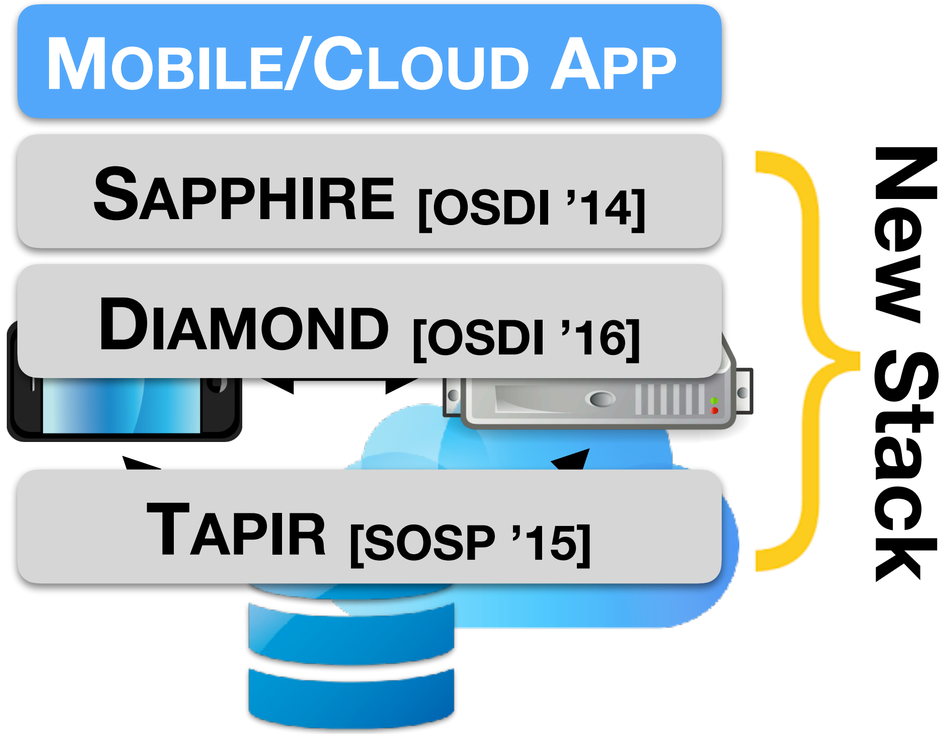
Mobile/cloud apps have a new set of challenges.



- How to deploy a single app on **multiple** devices and servers?
- How to share data and coordinate between **distributed** processes?
- How to persistently store **large-scale** data?

They need new systems to help them meet these challenges!

This Talk: New Systems for Mobile/Cloud Apps



General-purpose: Supports a wide range of apps

Easy-to-use: Does not require systems expertise

Efficient: Imposes low performance overhead

These systems are just a first step!

Talk Outline

- Introduction
- How to deploy a single app on **multiple** devices and servers?
- How to share data and coordinate between **distributed** processes?
- How to persistently store **large-scale** data?
- Future Work

SAPPHIRE [OSDI '14]

DIAMOND [OSDI '16]

TAPIR [SOSP '15]

Building a Mobile/Cloud App: The 100 Game

Multi-player, tu

Play

On each turn

numbe

First player to reach 100 wins

Availability: Games run even if some clients are offline

Fault-tolerance: Games run even if some clients crash

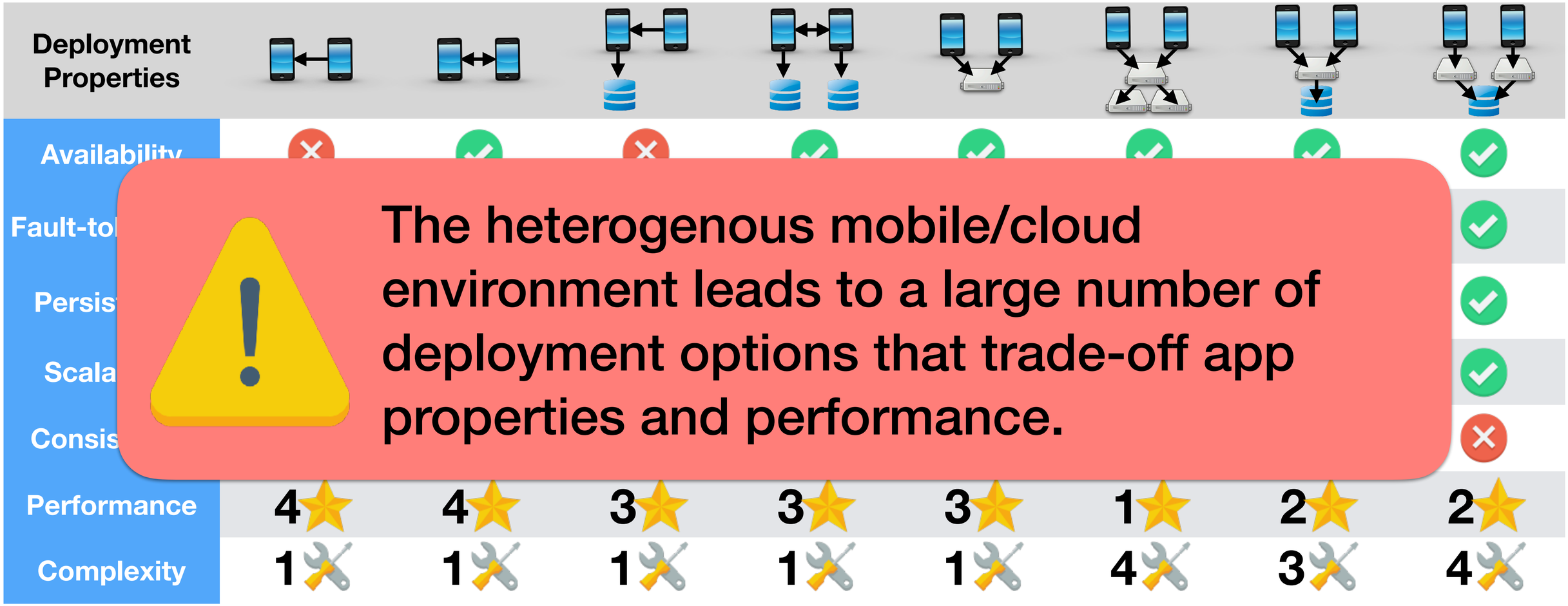
Persistence: Moves are never lost

Scalability: Many games can run at once

Consistency: Clients see moves in the same order

https://en.wikipedia.org/wiki/Nim#The_100_game

Background: Deploying the 100 Game



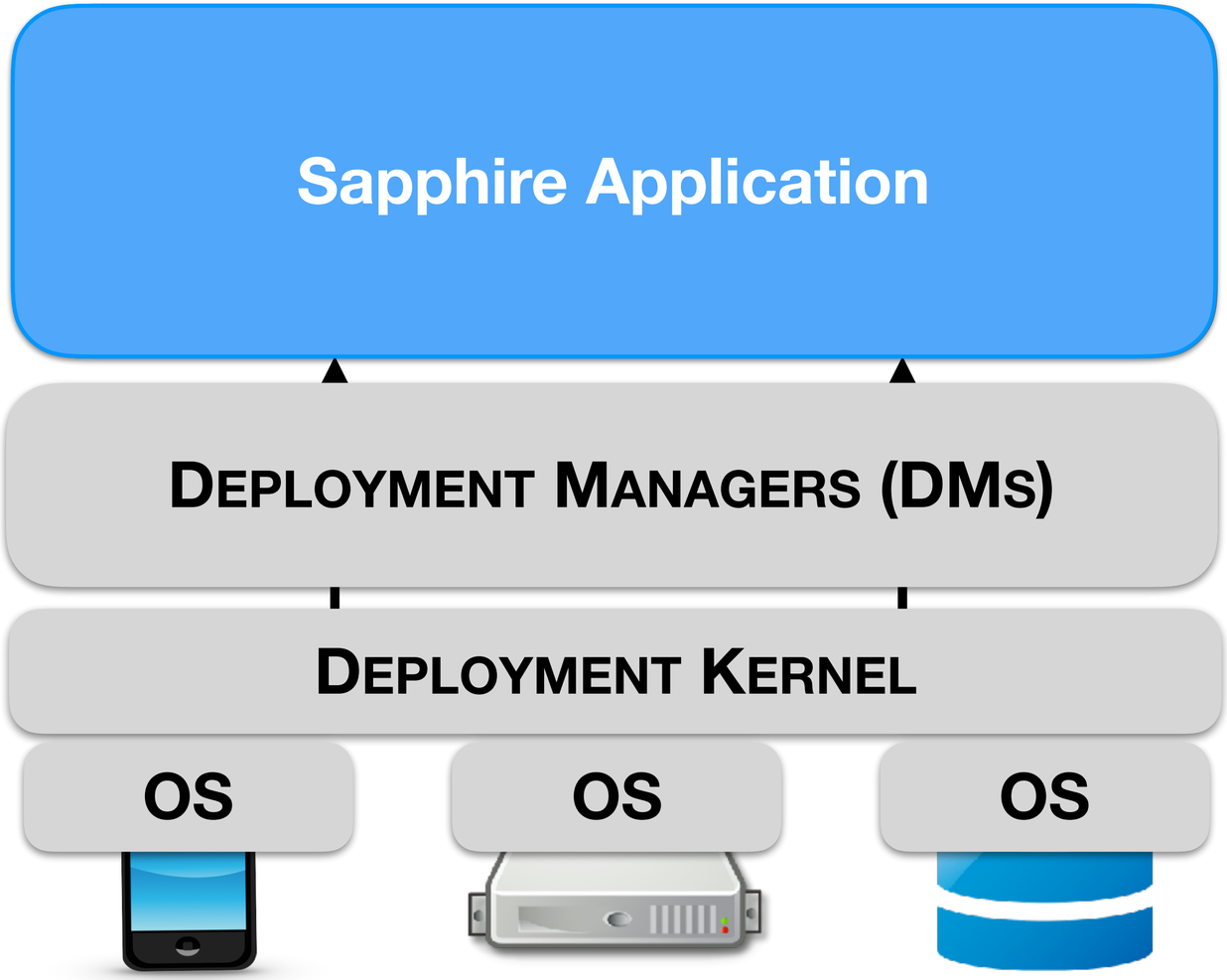
Sapphire

Sapphire is the first **deployment management system**, which has the following features:

- Automatically deploys apps across mobile devices, cloud servers and storage systems
- Allows programmers to easily choose, change, and reuse deployment options
- Enables advanced programmers to easily extend and customize deployment options

[Customizable and Extensible Deployment for Mobile/Cloud Applications. OSDI 2014]

Sapphire System

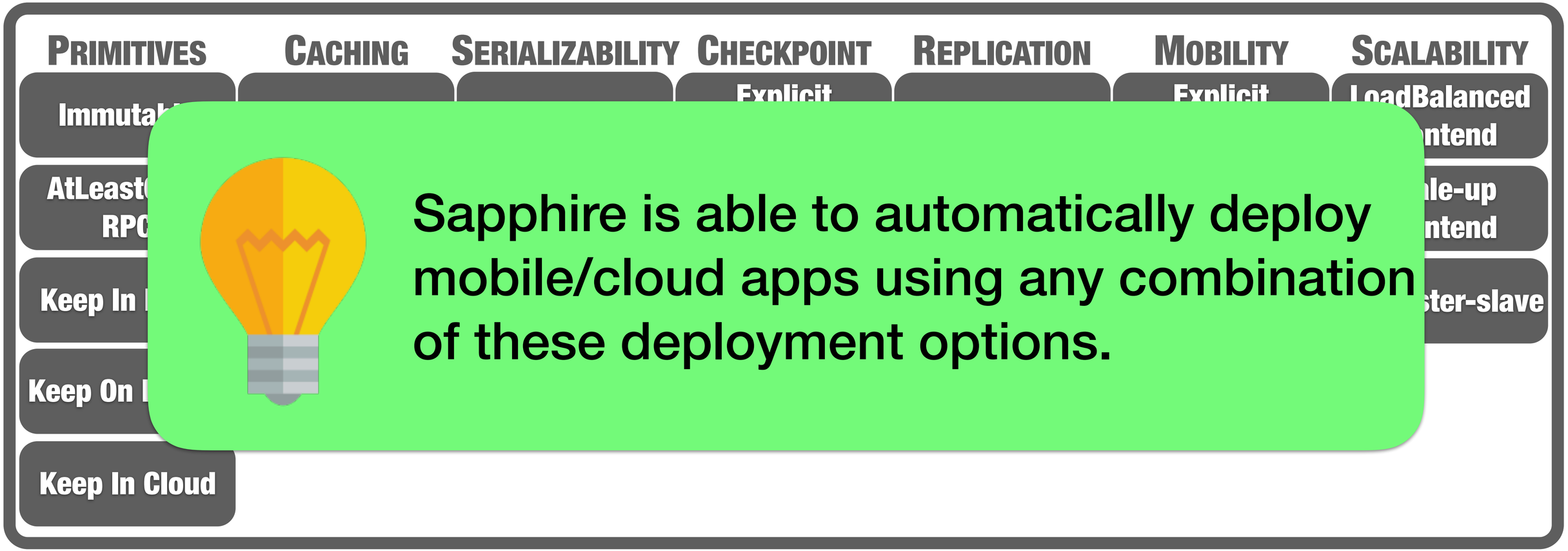


Partitioned into Sapphire Objects (SOs).

Extend Deployment Kernel to create custom runtime for each SO.

Provides basic services for SOs (e.g., creation, migration, best-effort RPC).

Sapphire Deployment Library



Sapphire provides easy and flexible deployment.

Talk Outline

- Introduction
- How to deploy a single app on **multiple** devices and servers?
- How to share data and coordinate between **distributed** processes?
- How to persistently store **large-scale** data?
- Future Work

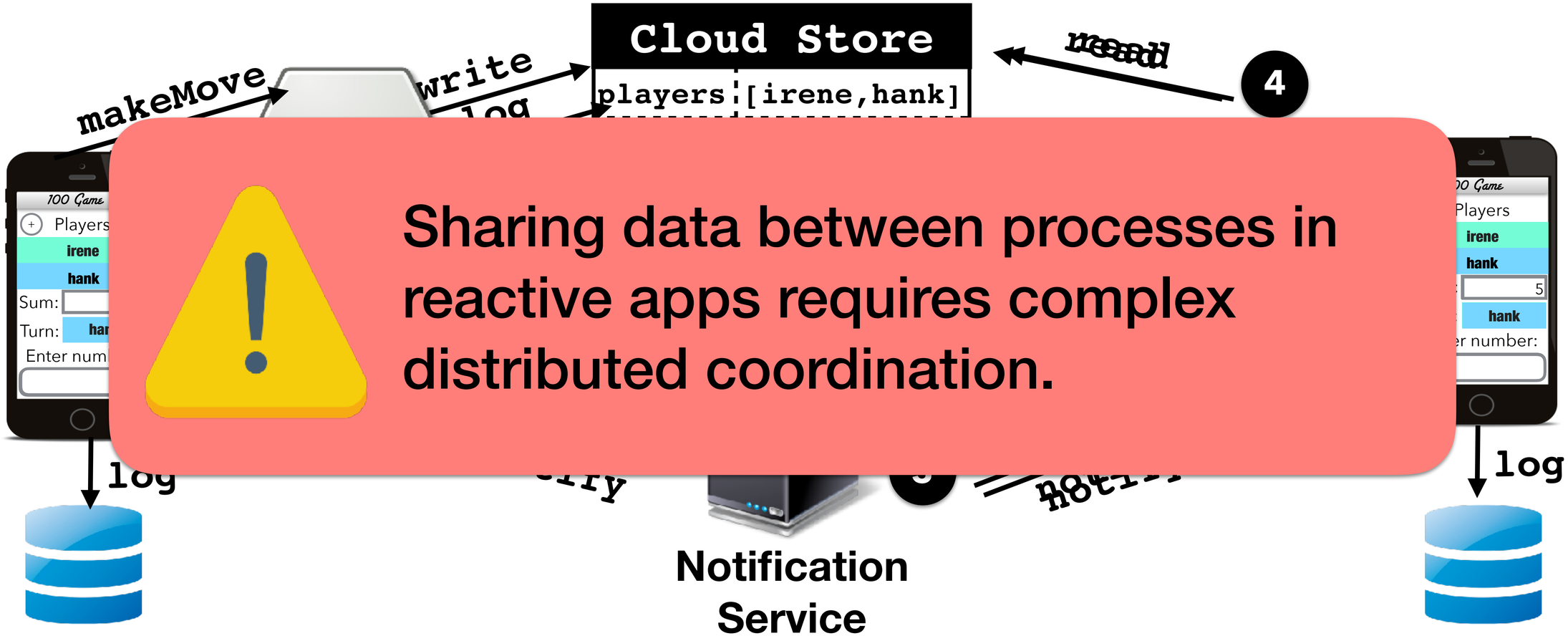
SAPPHIRE [OSDI '14]

DIAMOND [OSDI '16]

TAPIR [SOSP '15]

Mobile/cloud apps have a new property.

Background: Building a Reactive 100 Game



Diamond

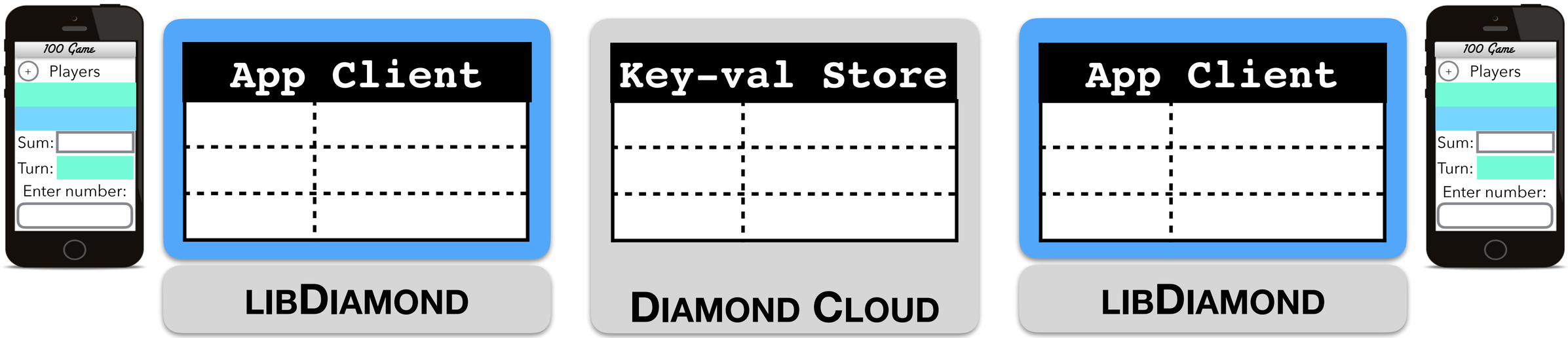
Diamond is the first **reactive data management system**, which provides the following guarantees:

- Ensures updates to shared data are consistent and durable
- Coordinates and synchronizes updates reliably across mobile clients and cloud storage
- Automatically triggers application code in response to updates to shared data

[Automating Data Management for Wide-area, Reactive Applications. OSDI 2016]

Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures



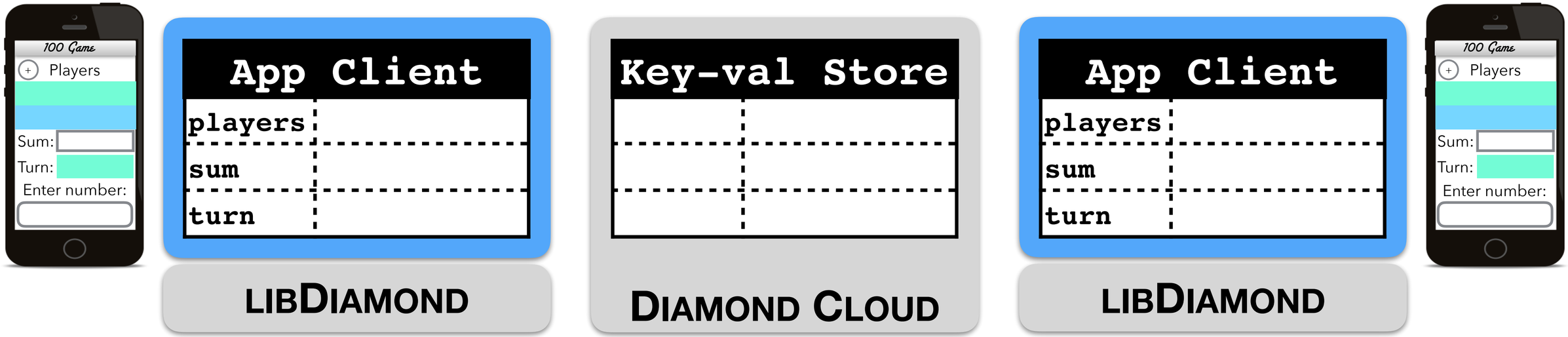
Reactive Data Types (RDTs)
Shared, persistent data structures

- Simple data structures including primitives (e.g., string, long), collections (e.g., list) and Conflict-free Data Types (e.g., counter, set)
- Data type semantics avoid false sharing and enable commutative operations
- Defined in libDiamond language bindings

Diamond Programming Model

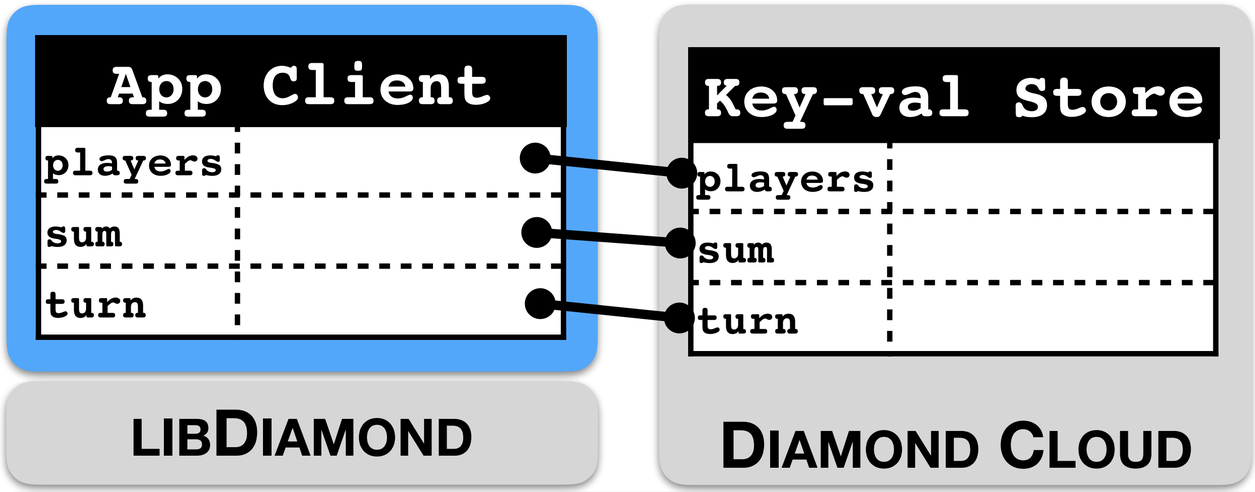
Reactive Data Types (RDTs)
Shared, persistent data structures

Reactive Data Map (rmap)
Binds RDTs in the app to the Diamond store



Reactive Data Map (rmap)
Binds RDTs in the app to the Diamond store

- Key abstraction for providing flexible, shared memory
- Gives apps control over what app data is shared and how it is organized
- Enables Diamond to automatically provide availability, fault-tolerance and consistency to RDTs



game.cc

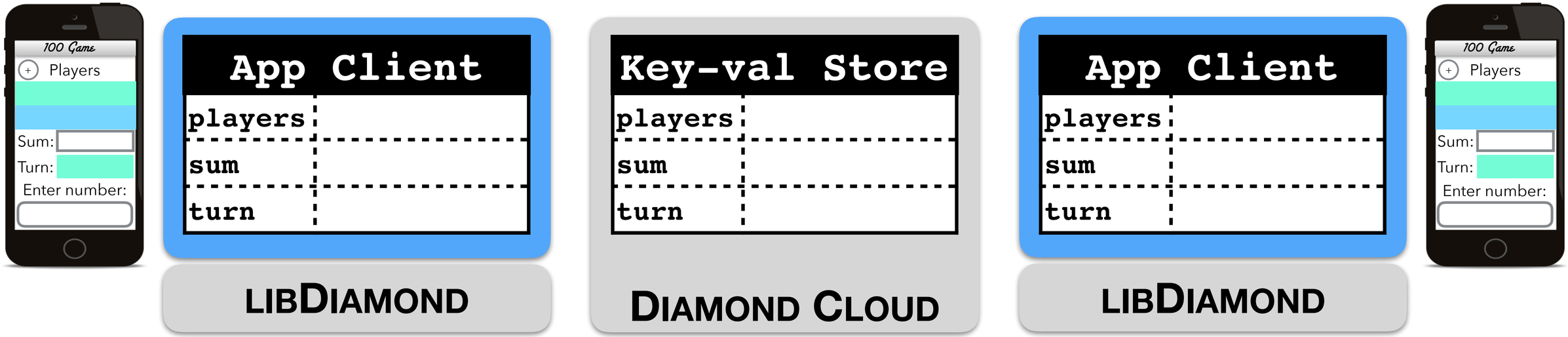
```
rmap(players, “players”);
rmap(sum, “sum”);
rmap(turn, “turn”);
```

Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Execute app code to update shared RDTs atomically and consistently

Reactive Data Map (rmap)
Binds RDTs in the app to the Diamond store



Read-write Transactions
Execute app code to update shared RDTs atomically and consistently

- Encapsulate reads and writes to shared RDTs with ACID guarantees
- Give application programmers control over isolation and atomicity of updates
- Ensure safe concurrent access to shared data from app clients

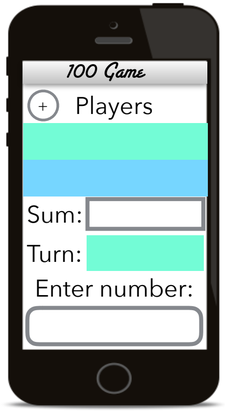
Diamond Programming Model

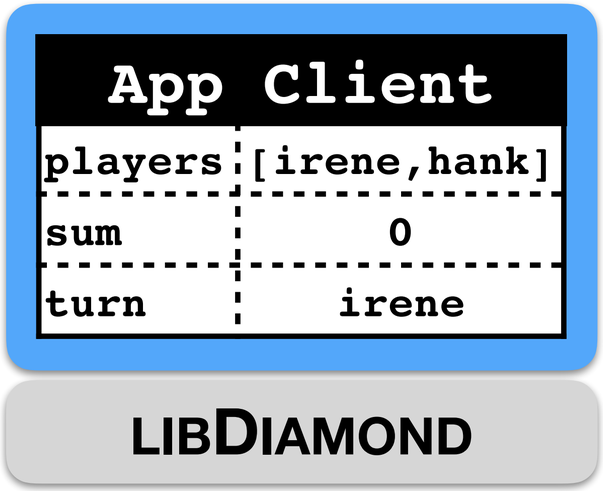
Reactive Data Types (RDTs)
Shared, persistent data structures

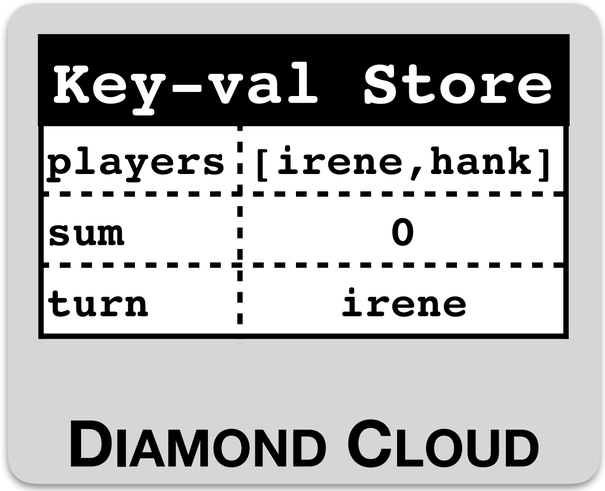
Read-write Transactions
Execute app code to update shared RDTs atomically and consistently

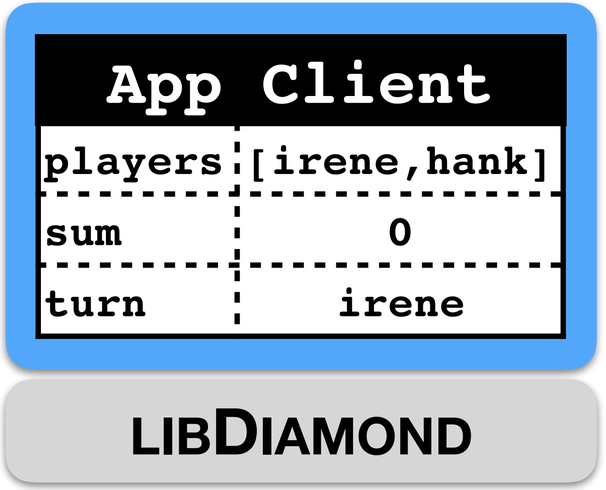
Reactive Data Map (rmap)
Binds RDTs in the app to the Diamond store

