CS5412: WHERE DID MY PERFORMANCE GO?

Lecture XVIII

Ken Birman

Suppose you follow the rules…

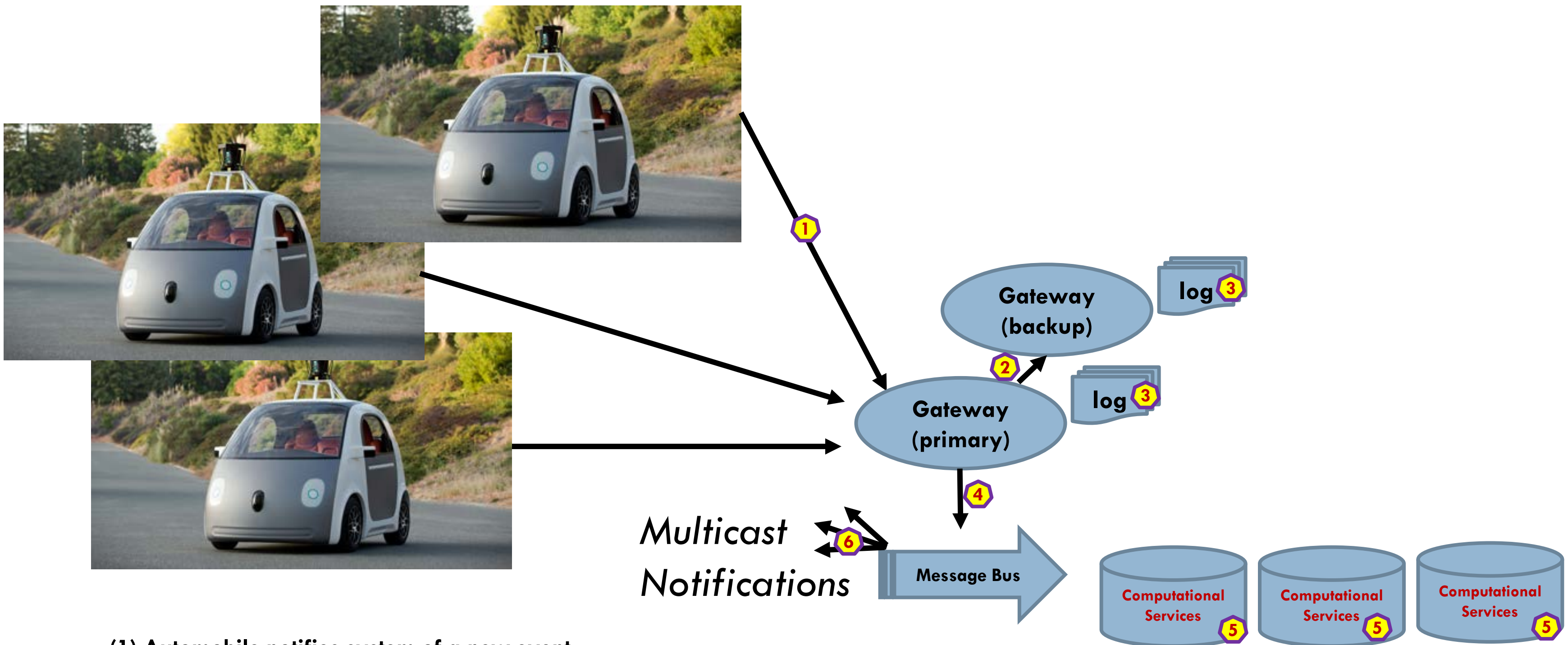
- You set out to build a fairly complex large-scale system for some kind of important task
  - Maybe not as mission-critical as a power grid or an air traffic control system…
  - … but on the other hand, smart cars are a hot topic, and robots, and many of these play safety critical roles

- You use clean-room techniques, object oriented programming, cutting edge quality-assurance

… and when you are done, the system is slow as molasses!

- What makes complex systems so slow?
- How can we run complex solutions in cloud settings without paying a huge performance cost?

Example: A smart car platform



(1) Automobile notifies system of a new event

(2, 3) System gateway accepts event, logs it locally and to a backup node

(4) Message bus (DDS) used to notify computational services

(5) Services compute routes, recommendations, etc.

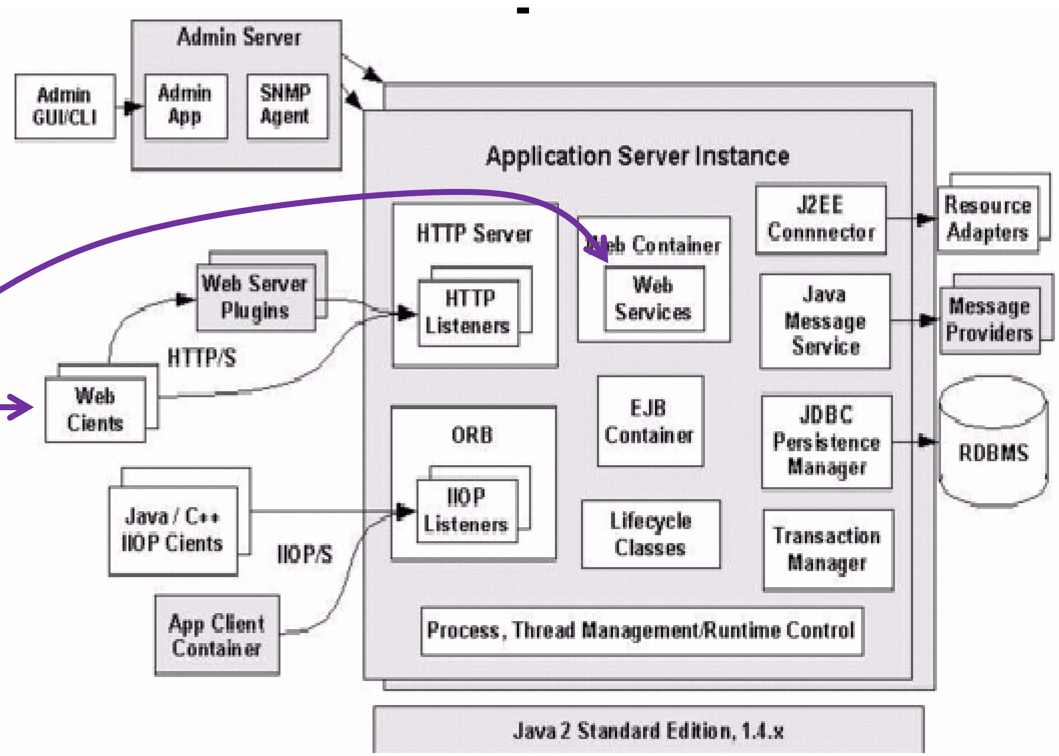
(6) Multicast used to update knowledge database in the vehicle and also in other vehicles impacted by the event

Componentized design

- There is a dominant trend towards building complex systems from "components", which can be entire programs and might be coded in different languages. Each element in this design is probably created from multiple components
- For example you could have a C# library used from C++/CLI and talking to other helper components written in C, standard C++ and Java, all on one platform
- This implies frequent "domain crossing" events, which also require serialization and deserialization

Componentized design

- This example comes from the ORACLE Java.com site
- Notice that in addition to **your code** there are many other helper components
- Every modern system looks like this!



Where would costs arise?

- Some events involve capturing images, video, lidar, etc. and might have large associated binary objects
- To send messages in an object oriented setting
  - *Need to "serialize" data into out-form, often costly and the out-form can be much larger than the in-form*
  - *Send it on the wire or log it to disk*
  - *Later on reception (or reading it) must de-serialize*
- Question: how many times might this occur in this kind of architecture?

Complex objects

- A first thing to realize is that most objects are fairly complex
- A lidar image captured by a smart car would have the radar data but might also include GPS coordinates, vehicle orientation and speed, altitude, angle of the sun, any filters being applied…
- So these have many fields that must be serialized

High costs of serialization

- We use the term ***serialization*** when a computing system converts data from its internal form to some kind of external form that can go on disk, on a network, or be passed to a component in a different language
- The external representation needs to be self-explanatory so that the receiving component can use it to build an object that matches what was sent
- A common style of representation is to use text and format it using XML, like a web page

