

Storage Systems (StoSys)

XM_0092

Lecture 10: Distributed / Storage Systems - II

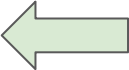
Animesh Trivedi

<https://stonet-research.github.io/>

Autumn 2023, Period 1



Syllabus outline

- ~~1. Welcome and introduction to NVM (today)~~
- ~~2. Host interfacing and software implications~~
- ~~3. Flash Translation Layer (FTL) and Garbage Collection (GC)~~
- ~~4. NVM Block Storage File systems~~
- ~~5. NVM Block Storage Key-Value Stores~~
- ~~6. Emerging Byte-addressable Storage~~
- ~~7. Networked NVM Storage~~
- ~~8. Trends: Specialization and Programmability~~
- ~~9. Distributed Storage / Systems - I~~
10. Distributed Storage / Systems - II 
11. Emerging Topics

What is the problem / observation

Distributed systems are large

Failures are common

How do we handle failures? **Ideas?**

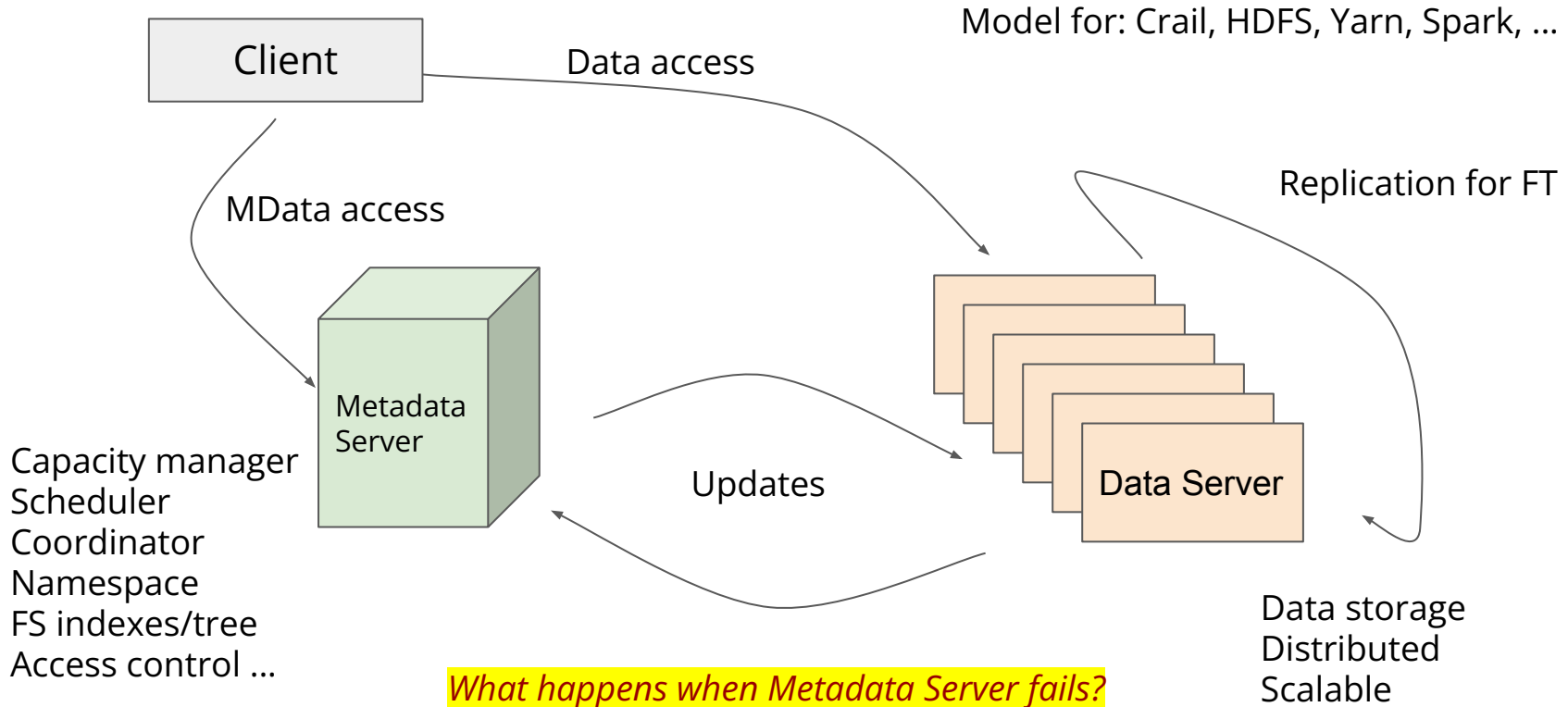
Make multiple copies

- Replication on multiple machines
- Checkpoint and recovery
- Consensus, ordering, and State Machine Replication (SMR)
- Log-based logging and recovery
- ...



<https://www.confluent.io/learn/distributed-systems/>

A Quick Distributed Storage Sketch



Often in the Literature

No clear answer, it is challenging to make Centralized Metadata service reliable and fault tolerant. Typically ...

- Write your specific custom fault tolerant service
 - Not recommended unless you know what you are doing!
- Use consensus based SMR like Paxos or Raft in your service
 - Needs service to be structured in a specific manner
- Use an external reliable service like Zookeeper
 - Need to organize its own metadata to fit in ZooKeepers' data type (tree node)
 - But what if you want a link list, hash map, array, or set?

Further Complexity

Concurrent, Distributed Data Structures are hard to get right. Now image a system in which you are building a Fault-tolerant metadata server with ...

- **Transactional (ACID) access to metadata**
 - Transaction Protocols like 2PC or 3PC
 - Concurrency control, locking, ordering
- **Sharding of metadata to do load balancing**
 - A single server cannot scale indefinitely with support a workload from a whole cluster
- **Perhaps caching, and replication**
 - How to keep multiple copies of metadata consistent

How do all these protocols interact with each other? Are they safe?

Corfu and Tango



Corfu and Tango (in Computer Science)

CORFU: A Shared Log Design for Flash Clusters

Mahesh Balakrishnan[§], Dahlia Malkhi[§], Vijayan Prabhakaran[§]
Ted Wobber[§], Michael Wei[‡], John D. Davis[§]

[§] Microsoft Research Silicon Valley

[‡] University of California, San Diego

Abstract

CORFU¹ organizes a cluster of flash devices as a single, shared log that can be accessed concurrently by multiple clients over the network. The CORFU shared log makes it easy to build distributed applications that require strong consistency at high speeds, such as databases, transactional key-value stores, replicated state machines, and metadata services. CORFU can be viewed as a distributed SSD, providing advantages over conventional SSDs such as distributed wear-leveling, network locality, fault tolerance, incremental scalability and geo-distribution. A single CORFU instance can support up to 200K appends/sec, while reads scale linearly with cluster size. Importantly, CORFU is designed to work directly over network-attached flash devices, slashing cost, power consumption and latency by eliminating storage servers.

1 Introduction

Traditionally, system designers have been forced to choose between performance and safety when building large-scale storage systems. Flash storage has the potential to dramatically alter this trade-off, providing persistence as well as high throughput and low latency. The advent of commodity flash drives creates new opportunities in the data center, enabling new designs that are impractical on disk or RAM infrastructure.

and geo-distribution [16]; and even a primary data store that leverages fast appends on underlying media. Flash is an ideal medium for implementing a scalable shared log, supporting fast, contention-free random reads to the body of the log and fast sequential writes to its tail.

One simple option for implementing a flash-based shared log is to outfit a high-end server with an expensive PCI-e SSD (e.g., Fusion-io [2]), replicating it to handle failures and scale read throughput. However, the resulting log is limited in append throughput by the bandwidth of a single server. In addition, interposing bulky, general-purpose servers between the network and flash can create performance bottlenecks, leaving bandwidth underutilized, and can also offset the power benefits of flash. In contrast, clusters of small flash units have been shown to be balanced, power-efficient and incrementally scalable [5]. Required is a distributed implementation of a shared log that can operate over such clusters.

Accordingly, we present CORFU, a shared log abstraction implemented over a cluster of flash units. In CORFU, each position in the shared log is mapped to a set of flash pages on different flash units. This map is maintained – consistently and compactly – at the clients. To read a particular position in the shared log, a client uses its local copy of this map to determine a corresponding physical flash page, and then directly issues a read to the flash unit storing that page. To append data, a client first determines the next available position in the shared log – using a sequencer node as an optimization for avoiding contention with other appending clients –

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 Silicon Valley

[‡] Microsoft Research Asia

[†] Microsoft

[§] University of California, San Diego

[¶] Cornell University

^{||} Tel-Aviv University

Abstract

Distributed systems are easier to build than ever with the emergence of new, data-centric abstractions for storing and computing over massive datasets. However, similar abstractions do not exist for storing and accessing metadata. To fill this gap, Tango provides developers with the abstraction of a replicated, in-memory data structure (such as a map or a tree) backed by a shared log. Tango objects are easy to build and use, replicating state via simple append and read operations on the shared log instead of complex distributed protocols; in the process, they obtain properties such as linearizability, persistence and high availability from the shared log. Tango also leverages the shared log to enable fast transactions across different objects, allowing applications to partition state across machines and scale to the limits of the underlying log without sacrificing consistency.

1 Introduction

Cloud platforms have democratized the development of scalable applications in recent years by providing simple, data-centric interfaces for partitioned storage (such as Amazon S3 [1] or Azure Blob Store [8]) and parallelizable computation (such as MapReduce [19] and Dred [28]). Developers can use these abstractions to

However, current cloud platforms provide applications with little support for storing and accessing metadata. Application metadata typically exists in the form of data structures such as maps, trees, counters, queues, or graphs; real-world examples include filesystem hierarchies [5], resource allocation tables [7], job assignments [3], network topologies [35], deduplication indices [20] and provenance graphs [36]. Updates to metadata usually consist of multi-operation transactions that span different data structures – or arbitrary subsets of a single data structure – while requiring atomicity and isolation; for example, moving a node from a free list to an allocation table, or moving a file from one portion of a namespace to another. At the same time, application metadata is required to be highly available and persistent in the face of faults.

