Frameworks for Automation

Abhishek Modi, Harshit Agarwal, Evan Fabry

Slicer: Auto-Sharding for Datacenter Applications

Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, Kfir Lev-Ari (Technion – Israel)

Google

Some slides and figures inspired by the original presentation

The Case for Slicer

Building a DNS Service

End-user devices

DNS Service

Virtual Machines

Cloud Platform

Issues with building a Scalable Service

Can replicate state on each server

However this is hard to scale when state is large.

Stateless services are Great

Issues with building a Scalable Service

Can replicate state on each server

However this is hard to scale when state is large.

Can use a database

However this affects availability

Issues with building a Scalable Service

Can replicate state on each server

However this is hard to scale when state is large.

Can use a database

However this affects availability

Can do static sharding

But this is not fault tolerant

Issues with building a Scalable Service

Can replicate state on each server

However this is hard to scale when state is large.

Can use a database

However this affects availability

Can do static sharding

But this is not fault tolerant

Can use consistent hashing for sharding

At this point we're just building infra. Overhead for building a service is too large

Stateful services seem inviable.

Where Slicer Comes In

The infrastructure for scaling a service is similar to that of a datastore.

Slicer is essentially refactored datastore infra.

Should provide high quality sharding with the consistency of a centralized system.

Should have low latency and high availability.

MAKING STATEFUL SERVICES PRACTICAL

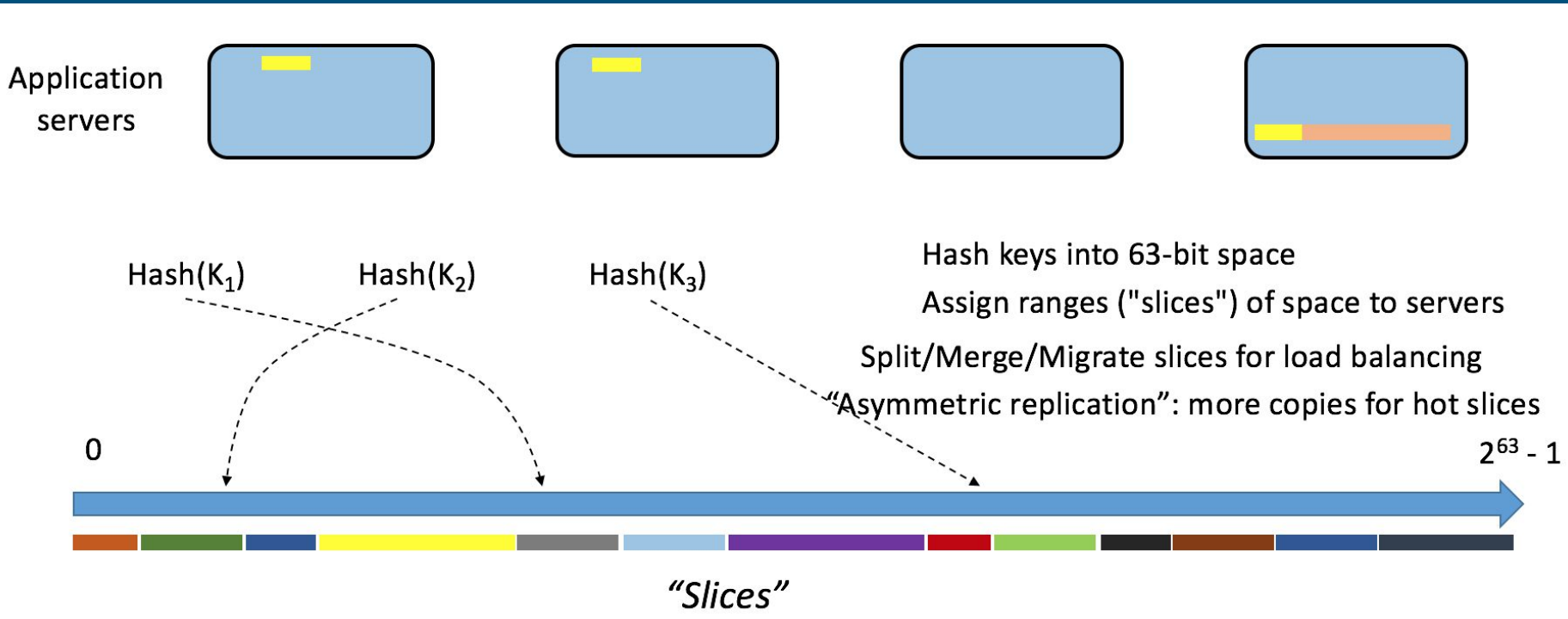
Slicer Sharding Model

Hash keys into 63 bit space.