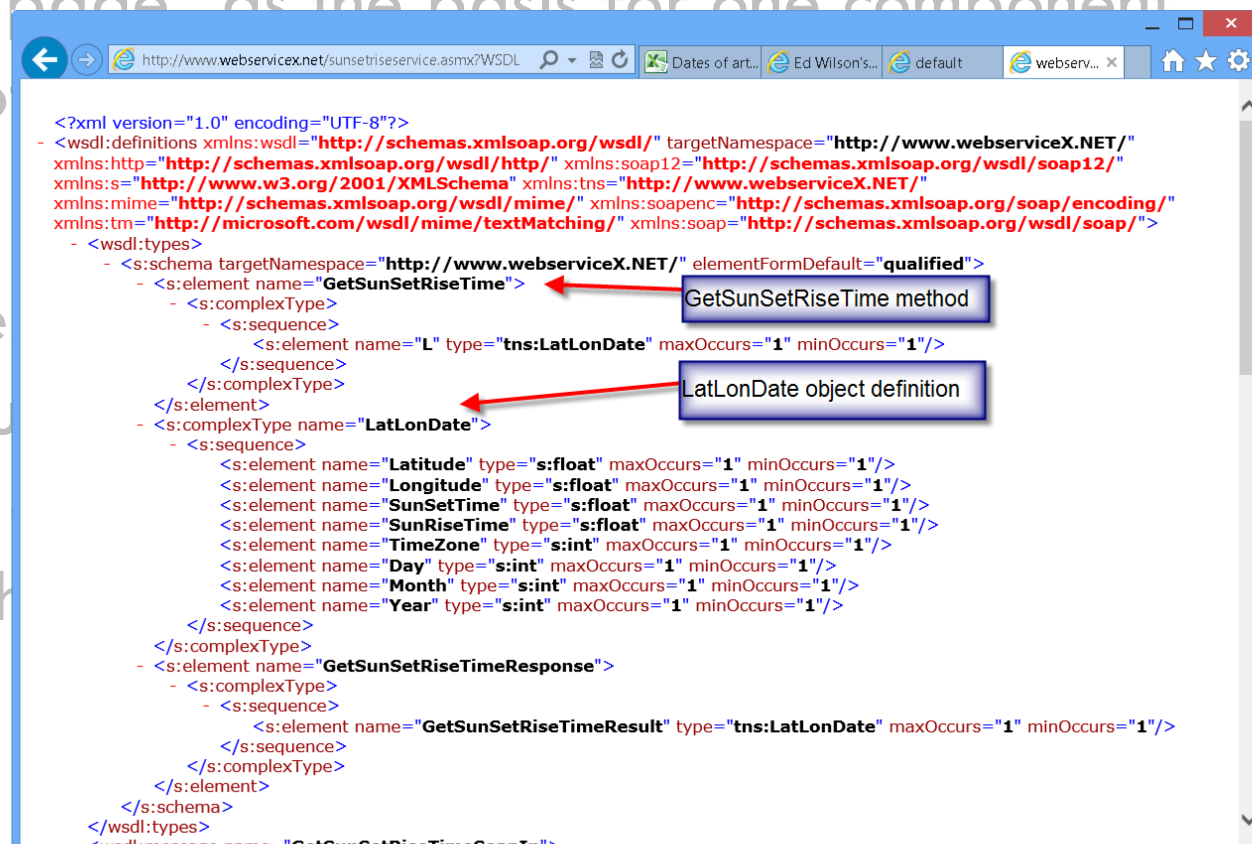
SOAP: Simple Object Access Protocol

- SOAP is a widely supported standard for using this kind of "web page" as the basis for one component accessing another component

- SOAP assumes an object to object style of interaction, but in practice a component could have many objects and can expose any of their **static** interfaces if the arguments are all **by value**.

SOAP: Simple Object Access Protocol

- SOAP is a widely supported standard for using this kind of “web page” as the basis for one component accessing ano
- SOAP assume interaction, bu many objects interfaces if th



SOAP representation

- The SOAP request format includes things like the service being accessed, the version number of the API that the caller was compiled against, the request being issued, and the arguments that were supplied to the request.
- Each argument could be a complex object, and it can include references to other objects as long as all of them are fully contained in a single "tree"
- XML nesting is used to represent inner objects

SOAP representation

- Later when the request finishes, the component can send back a reply
  - This is done in a similar manner, using a SOAP response object, again with a header and so forth
- SOAP type checks at every stage
  - If a type exception arises, SOAP always throws it on the caller side, not on the service side
  - This way if a server is upgraded, old clients that are launched accidentally won't crash it

What makes serialization costly?

- Generating the SOAP message can be surprisingly computationally expensive
  - Recursively we need to visit each element
  - For each one, make sure to output a “type description” and then emit the corresponding object
  - Any value types will need to be converted **accurately** into a text form. For example, we can’t lose floating point precision in a SOAP request/response, unlike when you print a floating point number on the console
- All of this makes messages big and slow to create

Why not use binary format?

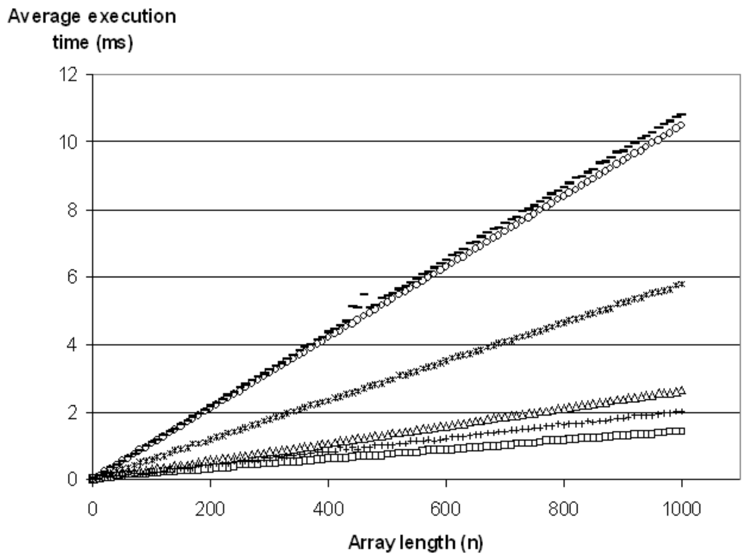
- Older systems often used binary representations and in fact there are many popular request/reply formats and representations

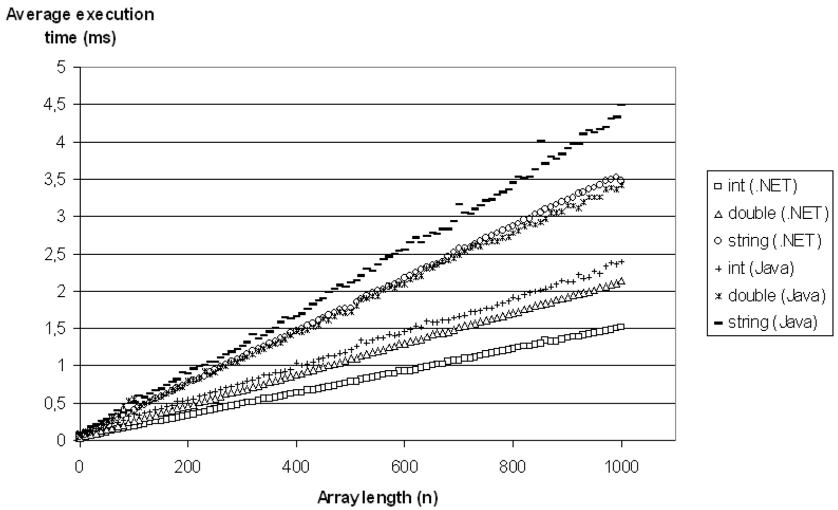
- The super efficient ones assume same data representations on source and destination: same programming language, version (patches included), hardware architecture and operating system

- But we can't always be so lucky. SOAP is universal.

Costs of serialization, deserialization

- CPU overheads to serialize (left) and deserialize (right), 10,000 times





Estimating the Cost of XML Serialization of Java Objects.
Imre, G. ; Charaf, H. ; Lengyel, L. IEEE Engineering of Computer Based Systems (ECBS-EERC), 2013.

Example: A beverage distribution center

- Suppose that we are just looking at a very simple case, like records sent from the cash-register at the Ithaca Imported Beverages company to the database it uses for inventory

- They specialize in imported beers, so consider costs of serialization of a "beer record"

- Example from M@X on DEV (www.maxondev.com)

Size overheads: A “beer” object

- C# example of a class that might describe a Belgian beer
- It has a brand, a level of alcohol, a brewery, etc.
- Notice that only some of these are fields with associated data and the data is very simple in this example!

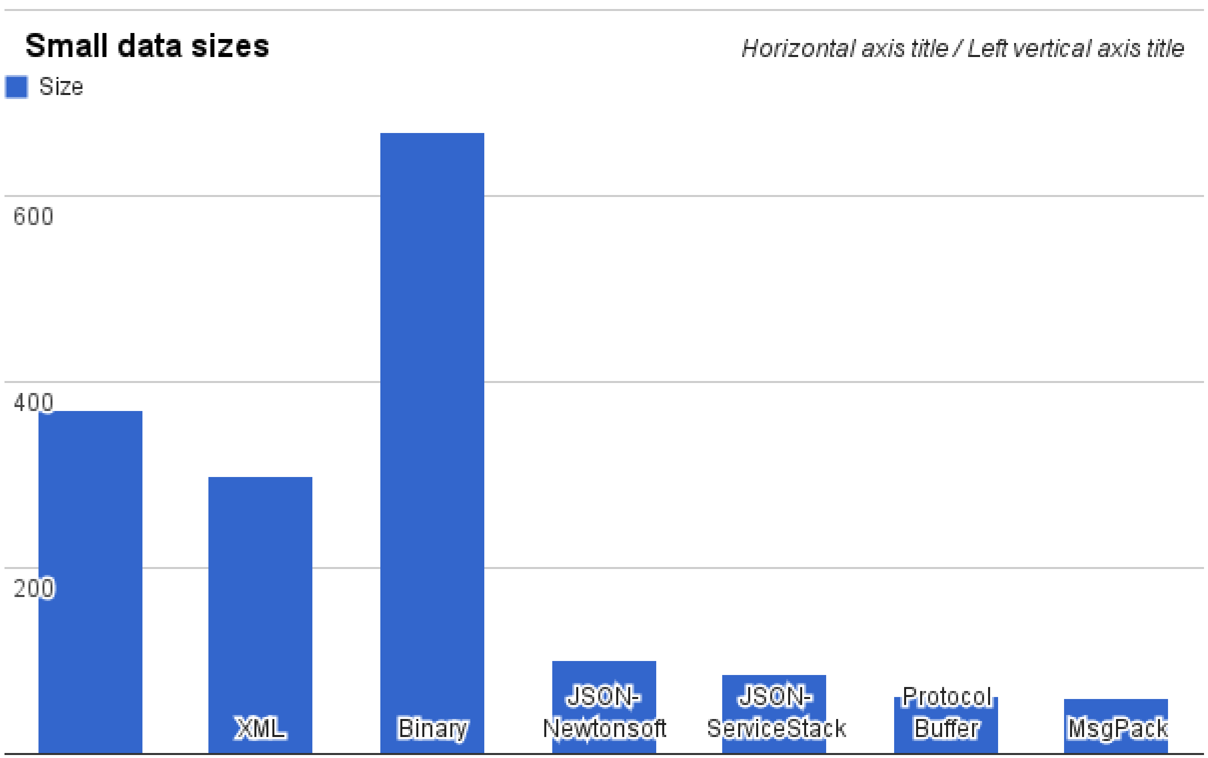
```
[Serializable, ProtoContract, DataContract]
21 references
public class Beer
{
    [ProtoMember(1), DataMember]
    1 reference
    public string Brand { get; set; }

    [ProtoMember(2), DataMember]
    1 reference
    public List<String> Sort { get; set; }

    [ProtoMember(3), DataMember]
    1 reference
    public float Alcohol { get; set; }

    [ProtoMember(4), DataMember]
    1 reference
    public string Brewery { get; set; }
}
```

