# Tango

Emmett Witchel

CS380L

# Faux Quiz Questions

- Compare/contrast Tango against LFS
- Compare/contrast Tango against TxOS
- Compare/contrast Tango against Spark/DryadLINQ
- How are streams used in Tango?
- Why do holes arise in a tango log? How does the system deal with them?
- How do streams complicate cross-object transactions in Tango?
- How does Tango's commit protocol differ from a traditional protocol like 2PC?
- Compare/contrast fault-tolerance techniques in Tango and Spark

# Tango: distributed data structures over a shared log

*Mahesh Balakrishnan*, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran
Michael Wei, John D. Davis, Sriram Rao, Tao Zou, Aviad Zuck

Microsoft Research

# big **meta**data

- design pattern: distribute data, *centralize metadata*
- schedulers, allocators, coordinators, namespaces, indices (e.g. HDFS namenode, SDN controller…)

# big **meta**data

- design pattern: distribute data, *centralize metadata*
- schedulers, allocators, coordinators, namespaces, indices (e.g. HDFS namenode, SDN controller…)
- usual plan: harden centralized service later

"***Coordinator failures will be handled safely*** using the ZooKeeper service [14]." Fast Crash Recovery in RAMCloud, Ongaro et al., SOSP 2011.

"***Efforts are also underway to address high availability*** of a YARN cluster by having passive/active failover of RM to a standby node." Apache Hadoop YARN: Yet Another Resource Negotiator, Vavilapalli et al., SOCC 2013.

"However, ***adequate resilience can be achieved*** by applying standard replication techniques to the decision element." NOX: Towards an Operating System for Networks, Gude et al., Sigcomm CCR 2008.

# big **meta**data

- design pattern: distribute data, *centralize metadata*
- schedulers, allocators, coordinators, namespaces, indices (e.g. HDFS namenode, SDN controller…)
- usual plan: harden centralized service later

"***Coordinator failures will be handled safely*** using the ZooKeeper service [14]." Fast Crash Recovery in RAMCloud, Ongaro et al., SOSP 2011.

"***Efforts are also underway to address high availability*** of a YARN cluster by having passive/active failover of RM to a standby node." Apache Hadoop YARN: Yet Another Resource Negotiator, Vavilapalli et al., SOCC 2013.

"However, ***adequate resilience can be achieved*** by applying standard replication techniques to the decision element." NOX: Towards an Operating System for Networks, Gude et al., Sigcomm CCR 2008.

- … but hardening is difficult!

# the abstraction gap for metadata

centralized metadata services are built using in-memory data structures (e.g. Java / C# Collections)

- state resides in maps, trees, queues, counters, graphs…
- transactional access to data structures
    - example: a scheduler atomically moves a node from a free list to an allocation map

# the abstraction gap for metadata

centralized metadata services are built using in-memory data structures (e.g. Java / C# Collections)

- state resides in maps, trees, queues, counters, graphs…
- transactional access to data structures
  - example: a scheduler atomically moves a node from a free list to an allocation map

adding high availability requires different abstractions

- move state to external service like ZooKeeper
- restructure code to use state machine replication
- implement custom replication protocols