Existing solutions for storing metadata do not provide transactional access to arbitrary data structures with persistence and high availability. Cloud storage services (e.g., SimpleDB [2]) and coordination services (e.g., ZooKeeper [27] and Chubby [14]) provide persistence, high availability, and strong consistency. However, each system does so for a specific data structure, and with limited or no support for transactions that span multiple operations, items, or data structures. Conventional databases support transactions, but with limited scalability and not over arbitrary data structures.

Log: A very powerful data structure

A sequential appending data structure

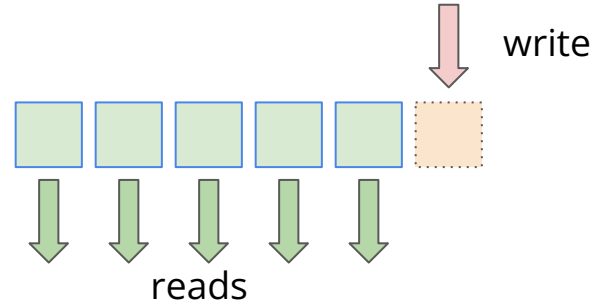
- Write only at the end (tail)
- Read from anywhere

Many unique properties

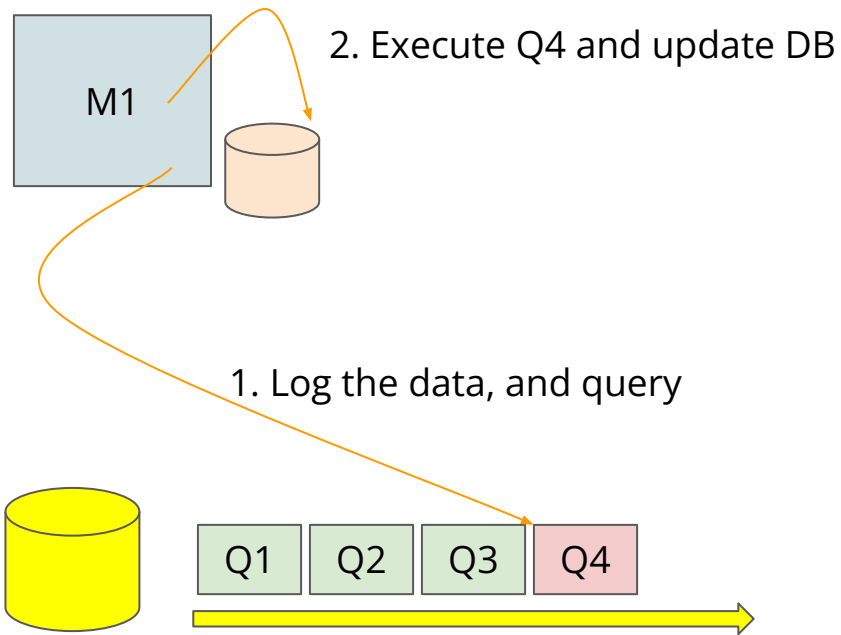
- No in-place updates, once written, the data becomes immutable
- Serialized writing, one point of writing, the tail - either the write succeeds or not (atomicity)
- Logging events (used in DBs, file systems) for failure recovery
- Ordering of events (writes, transactions, or whatever)

For NAND flash

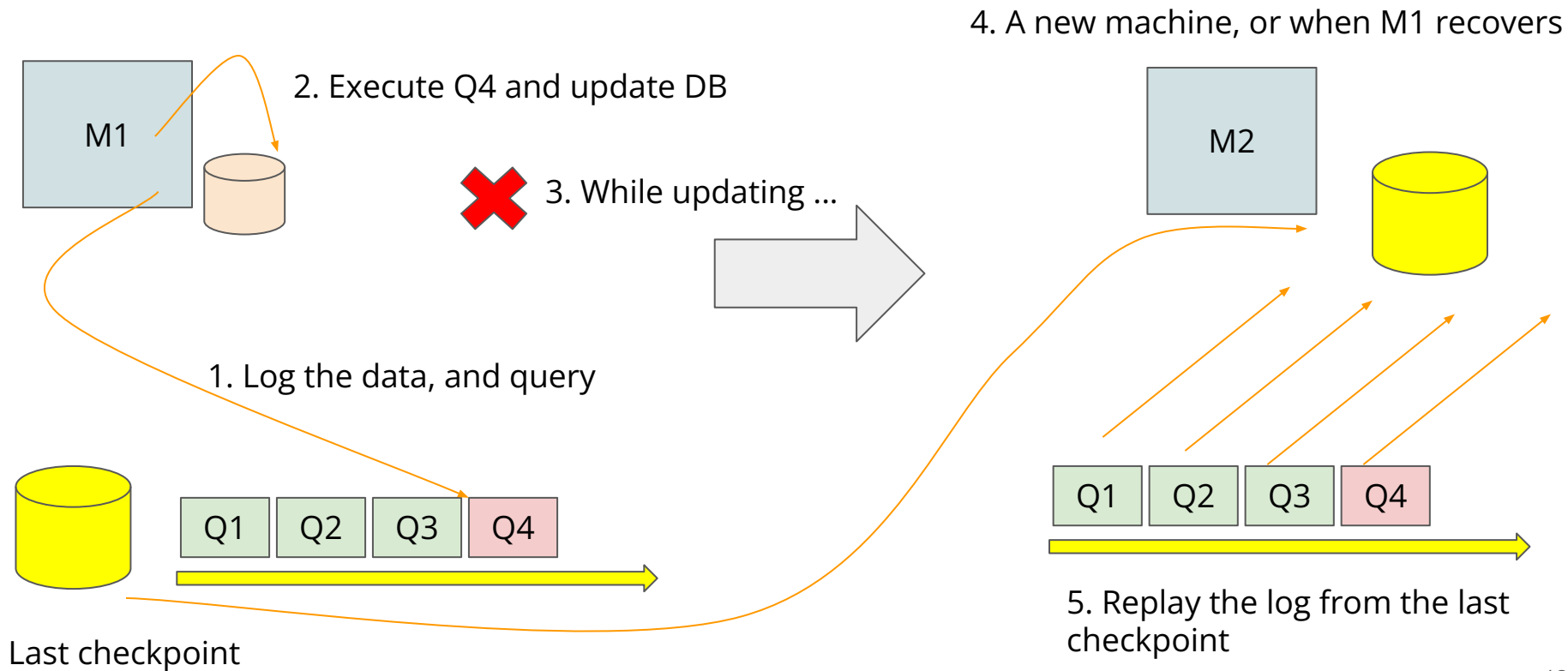
- Converts a random write to a sequential one
- Parallel reading



Log usage example - I : Recovery and Write-Ahead-Log (WAL)



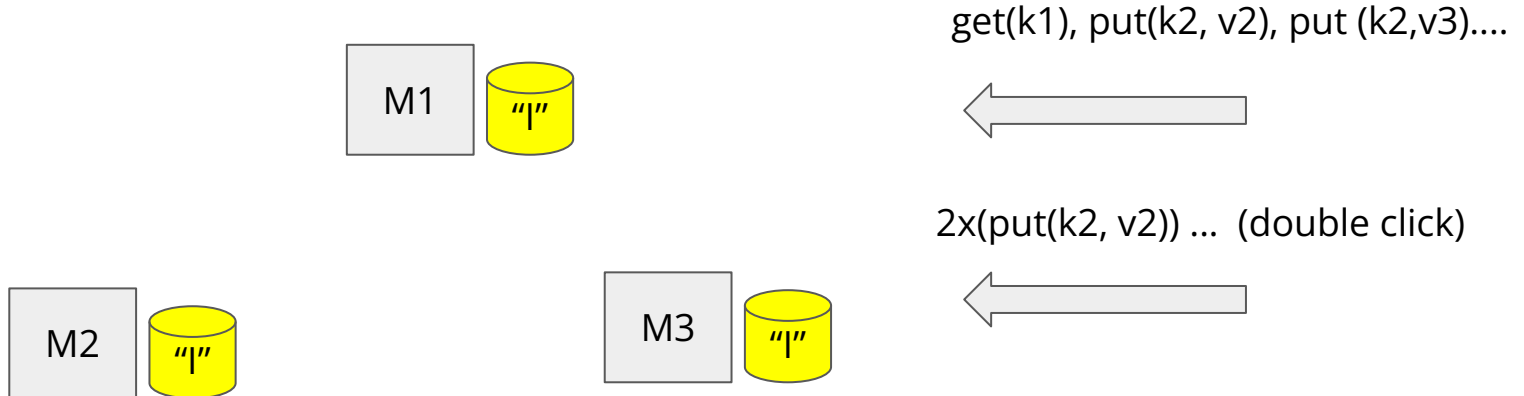
Log usage example - I : Recovery and Write-Ahead-Log (WAL)



Log usage example - II : State Machine Replication

Given a common starting state say "I", if all machines apply the same deterministic transformation to their data they will stay maintain a replicated copy of the data - *hence can handle failure of machines*

Challenges: in an unreliable distributed environment how to come up with ordering of transformation?