Tabular summary of costs

- Space costs in bytes, time costs in ms

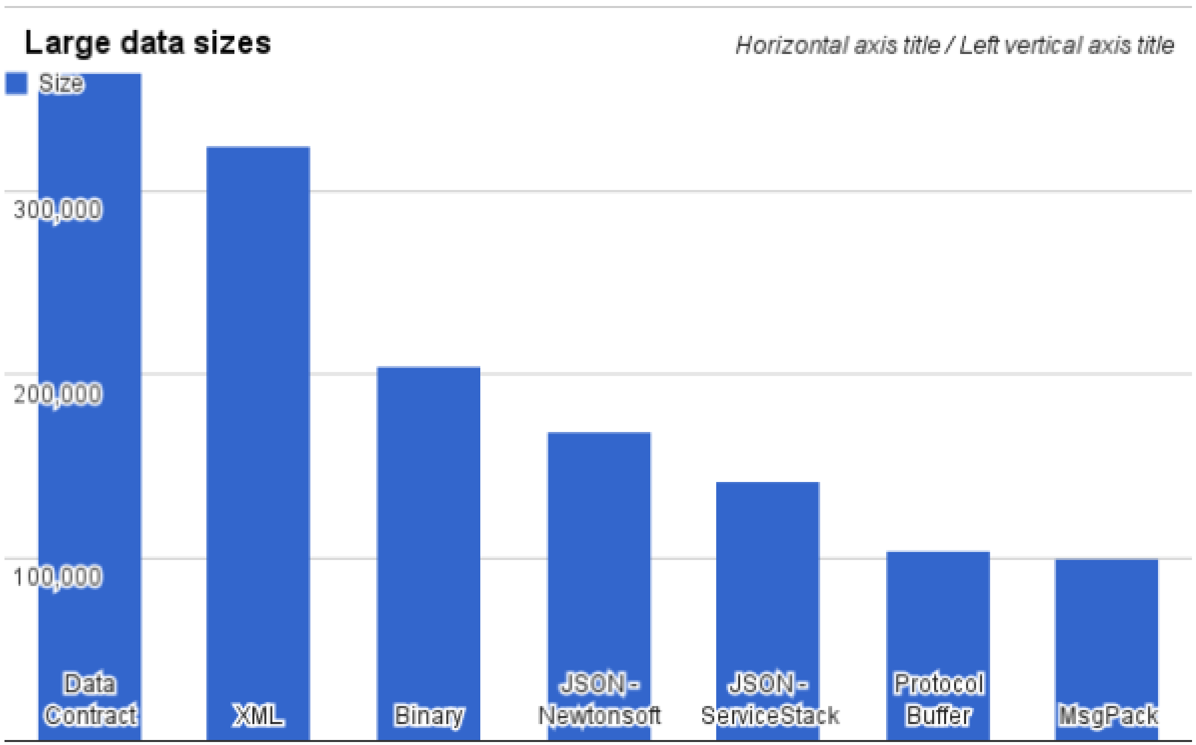
	Data Contract	XML	Binary	JSON - Newtonsoft	JSON - ServiceStack	Protocol Buffer	MsgPack
Size (Large)	364,299	323,981	204,793	168,429	141,863	104,191	99,670
Deserialize (Large)	11.469048	7.889384	19.39763	10.715157	5.731472	3.82069	6.778702
Serialize (Large)	4.443877	5.508091	13.700064	8.025799	3.559688	1.447036	1.431415
Size (Small)	370	298	669	102	86	62	61
Deserialize (Small)	0.012718	0.015977	0.019405	0.007171	0.00174	0.003883	0.002664
Serialize (Small)	0.004413	0.021897	0.021023	0.007081	0.003645	0.000989	0.000907

Time cost: Serialize a “beer” object

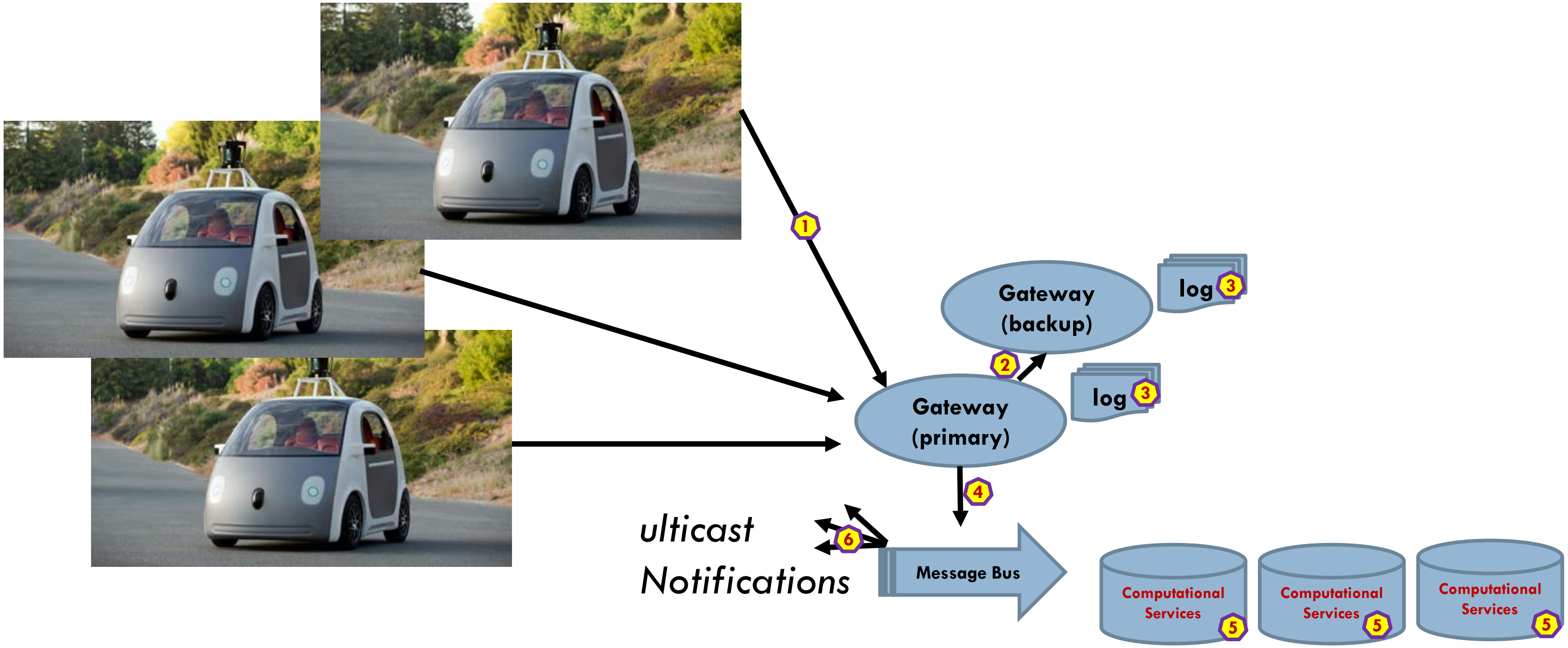


Time cost: List of all 1610 Belgian beers

http://en.wikipedia.org/wiki/List_of_Belgian_beer



How many such operations occur?



- We identified 6 steps, each requiring serialization/deserialization, but if elements are componentized, the total could be 5x or 10x more!

What can we do?

- Even binary serialization wasn't really so cheap

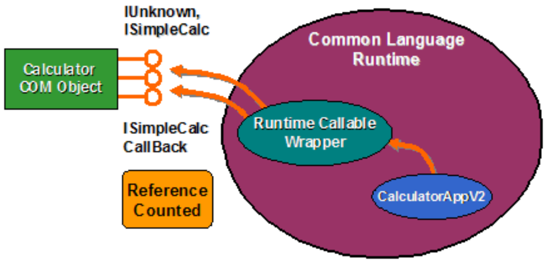
- The only thing that turns out to be cheap is to send very simple messages with very simple content, like "one string"

- So… can we magically transform our code into very simple code? Introducing… **logging**!