# the Tango abstraction

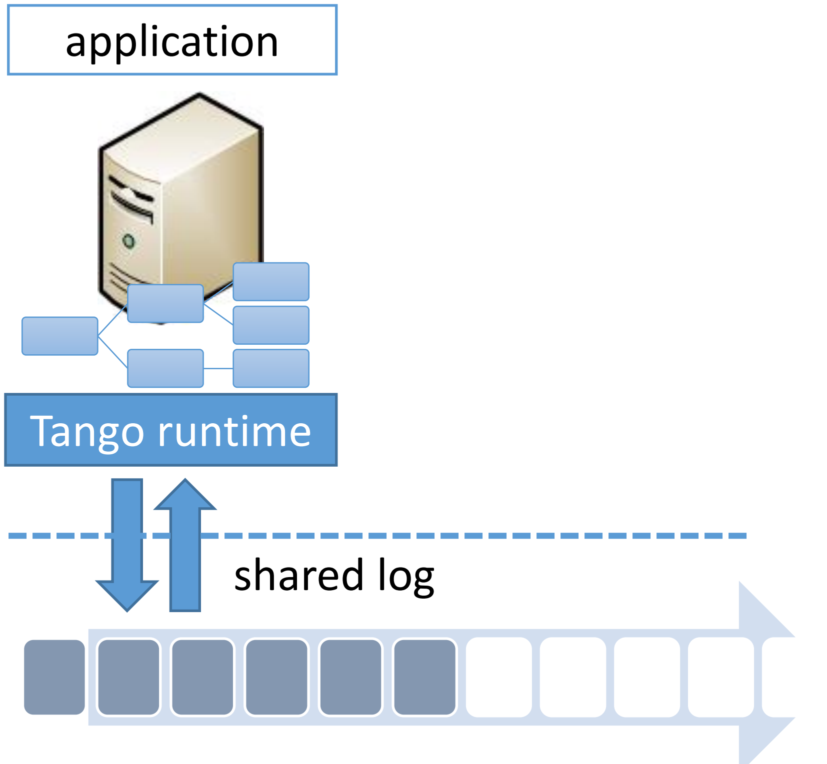
a Tango object

**=**

**view**
in-memory
data structure

**+**

**history**
ordered
updates in
shared log



# the Tango abstraction

a Tango object

**=**

**view**

in-memory

data structure

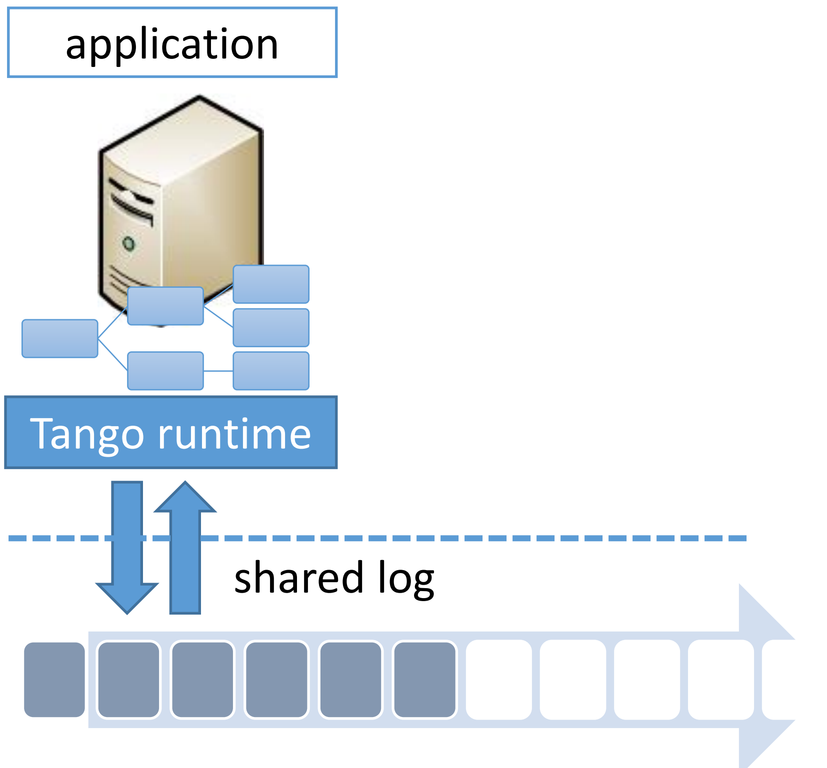
**+**

**history**

ordered

updates in

shared log




the shared log is the source of

- persistence

# the Tango abstraction

a Tango object

**=**

**view**

in-memory

data structure

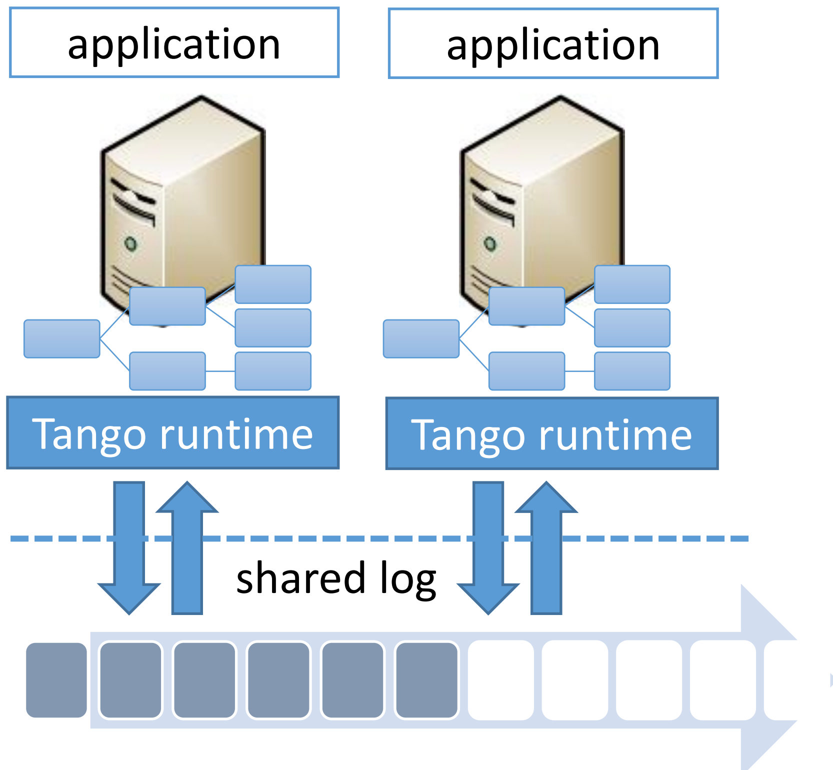
**+**

**history**

ordered

updates in

shared log




the shared log is the source of

- persistence
- availability

# the Tango abstraction

a Tango object

**=**

**view**

in-memory

data structure

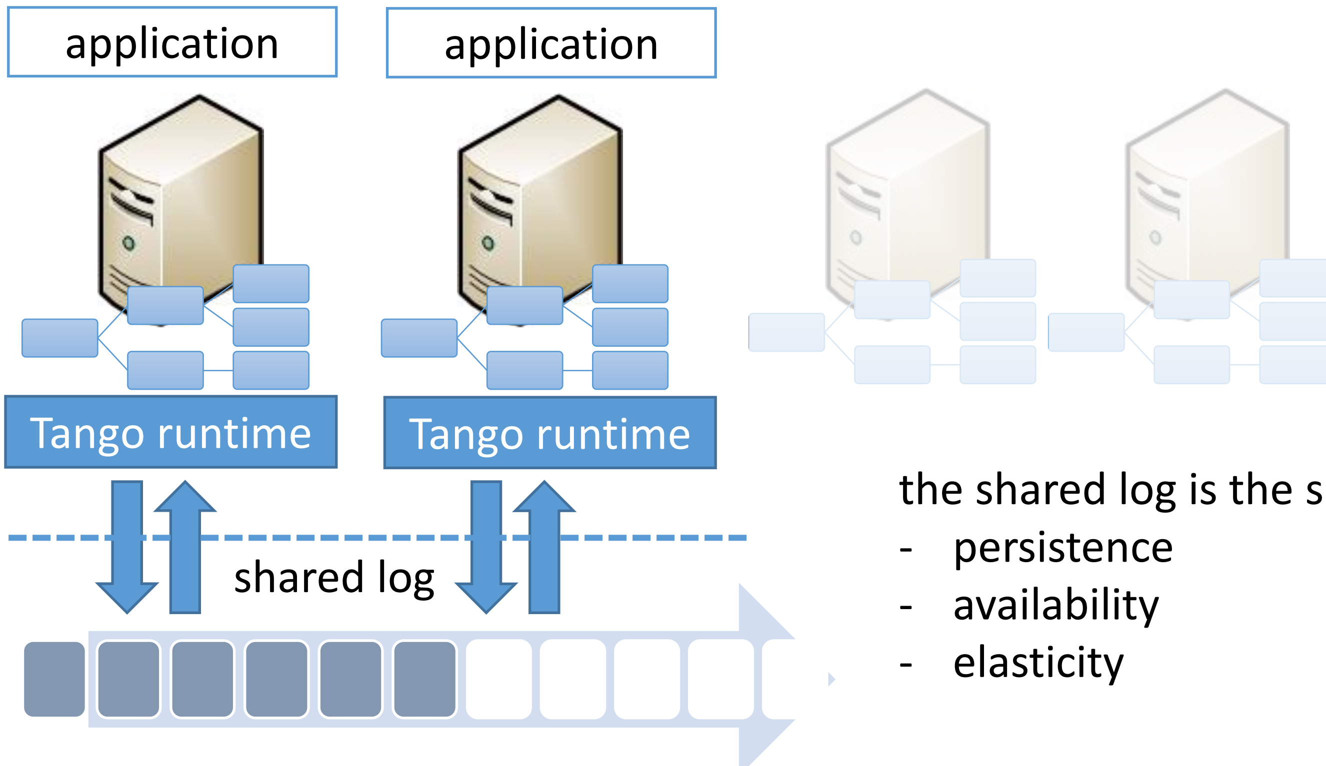
**+**

**history**

ordered

updates in

shared log

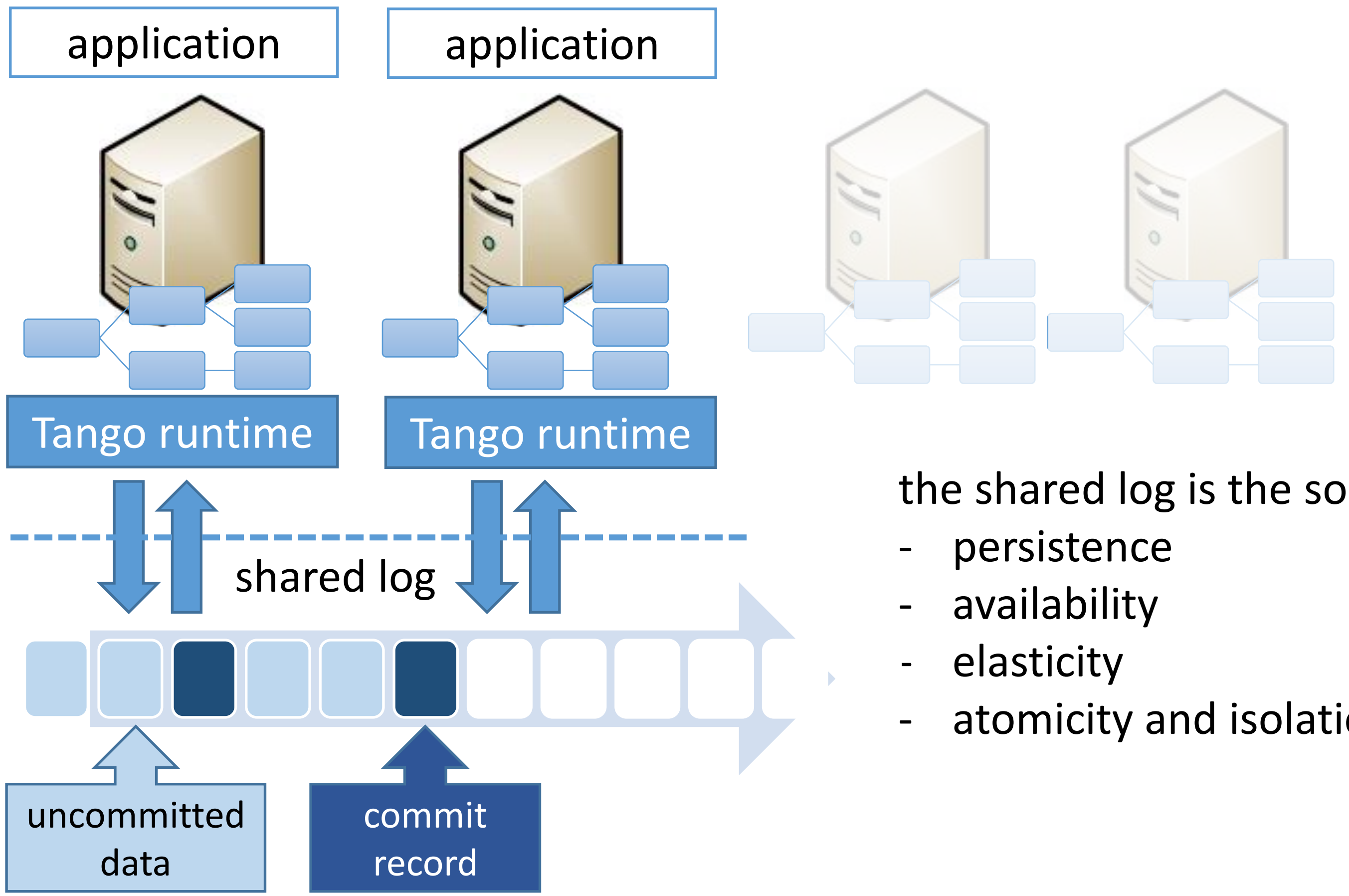



the shared log is the source of

- persistence
- availability
- elasticity

# the Tango abstraction

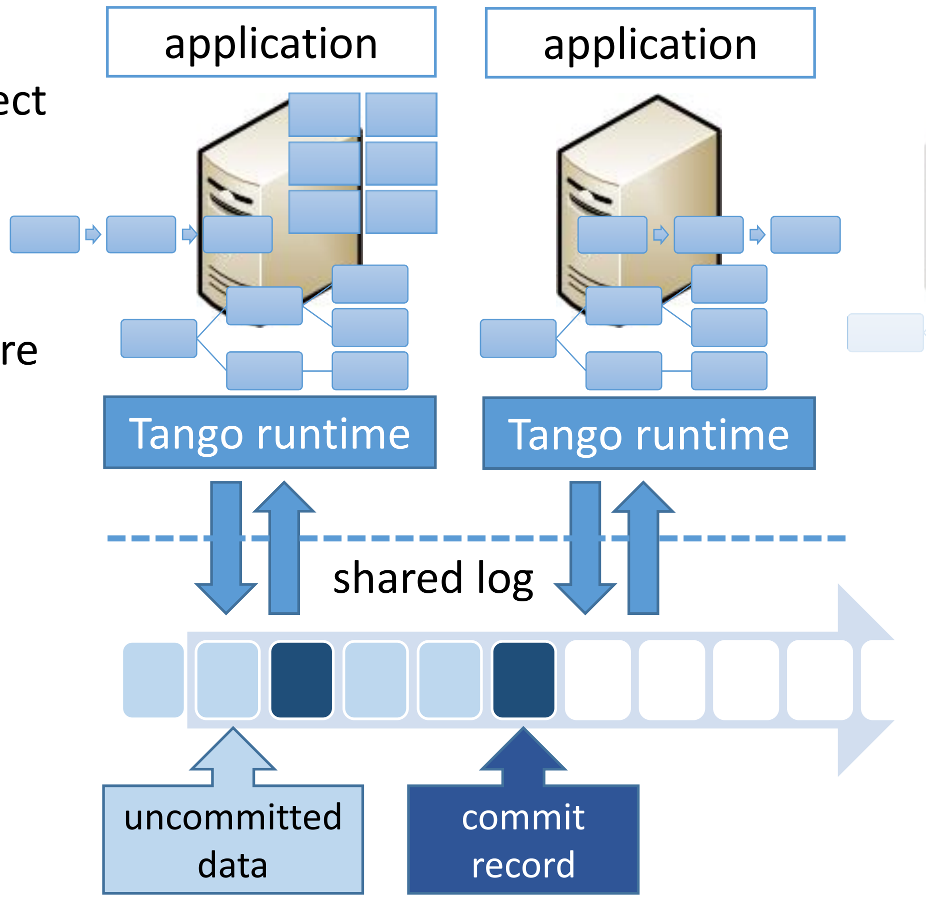
a Tango object
=
**view**
in-memory
data structure
**+**
**history**
ordered
updates in
shared log




the shared log is the source of
- persistence
- availability
- elasticity
- atomicity and isolation

# the Tango abstraction

a Tango object
**=**
**view**
in-memory
data structure
**+**
**history**
ordered
updates in
shared log




the shared log is the source of
- persistence
- availability
- elasticity
- atomicity and isolation
  … across multiple objects

# the Tango abstraction

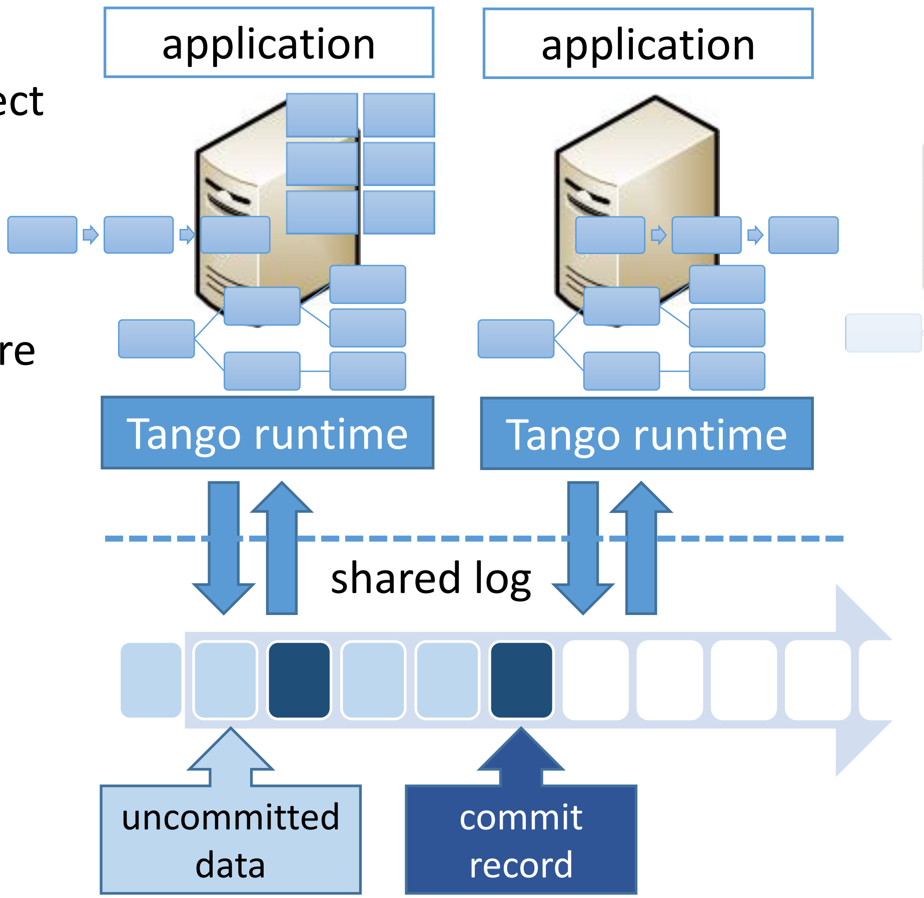
a Tango object

**=**

**view**
in-memory
data structure

**+**

**history**
ordered
updates in
shared log



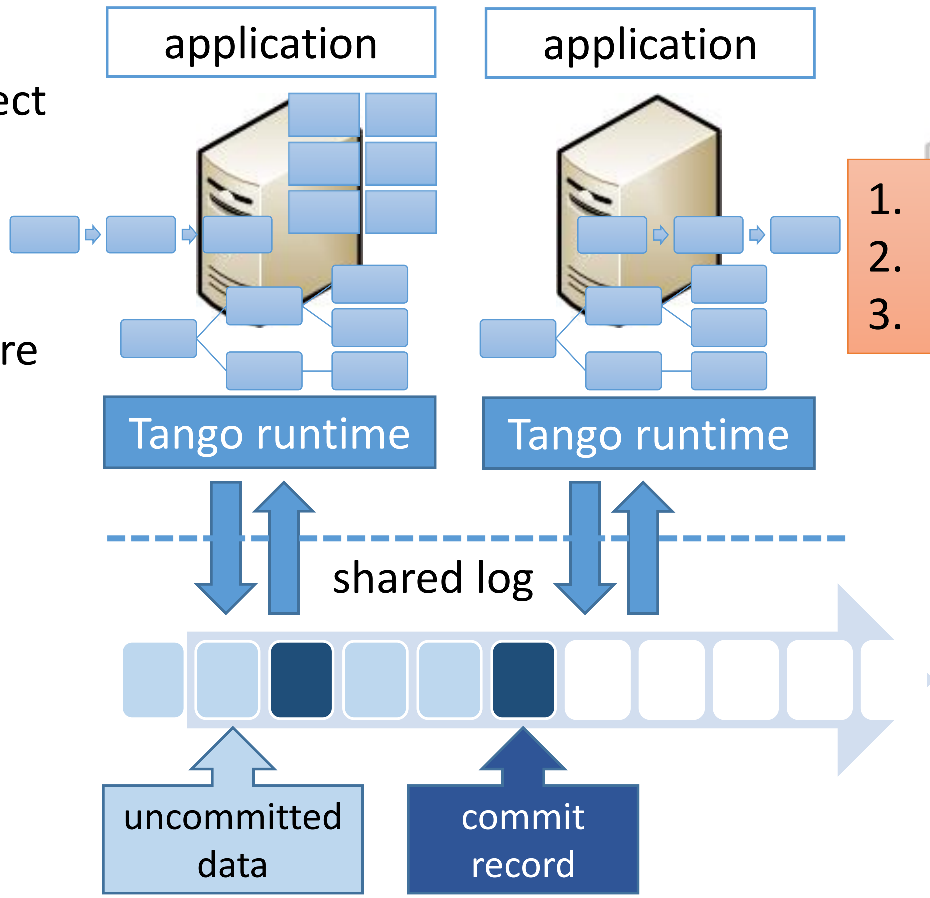

the shared log is the source of

- persistence
- availability
- elasticity
- atomicity and isolation
  … across multiple objects

no messages… only appends/reads on the shared log!

# the Tango abstraction

a Tango object
**=**
**view**
in-memory
data structure
**+**
**history**
ordered
updates in
shared log




1. Tango objects are **easy to use**
2. Tango objects are **easy to build**
3. Tango objects are **fast and scalable**

the shared log is the source of
- persistence
- availability
- elasticity
- atomicity and isolation
  … across multiple objects

no messages… only appends/reads on the shared log!

# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations

**example:**

```
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
```

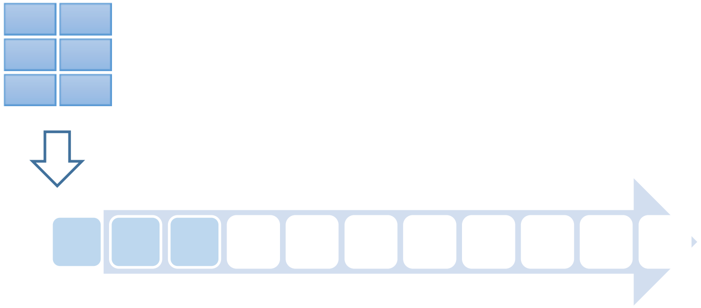
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations

**example:**

```
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
```

**under the hood:**



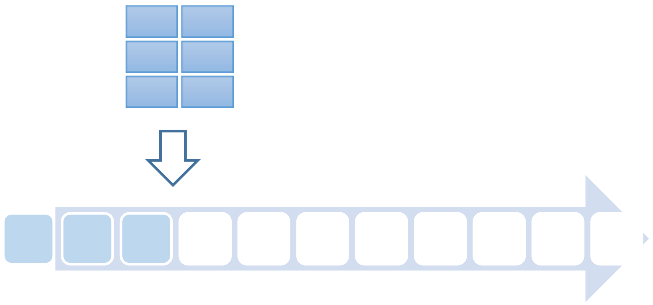
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations

**example:**

```
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
```

**under the hood:**



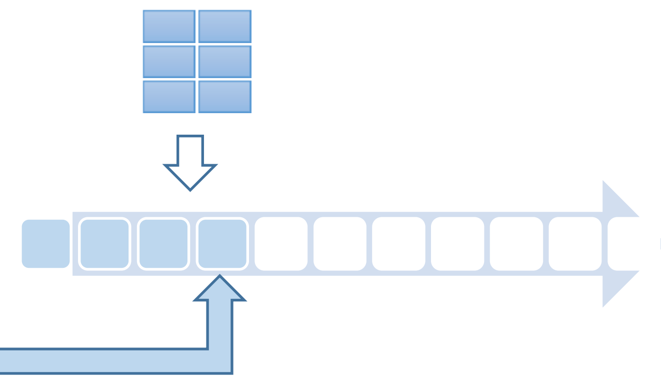
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations

**example:**

```
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
```

**under the hood:**



# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

```
TR.BeginTX();
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
status = TR.EndTX();
```

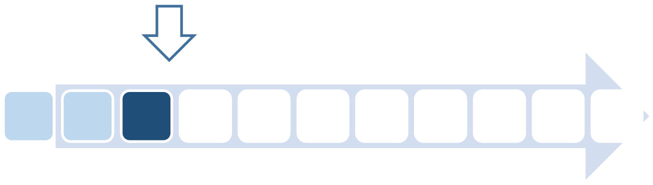
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

TR.**BeginTX();**
curowner = *ownermap*.get("ledger");
if(curowner.equals(myname))
    *ledger*.add(item);
status = TR.**EndTX();**

**under the hood:**



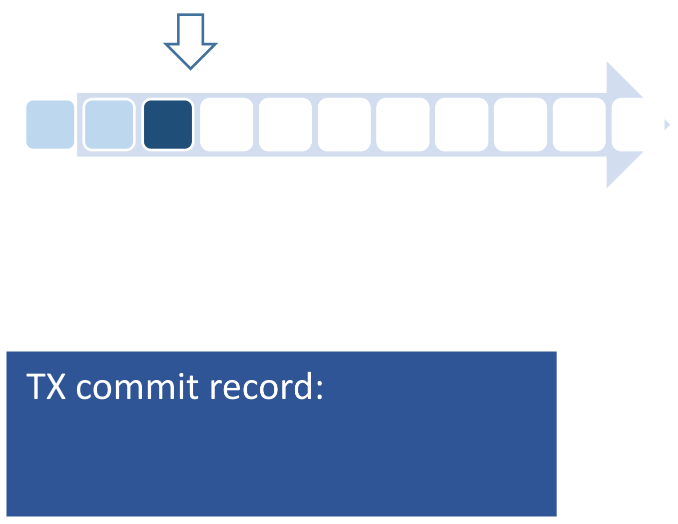
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

```
TR.BeginTX();
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
status = TR.EndTX();
```

**under the hood:**



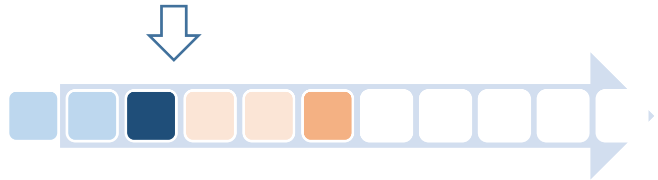
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

```
TR.BeginTX();
curowner = ownermap.get(“ledger”);
if(curowner.equals(myname))
        ledger.add(item);
status = TR.EndTX();
```

**under the hood:**



TX commit record:

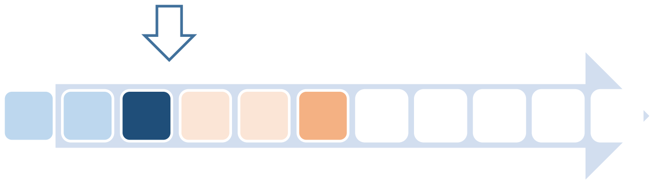
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

TR.**BeginTX();**
curowner = *ownermap*.get("ledger");
if(curowner.equals(myname))
*ledger*.add(item);
status = TR.**EndTX();**

**under the hood:**



TX commit record:
read-set: (ownermap, ver:2)