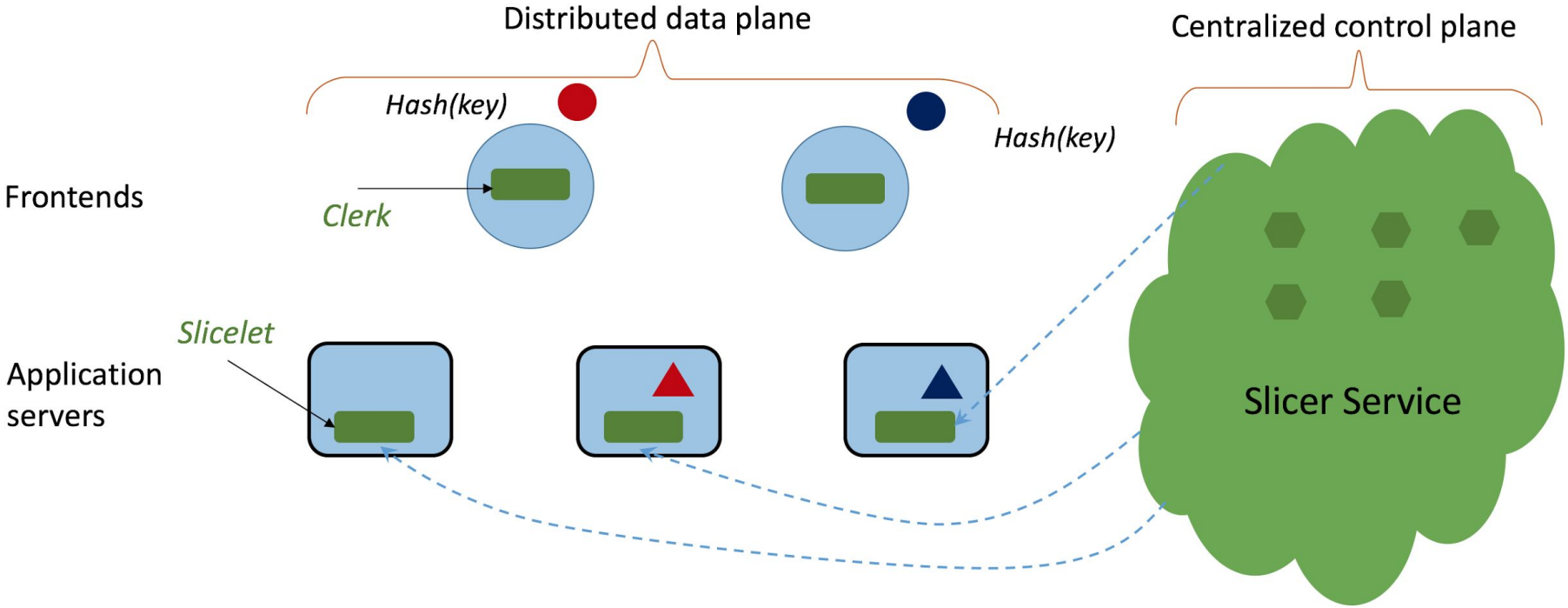
Assign ranges of this space to servers.

Can split/merge/migrate slices for load balancing.

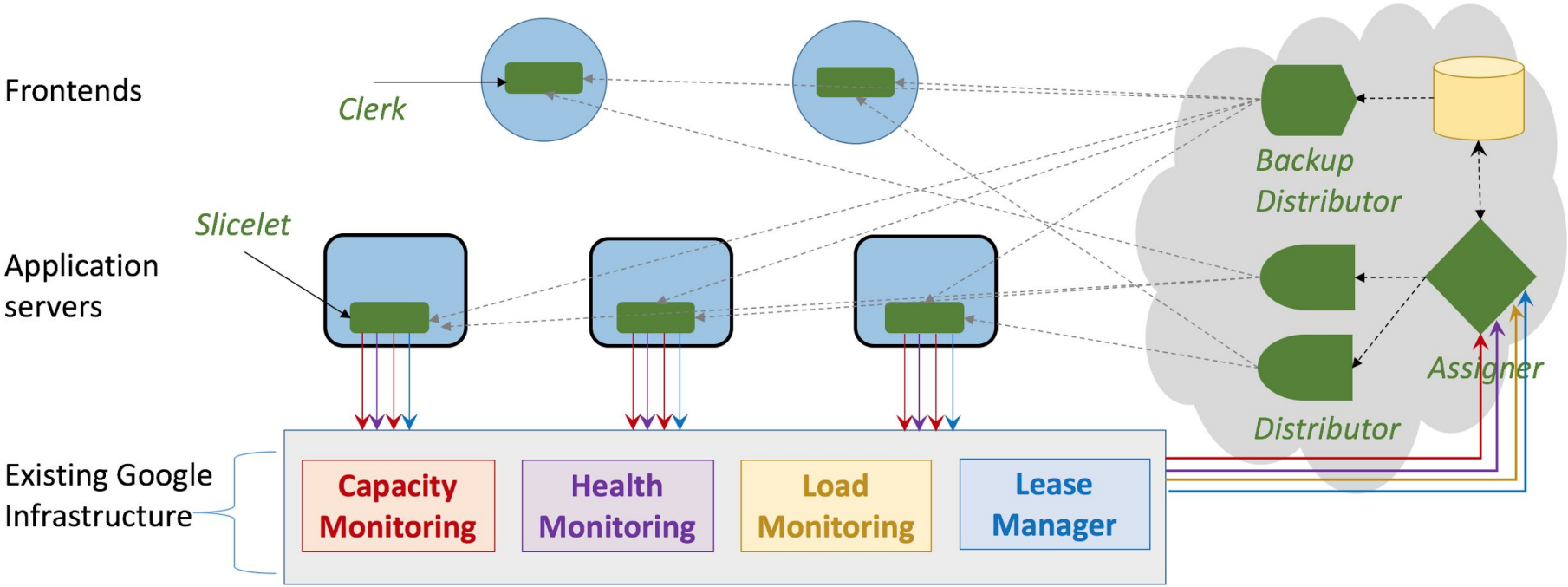
“Asymmetric replication” to deal with hot slices


Application servers
Hash(K_1)
Hash(K_2)
Hash(K_3)
Hash keys into 63-bit space
Assign ranges ("slices") of space to servers
Split/Merge/Migrate slices for load balancing
"Asymmetric replication": more copies for hot slices
0
$2^{63} - 1$
"Slices"