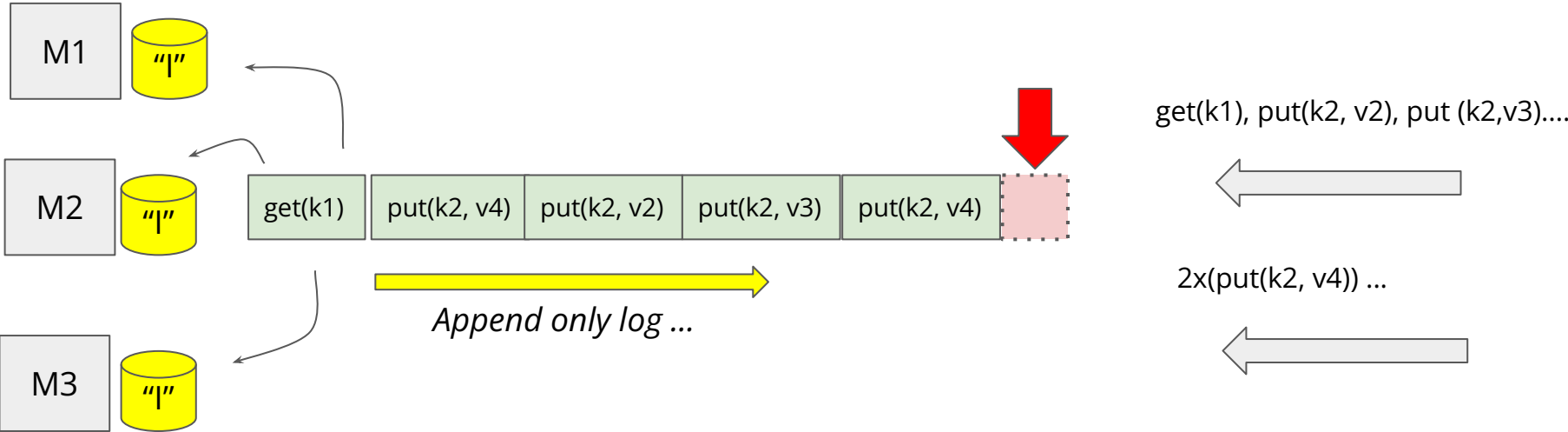


Ideas?

Consensus Algorithms

Paxos, Raft, Viewstamp Replication, Zookeeper Atomic Broadcast, ...

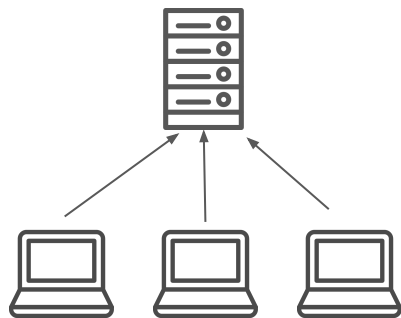
Raft (USENIX 2015): a log can be used as a basis of building consensus



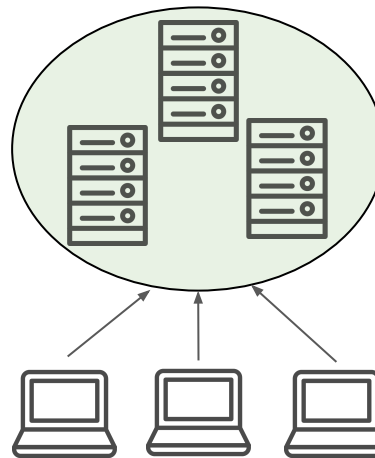
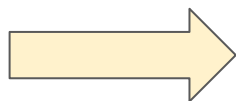
SMR for Highly Available Services

building a highly reliable storage service

key-value store



client machines

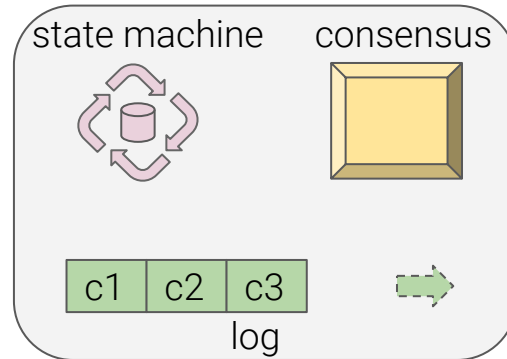
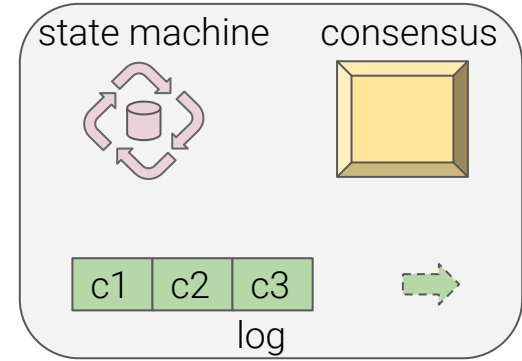
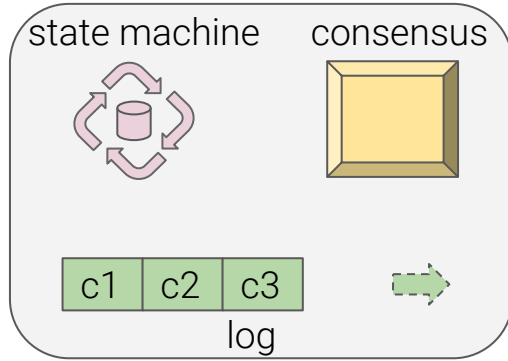


client machines

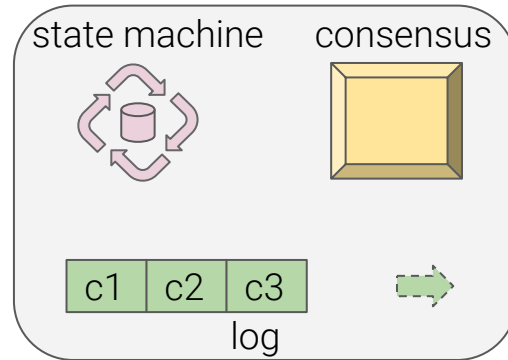
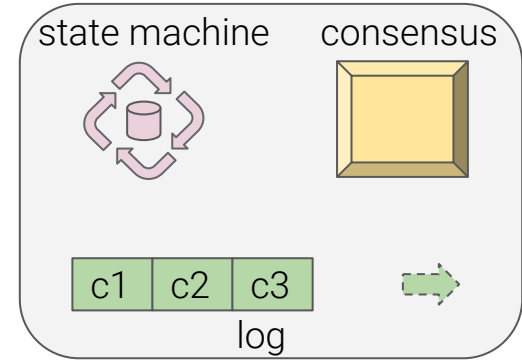
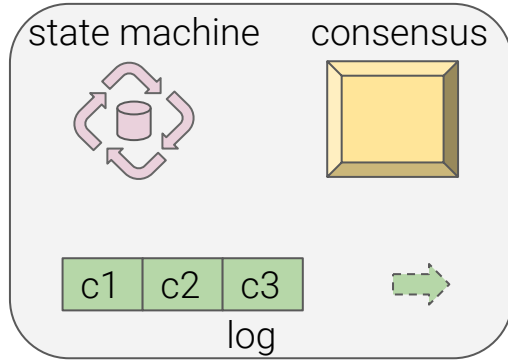
consensus to reason about
the state of the replicated system