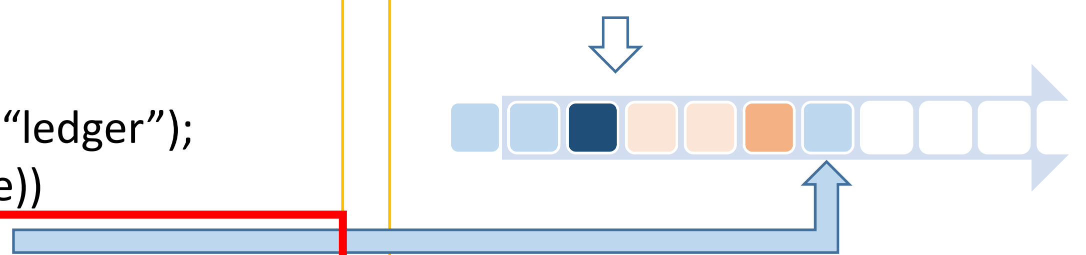
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

TR.**BeginTX();**
curowner = *ownermap*.get("ledger");
if(curowner.equals(myname))
        *ledger*.add(item);
status = TR.**EndTX();**

**under the hood:**



TX commit record:
read-set: (ownermap, ver:2)
write-set: (ledger, ver:6)

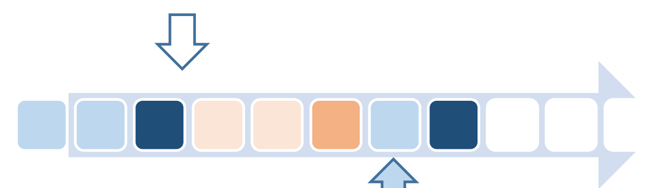
# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

```
TR.BeginTX();
curowner = ownermap.get("ledger");
if(curowner.equals(myname))
        ledger.add(item);
status = TR.EndTX();
```

**under the hood:**



TX commit record:
read-set: (ownermap, ver:2)
write-set: (ledger, ver:6)

# Tango objects are easy to use

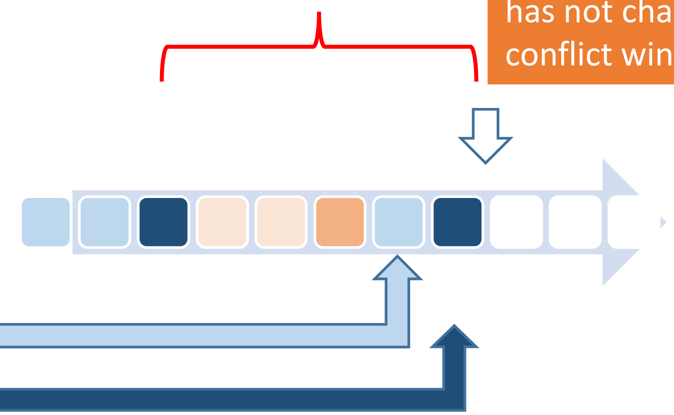
- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

**example:**

TR.**BeginTX();**
curowner = *ownermap*.get("ledger");
if(curowner.equals(myname))
    *ledger*.add(item);
status = TR.**EndTX();**

**under the hood:**

TX commits if read-set (*ownermap*) has not changed in conflict window



TX commit record:
read-set: (ownermap, ver:2)
write-set: (ledger, ver:6)

# Tango objects are easy to use

- implement standard interfaces (Java/C# Collections)
- linearizability for single operations
- serializable transactions

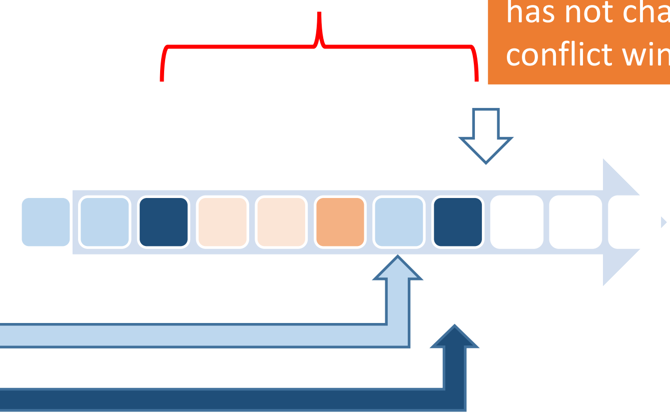
**example:**

TR.**BeginTX();**
curowner = *ownermap*.get("ledger");
if(curowner.equals(myname))
        *ledger*.add(item);
status = TR.**EndTX();**

speculative commit records: each client decides if the TX commits or aborts **independently but deterministically**
[similar to Hyder (Bernstein et al., CIDR 2011)]

**under the hood:**

TX commits if read-set (*ownermap*) has not changed in conflict window



TX commit record:
read-set: (ownermap, ver:2)
write-set: (ledger, ver:6)

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

```
class TangoRegister {
        int oid;
        TangoRuntime *T;
        int state;
        void apply(void *X) {
                state = *(int *)X;
        }
        void writeRegister (int newstate) {
                T->update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T->query_helper(oid);
                return state;
        }
}
```

simple API exposed by runtime to object: 1 upcall + two helper methods
arbitrary API exposed by object to application: mutators and accessors

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

```
class TangoRegister {
        int oid;
        TangoRuntime ∗T;
        int state;
        void apply(void ∗X) {
                state = ∗(int ∗)X;
        }
        void writeRegister (int newstate) {
                T−>update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T−>query_helper(oid);
                return state;
        }
}
```

object-specific state

simple API exposed by runtime to object: 1 upcall + two helper methods
arbitrary API exposed by object to application: mutators and accessors

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

```
class TangoRegister {
        int oid;
        TangoRuntime ∗T;
        int state;
        void apply(void ∗X) {
                state = ∗(int ∗)X;
        }
        void writeRegister (int newstate) {
                T−>update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T−>query_helper(oid);
                return state;
        }
}
```

invoked by Tango runtime on EndTX to change state

simple API exposed by runtime to object: 1 upcall + two helper methods

arbitrary API exposed by object to application: mutators and accessors

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

```
class TangoRegister {
        int oid;
        TangoRuntime *T;
        int state;
        void apply(void *X) {
                state = *(int *)X;
        }
        void writeRegister (int newstate) {
                T->update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T->query_helper(oid);
                return state;
        }
}
```

mutator: updates TX write-set, appends to shared log

simple API exposed by runtime to object: 1 upcall + two helper methods

arbitrary API exposed by object to application: mutators and accessors

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

```
class TangoRegister {
        int oid;
        TangoRuntime ∗T;
        int state;
        void apply(void ∗X) {
                state = ∗(int ∗)X;
        }
        void writeRegister (int newstate) {
                T−>update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T−>query_helper(oid);
                return state;
        }
}
```

accessor: updates TX read-set, returns local state

simple API exposed by runtime to object: 1 upcall + two helper methods
arbitrary API exposed by object to application: mutators and accessors

# Tango objects are easy to build

15 LOC == persistent, highly available, transactional register

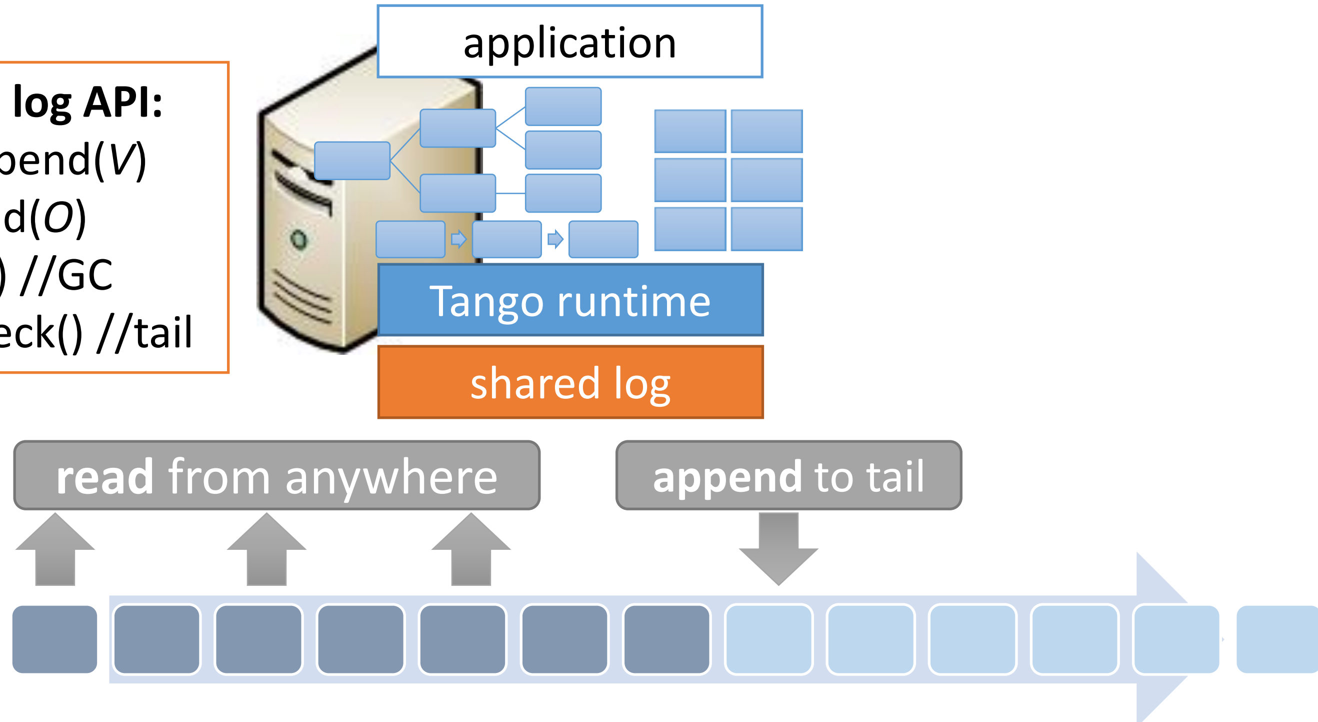
```
class TangoRegister {
        int oid;
        TangoRuntime ∗T;
        int state;
        void apply(void ∗X) {
                state = ∗(int ∗)X;
        }
        void writeRegister (int newstate) {
                T−>update_helper(&newstate , sizeof (int) , oid);
        }
        int readRegister () {
                T−>query_helper(oid);
                return state;
        }
}
```

Other examples:
Java ConcurrentMap: 350 LOC
Apache ZooKeeper: 1000 LOC
Apache BookKeeper: 300 LOC

simple API exposed by runtime to object: 1 upcall + two helper methods
arbitrary API exposed by object to application: mutators and accessors