Slicer Overview



Slicer Architecture



Fault Tolerance

Localized failures

Each component of the architecture can live on separate datacenters.
Any distributor/assigner can be tasked to handle any job

Correlated failures

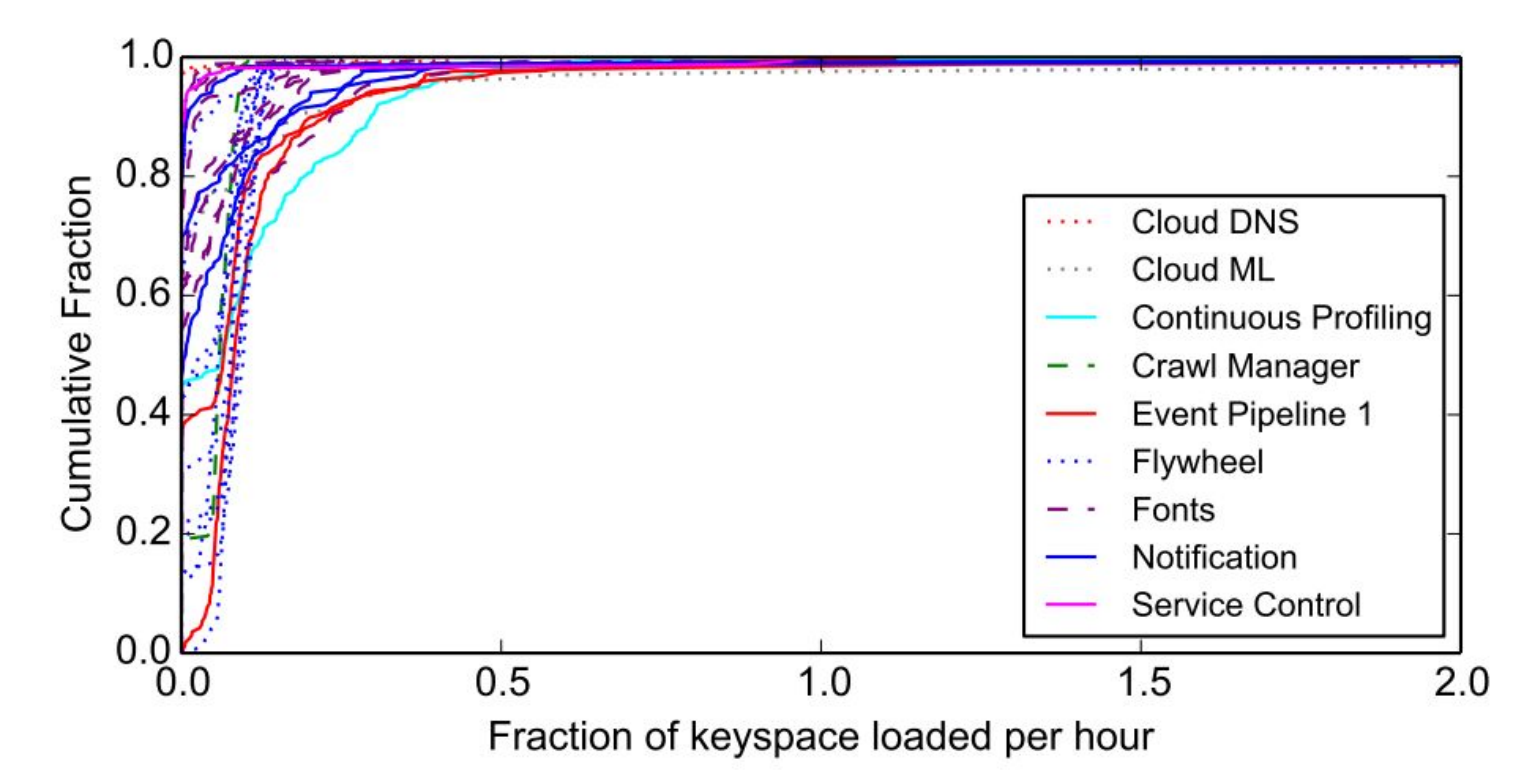
Alive servers have already cached the assignments. Can still serve for a while.
Backup Distributors essentially keep this cache alive for longer.
The system won’t recover on its own, but it helps buy time.