- + Pros: easy, and consistent
- Cons: Failure

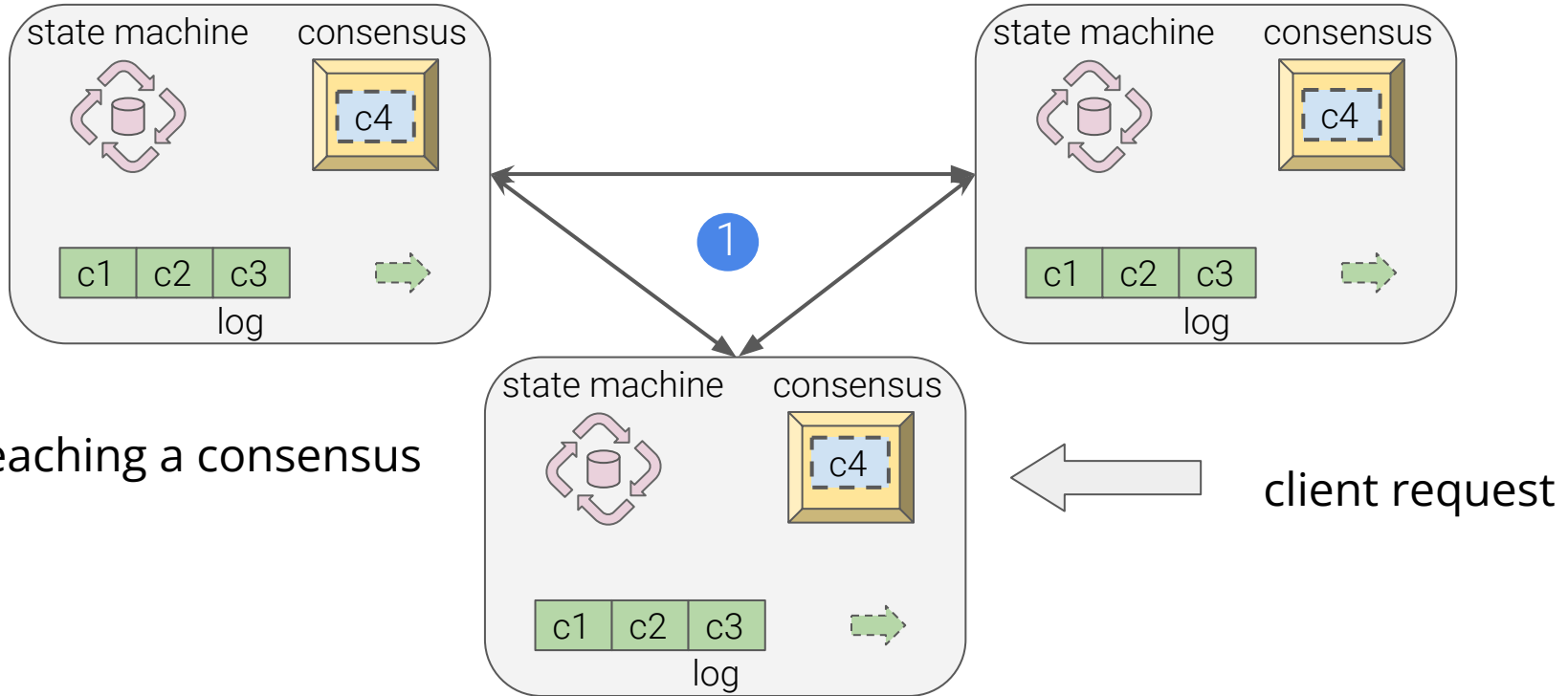
State-Machine Replication using Log



State-Machine Replication using Log

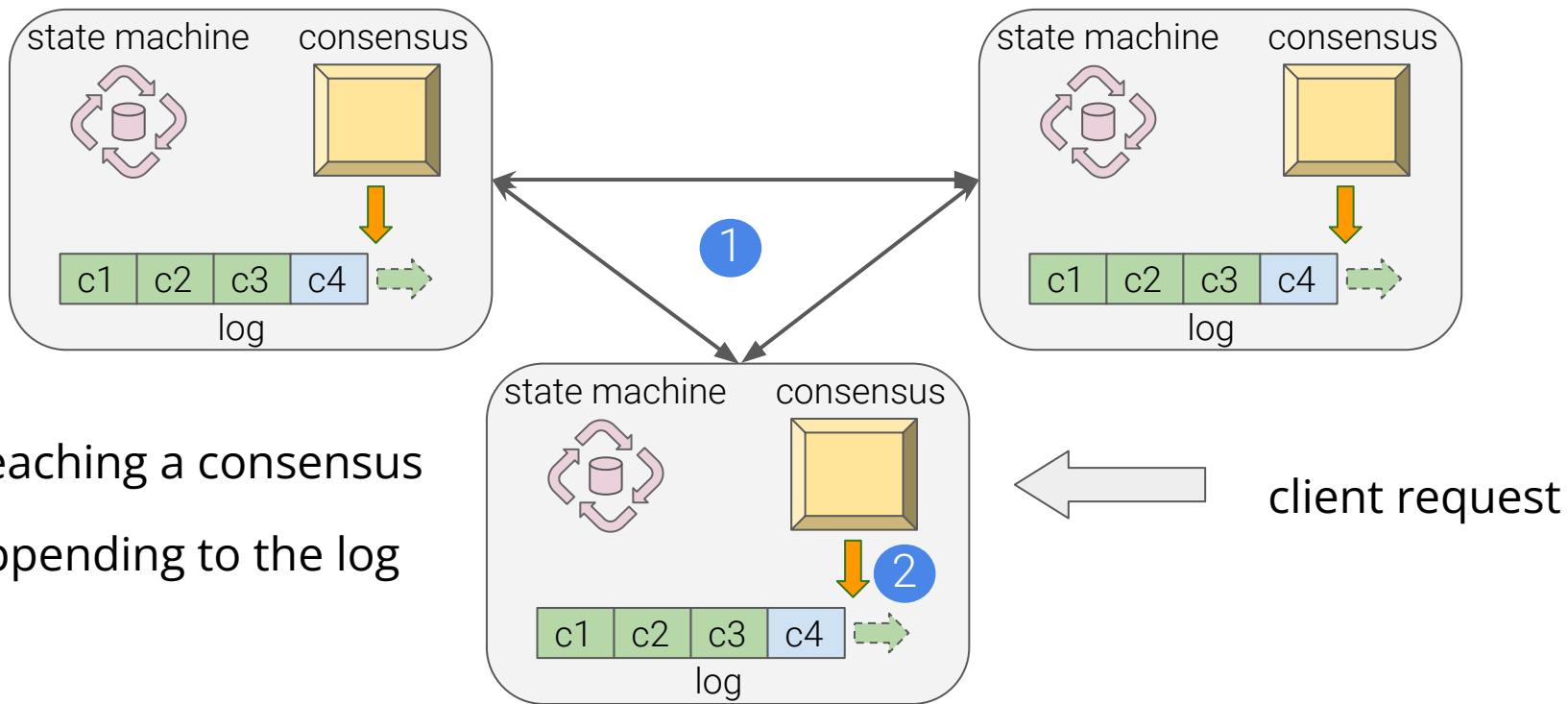


State-Machine Replication using Log



1 Reaching a consensus

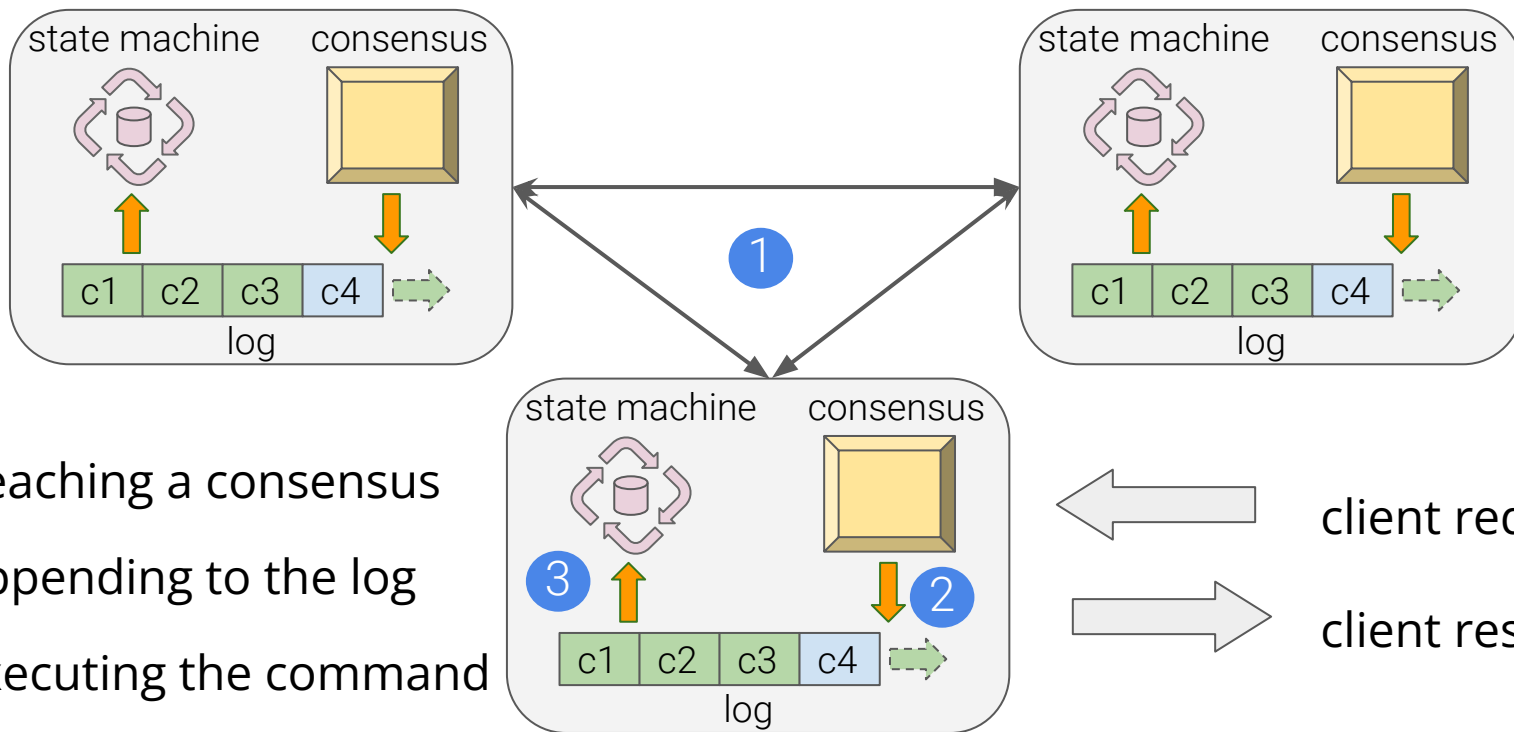
State-Machine Replication using Log



1 Reaching a consensus

2 Appending to the log

State-Machine Replication using Log



Corfu: A Shared Distributed Log over Flash

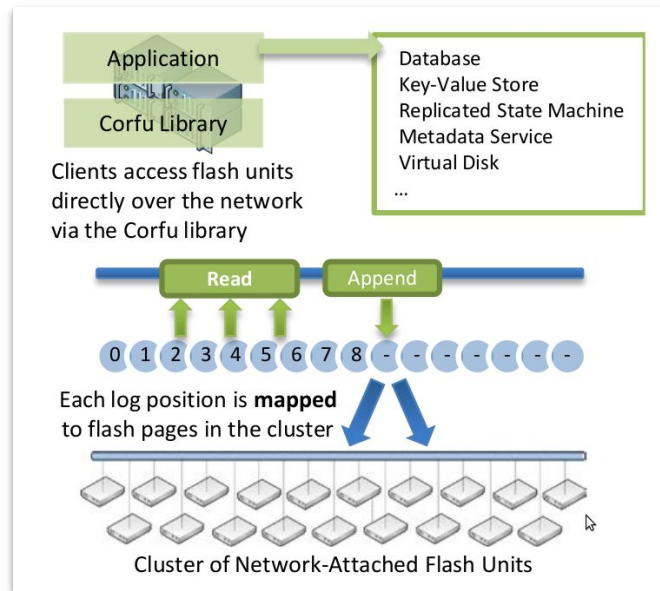
Seems like a perfect match to put Flash + Log together