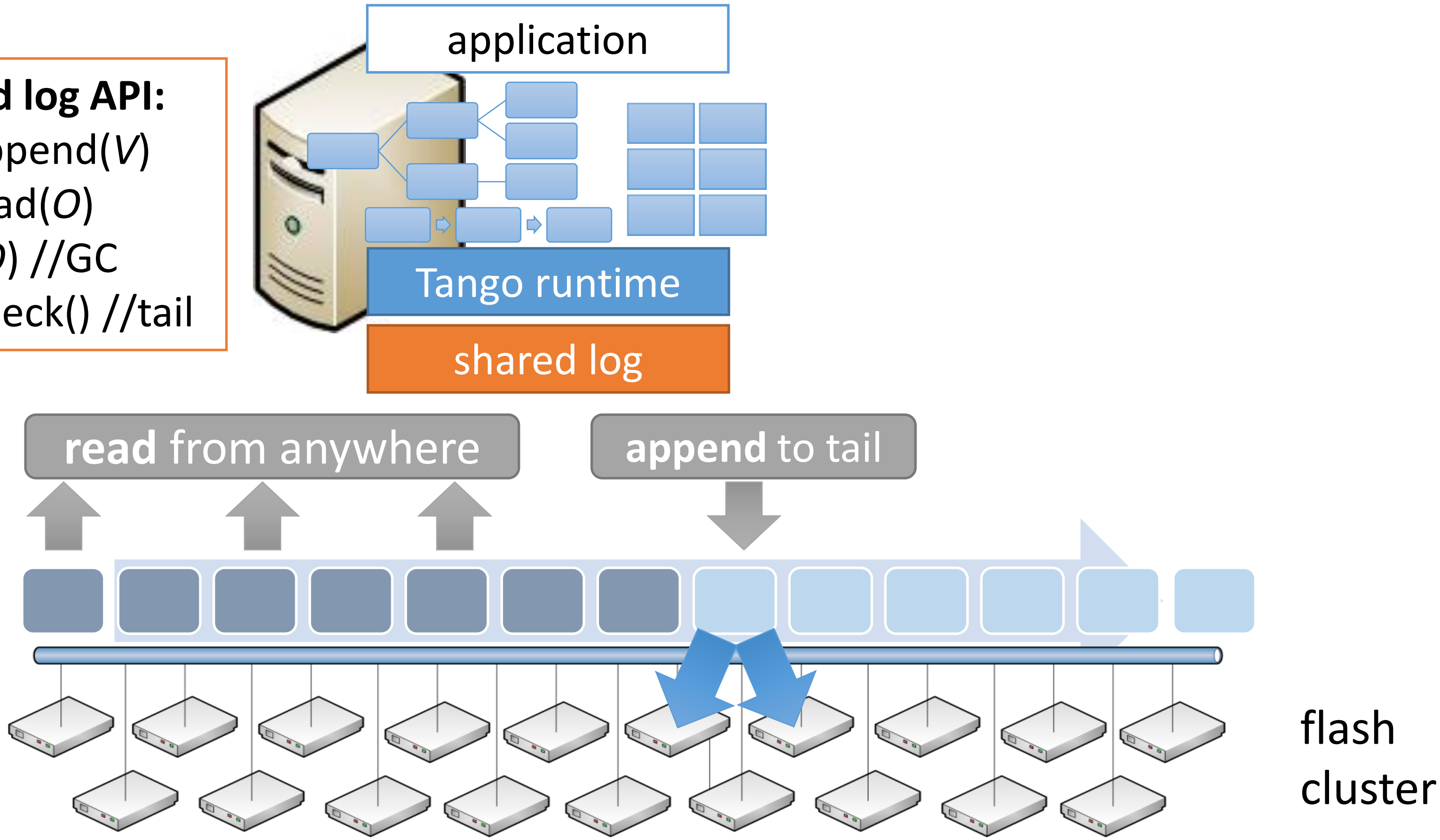
# the secret sauce: a fast shared log

**shared log API:**
O = append(*V*)
V = read(*O*)
trim(*O*) //GC
O = check() //tail



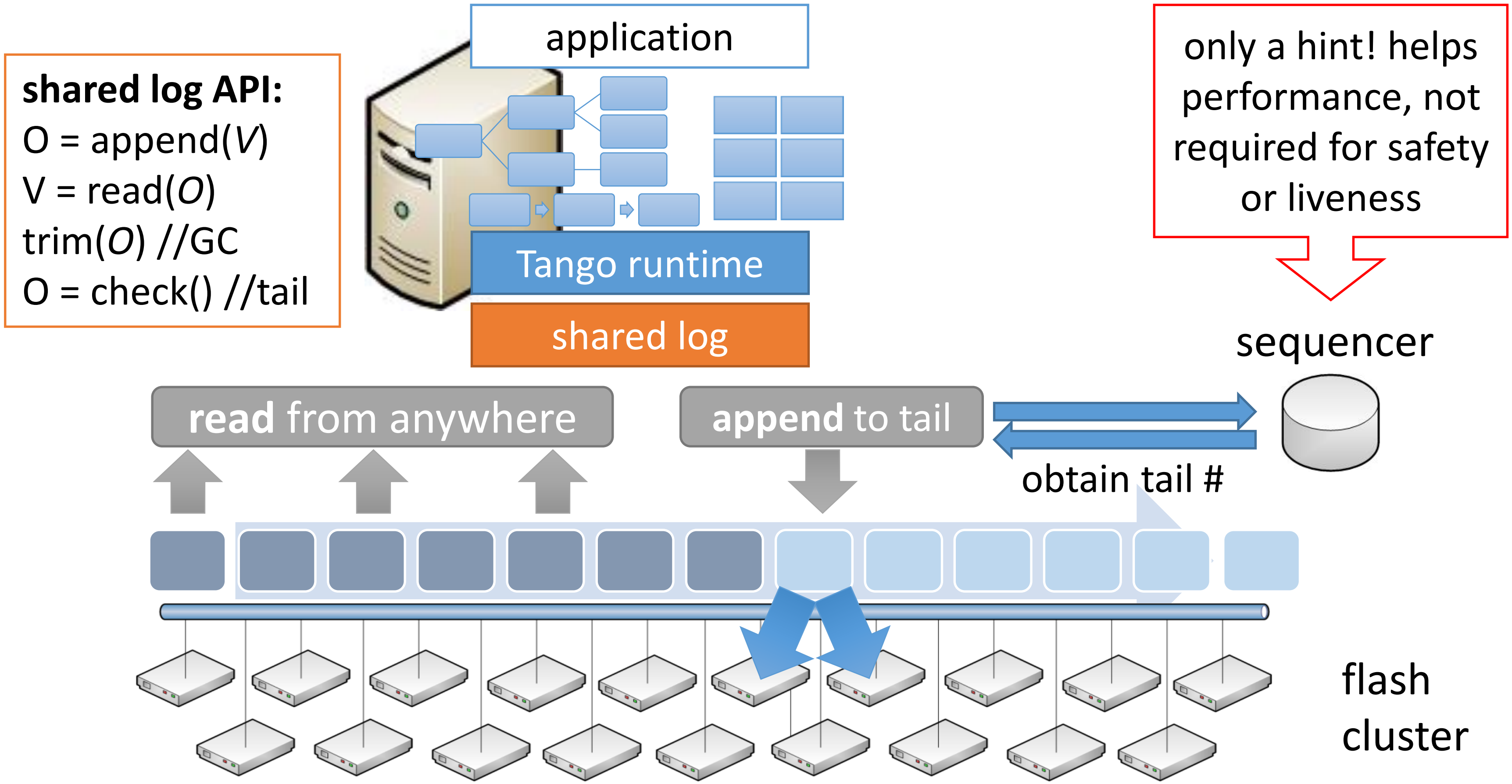
# the secret sauce: a fast shared log

**shared log API:**
O = append(*V*)
V = read(*O*)
trim(*O*) //GC
O = check() //tail



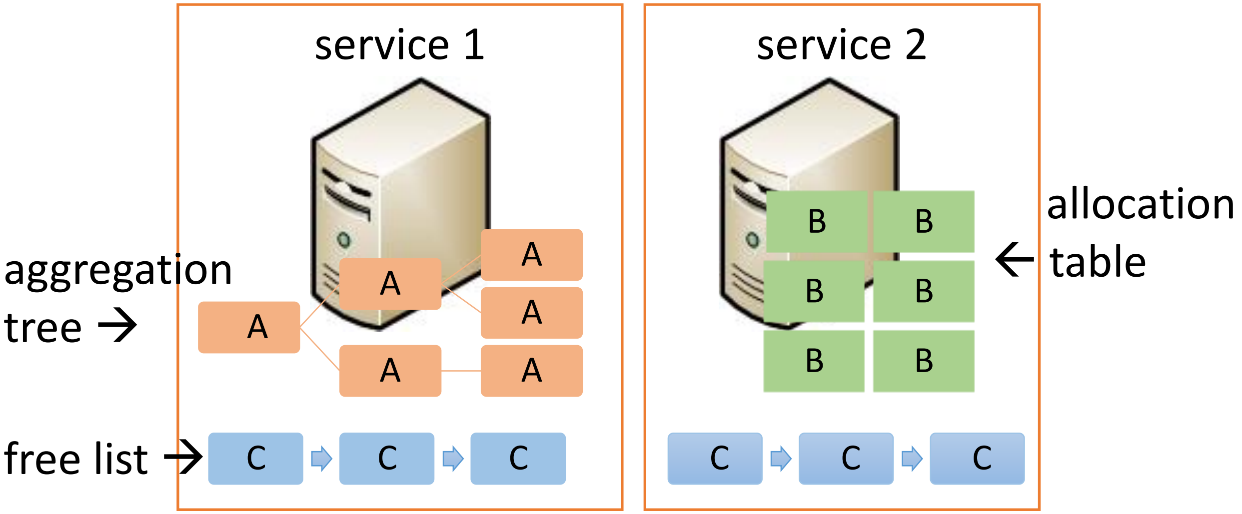

the CORFU decentralized shared log [NSDI 2012]:
- reads scale linearly with number of flash drives

# the secret sauce: a fast shared log

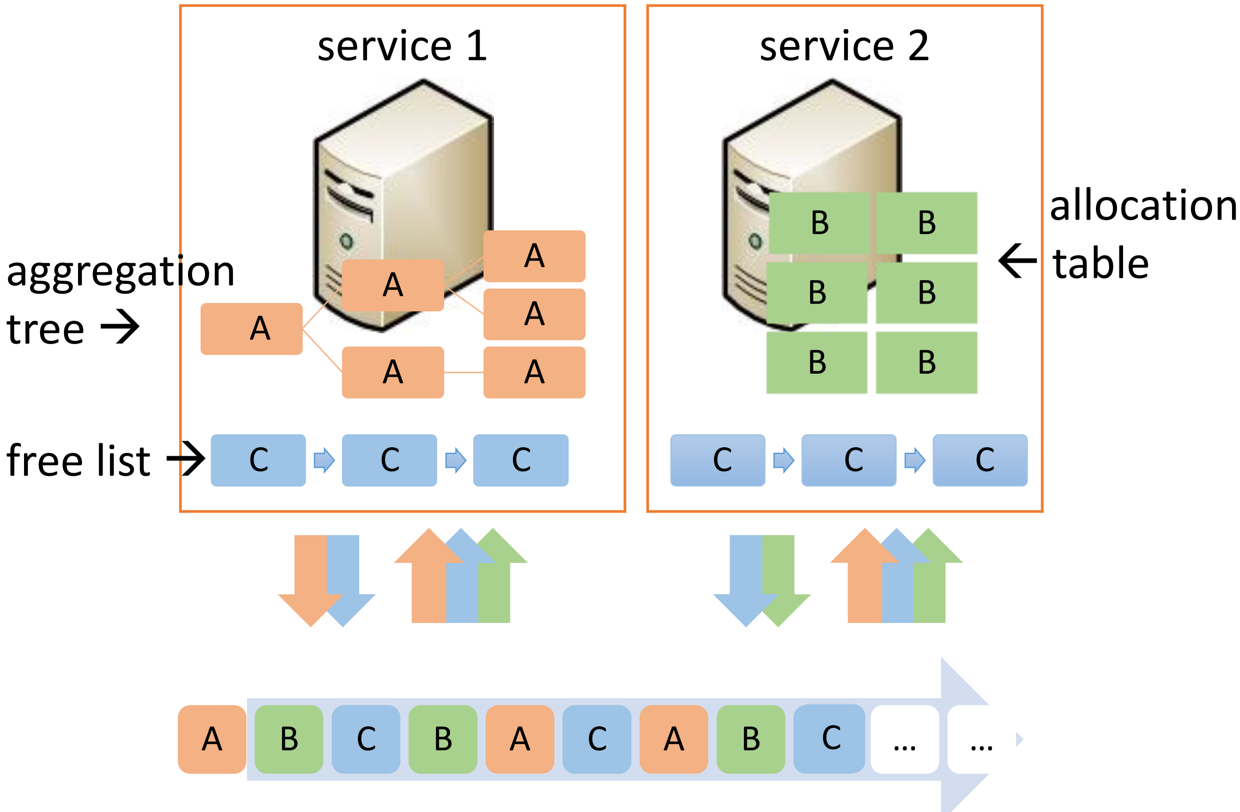



the CORFU decentralized shared log [NSDI 2012]:
- reads scale linearly with number of flash drives
- 600K/s appends (limited by sequencer speed)