Evaluation

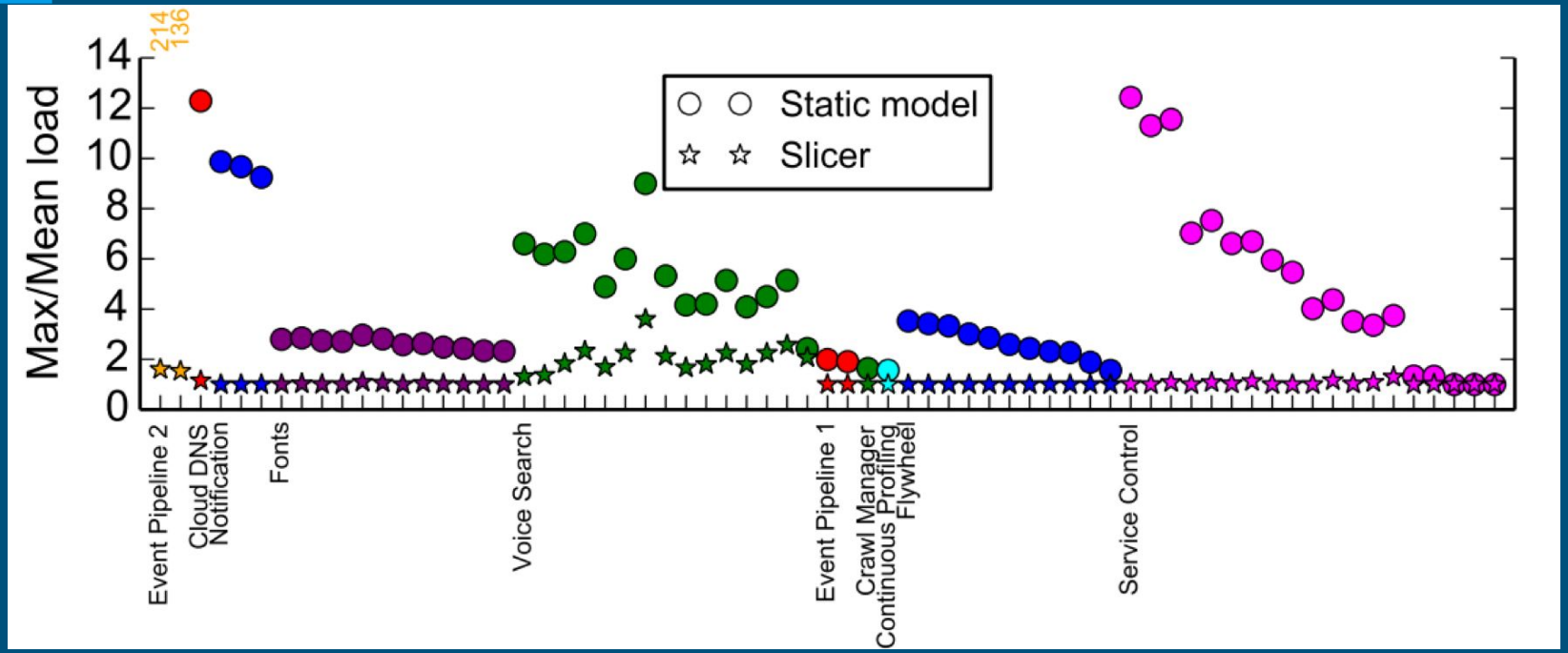
Used Across > 20 services

Availability: Integration with Stubby: 99.98% allocation (pessimistic)
Stubby uses local decisions.

Evaluation: Load Balancing



Evaluation: Load Balancing



Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications

Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, Henry M. Levy
Presented by: Harshit Agarwal