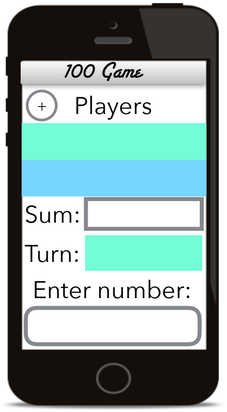
Reactive Transactions (read-only)
Automatically re-execute app code when the read set updates











Reactive Transactions (read-only)
Automatically re-execute app code when the read set updates

- Key abstraction for automatically propagating updates to local data
- Gives apps a consistent view of shared data and control over what to sync
- Automatically triggers app code in response to updates from read-write transactions

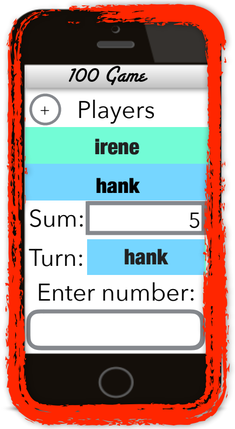
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Execute app code to update shared RDTs atomically and consistently

Reactive Data Map (rmap)
Binds RDTs in the app to the Diamond store

Reactive Transactions (read-only)
Automatically re-execute app code when the read set updates



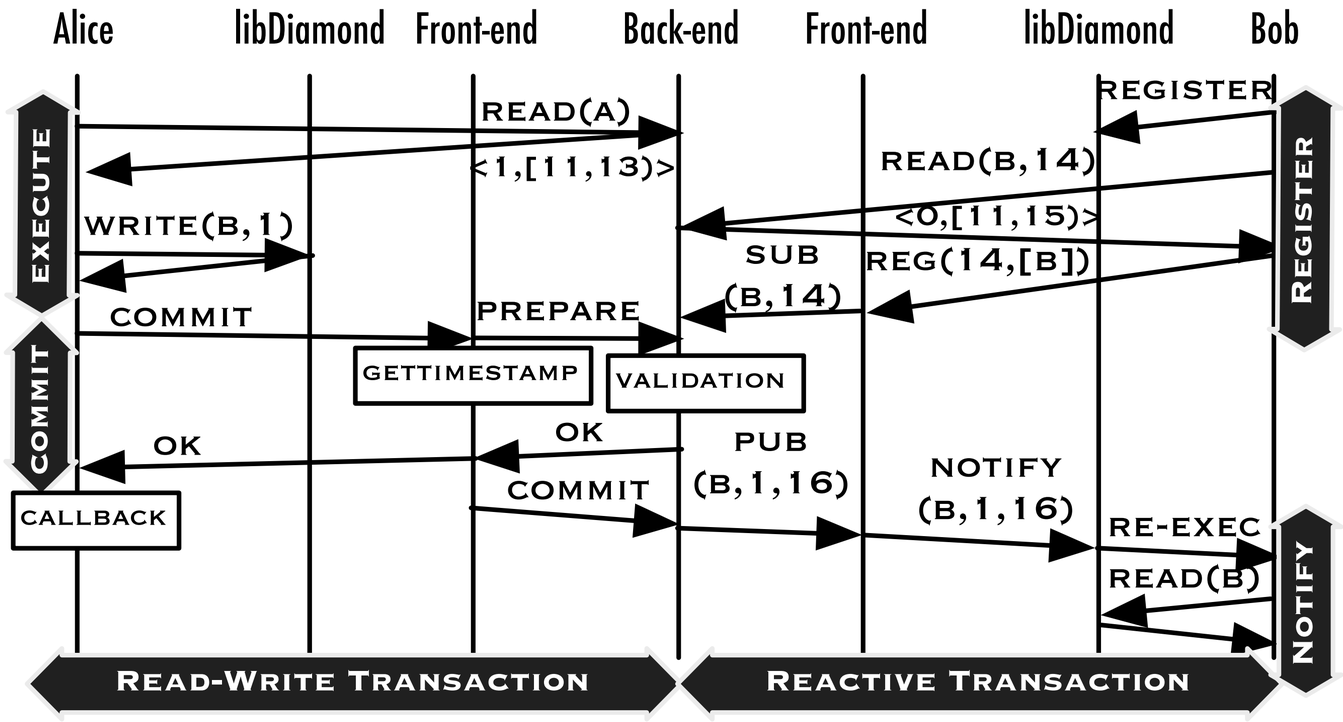
Diamond replaces distributed coordination in reactive apps with distributed, shared data structures.

Diamond Implementation

Prototype features:

- New data-type optimistic concurrency control (DOCC) mechanism to reduce wide-area aborts by leveraging CRDT semantics
- Front-end language bindings for C++, Java/Android and Python
- Cross-compiles for both x86 and ARM architectures
- Implemented in 11,795 lines of C++

Diamond Transaction Protocol



Evaluation Overview

Does Diamond simplify reactive mobile/cloud applications?

How does Diamond perform compare to a hand-coded implementation?

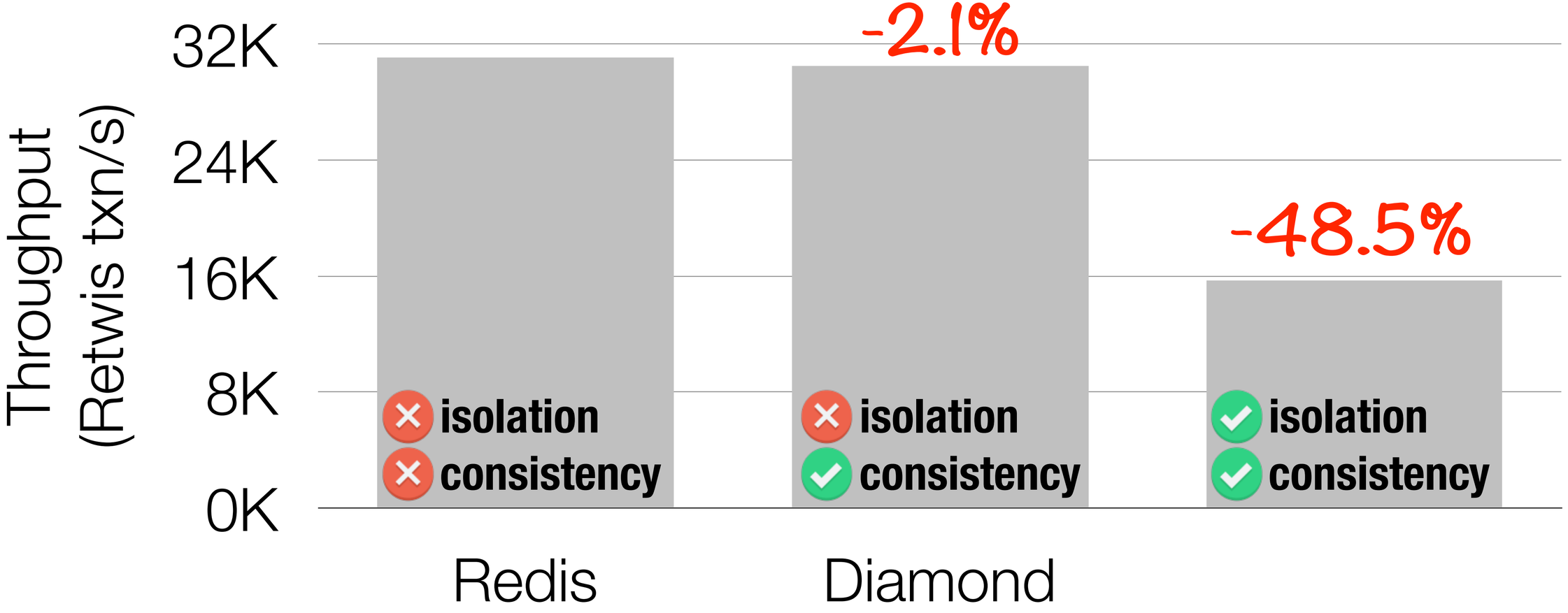
Workload: Retwis-based Twitter benchmark

Testbed: Google Compute Engine VMs (5 shards x 3 replicas)

Diamond reduces the complexity and improves the guarantees of reactive apps.

	100 Game		Chat Room		Scrabble		Twitter Clone	
	Original	Diamond	Original	Diamond	Original	Diamond	Original	Diamond
Complexity (LoC)	46	34	335	225	8729	7603	14278	12554
Availability	✅	✅	✅	✅	✅	✅	✅	✅
Fault-tolerance	❌	✅	❌	✅	❌	✅	❌	✅
Persistence	✅	✅	❌	✅	❌	✅	✅	✅
Scalability	✅	✅	❌	✅	❌	✅	✅	✅
Consistency	❌	✅	❌	✅	✅	✅	❌	✅
Reactivity	❌	✅	✅	✅	✅	✅	❌	✅

Diamond's data management has low overhead.