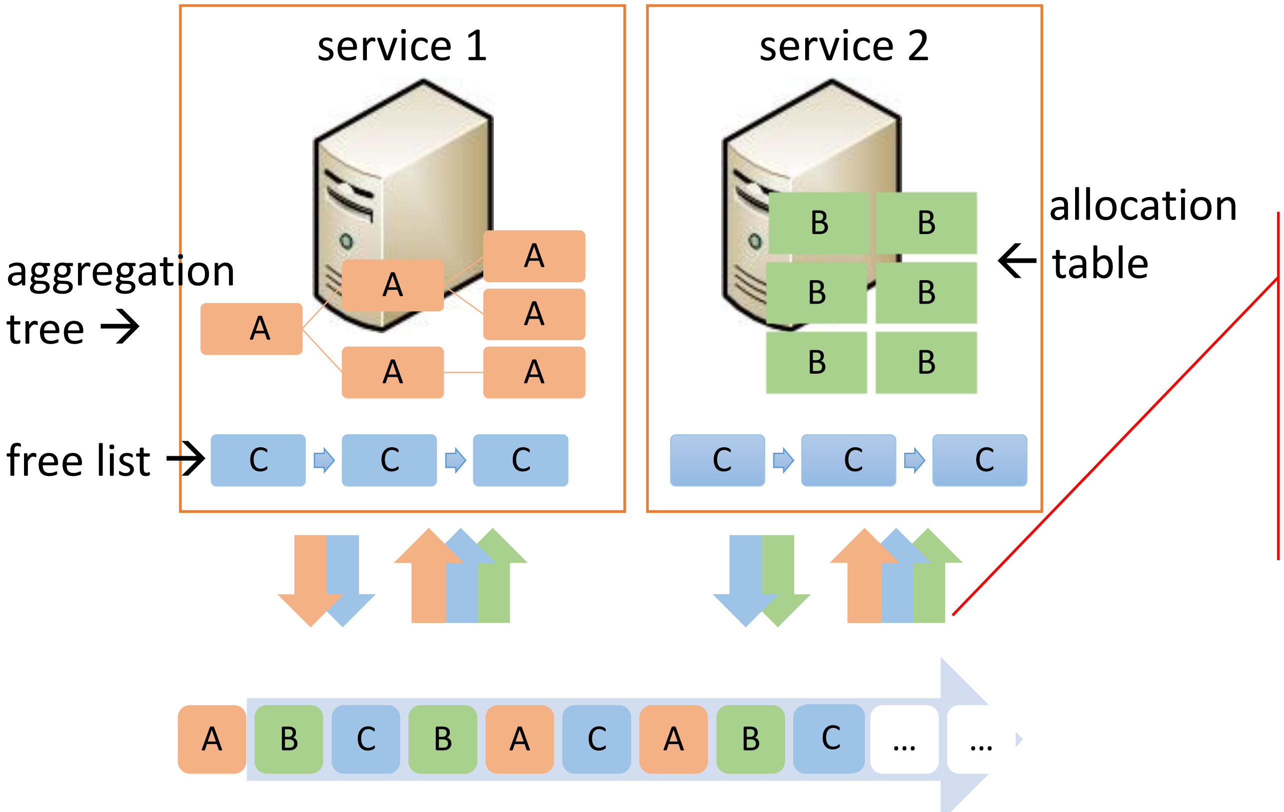
# a fast shared log isn't enough…



# a fast shared log isn't enough…

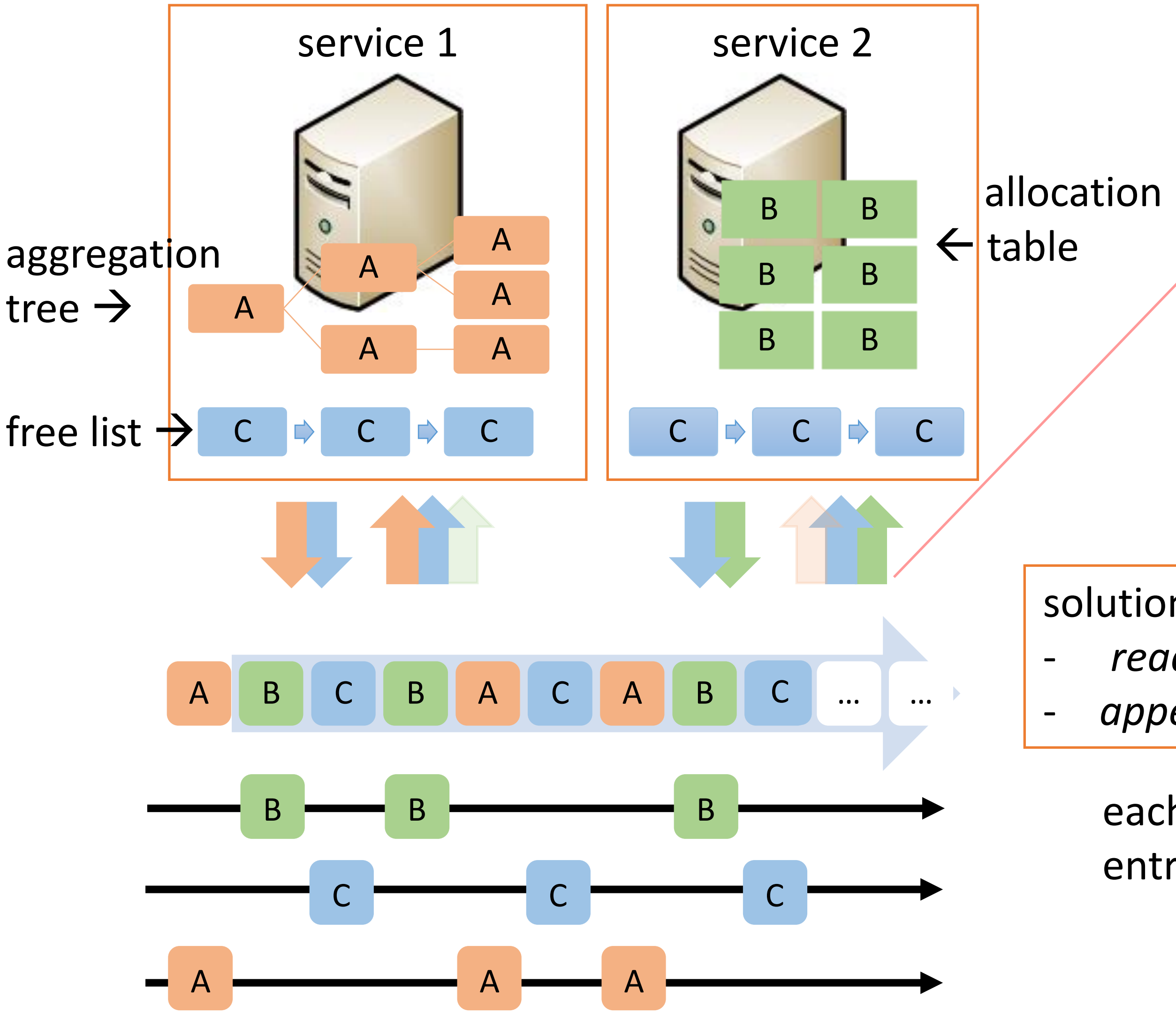


# a fast shared log isn't enough…




**the playback bottleneck**: clients must read all entries → inbound NIC is a bottleneck

# a fast shared log isn't enough…

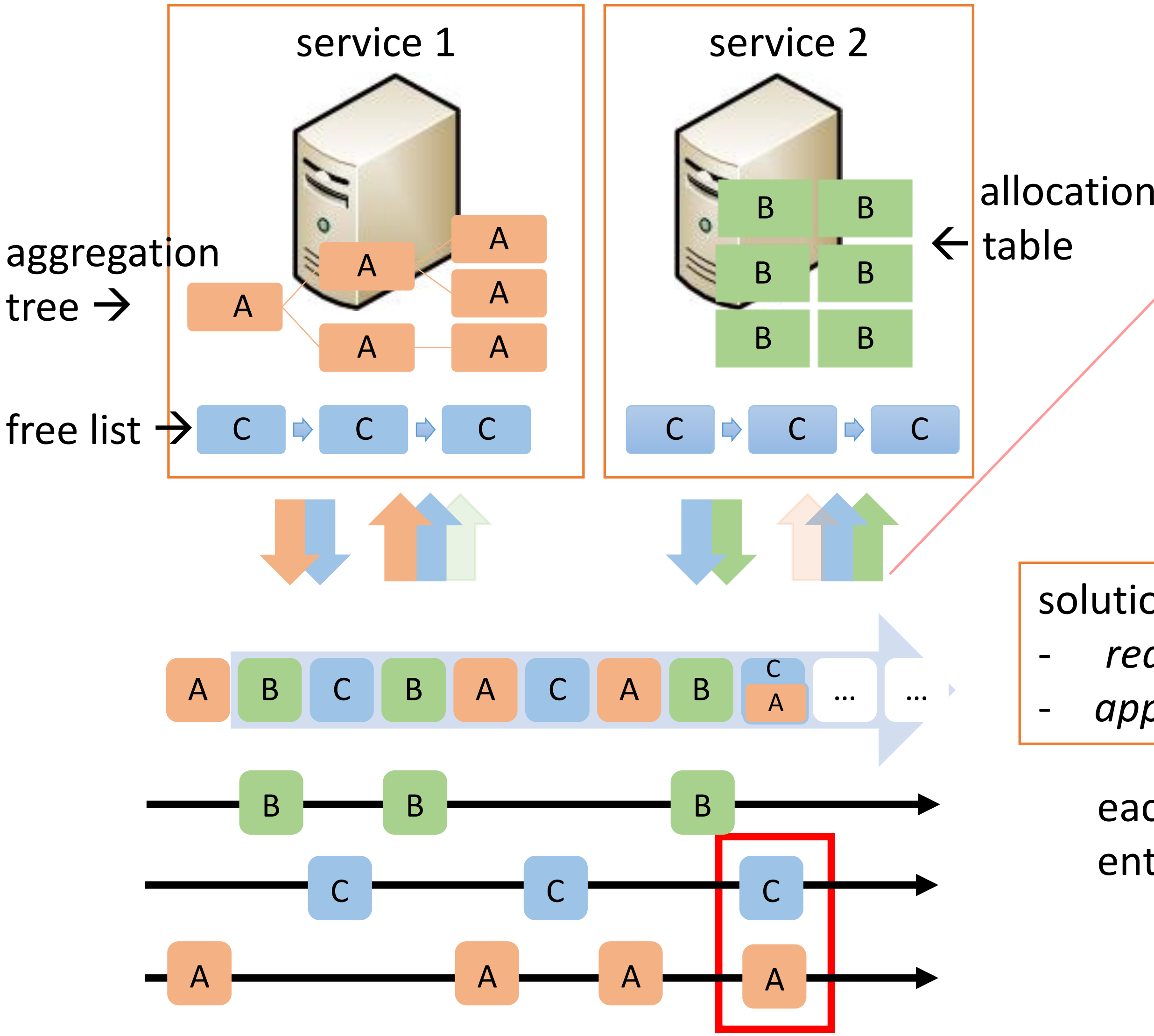



**the playback bottleneck**:
clients must read all entries → inbound NIC is a bottleneck

solution: **stream** abstraction
- *readnext(streamid)*
- *append(value, streamid1, … )*

each client only plays entries of interest to it

# a fast shared log isn't enough…

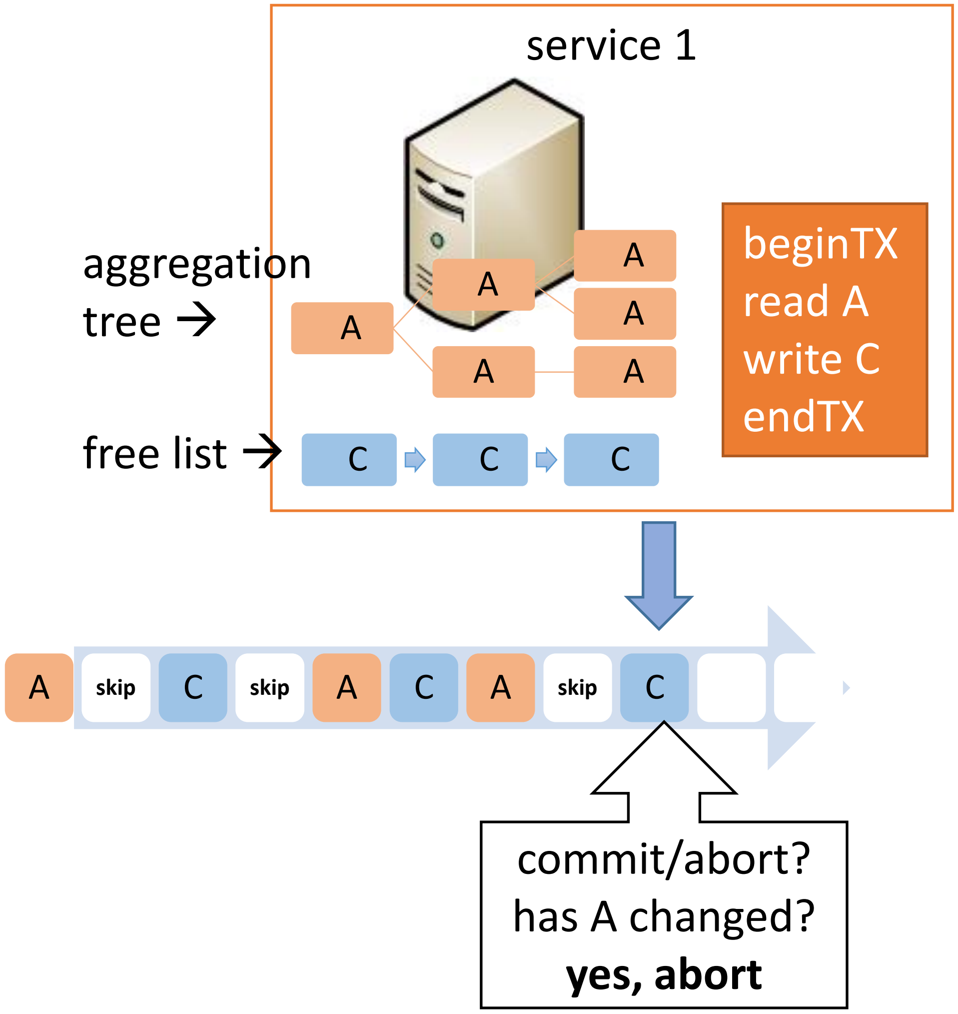



**the playback bottleneck:**
clients must read all entries → inbound NIC is a bottleneck

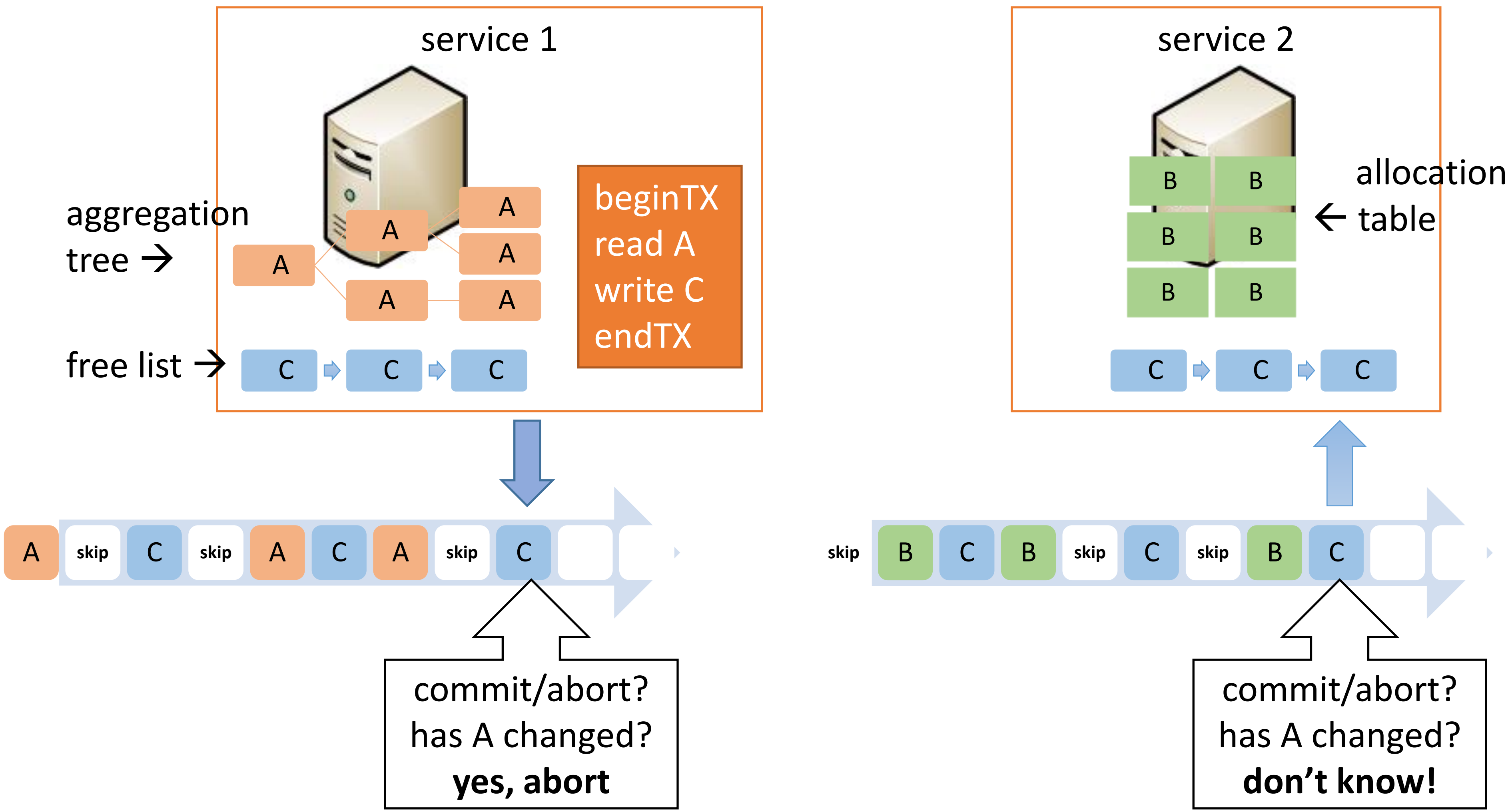
solution: **stream** abstraction
- *readnext(streamid)*
- *append(value, streamid1, … )*