Some slides and figures inspired by the original presentation

Reactive Applications

Frameworks for Automation

Abhishek Modi, Harshit Agarwal, Evan Fabr

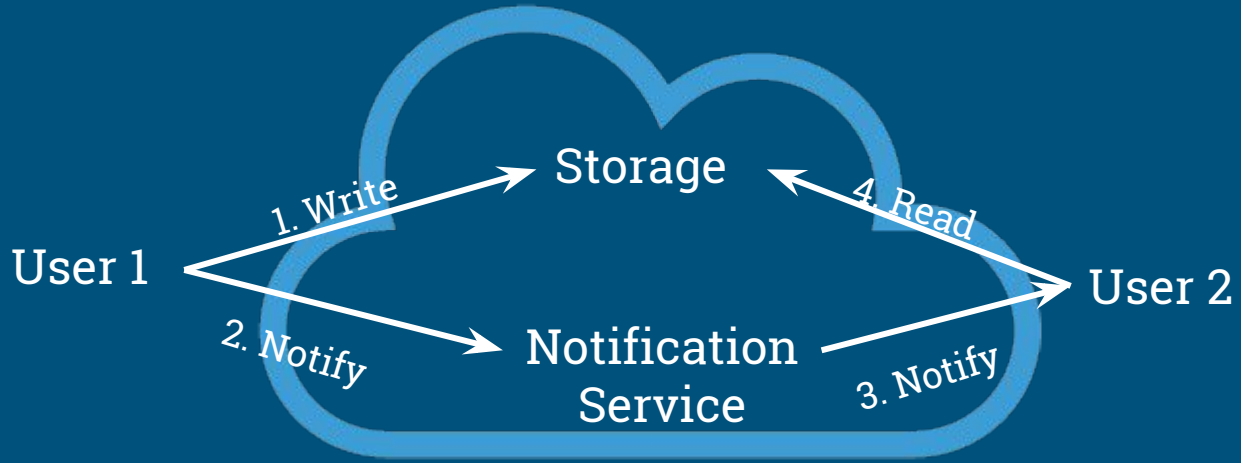
Abhishek Modi

Reactive Applications

Automatically propagate updates across mobile devices and the cloud

Reactive Applications - Challenges

Automatic propagation of updates is challenging for application programmers



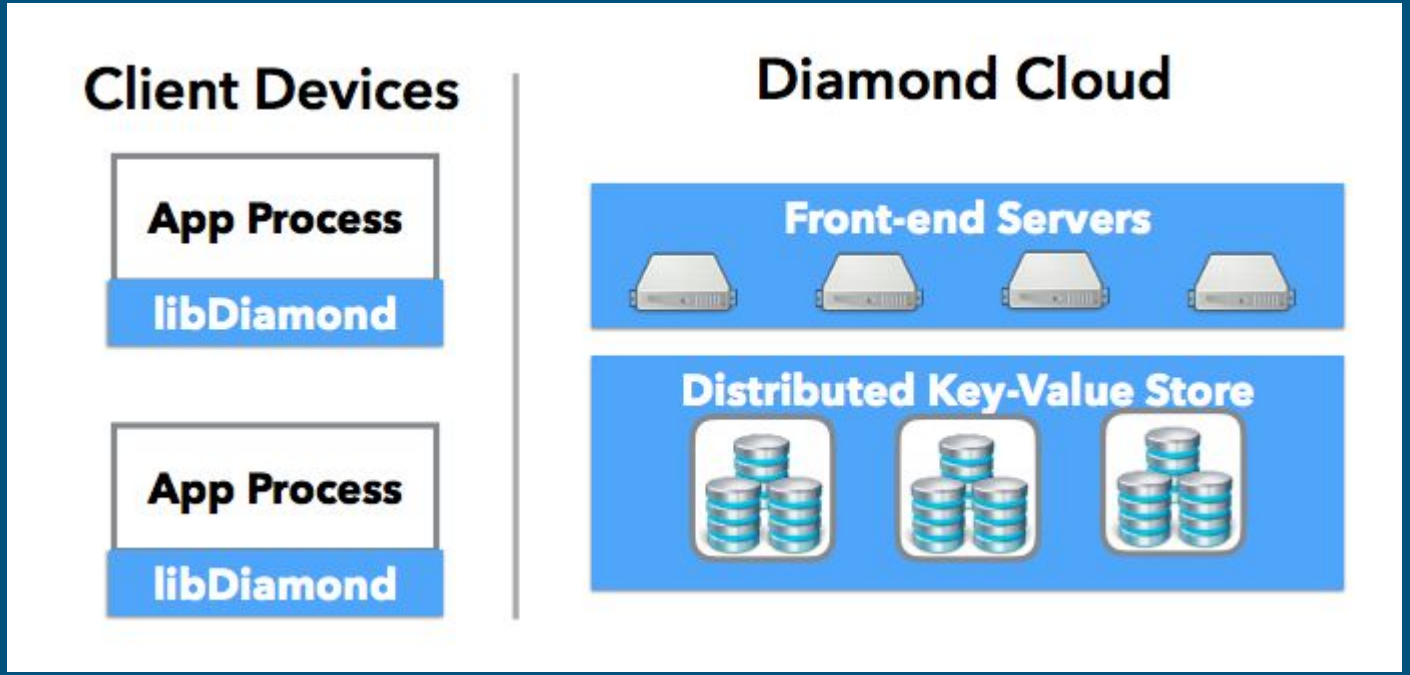
Reactive applications require end-to-end data management with strong guarantees

Diamond

First reactive data management service

- 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

Diamond System Model



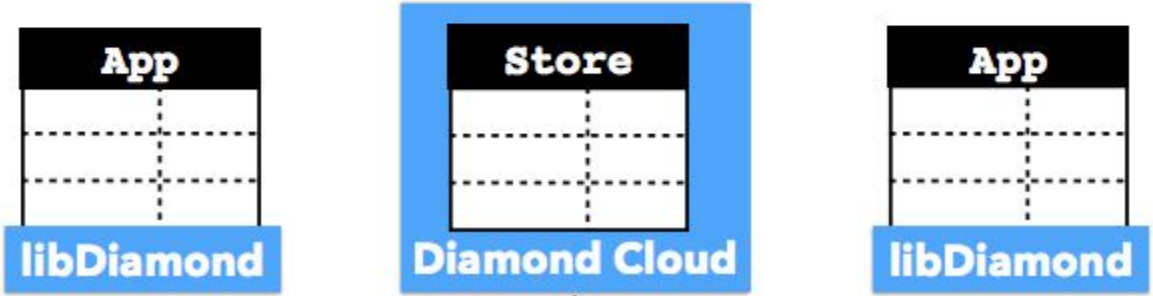
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Read-write transactions to update shared RDTs.

Reactive Data Map (rmap)
Binding between RDTs in the app and the Diamond store

Reactive Transactions
Read-only transactions that re-execute app code when the read set updates.



Reactive Data Types (RDT)

Shared, persistent data structures

- Simple data structures - enable primitives, collections and more complicated data types (boolean, lists, int, long, etc)
- Data type semantics avoid false sharing and enable commutative operations
- Defined in libDiamond language bindings

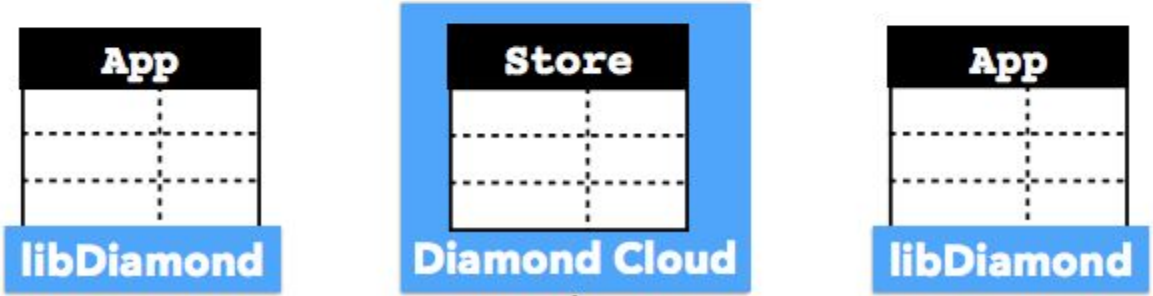
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Read-write transactions to update shared RDTs.