Key ideas: Very simple

- Write the large complex objects into a reliable log service, just once.
  - Logging means "append only, durable, file"
  - You **write it once**, can read it later
- Now we *substitute a URL for the large object.*
  - We could modify the application itself
  - Or we could create a "wrapper" for the object itself or for the libraries used in the application

Concept: A "wrapper"



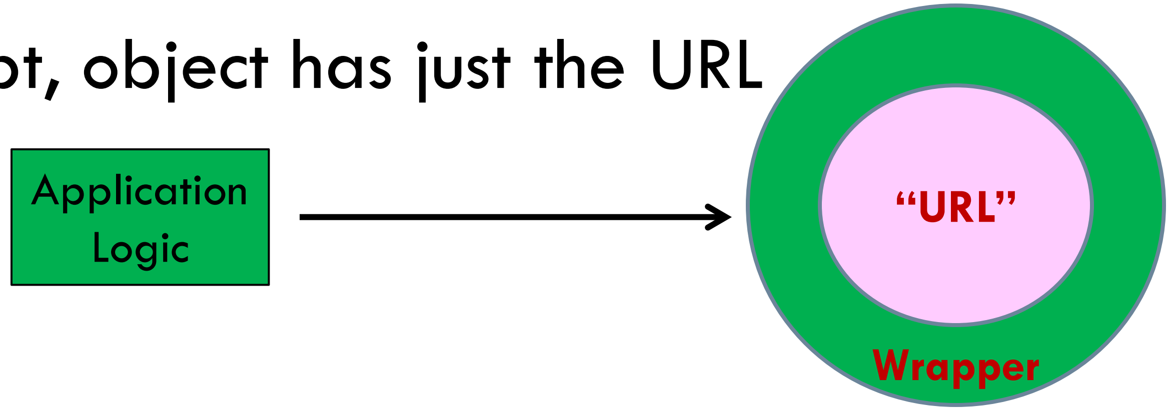
- Start with a complex application…
  you really don't want to modify it

- Identify some big objects it sends, and modify the setter/getter methods to first "memory-fy" it
  - If we have the URL but not the object, fetch the object
  - Then perform action as usual

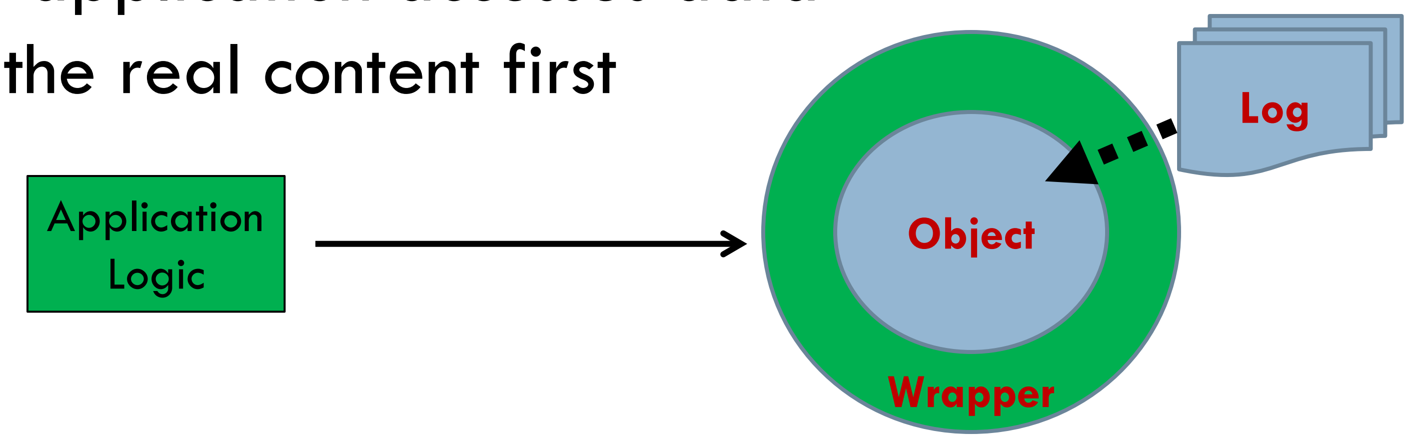
- A *lazy fetch!* **Question: why will this help?**

Concept: A "wrapper"

- On receipt, object has just the URL



- But if the application accesses data we load the real content first



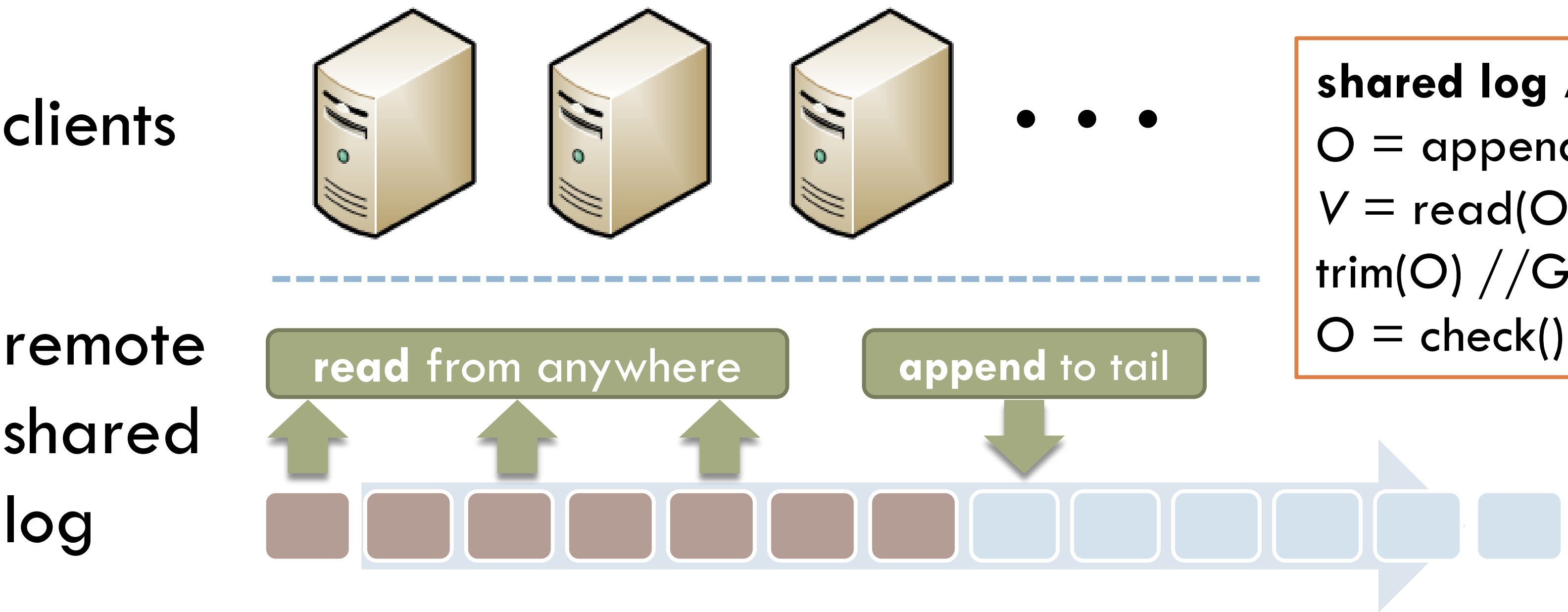
Can it be totally transparent?

- In many cases, a wrapper can completely hide the log from the real application
- But if the object is modified, then transmitted, we need to create a new logged version, and use a new URL for it.
- The log service won't allow you to modify a logged object, only to create "new" logged objects

Data center logging services

- This area was very ad-hoc for a while
- Then the Berkeley "log structured file system" was proposed. LFS was really popular.
- More recently, Corfu and Tango were introduced by Microsoft. These are logging services for situations where reliability and speed are paramount
  - The slides that follow are from Mahesh Balakrishnan, one of the team leaders for this project at MSR

The shared log abstraction



shared log API:

O = append(*V*)

V = read(O)

trim(O) //GC

O = check() //tail

clients can concurrently **append** to the log, **read** from anywhere in its body, **check** the current tail, and **trim** entries that are no longer needed.

Outline

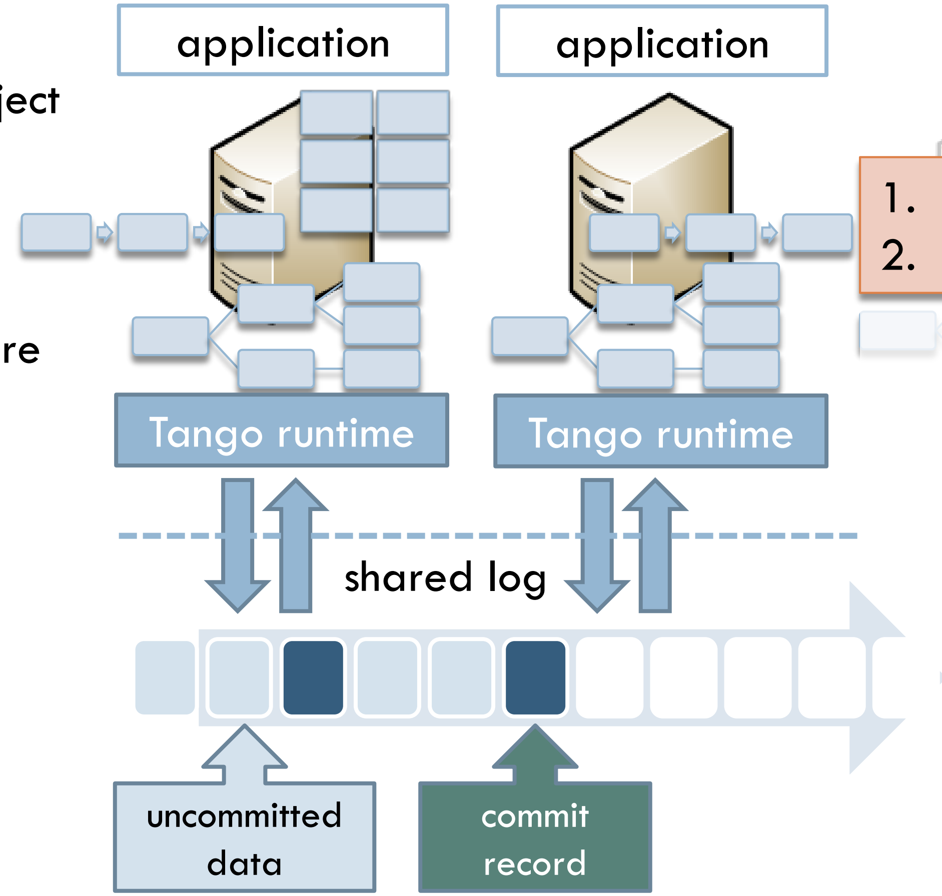
- *a shared log is a powerful and versatile abstraction.* **Tango** (SOSP 2013) provides transactional in-memory data structures backed by a shared log.