each client only plays entries of interest to it

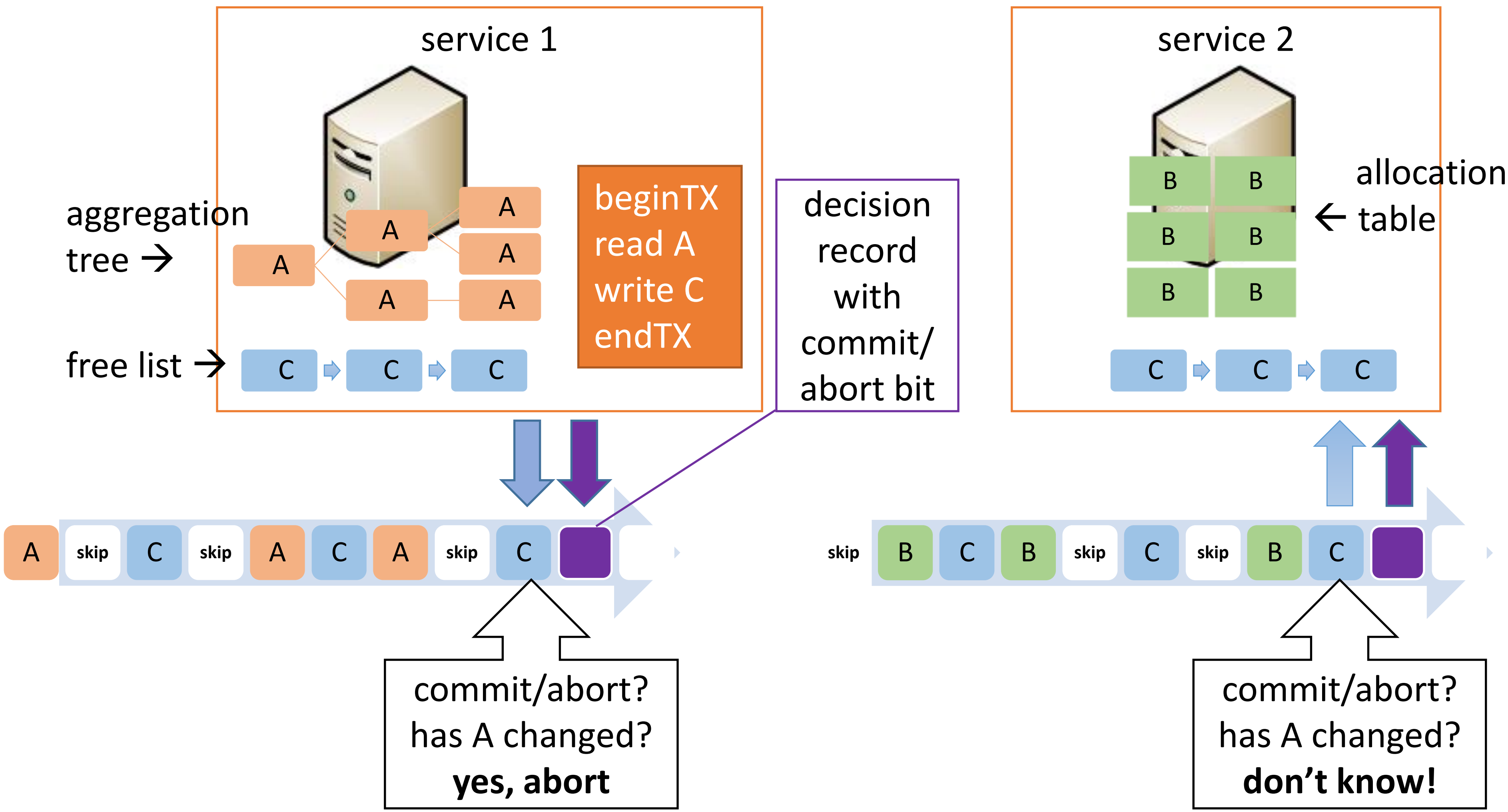
# transactions over streams



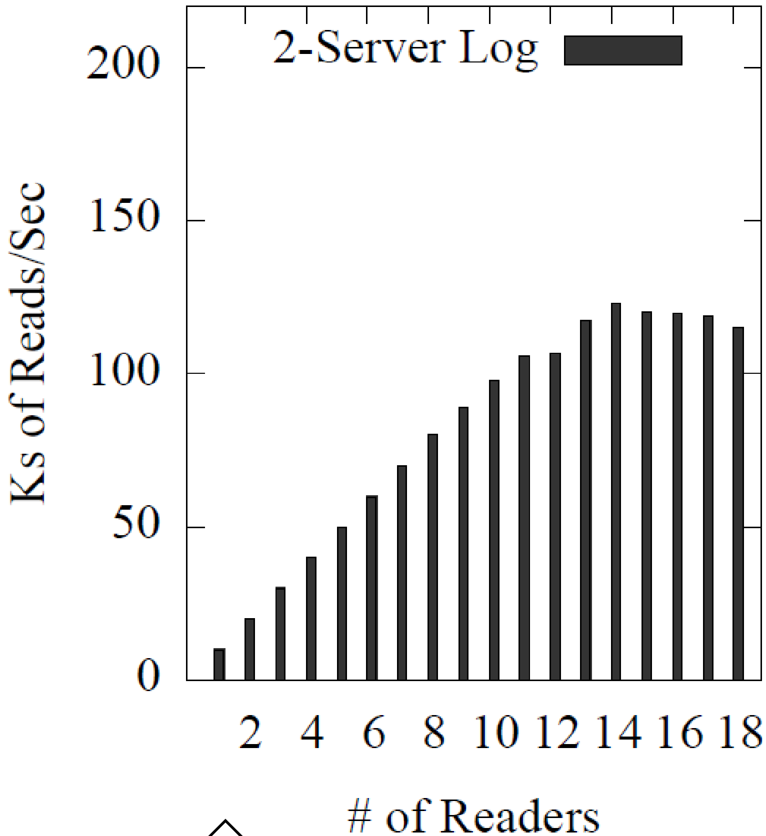
# transactions over streams



# transactions over streams



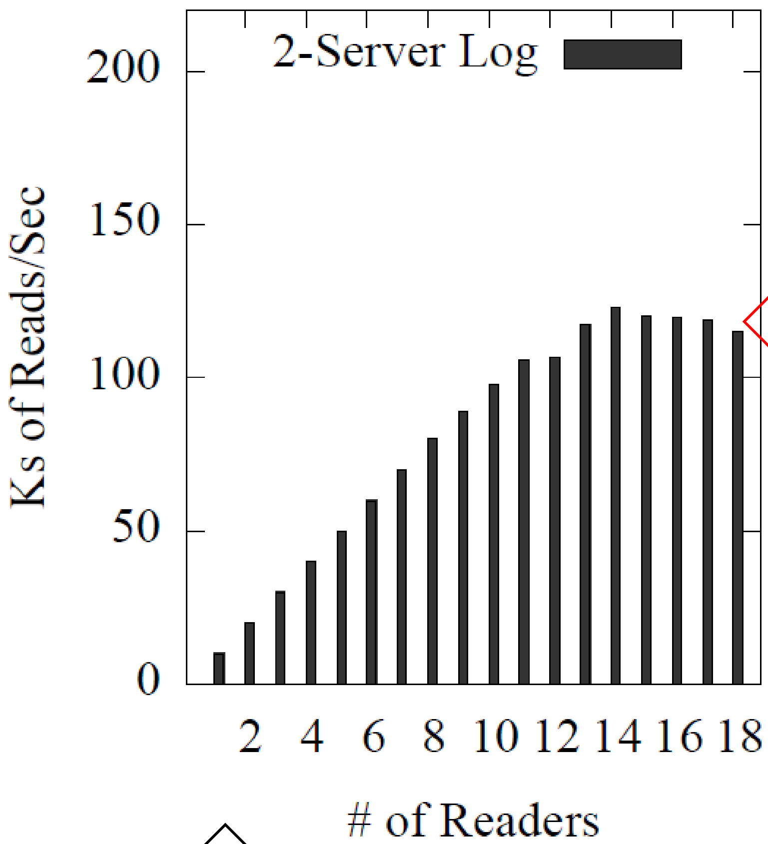
# evaluation: linearizable operations




constant write load (10K writes/sec), each client adds 10K reads/sec

a Tango object provides elasticity for strongly consistent reads

# evaluation: linearizable operations

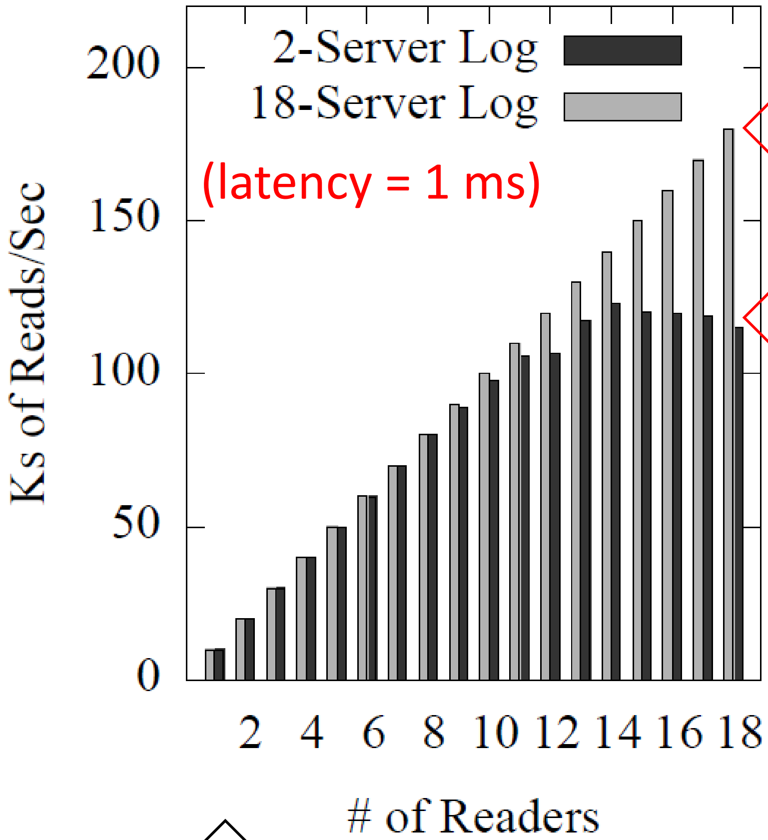



adding more clients → more reads/sec
… until shared log is saturated

a Tango object provides elasticity for strongly consistent reads

constant write load (10K writes/sec), each client adds 10K reads/sec

# evaluation: linearizable operations



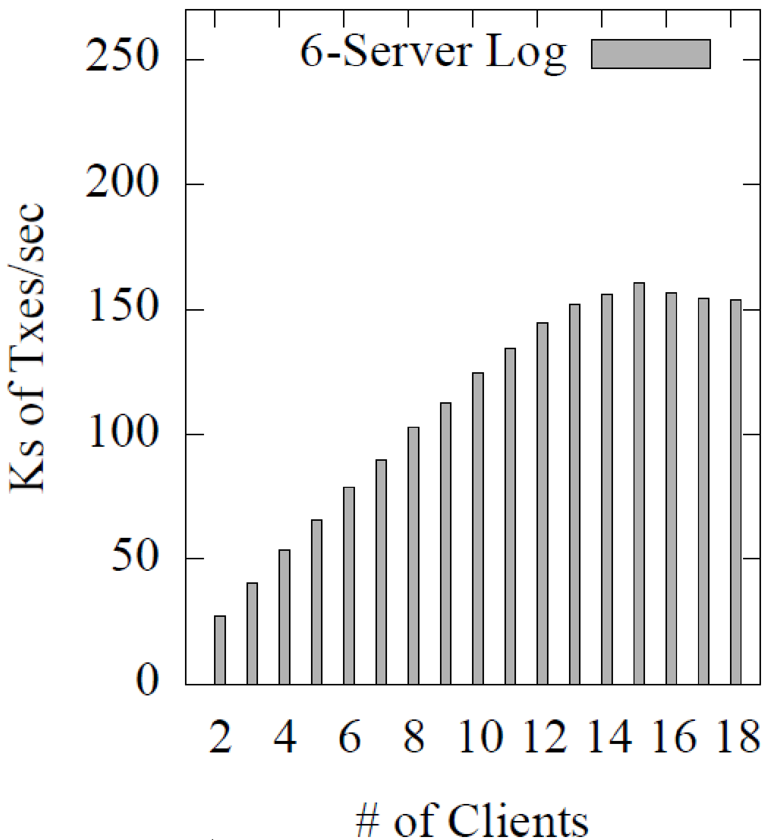

beefier shared log → scaling continues…
ultimate bottleneck: sequencer

adding more clients → more reads/sec
… until shared log is saturated

a Tango object provides elasticity for strongly consistent reads

constant write load (10K writes/sec), each client adds 10K reads/sec

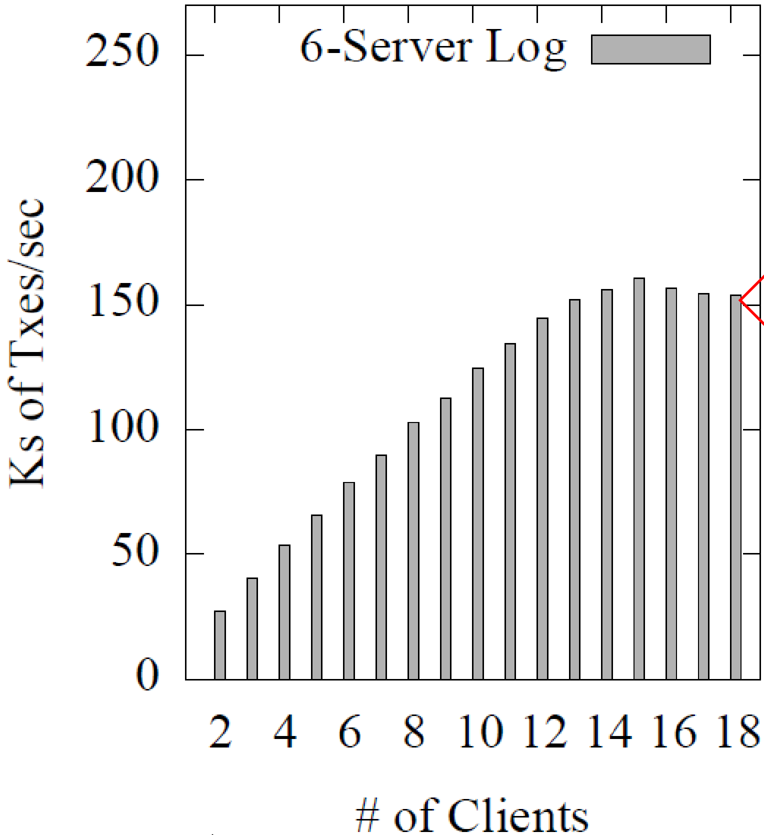
# evaluation: single object txes




scales like conventional partitioning…
but there's a cap on aggregate throughput

each client does transactions over its own TangoMap

# evaluation: single object txes

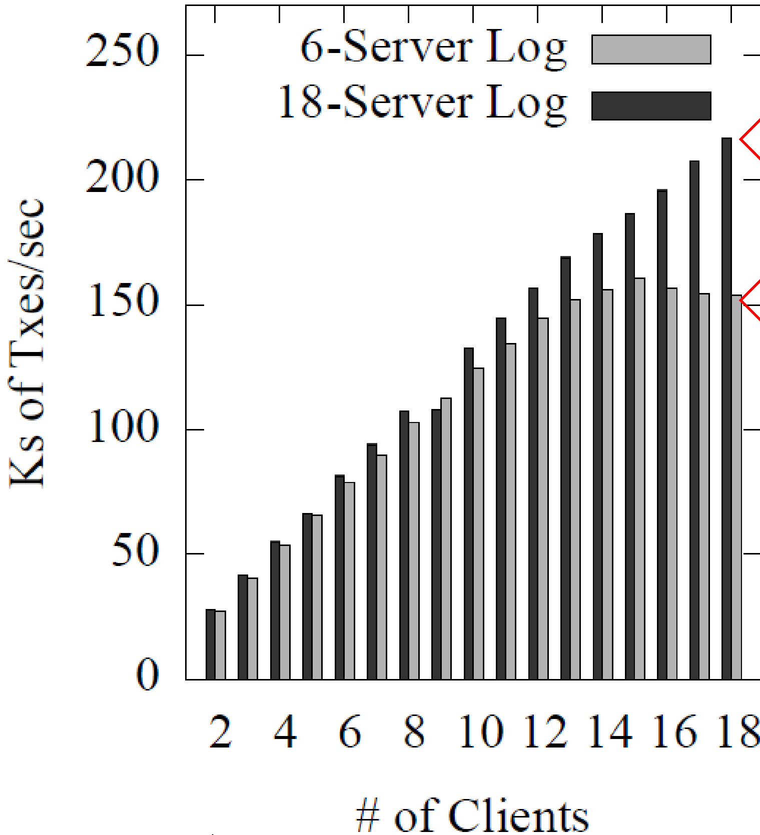



adding more clients → more transactions
… until shared log is saturated

scales like conventional partitioning…
but there's a cap on aggregate throughput

each client does transactions over its own TangoMap

# evaluation: single object txes



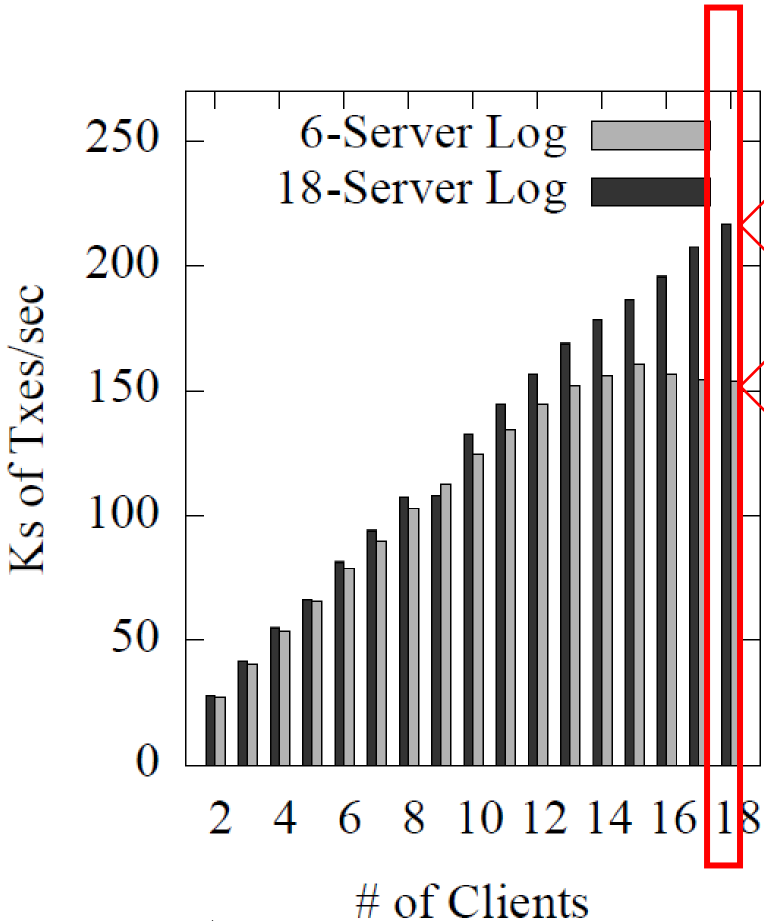

beefier shared log → scaling continues…
ultimate bottleneck: sequencer

adding more clients → more transactions
… until shared log is saturated

scales like conventional partitioning…
but there's a cap on aggregate throughput

each client does transactions over its own TangoMap