Reactive Data Map (rmap)
Binding between RDTs in the app and the Diamond store

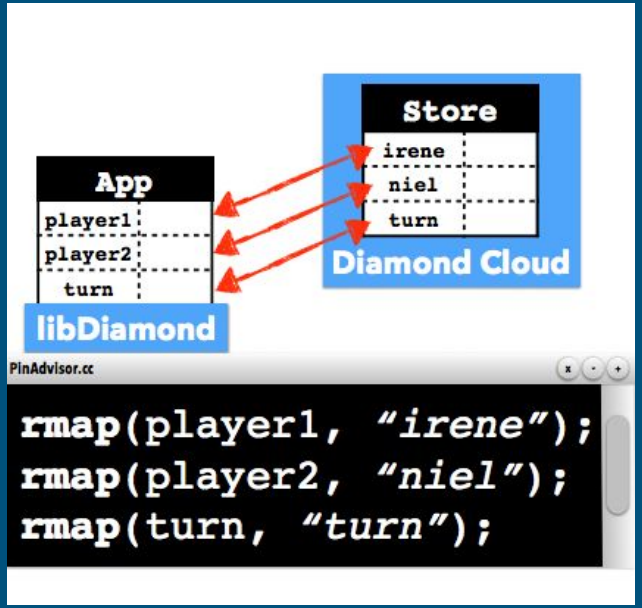
Reactive Transactions
Read-only transactions that re-execute app code when the read set updates.



Reactive Data Map (rmap)

Binding between RDTs in the app and keys in the Diamond store

- Key abstraction for flexible, shared member
- Gives apps control over data shared
- Enables automatic availability, fault tolerance and consistency to RDTs



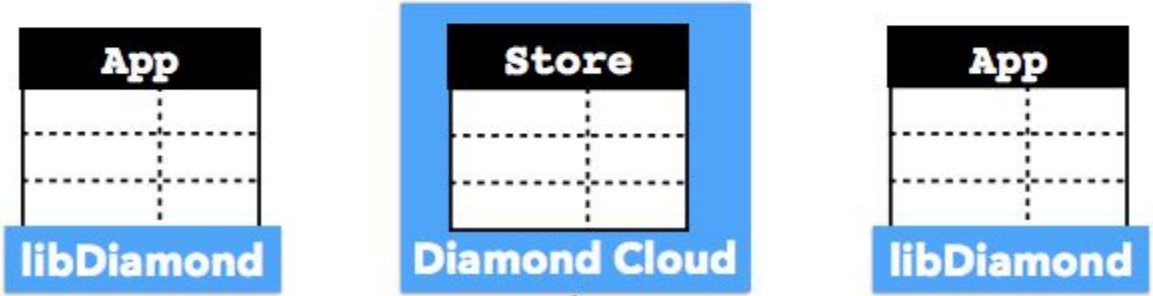
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Reactive Data Map (rmap)
Binding between RDTs in the app and the Diamond store

Read-write Transactions
Read-write transactions to update shared RDTs.

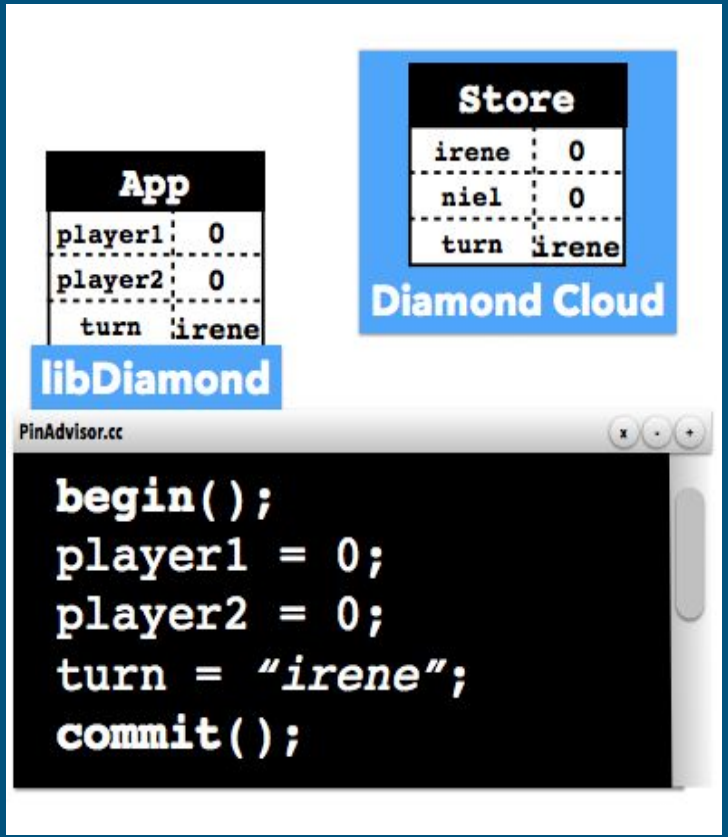
Reactive Transactions
Read-only transactions that re-execute app code when the read set updates.



Read-write Transactions

Read-write transactions to update shared RDTs.

- Execute application code to update rmapped RDTs
- Gives application creators ability to choose when data sync takes place
- Ensures safe concurrent access to shared data



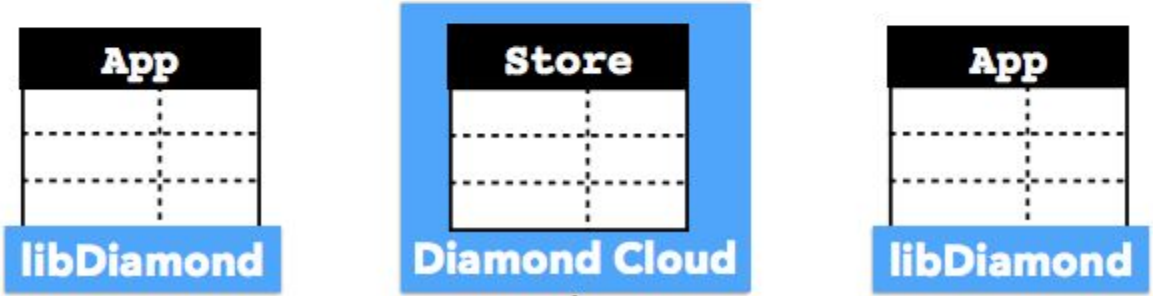
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Read-write transactions to update shared RDTs.