- *the shared log abstraction can be implemented efficiently.* **CORFU** (NSDI 2012) is a scalable, distributed shared log that supports millions of appends/sec.

- *a fast, scalable shared log enables fast, scalable distributed services.* **Tango+CORFU** supports millions of transactions/sec.

The shared log approach

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



1. Tango objects are **easy to use**
2. Tango objects are **easy to build**

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

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

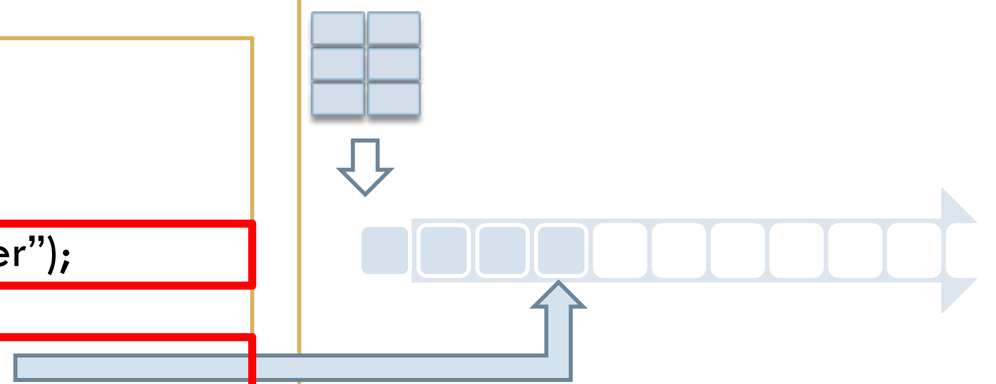
Tango objects are easy to use

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

example:

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

under the hood:

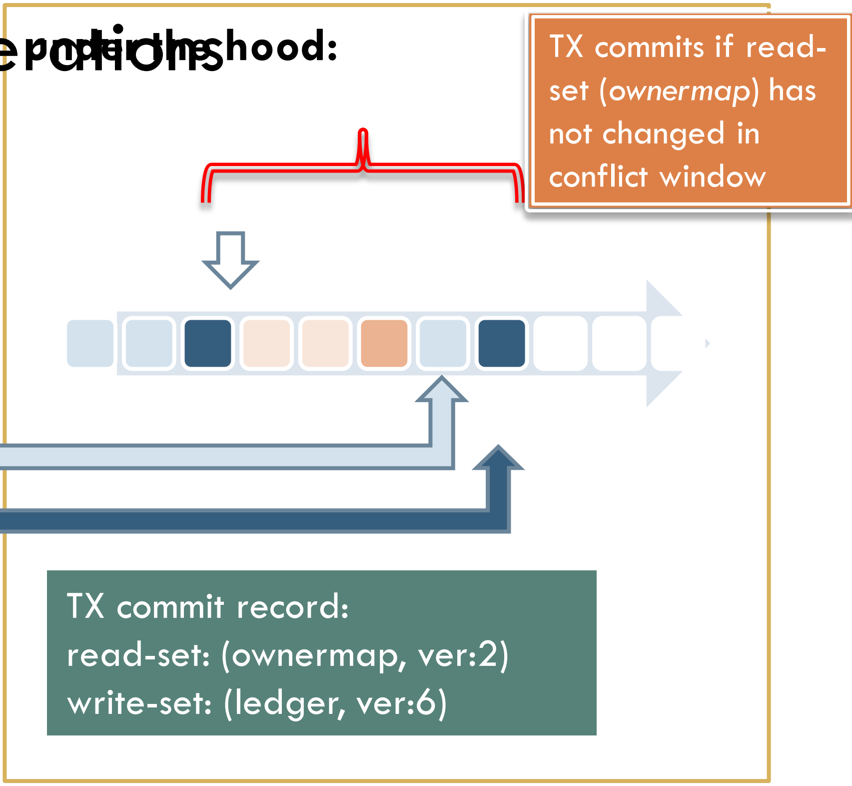


Tango objects are easy to use

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

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)]

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

invoked by Tango runtime on EndTX to change state

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

accessor: updates
t,
tate

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

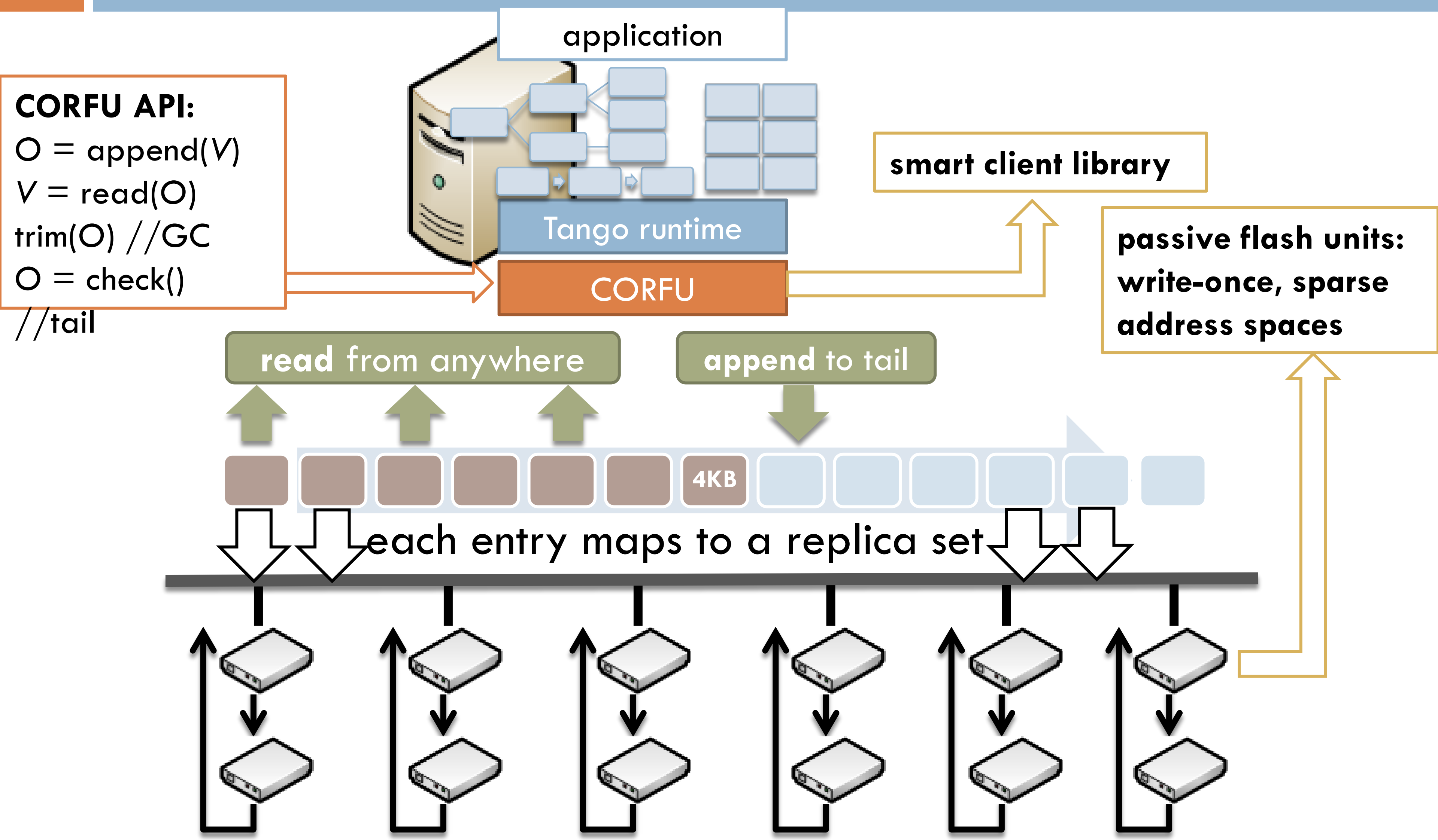
Outline

- *a shared log is a powerful and versatile abstraction.* **Tango** (SOSP 2013) provides transactional in-memory data structures backed by a shared log.