# evaluation: single object txes




beefier shared log → scaling continues…

ultimate bottleneck: sequencer

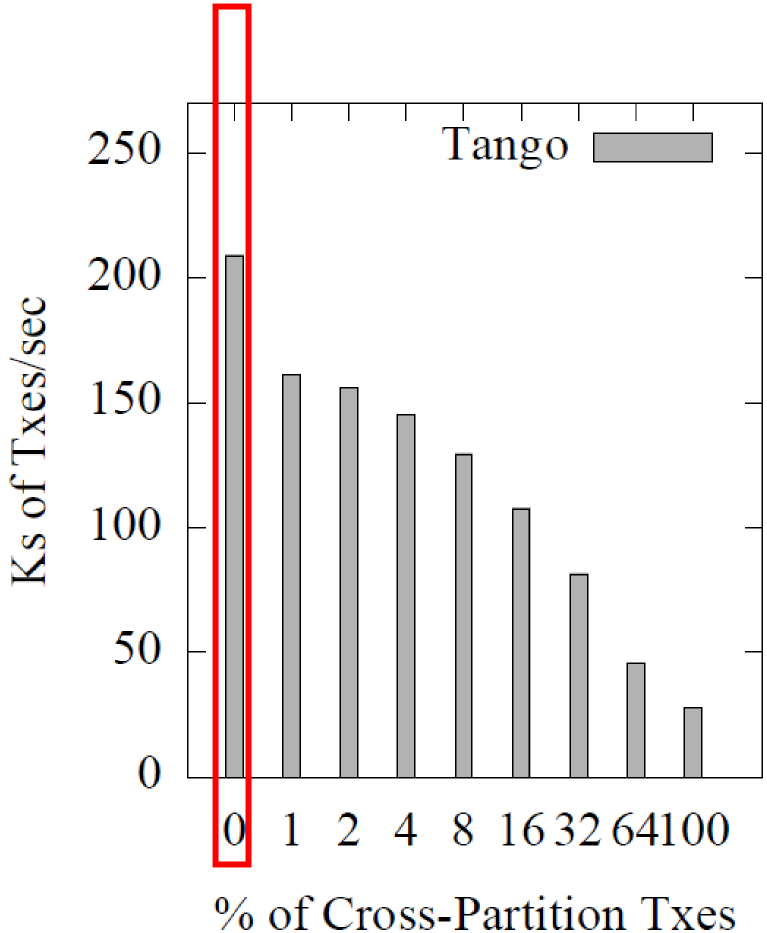
adding more clients → more transactions

… until shared log is saturated

scales like conventional partitioning…
but there's a cap on aggregate throughput

each client does transactions over its own TangoMap

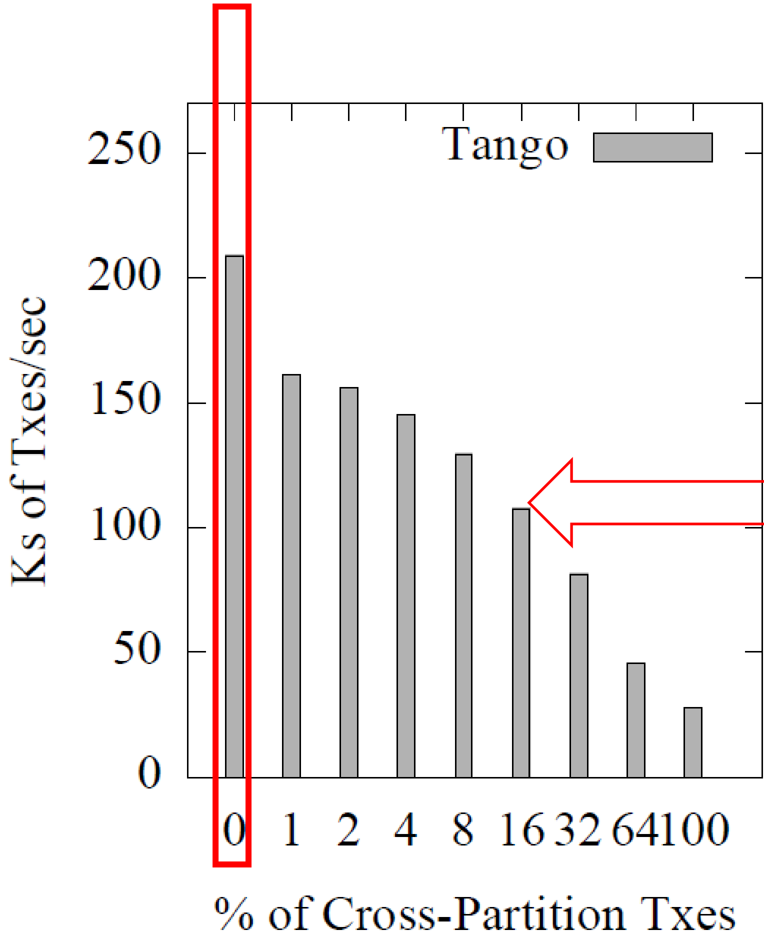
# evaluation: multi-object txes




Tango enables fast, distributed transactions across multiple objects

18 clients, each client hosts its own TangoMap
cross-partition tx: client moves element from its own TangoMap to some other client's TangoMap

# evaluation: multi-object txes

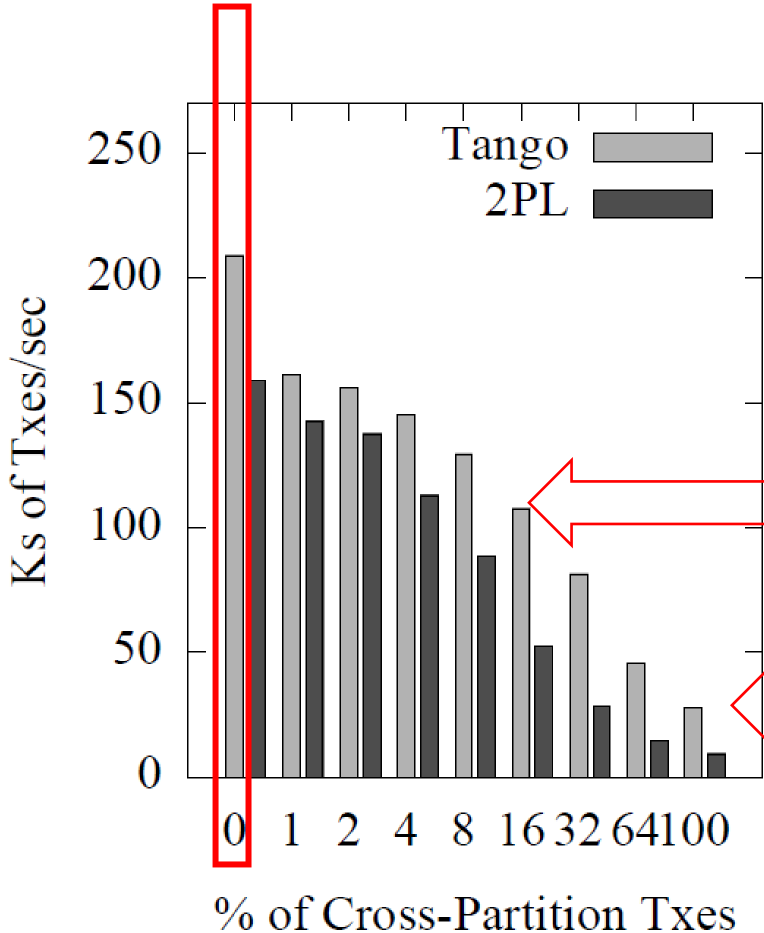



Tango enables fast, distributed transactions across multiple objects

over 100K txes/sec when 16% of txes are cross-partition

18 clients, each client hosts its own TangoMap
cross-partition tx: client moves element from its own TangoMap to some other client's TangoMap

# evaluation: multi-object txes




Tango enables fast, distributed transactions across multiple objects

over 100K txes/sec when 16% of txes are cross-partition

similar scaling to 2PL…
without a complex distributed protocol

18 clients, each client hosts its own TangoMap
cross-partition tx: client moves element from its own TangoMap to some other client's TangoMap

# conclusion

Tango objects: data structures backed by a shared log

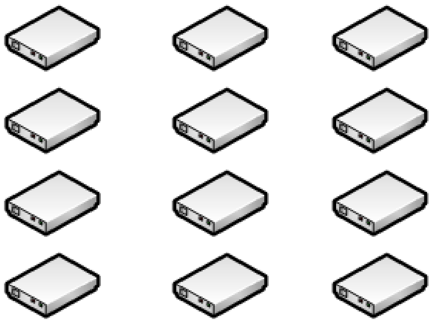
key idea: the shared log does all the heavy lifting

(persistence, consistency, atomicity, isolation, history, elasticity…)

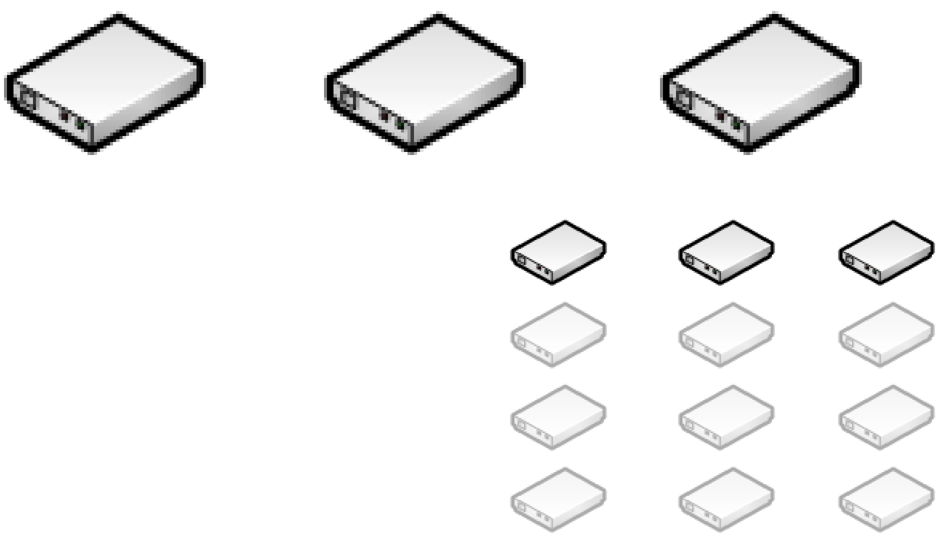
Tango objects are easy to use, easy to build, and fast!

Tango democratizes the construction of highly available metadata services thank you!

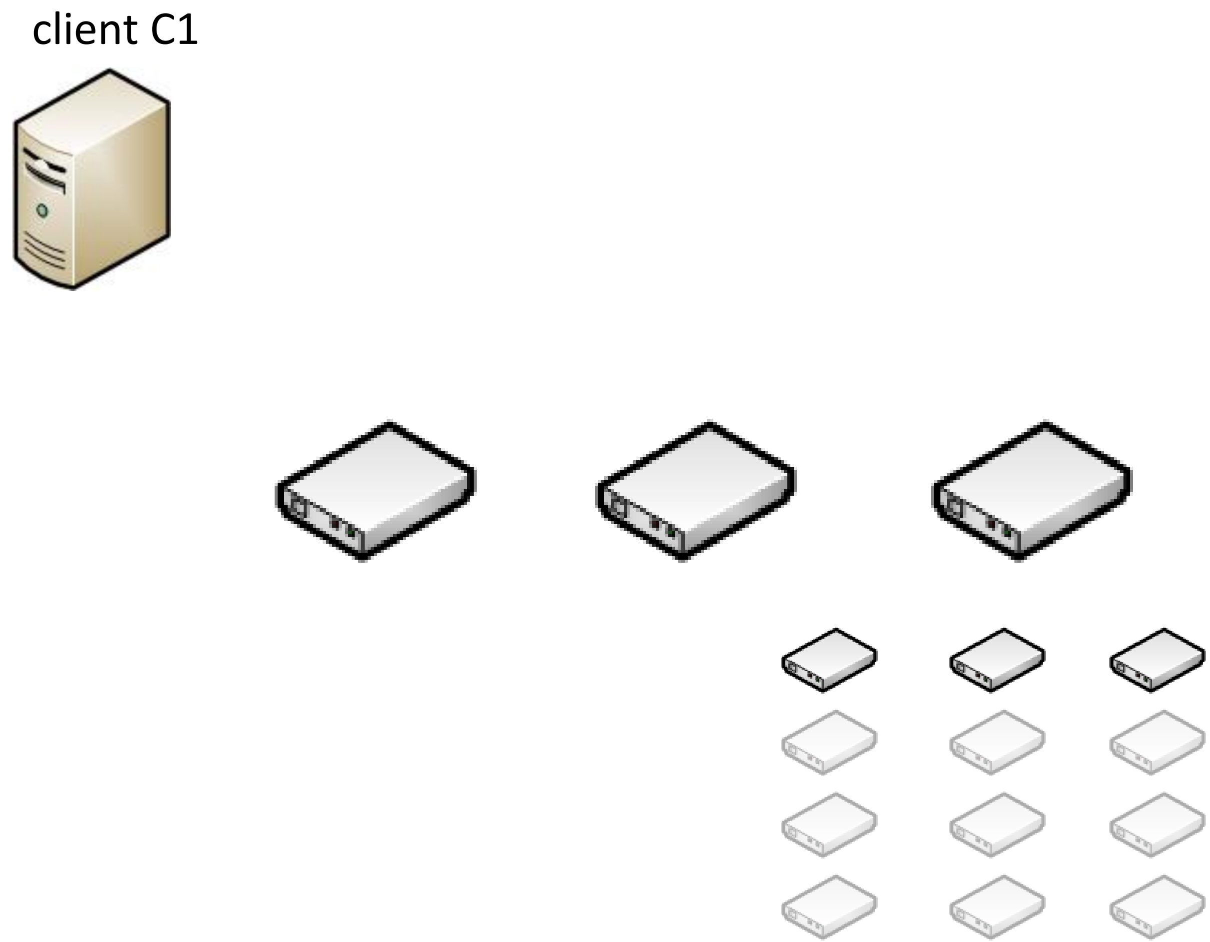
# the CORFU protocol: (chain) replication



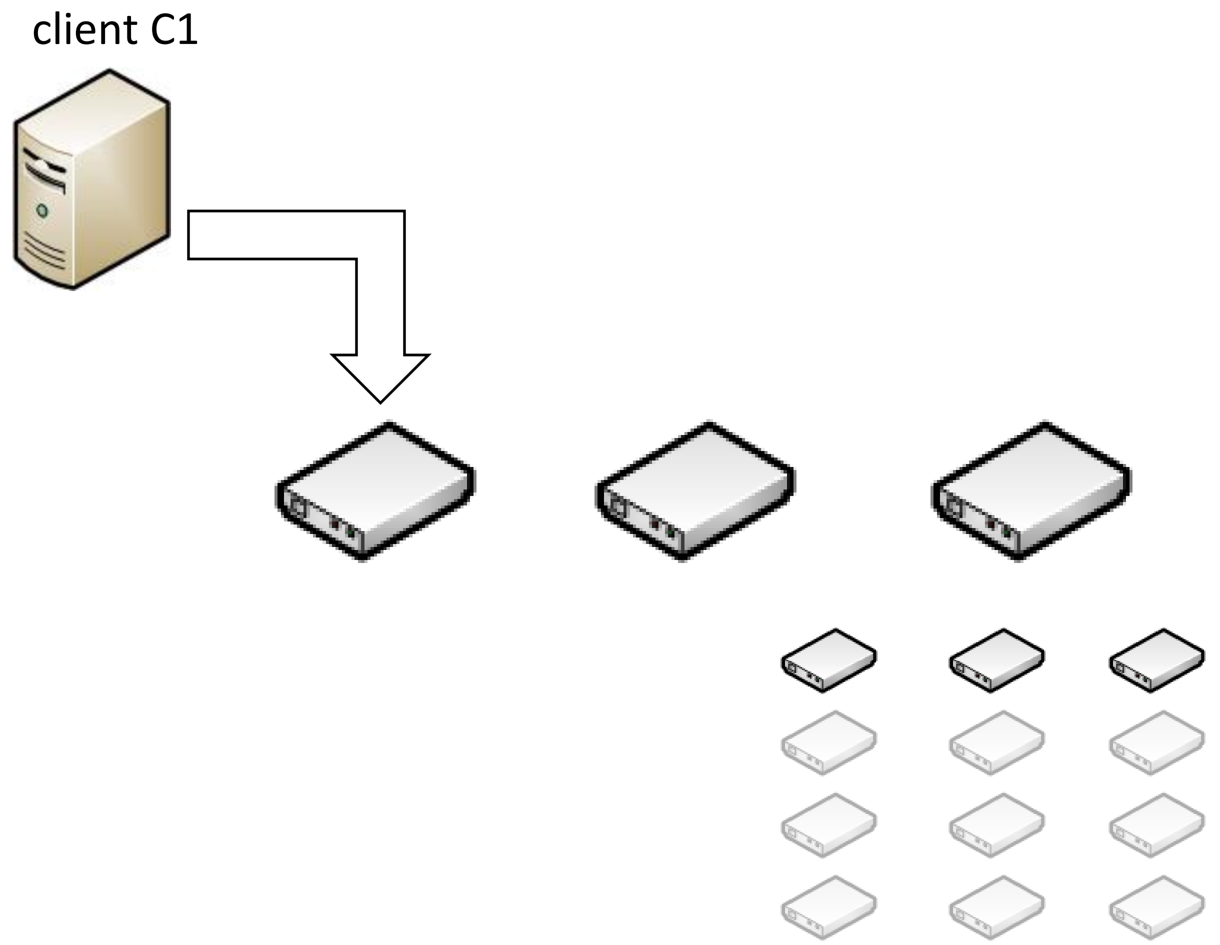
# the CORFU protocol: (chain) replication