Reactive Data Map (rmap)
Binding between RDTs in the app and the Diamond store

Reactive Transactions
Read-only transactions that re-execute app code when the read set updates.



Reactive Transactions

Read-only transactions that 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 data synced
- Automatically triggers app code, responding to read-write transactions to shared RDTs

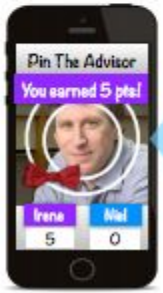
Diamond Programming Model

Reactive Data Types (RDTs)
Shared, persistent data structures

Read-write Transactions
Read-write transactions to update shared RDTs.

Reactive Data Map (rmap)
Binding between RDTs in the app and the Diamond store

Reactive Transactions
Read-only transactions that re-execute app code when the read set updates.



Automated end-to-end data management and storage with fault-tolerance, availability and consistency

Diamond ACID+R Guarantees

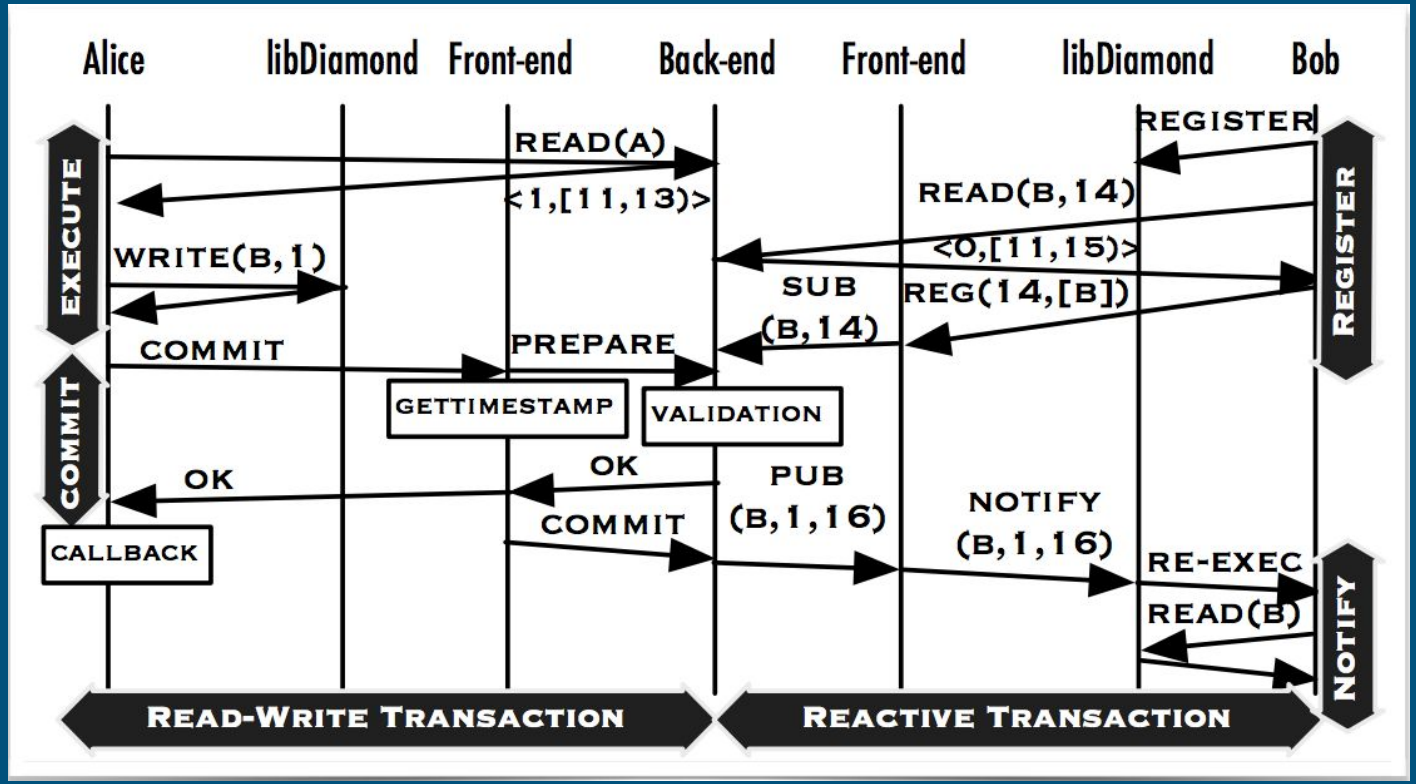
Diamond ACID+R Guarantees

- **Atomicity:** All or no updates to shared data in a read-write transaction complete
- **Consistency:** All accesses in a transaction (read-write or reactive) reflect a single, point-in-time view of shared data
- **Isolation**: All transactions reflect a serial execution order over shared data
- **Durability:** All updates in committed transactions are never lost
- **Reactivity:** All accesses in reactive transactions will eventually reflect the latest updates.