Talk Outline

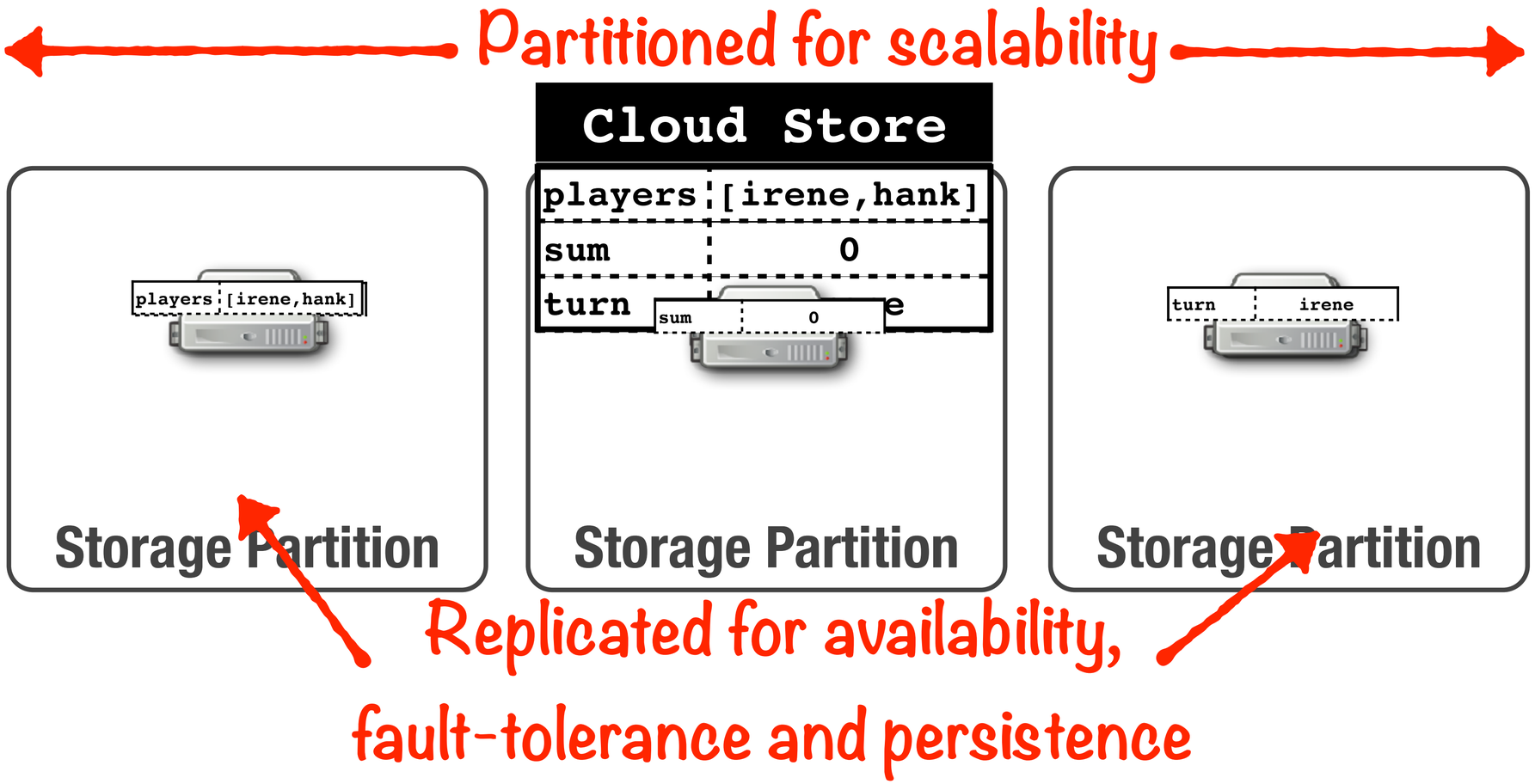
- Introduction
- How to deploy a single app on **multiple** devices and servers?
- How to share data and coordinate between **distributed** processes?
- How to persistently store **large-scale** data?
- Future Work

SAPPHIRE [OSDI '14]

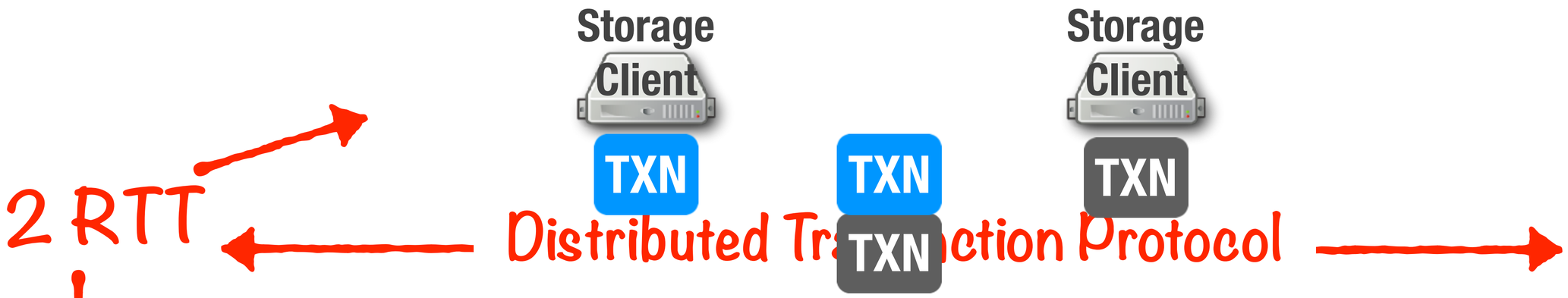
DIAMOND [OSDI '16]

TAPIR [SOSP '15]

Background: Distributed Storage Systems



Distributed transactions with strict ordering are easy to use but inefficient to coordinate.



Existing systems waste work using distributed transaction and replication protocols that both enforce strict ordering.

Replication Protocol (e.g., Paxos)

TAPIR

A new transaction protocol designed to enforce a linearizable transaction ordering atop a replication protocol with no ordering guarantees.

[Building Consistent Transactions with Inconsistent Replication. SOSP 2015]

Inconsistent Replication (IR)

New replication protocol providing ***unordered operations*** where replicas agree on operation results.

Inconsistent Replication (IR) Guarantees

IR provides a fault-tolerant, unordered operation set with the following guarantees:

IR's guarantees support test-and-set operations, which are sufficient for applications to detect and avoid conflicts without ordering.

Overlap with every previously added operation on at least one replica out of every quorum.

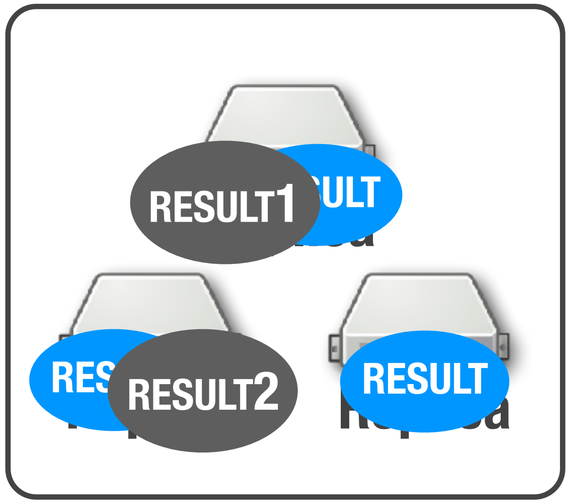
The IR Protocol (simplified)

1. Execute operation at replicas.

2. If results from a *super-majority* match, return result.

3. If not, application protocol picks a result.

4. Update result at replicas.



IR Pros & Cons

Fast: 1 round-trip fast path, 2 round-trip slow path

Efficient: No cross-replica coordination or leader needed to complete operations

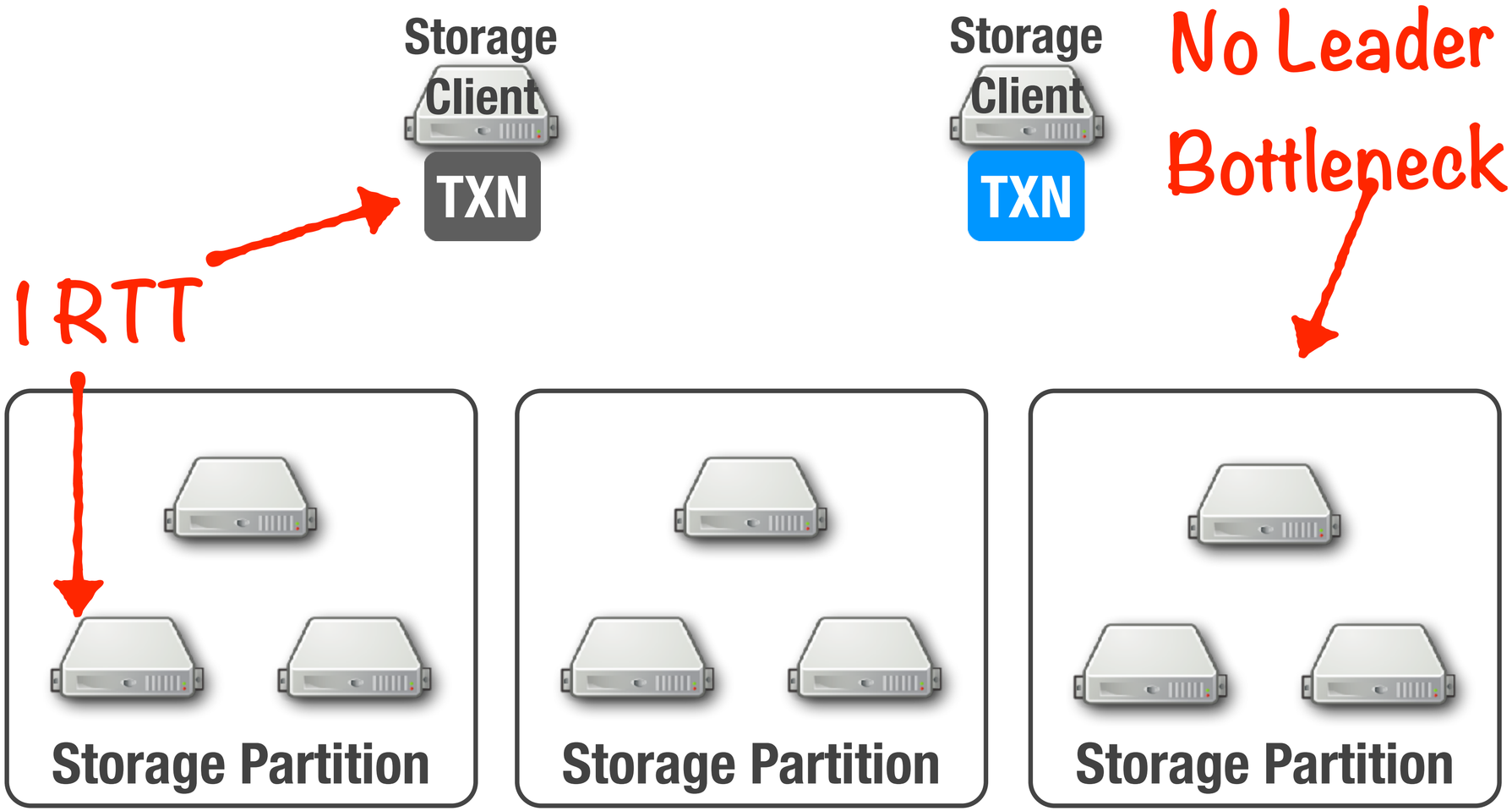
Less general: Does not ensure replicas appear as a single machine