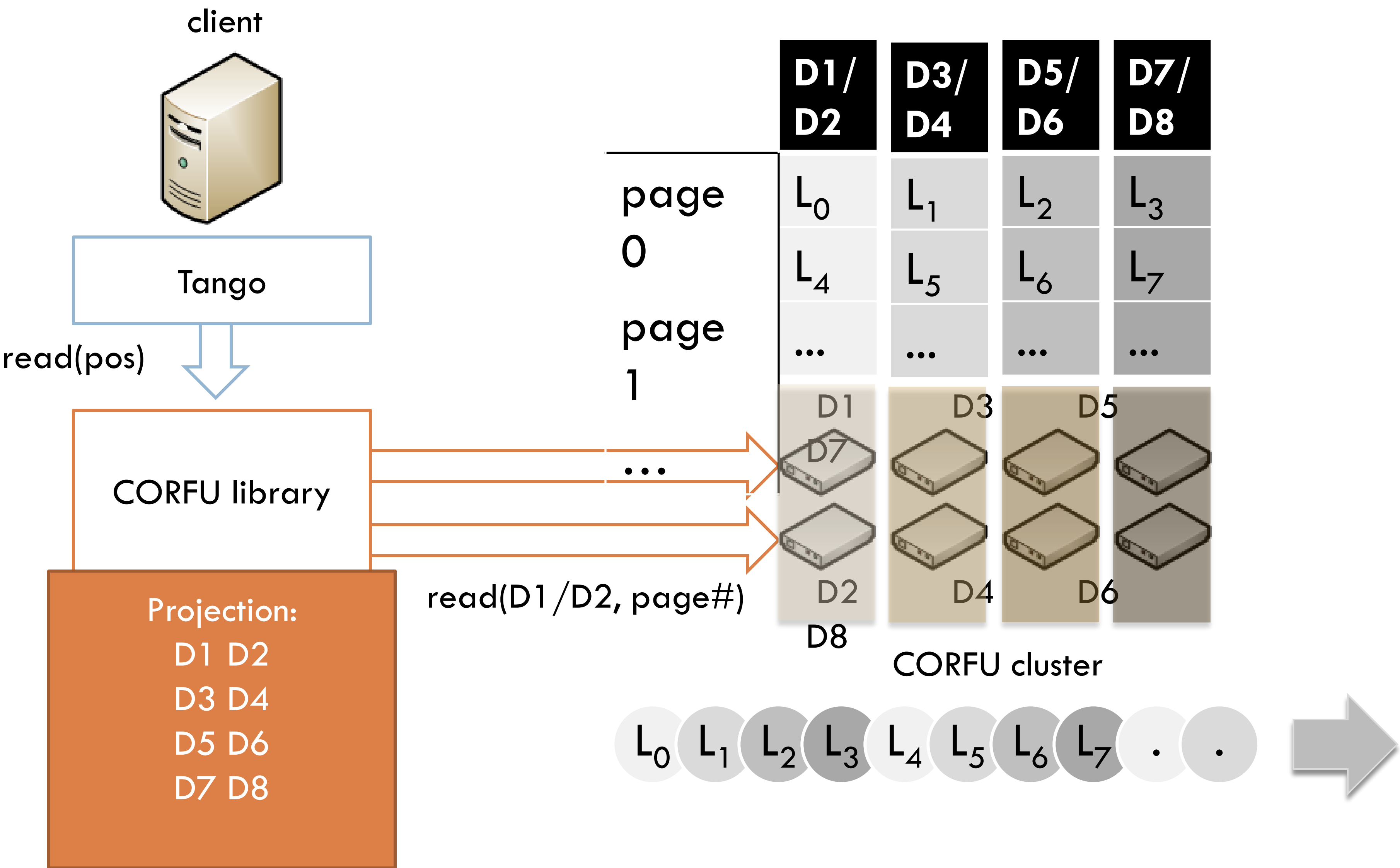
- *the shared log abstraction can be implemented efficiently.* **CORFU** (NSDI 2012) is a scalable, distributed shared log that supports millions of appends/sec.

- *a fast, scalable shared log enables fast, scalable distributed services.* **Tango+CORFU** supports millions of transactions/sec.

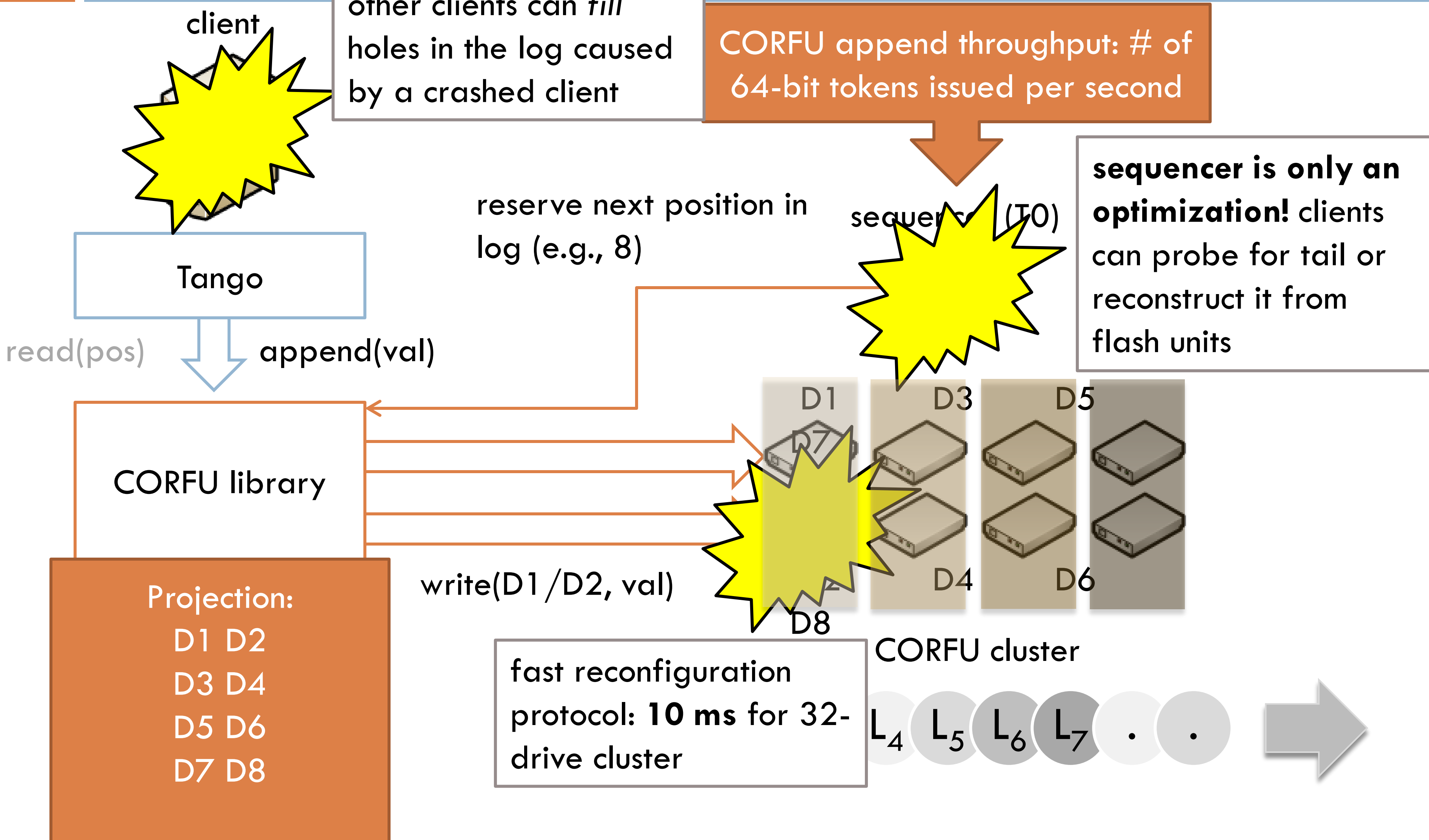
The CORFU design



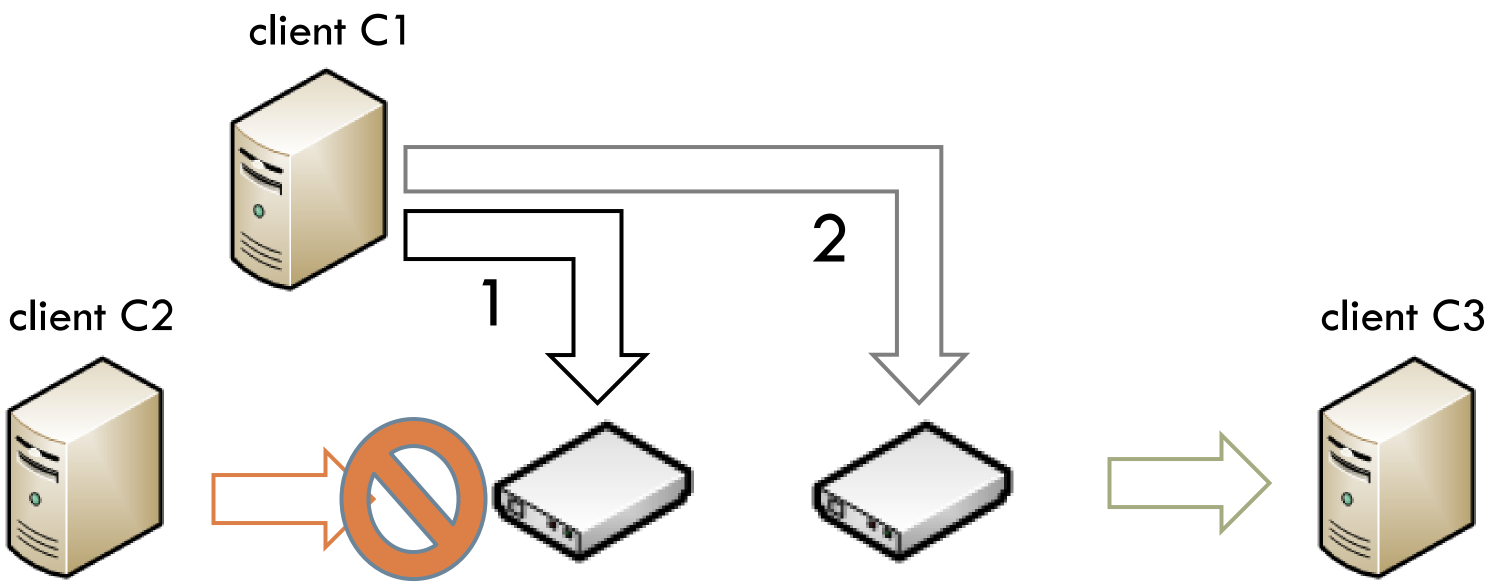
The CORFU protocol: reads



The CORFU protocol: appends



Chain replication in CORFU



safety under contention:
if multiple clients try to write to same log position concurrently, only one wins
writes to already written pages => error