Diamond Isolation Levels

Read-write Isolation Level	Reactive Isolation Level
Strict Serializability	Serializable Snapshot
Snapshot Isolation	Serializable Snapshot
Read Committed	Read Committed

Diamond Implementation - Transaction Protocol

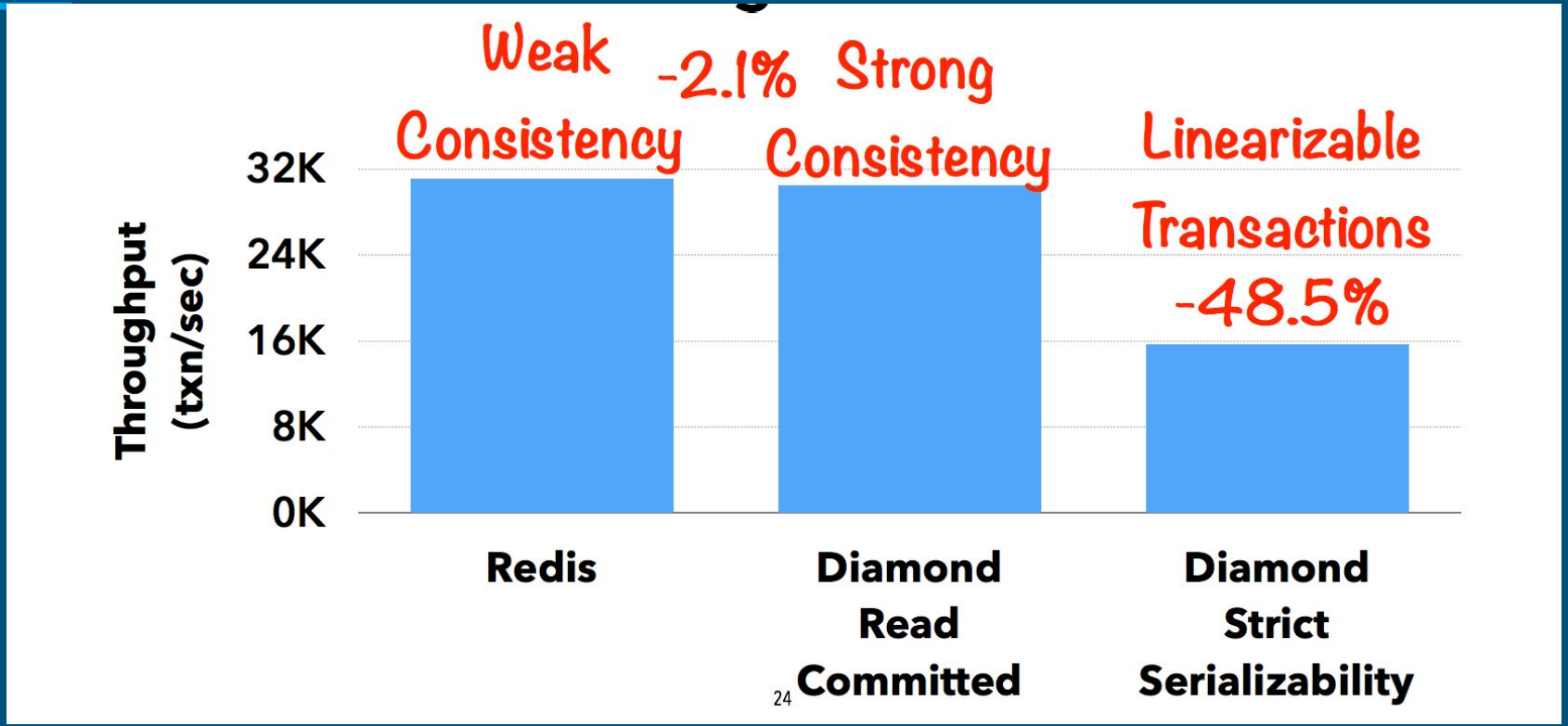


Evaluation

Reduces Complexity, Improves Guarantees

Application	Original LoC	Diamond LoC	%Saved
Multi-player Game	46	34	26%
Chat Room	335	225	33%
Scrabble Clone	8729	7603	13%
Twitter Clone	14278	12554	13%

Low Data Management Overhead



Conclusion

- Reactive applications are complicated to build
- Diamond makes it easier by giving end to end data management and storage and provides strong transactional ACID+R guarantees
- Diamond simplifies reactive apps, with low overhead

Related Work

- Distributed Programming Frameworks : Meteor, Parse, Firebase, Mjolnir, Mapjax, RethinkDB
- Client-side Programming Frameworks : React, Angular, Blaze, ReactiveX
- Distributed Storage Systems : Redis, MongoDB, Dropbox
- Notification/Pub-Sub/Streaming Services : Thialfi, Apache Kafka, Amazon Kinesis

Questions (Diamond):

What are the needs of a similar system that attempts to extend offline support?

For what sort of applications is this possible and when is it best just to evict unresponsive members?

Comment (Diamond):

At a higher level than ACID+R, the core problem that Diamond attempts to solve is computation in a user-driven system (non-replicable nodes) where nodes depend upon each other

Questions (Diamond):

Is there a class of applications for which harder constraints on responsiveness might be necessary?

Is this possible when some nodes are end users, hence non-replicable?

Questions (Diamond):

How is Diamond different from triggers that exist in other database systems?

Is there any way to rollback updates made to RDTs?

Questions (Slicer):

The Backup Distributor is simple, it just uses trivial static sharding with stale information.

This is robust, but is there a better approach?

Questions (Slicer):

Under limitations, they mention that task heterogeneity is somewhat managed by CPU balancing but RAM usage is not balanced.

Also, differences in amount of time to complete different requests are not considered.

What are some challenges of implementing the above?

Questions (Slicer):

This system uses 5min windows to assess CPU load.

This results in somewhat slow response to load changes.

In practice, some window like this is needed to smooth smaller peaks in load, however is there a reason that we can’t do better?