Requires co-design: Need to carefully co-design application protocol for both correctness and performance

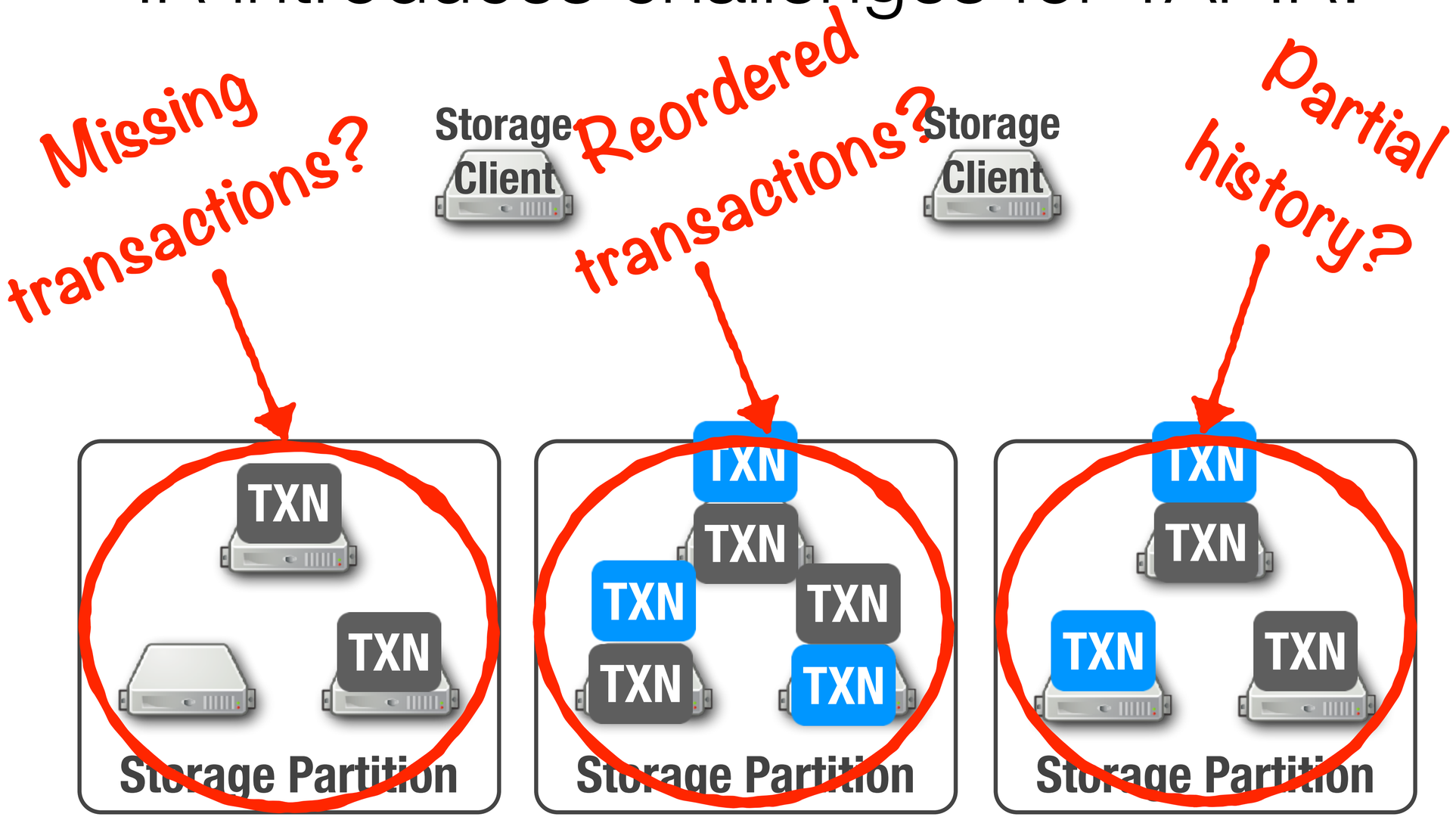
TAPIR

- New transaction protocol designed to work with inconsistent replication.
- Enforces a linearizable transaction ordering (i.e., strict serializability), like existing transactional storage systems (e.g., Google's Spanner).
- Uses two-phase commit for atomicity, optimistic concurrency control, and IR for replication.

TAPIR works with IR to provide more efficient distributed transactions.

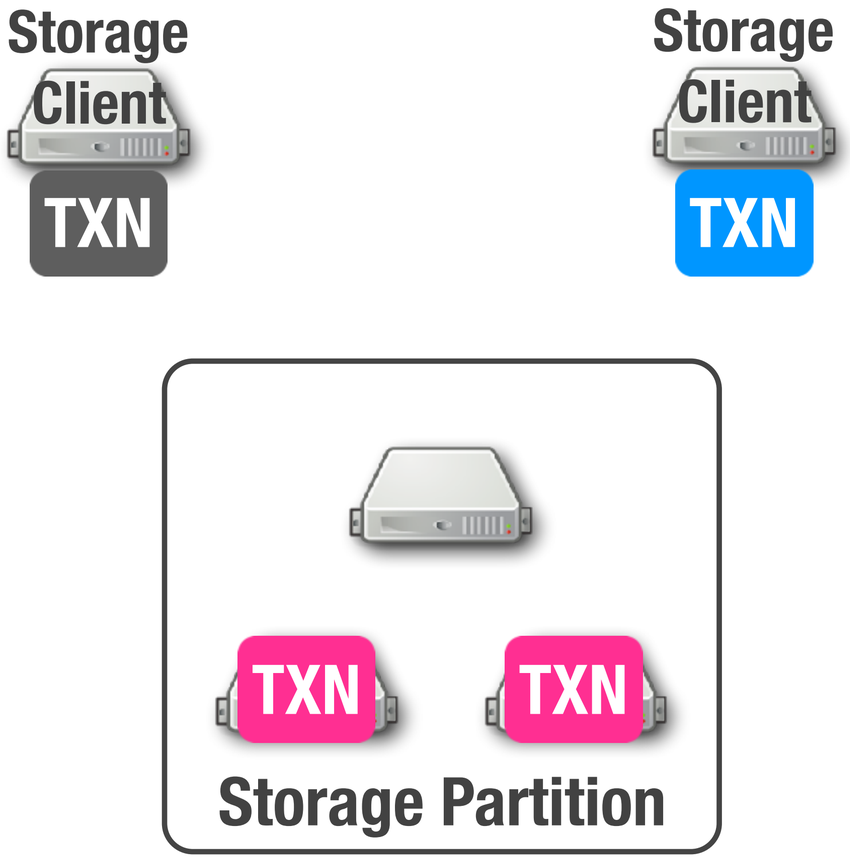


IR introduces challenges for TAPIR.



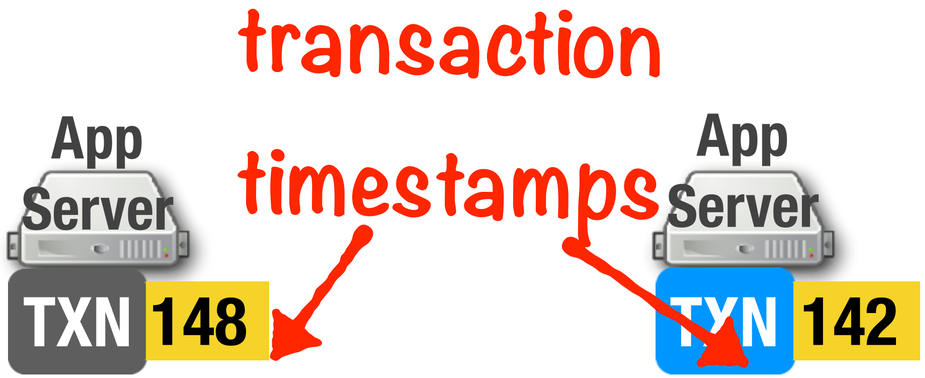
TAPIR uses optimistic concurrency control (OCC) to detect conflicts on IR.

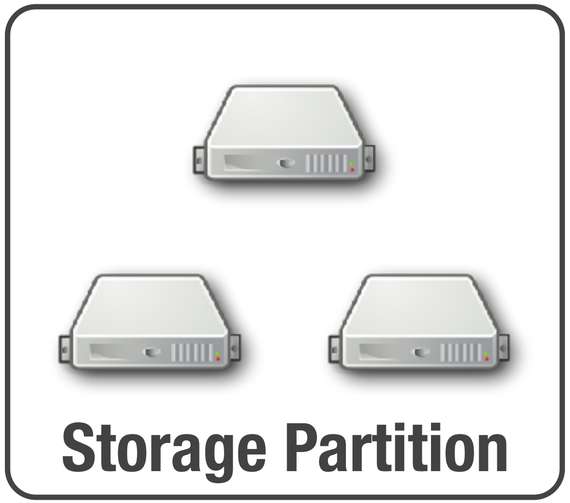
- OCC checks **one transaction at a time**, so a full transaction history is not necessary.
- IR ensures every pair of transactions is checked on at least one replica.
- OCC+IR ensures that every conflict is detected.