A shared log abstraction over distributed flash

The log can be used as an ordering, arbiter

Using log one can build **a strongly consistent** systems

- *What is a strongly consistent system?*
- *Why do we need a strongly consistent system?*



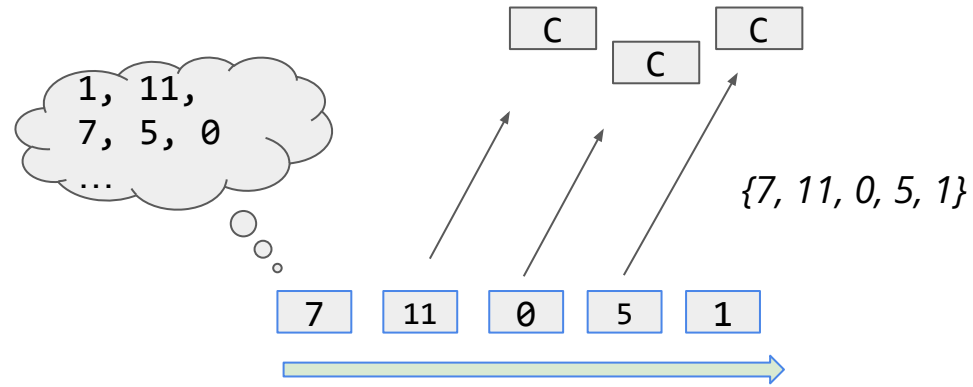
Why a Single Shared Log?

A single shared log allows multiple entities in a distributed system to access **the total order**

Easy reasoning about the order and program semantics - **Strong Consistency**

A single Shared log has been used before in DBs, shared file systems

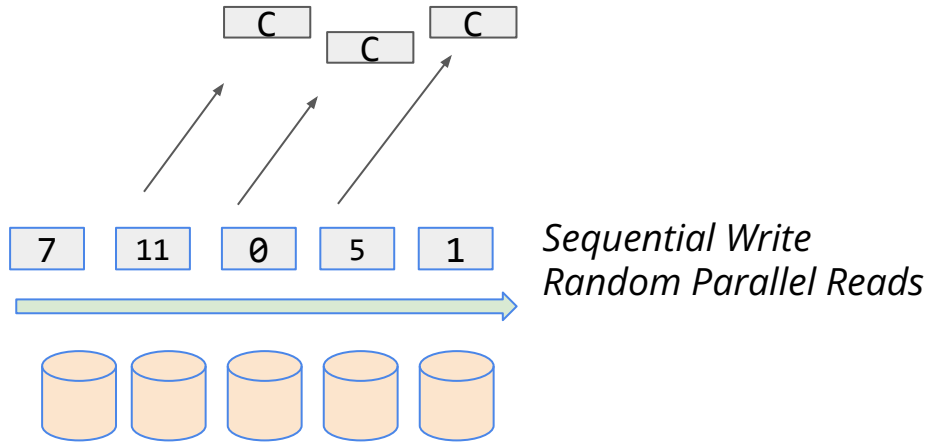
What challenges you might think when designing a shared log system on top of disks, that a flash might help with?



All the client would see some total order of operations - but always the SAME order

Strong consistency: *all accesses are seen by all parallel clients in the same order (sequentially)*

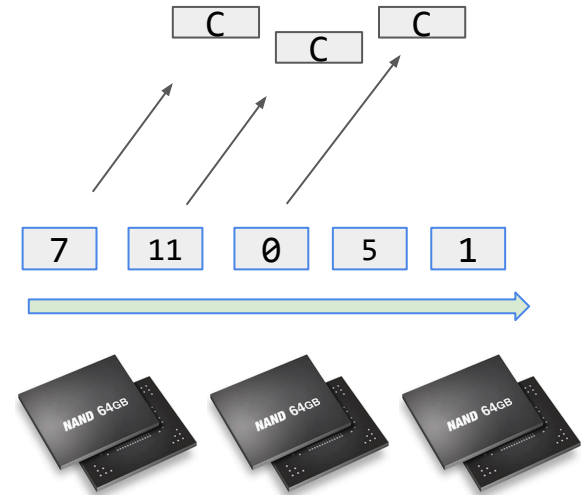
Logs on Disk vs. Flash



Poor bandwidth with concurrent accesses, lots of seeks, will destroy write performance too

How did LogFS solve this problem?

They made a case that reads will be mostly served from large DRAM cache. Disk mostly need to serve large sequential writes.



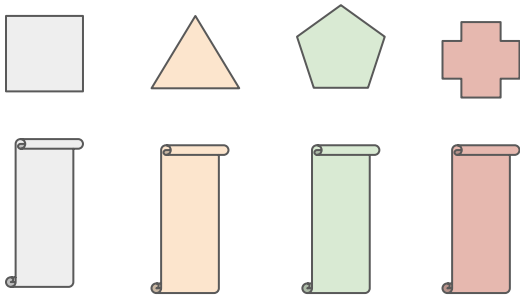
Parallel Concurrent reads
Absolutely not a problem :)

Sequential writes good :)

Why a **Single** Distributed Log?

Why cannot we store the log in a single machine → Failure Handling

Should all object go in a single log? Give objects their own private log of updates and save these logs to individual flash device.



- Cannot provide strong consistency across objects
 - Transactions with multiple objects (no order)
- Performance bottleneck by flash devices
- Hot objects - lots of updates
 - The flash log device will wear out

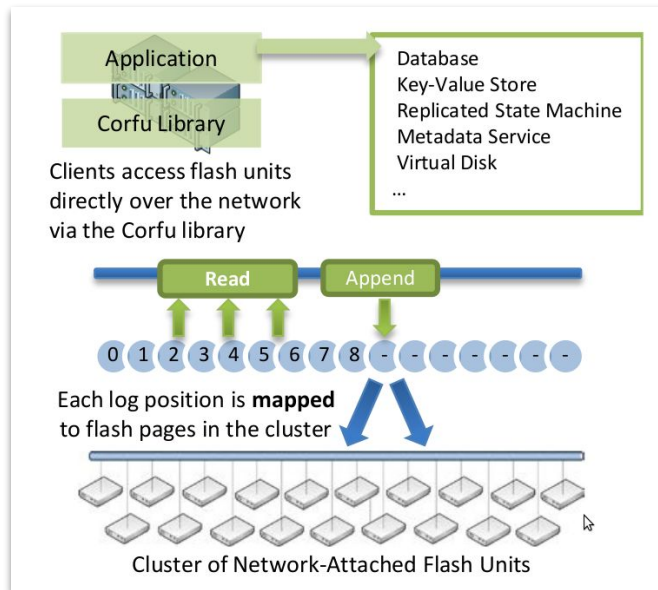
Corfu: A Shared + Distributed Log over Flash

Seems like a perfect match to put Flash + Log together

A shared log abstraction over distributed flash

Key design choices:

- Why a single shared log?
 - A total order based strong consistency model
- Why a single distributed log?
 - Storing log on a single flash drive would be bottleneck by its performance
 - [Flash] If partitioned per-key or per-SDD, then hot logs will wear off flash



Corfu Design and API

Build a shared distributed log abstraction over distributed SSDs

Keep the devices simple, hence, **client-centric/driven design** - SSD themselves do no do much (passive storage)

A basic log API : append, and read

Two new calls: **trim** and **fill**

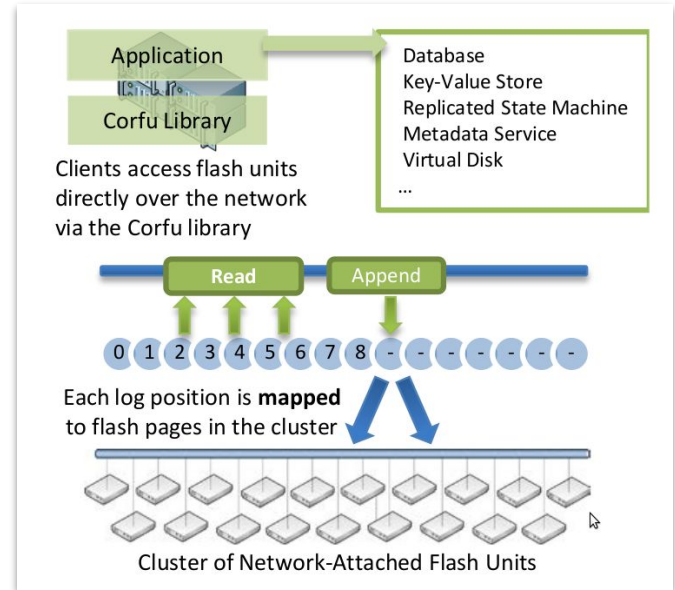
- **Trim**: like the trim command
- **Fill** : is to make an area filled with junk (to ensure it is considered written)

<i>append(b)</i>	Append an entry b and return the log position ℓ it occupies
<i>read(\ell)</i>	Return entry at log position ℓ
<i>trim(\ell)</i>	Indicate that no valid data exists at log position ℓ
<i>fill(\ell)</i>	Fill log position ℓ with junk

What is needed to Design Corfu?

1. On which flash device and flash page, the log offset "0" is stored?
2. Where is the current tail of the log? Or where does a client write next
3. What happens when there is a failure of a flash device(s)?

Any ideas?