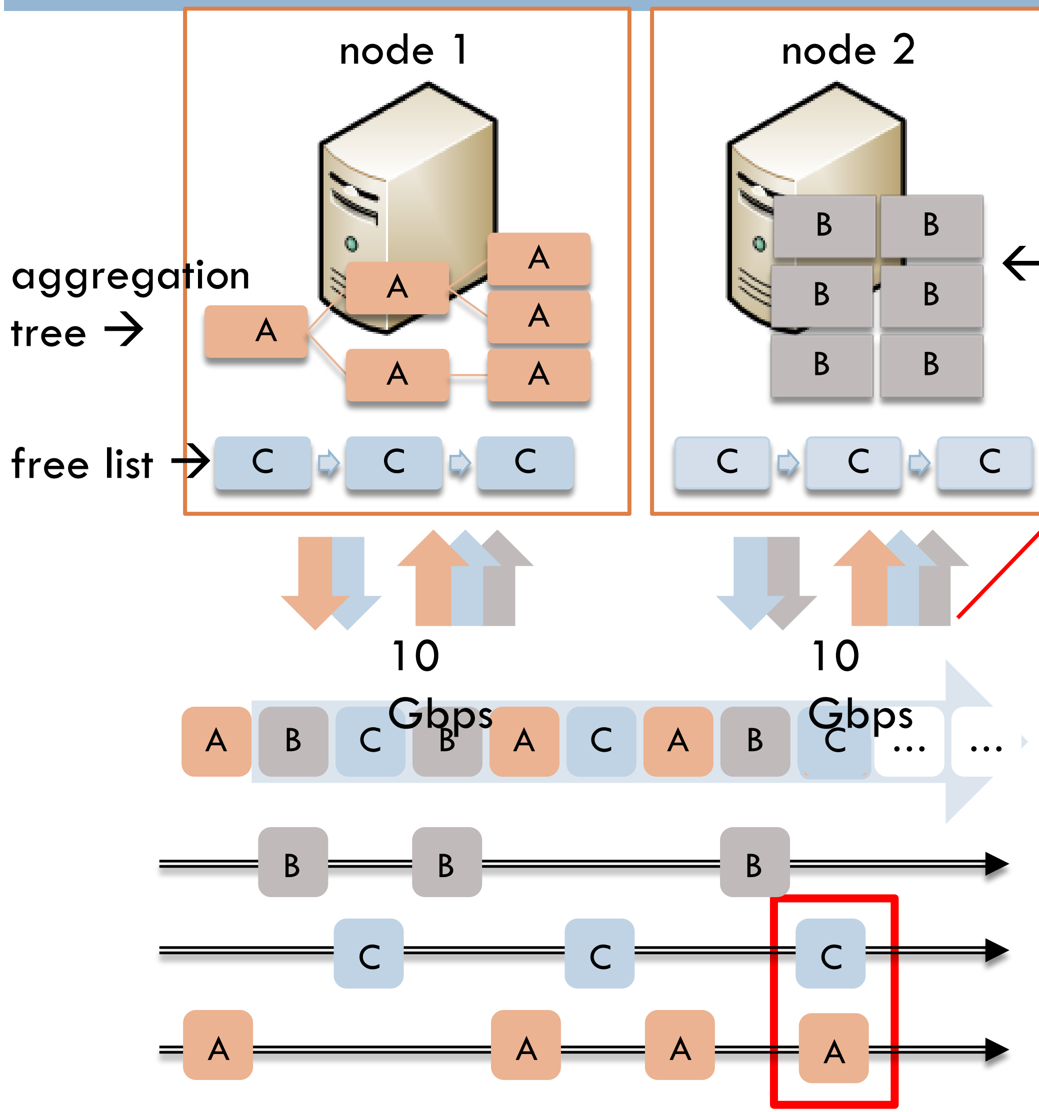
durability:
data is only visible to reads if entire chain has seen it
reads on unwritten pages => error

requires **write-once** semantics from flash unit

Outline

- *a shared log is a powerful and versatile abstraction.* **Tango** (SOSP 2013) provides transactional in-memory data structures backed by a shared log.
- *the shared log abstraction can be implemented efficiently.* **CORFU** (NSDI 2012) is a scalable, distributed shared log that supports millions of appends/sec.
- *a fast, scalable shared log enables fast, scalable distributed services.* **Tango+CORFU** supports millions of transactions/sec.

a fast shared log isn't enough…



the playback bottleneck:

clients must read all entries →

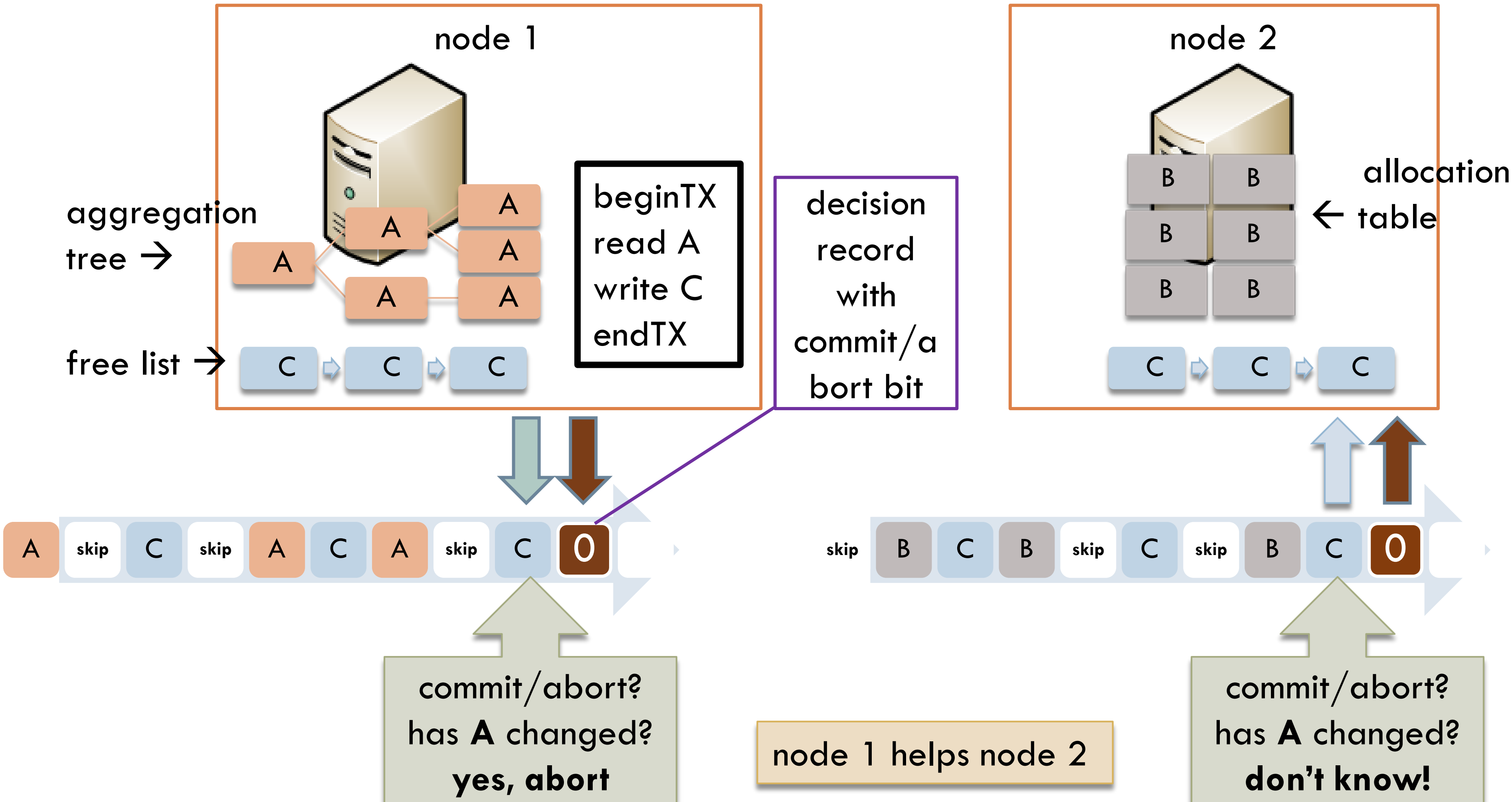
inbound NIC is a bottleneck

solution: **stream** abstraction

- *readnext(streamid)*
- *append(value, streamid1, …)*

each client only plays entries of interest to it

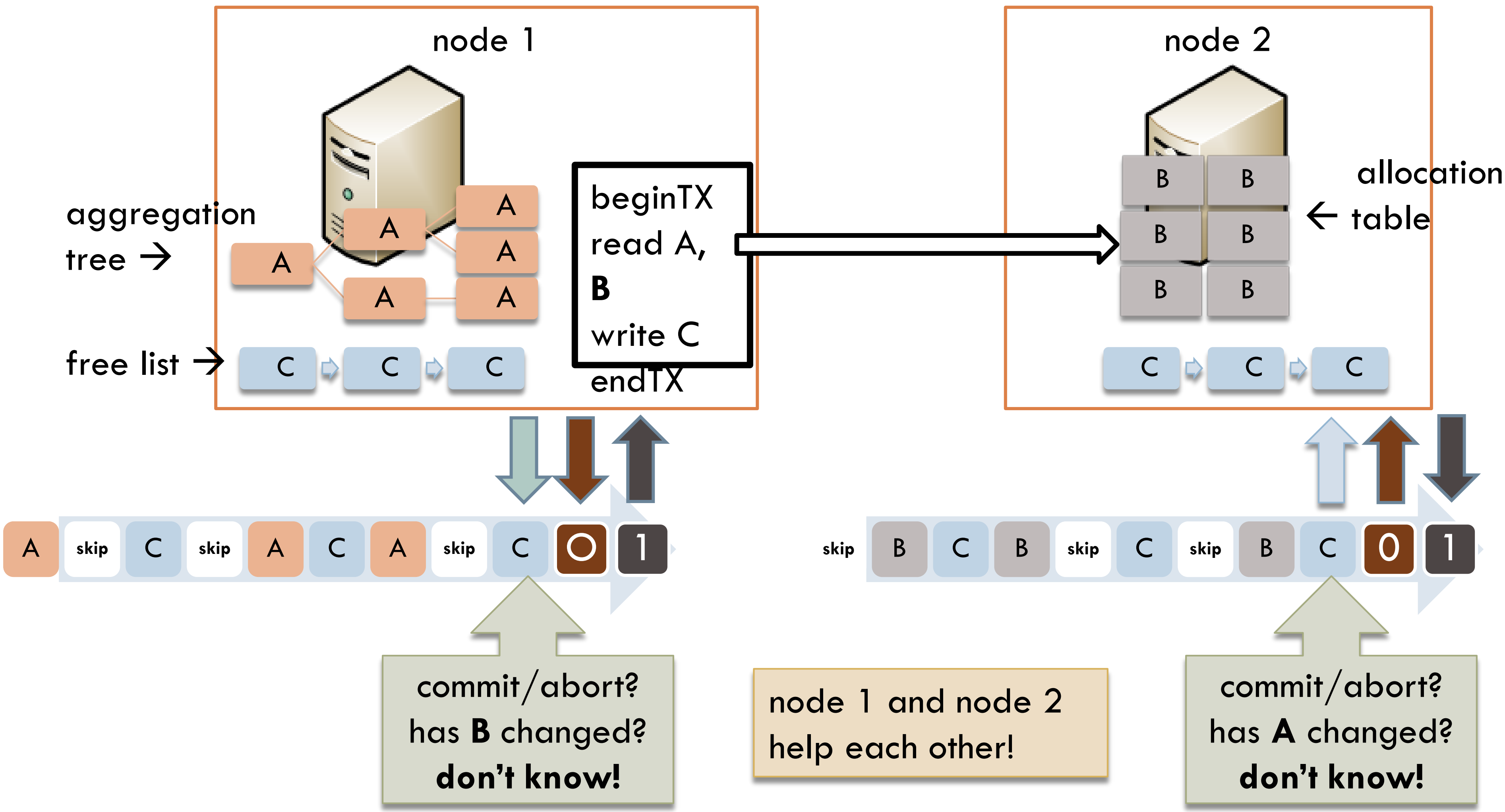
txes over streams



What about transactions?

- Recent work (Tango, aka "Corfu-DB") looked at this
- They focused on back-end applications, but in fact there is some talk of experimenting with this idea in the first tier as well because it really is very fast
- Basically, modified transactional implementation uses Corfu for the "state of the transactional DB"

distributed txes over streams



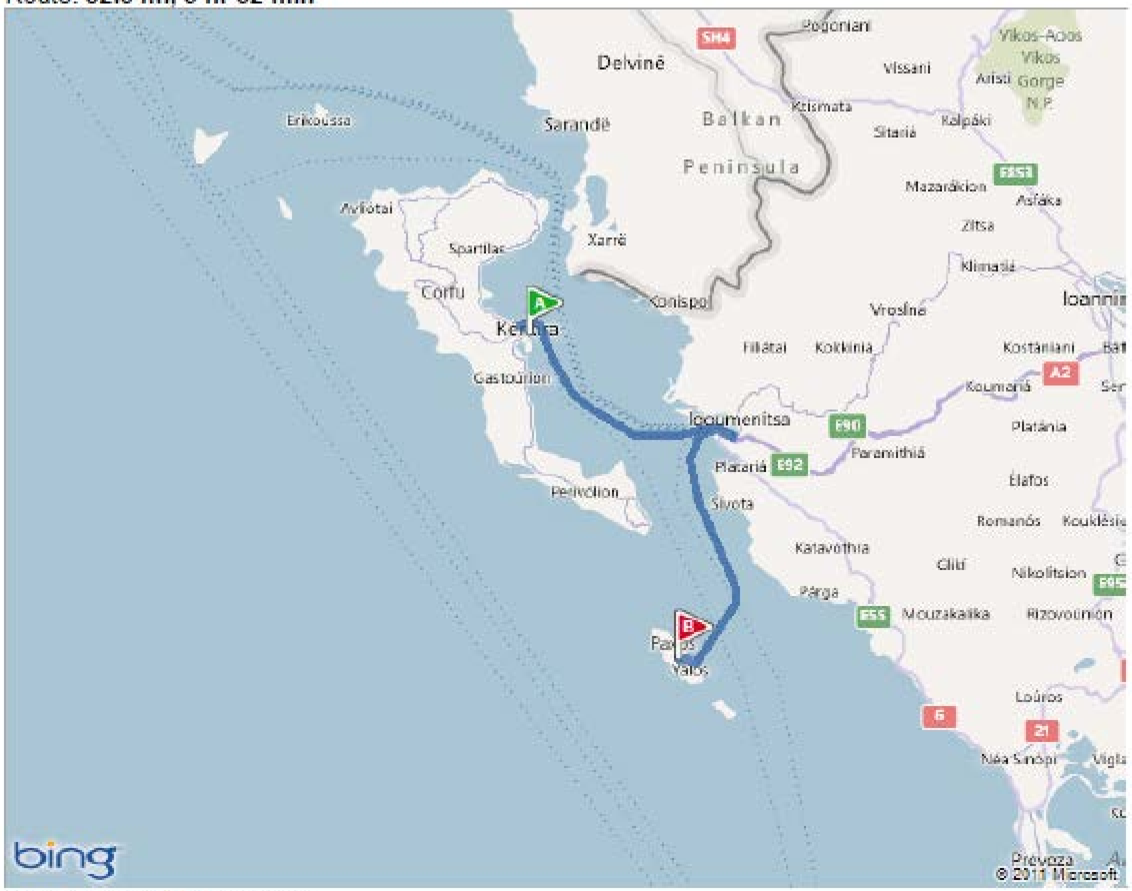
distributed transactions without a distributed (commit) protocol!

Research insights

- A durable, iterable total order (i.e., a shared log) is a unifying abstraction for distributed systems, subsuming the roles of many distributed protocols
- It is possible to impose a total order at speeds exceeding the I/O capacity of any single machine
- A total order is useful even when individual nodes consume a subsequence of it

how far is CORFU from Paxos?

Route: 52.8 mi, 3 hr 32 min

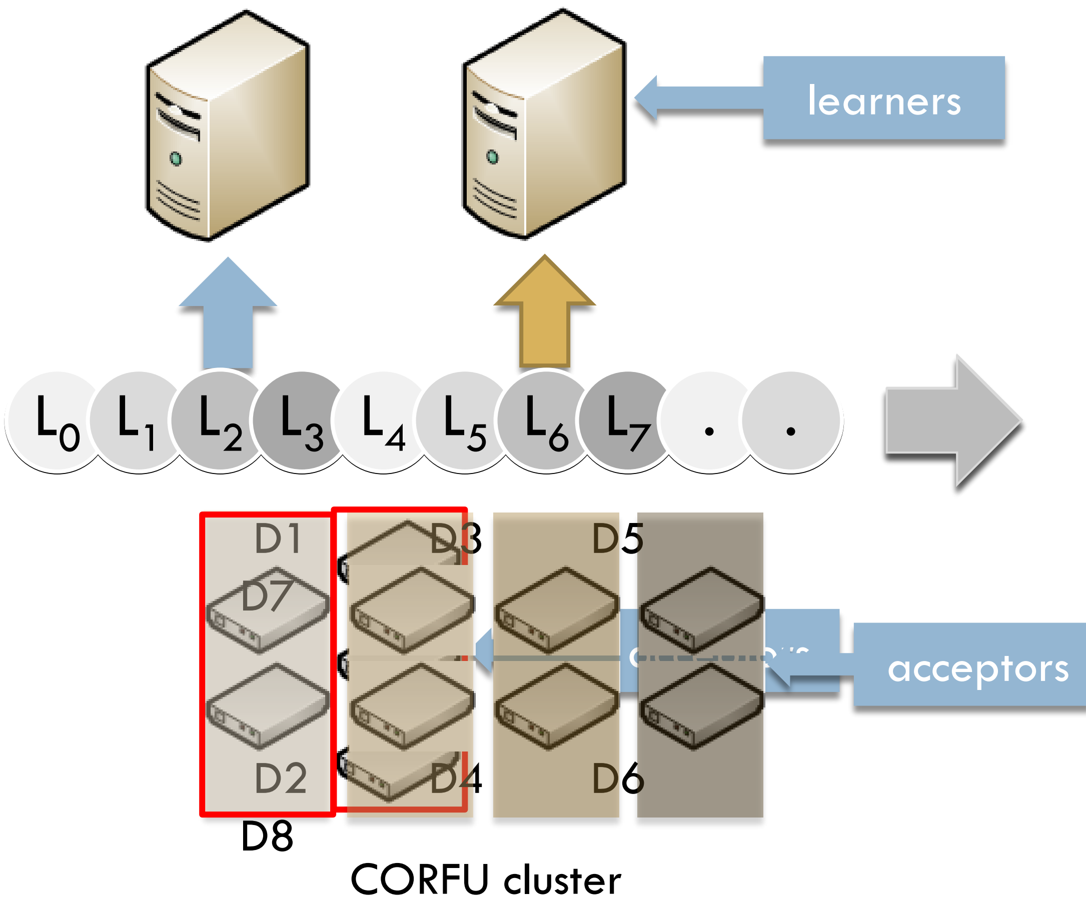


This was your map view in the browser window.

how far is CORFU from Paxos?

CORFU scales the Paxos acceptor role:
each consensus decision is made by a different set of acceptors

streaming CORFU scales the Paxos learner role:
each learner plays a subsequence of commands



Conclusions

- Wrap objects and use a logging service for higher performance in cloud settings

- Tango objects: data structures backed by a shared log

- key idea: the shared log does all the heavy lifting

(durability, consistency, atomicity, isolation, elasticity…)

- Tango objects are easy to use, easy to build, and fast…

… thanks to CORFU, a shared log without an I/O bottleneck