TAPIR uses loosely synchronized clocks to efficiently order transactions.

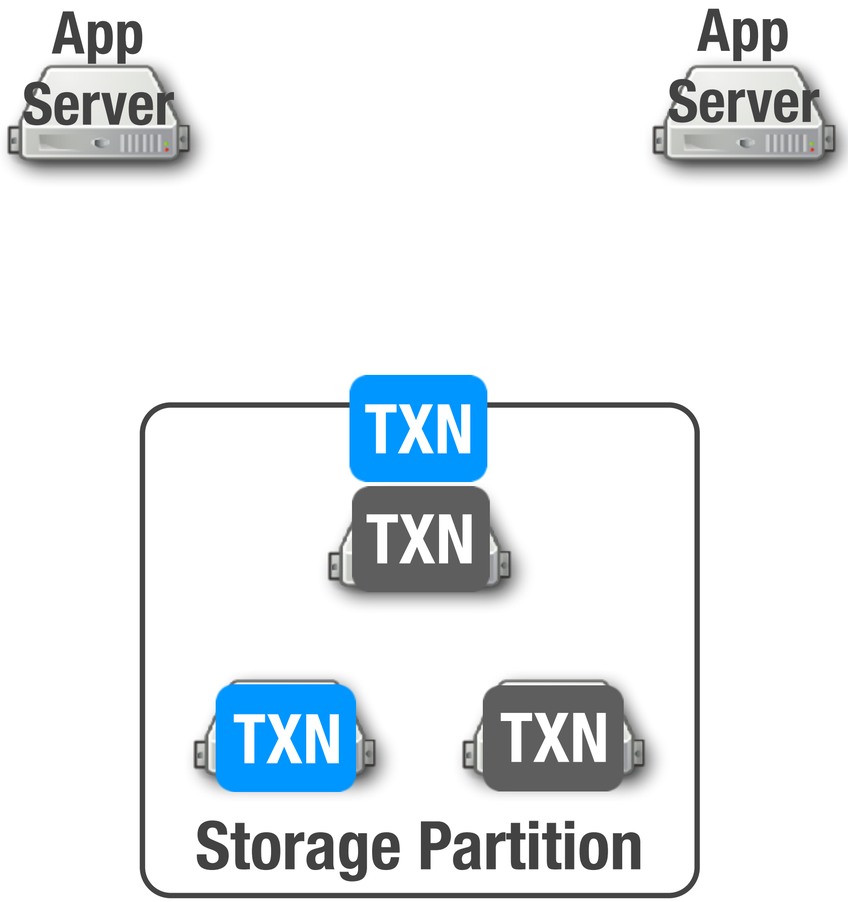
- Clients pick transaction timestamp using local clock.
- Replicas validate transaction at timestamp, regardless of when they receive the transaction.
- Clock synchronization for performance, not correctness.





TAPIR uses multi-versioning to reconcile inconsistent replicas.

- IR periodically synchronizes inconsistent replicas.
- TAPIR inserts versions using the transaction timestamp.
- OCC prevents inconsistent replicas from violating transaction ordering.



Experimental Overview

Does TAPIR outperform existing distributed transactional storage?

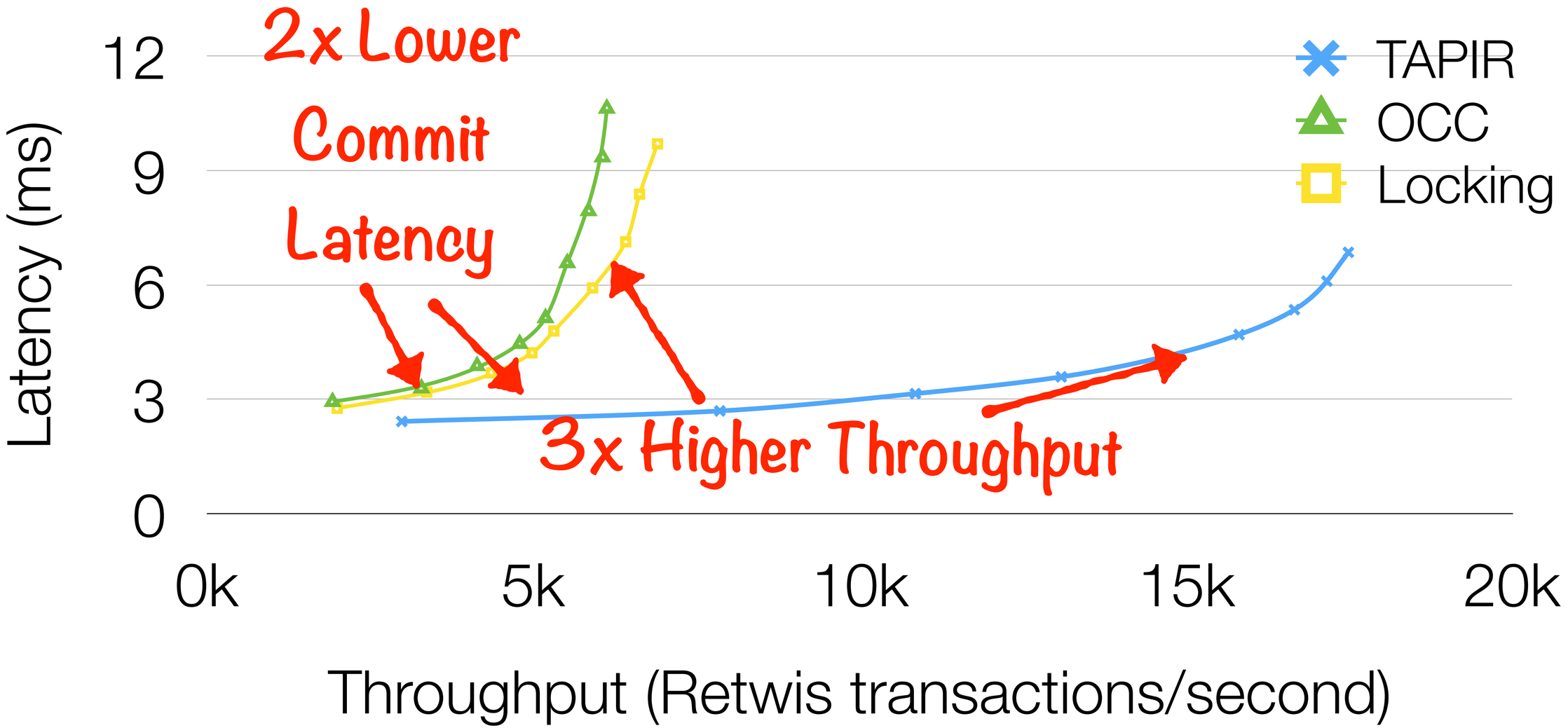
Can TAPIR make strong ordering guarantees more affordable?

Implementation: Transactional key-value store

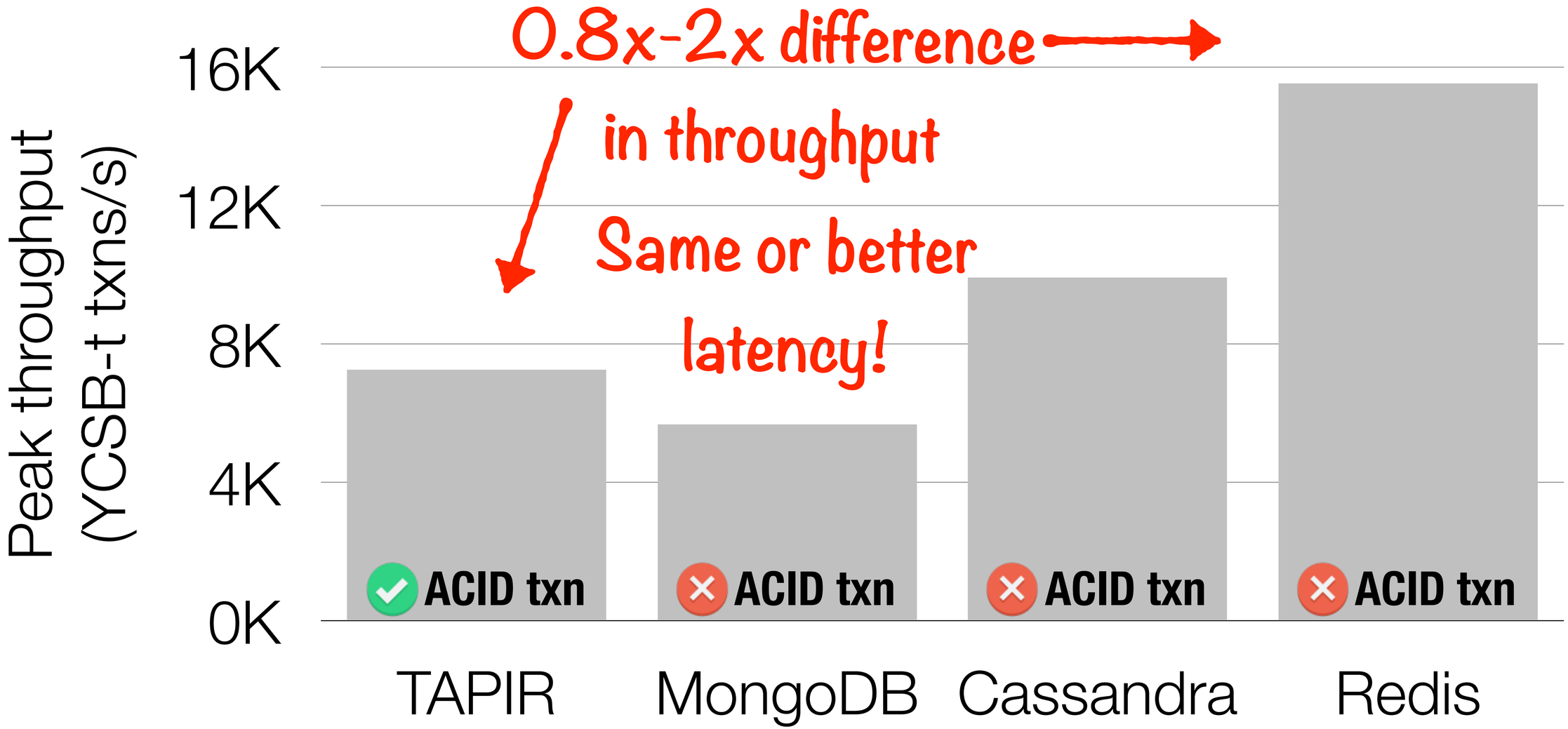
Workloads: Retwis Twitter clone & YCSB-t.