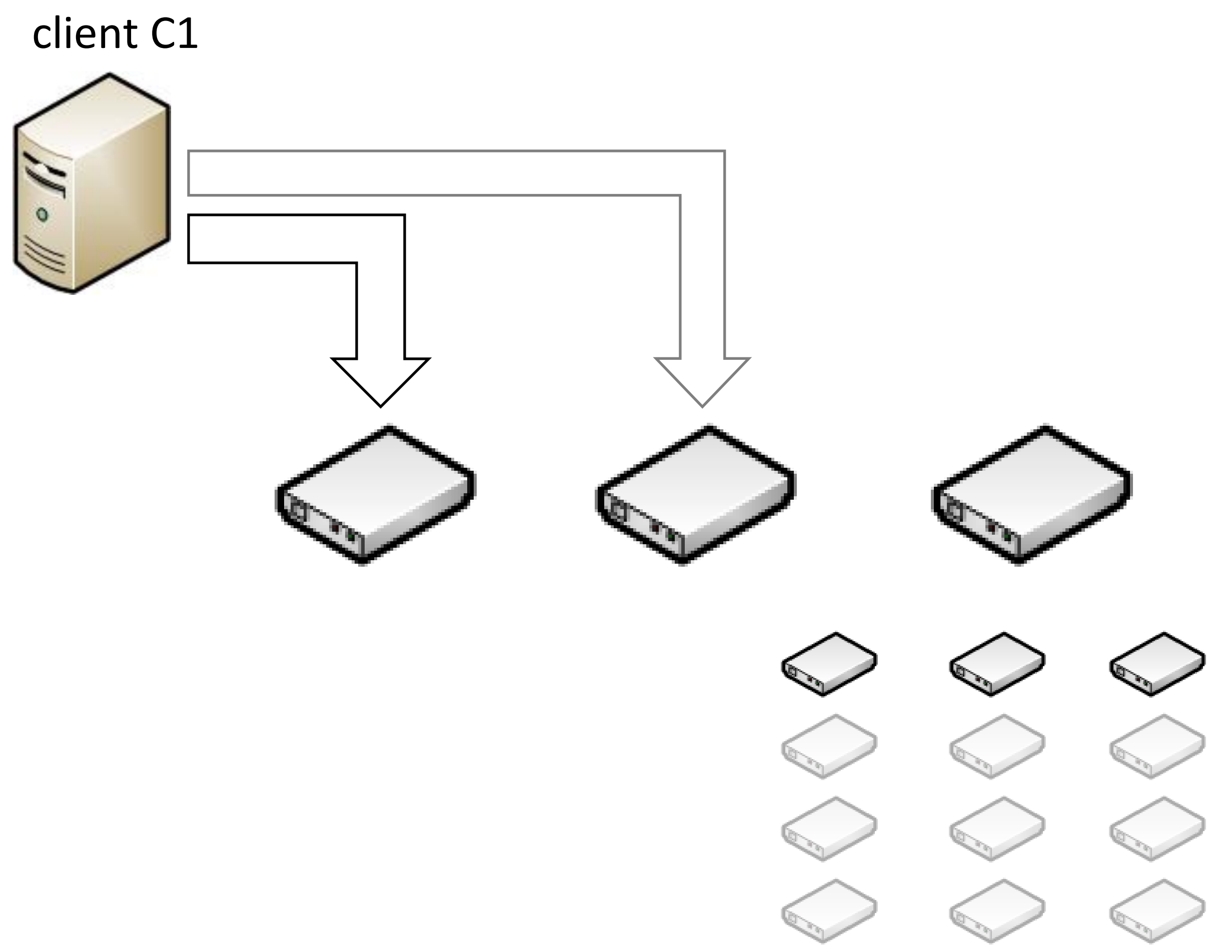
# the CORFU protocol: (chain) replication



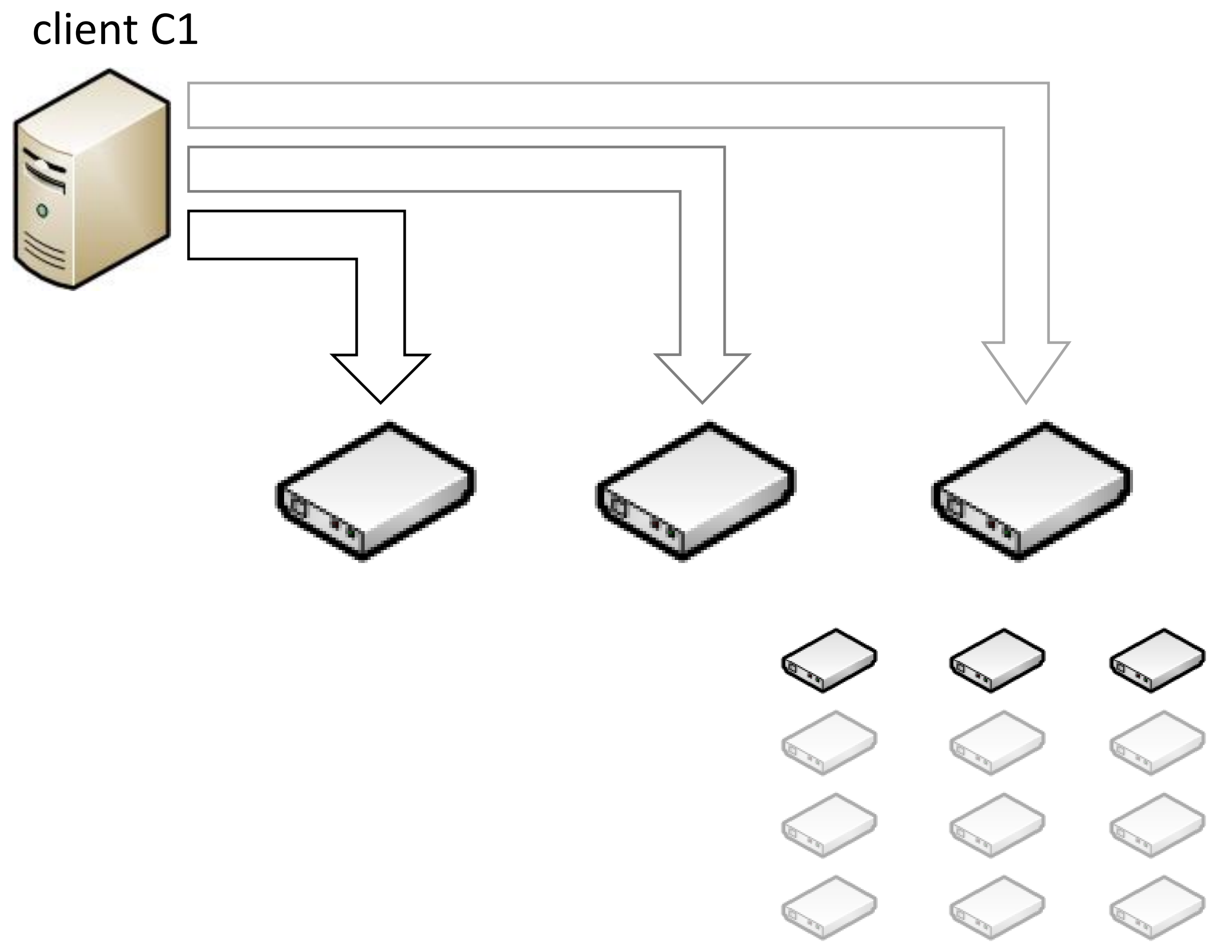
# the CORFU protocol: (chain) replication



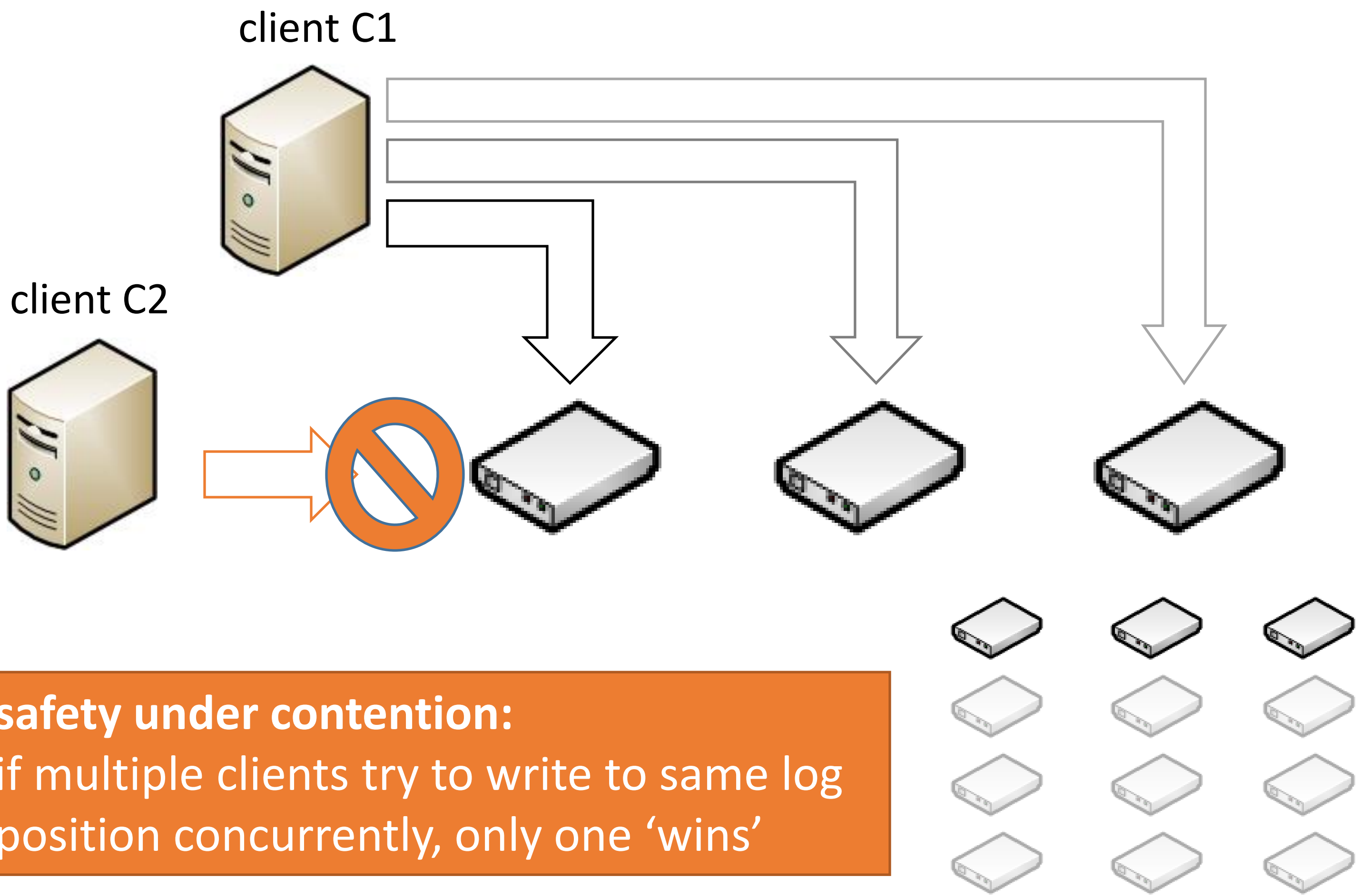
# the CORFU protocol: (chain) replication



# the CORFU protocol: (chain) replication

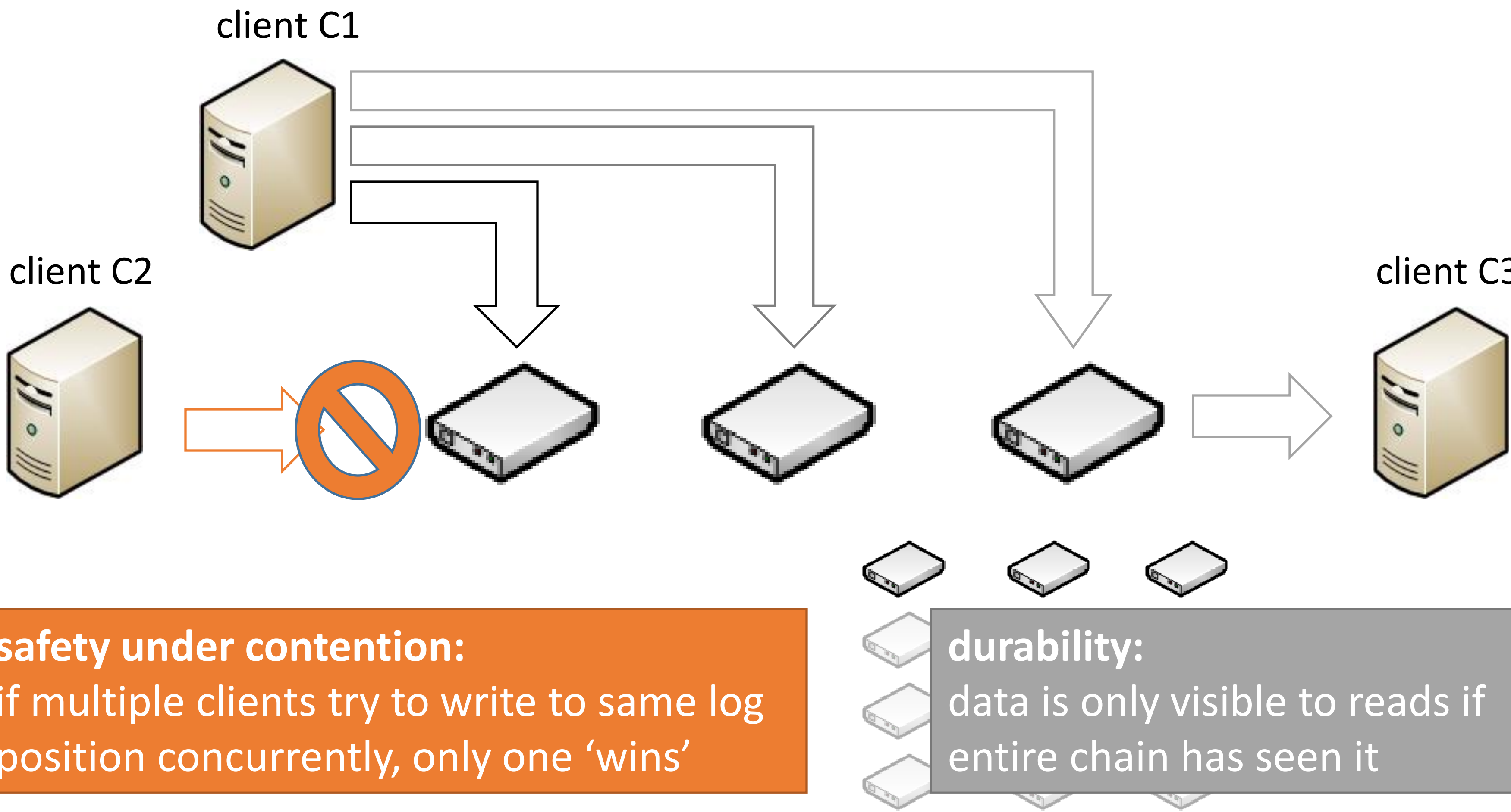


# the CORFU protocol: (chain) replication




**safety under contention:**
if multiple clients try to write to same log position concurrently, only one 'wins'

# the CORFU protocol: (chain) replication




**safety under contention:**
if multiple clients try to write to same log position concurrently, only one 'wins'

**durability:**
data is only visible to reads if entire chain has seen it