How to find data location in Corfu

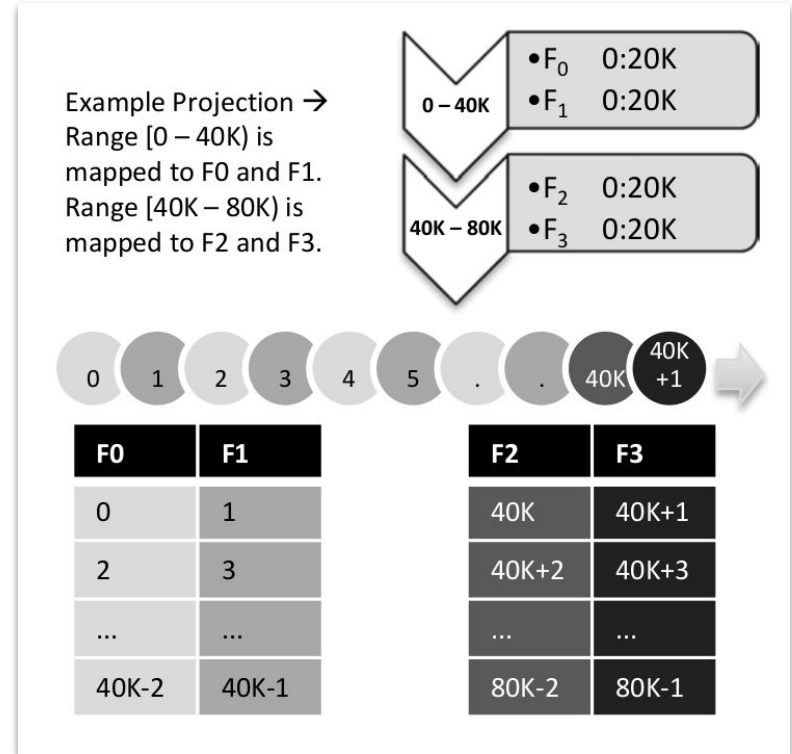
Uses simple deterministic extents projections that can be calculated by clients, here:

- 0 to 40K is stored on F0 and F1
 - Between F0 and F1 it can be **RR** or one after another
 - Equivalent of sharding or partitioning
- 40K-80K is stored on F2 and F3

When reading, client can calculate which flash page stores which log address

Each extent (0-40K, 40K-80K) are associated with a **replica flash devices for failure handling**

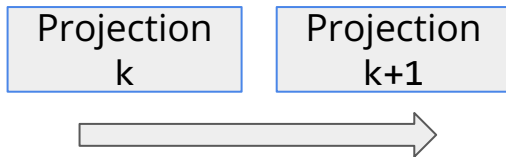
How does a client know which projection to use if there was a failure?



Changing the Projection

In case of an extent completion, SSD failure, or node crashes or joining - we need to change the projection

1. Give projection numbers
 - a. Every time project is changed, its version/epoch is changed
 - b. **Any client can change the projection**



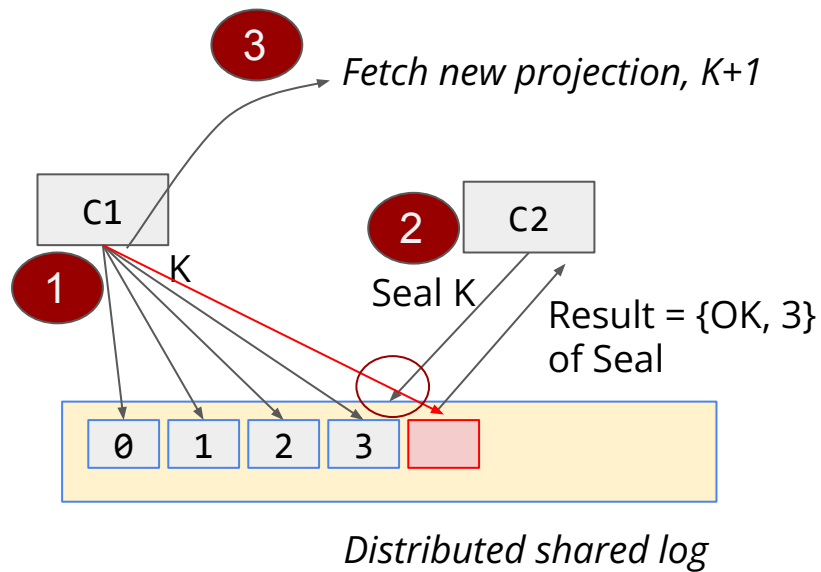
2. Store projections separately (vertical Paxos)
 - a. Disk volume (RAID)
 - b. Can use Paxos state to store reliably
 - c. Can use Corfu itself but ONLY with static projection



Is this enough for safely operate Corfu? How do clients know (apart from the one who changed it) a projection has changed? How do SSDs know a projection has changed?

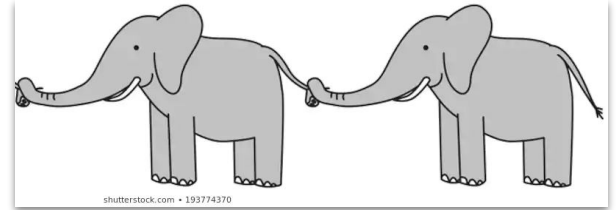
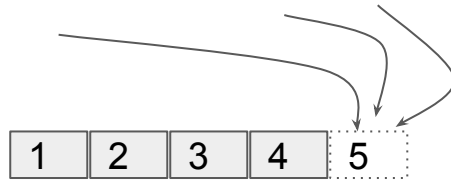
Additional Support from SSDs

1. Only **write-once** semantics on a page
 - a. Can be content or "fill junk" (see next)
2. **Read** function for written pages
 - a. Page written = OK
 - b. Page not written = Not Written
 - c. Page trimmed (in case of GC) or Junk
3. Expose storage space as a logical contiguous address space
4. Support for a **seal** command
 - a. All I/O from previous generation are stopped and rejected
 - b. Hence, all client coordinate to find the ground truth about the projection generation

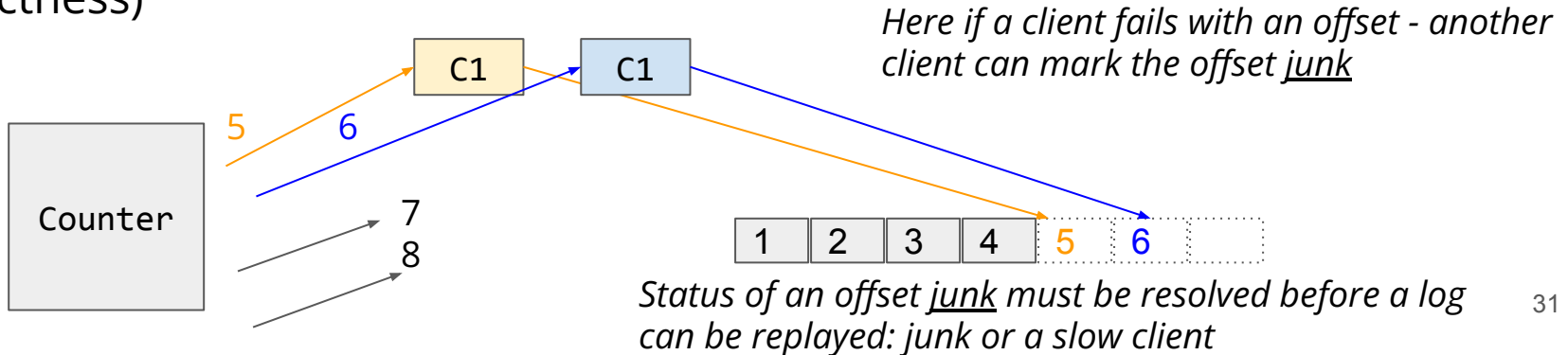


How does a client find the tail?

Design 1 : Client content for the tail only one will succeed (write-once)



Design 2 : Build a network sequencer (optimizer, not needed for correctness)

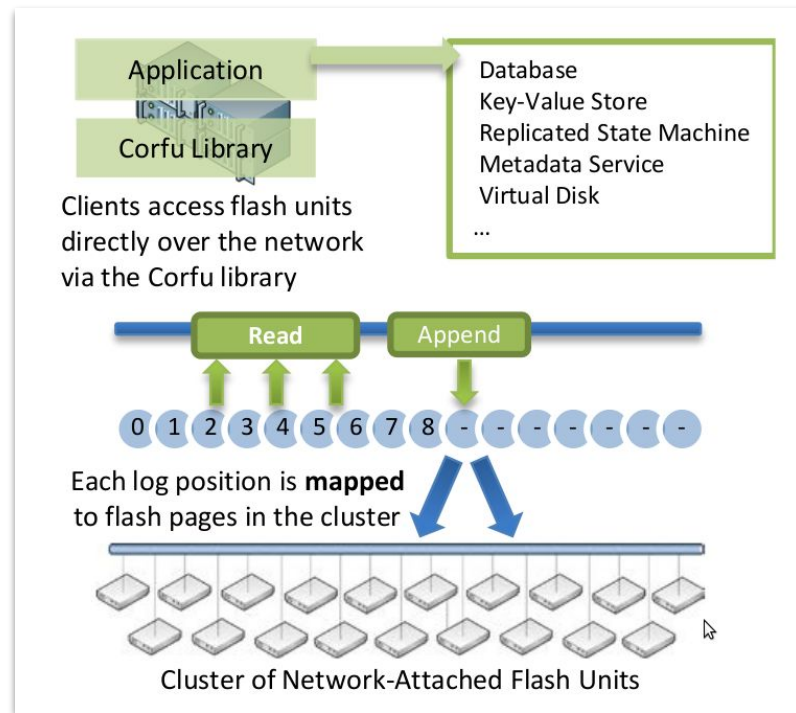


What Kind of Properties can Corfu Provide?