Testbed: Google Compute Engine VMs (5 shards x 3 replicas)

TAPIR improves both latency and throughput.



TAPIR makes distributed transactions affordable.



Talk Outline

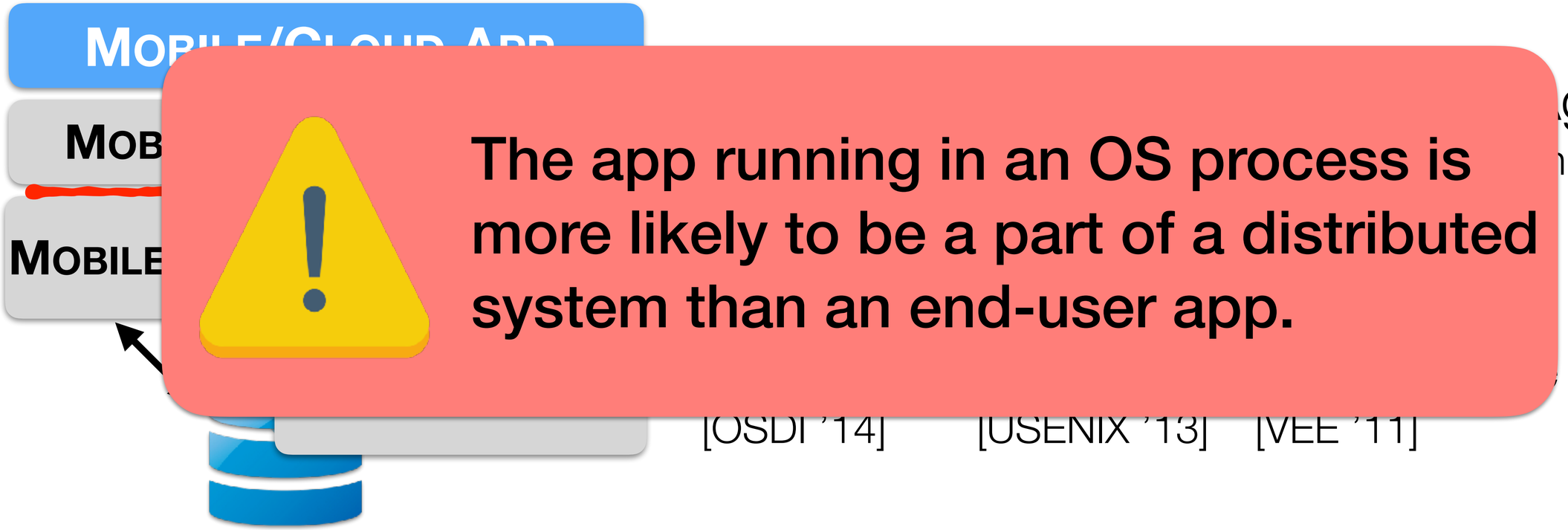
- Introduction
- How to deploy a single app on **multiple** devices and servers?
- How to share data and coordinate between **distributed** processes?
- How to persistently store **large-scale** data?
- Future Work

SAPPHIRE [OSDI '14]

DIAMOND [OSDI '16]

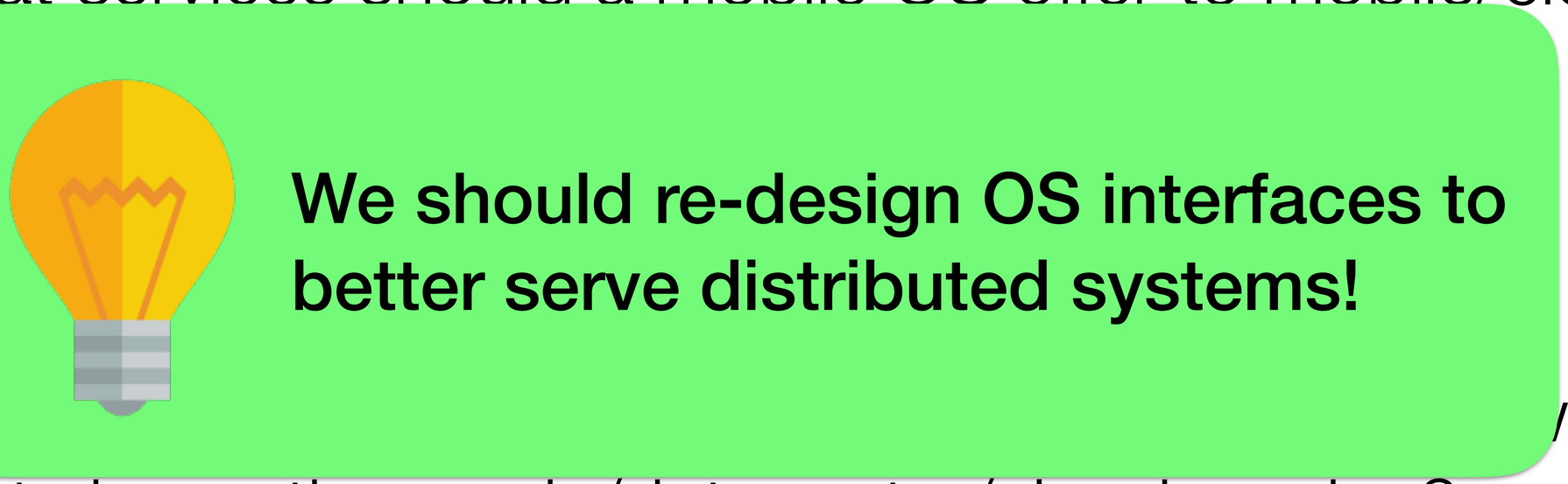
TAPIR [SOSP '15]

Looking down the stack …

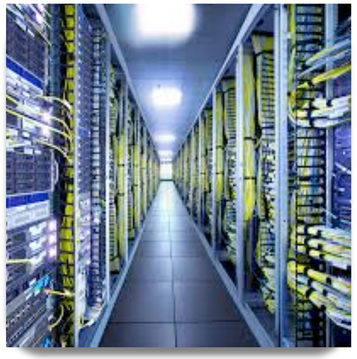


Modern systems need to cooperate across layers and distributed nodes.

- What services should a mobile OS offer to mobile/cloud sy
- H
m
- A ways exists in another node/datacenter/cloud service?



User devices will see more fundamental changes



Cloud

2000s

Mobile

2010s

Internet of Things

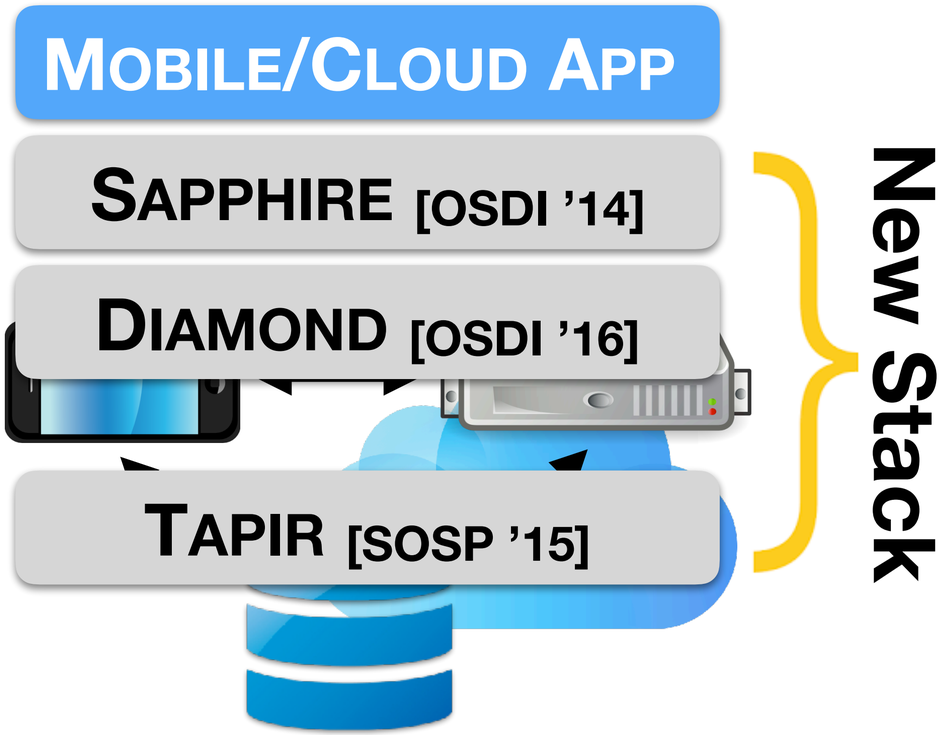
2015s

Home Assistants

2020s

Their apps will have new challenges that need new systems!

Summary



I design and build general-purpose systems to help programmers tackle new application challenges.

https://github.com/iyzhang