1. A distributed total order
2. A log for transactional updates
3. Log for play forward/undo updates
4. A consensus service
 - a. State Machine Replication (SMR)

You can build

1. Key-Value Store
2. Databases
3. File Systems
4. Fault Tolerant Data Structures



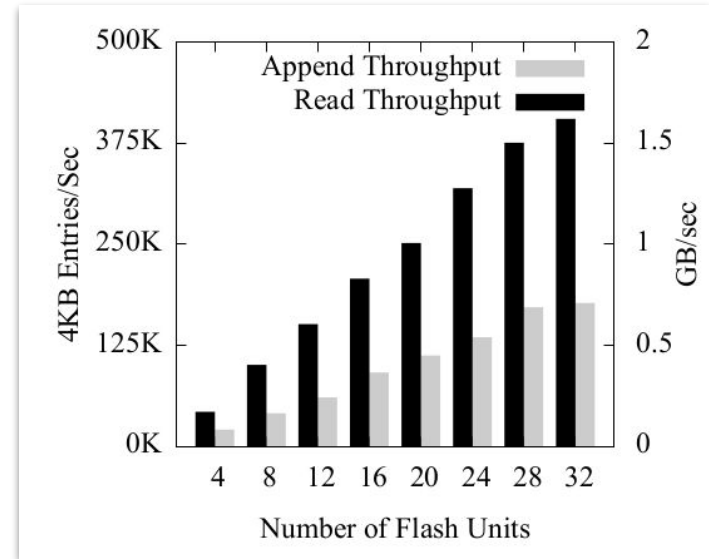
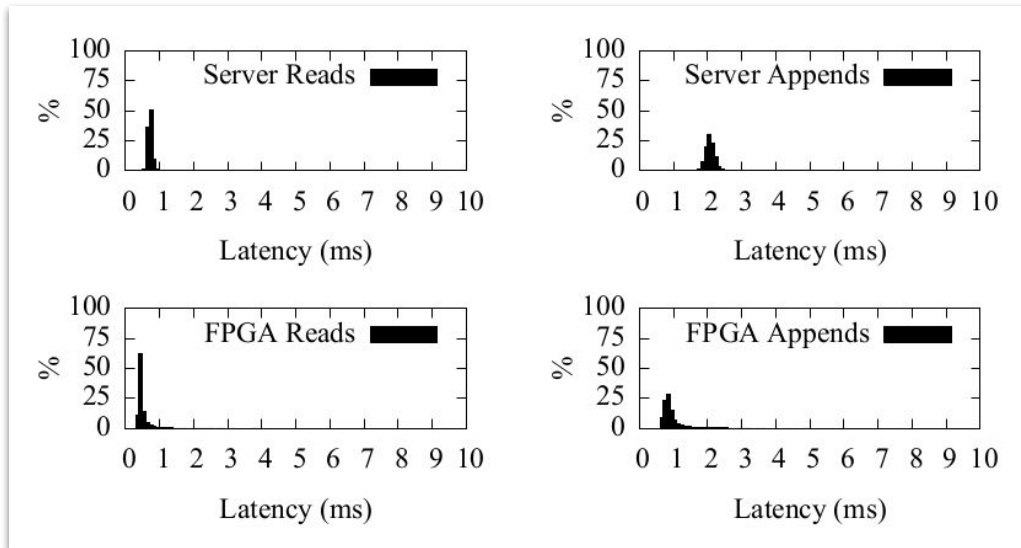
The Unique Role Flash Plays Here

1. The ability to provide **large random, parallel read bandwidth** from the log
2. The ability to build the log abstraction which can provide **single point of writing at the tail**
3. The ability to seal / trim log - **matches nicely with what SSDs do internally**

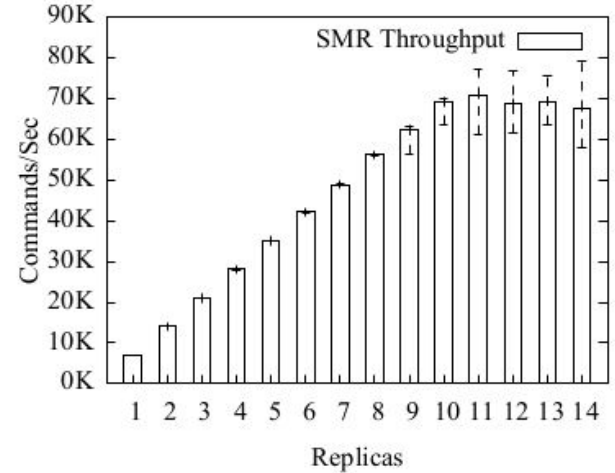
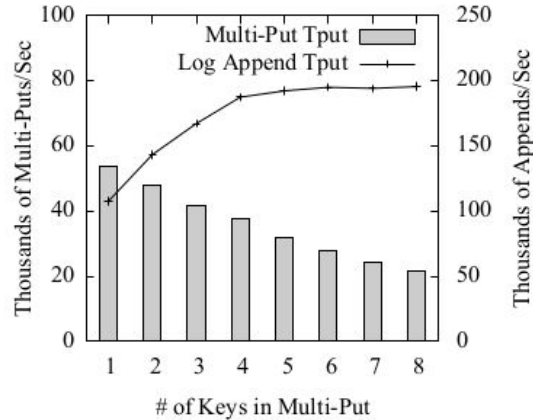
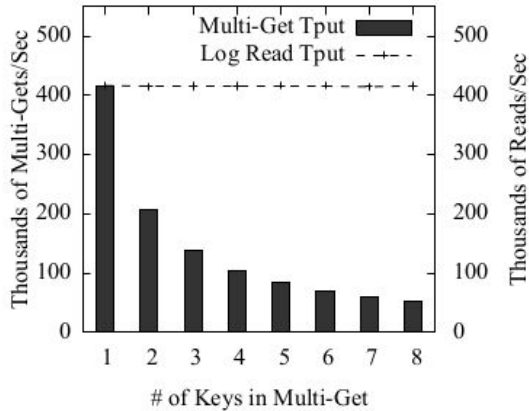
You can potentially build Corfu with disks as well, but the performance will not be good. In that case the application need to partition data to perform at the scale (partition = parallelism), thus sacrificing “C” from the CAP theorem

Corfu Performance

(i) Server-attached SSDs; (ii) FPGA-attached



Corfu Applications



- Atomic, multi-keys puts and gets for KV
- Scalable SMR implementation with replication
 - Typically very difficult to get it right!

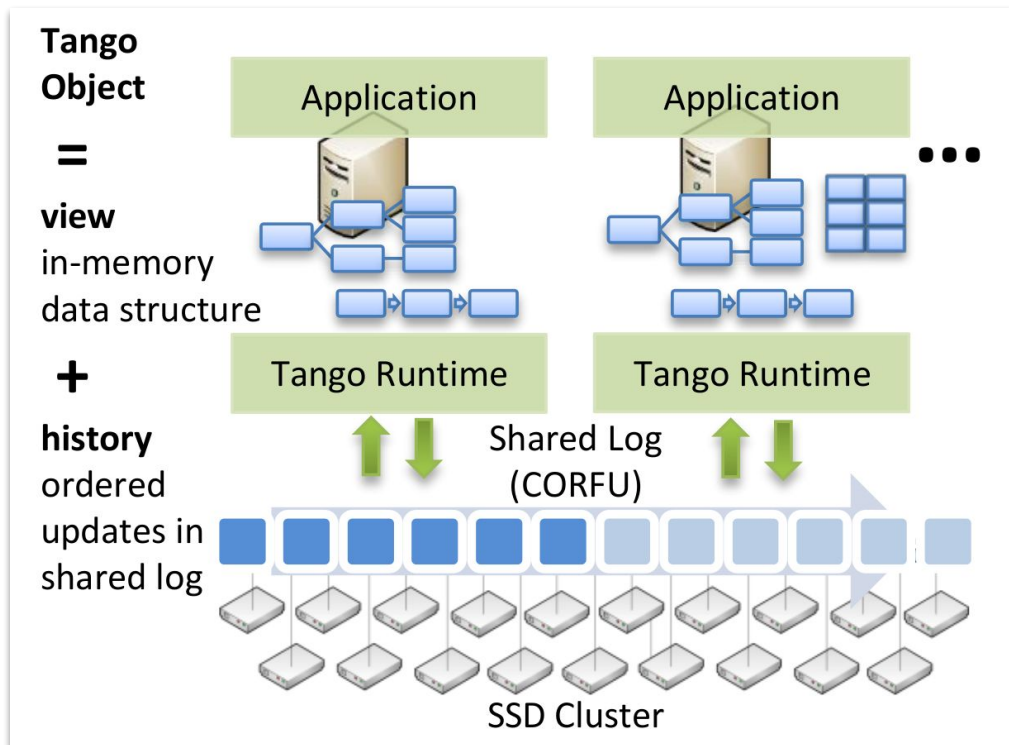
Tango: Distributed Data Structures over a Shared Log

Tango is any arbitrary object which has

- In-memory view
- History of updates stored on the shared corfu log

Corfu provides:

- Persistency
- Consistency
- Atomicity and fault tolerance



Tango Objects

```
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 ID

Attached Tango Runtime

Actual object state

Upcall function called from Tango
for state update

Write, that takes an opaque buffer and
"append" in Corfu via update_helper

Read, uses query_helper to "read"
new appends from the log. It can
"check" if the tail of the log has
moved and then replay the changes
from it

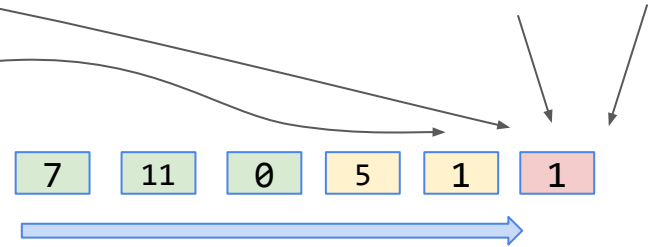
Tango Objects

```
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;
    }
}
```

Consistency: total order from the log → for single object linearizability

Durability: if the machine crashes, reconstruct the object on another machine

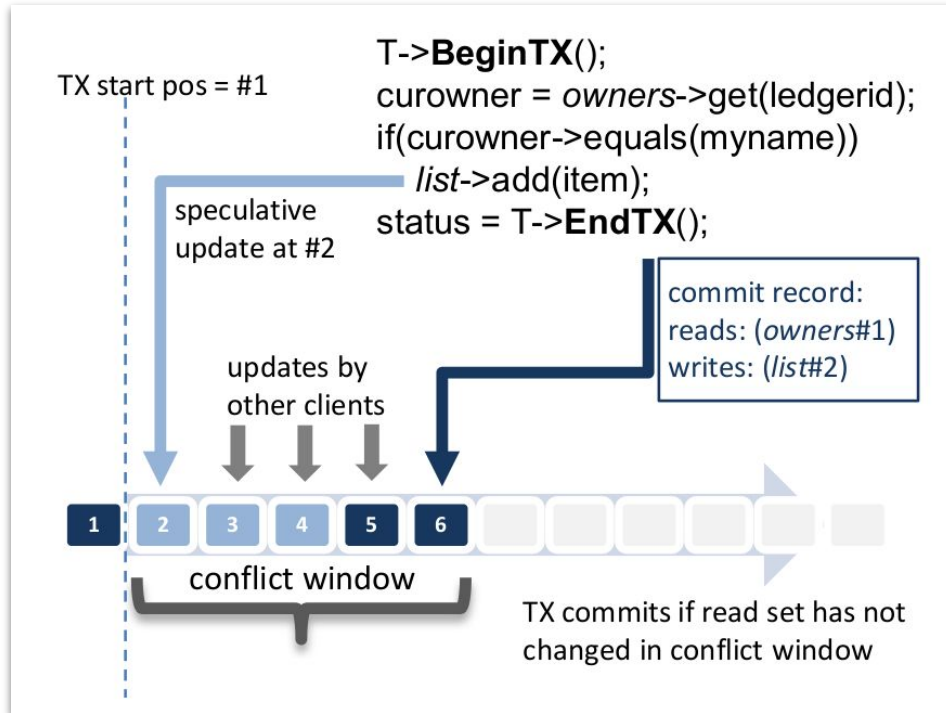
History: all versions of the objects are accessible from the log



Corfu distributed shared log

Multiple Objects and Transactions

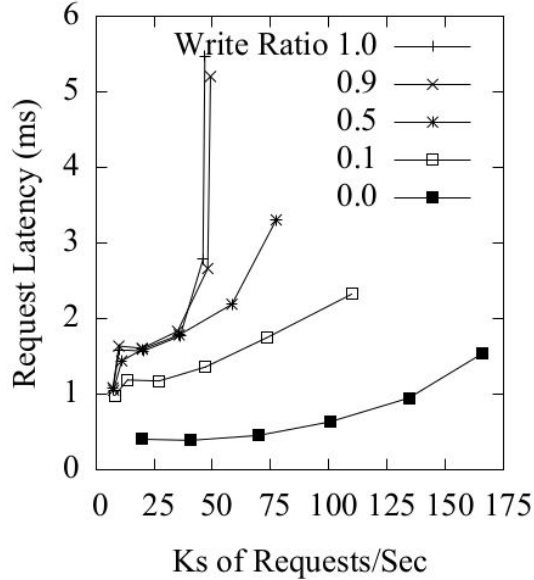
How do we atomically update **multiple objects** at the same time: Transactions!



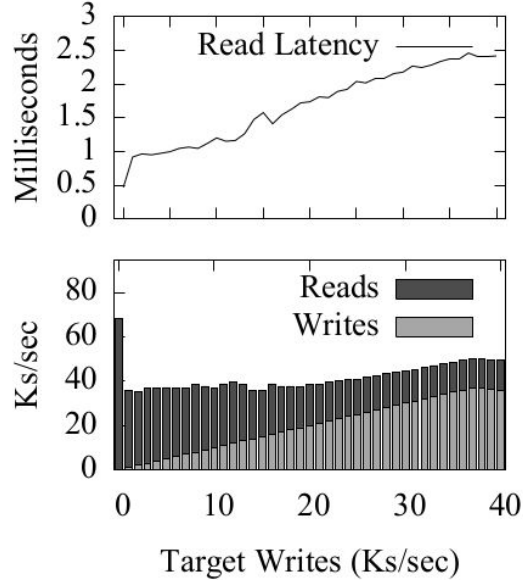
1. Mark the starting block for the transaction -- speculative commit
2. Keep track of read/write sets
3. Check at the time of commit any conflicts with the read/write set

When other machines see the speculative commits for objects, they buffer it internally and only make it visible (Atomicity) once they encounter a commit block

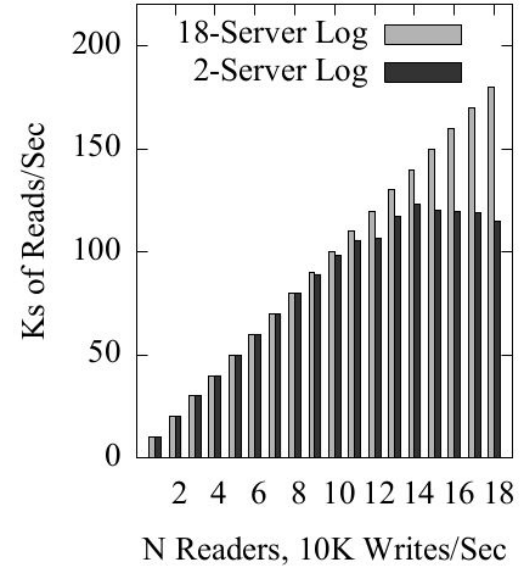
Performance: Scalability (Views, or copies)



Single view



Two views



Multiple views

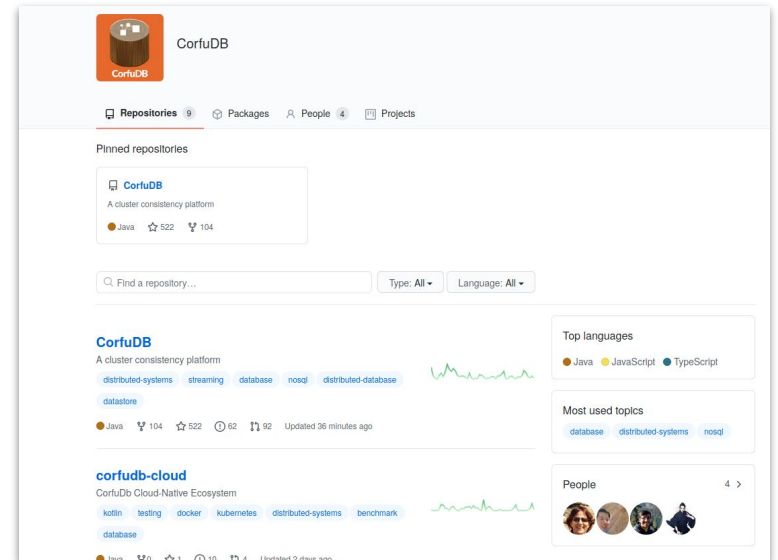
Corfu + Tango

Very powerful way of building fault tolerant data structures

Provides fault tolerance, scalability, ordering, transactions -- all in a single system

Shared distributed log is very powerful

Shared distributed log can be build efficiently using unique properties of flash storage



What you should know from this lecture

How logs are used

How does a log help in distributed settings (what is total order and why is it important)

What are Corfu and Tango

Why building a shared, and distributed log (Corfu) make sense

How does flash uniquely help in building of a shared and distributed log

What are the Corfu API calls, projection, and what are the purpose of seal and trim commands

How does Tango provide distributed fault tolerance data structures on Corfu

How does Tango provide multi-object transactions on the Corfu log

Further References

- Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: a shared log design for flash clusters. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12). <https://dl.acm.org/doi/10.5555/2228298.2228300>
- Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: distributed data structures over a shared log. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). Association for Computing Machinery, New York, NY, USA, 325–340. DOI:<https://doi.org/10.1145/2517349.2522732>
- Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: fast remote memory. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14). USENIX Association, USA, 401–414.
- David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: a fast array of wimpy nodes. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). Association for Computing Machinery, New York, NY, USA, 1–14. DOI:<https://doi.org/10.1145/1629575.1629577>
- Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 147–160.
- The RAMCloud Storage System, <https://dl.acm.org/doi/10.1145/2806887>
- Data Storage Research Vision 2025, <https://par.nsf.gov/servlets/purl/10086429>

Projection Changing Protocol

1. Client decides to seal P_i and install P_{i+1}
2. Send seal command to all SSDs that have a page mapped in P_i but not in the same position in P_{i+1}
3. Other clients know from the rejection that P_i is sealed
4. The client will know after receiving the ACK that P_i is sealed
 - a. It also knows the highest written log offset in P_i
5. The client installs the new P_{i+1} projection at the $(i + 1)$ th position in the projection log
 - a. If multiple clients start the reconfiguration at the same time, only one will succeed at at the $(i + 1)$ th position

