Bigtable, Spanner and Flat Datacenter Storage

by Onur Karaman and Karan Parikh

Introducing Bigtable

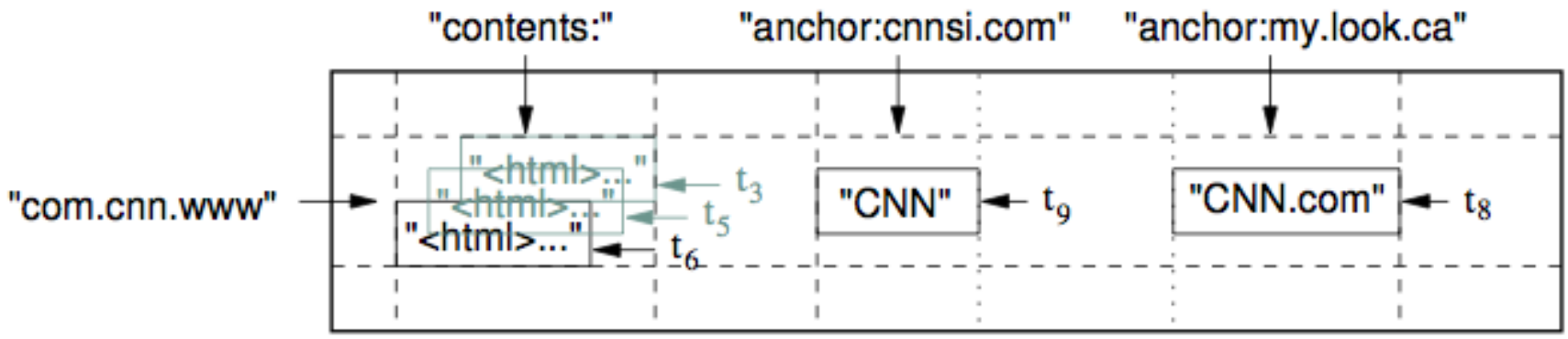
Why Bigtable?

- Store lots of data
- Scalable
- Simple yet powerful data model
- Flexible workloads: high throughput batch jobs to low latency querying

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Data Model

- "Sparse, distributed, persistent, multidimensional sorted map"
- (row: string, column: string, time: int64) → string
- Main semantics are: Rows, Column Families, Timestamps



Interacting with your beloved data

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Implementation

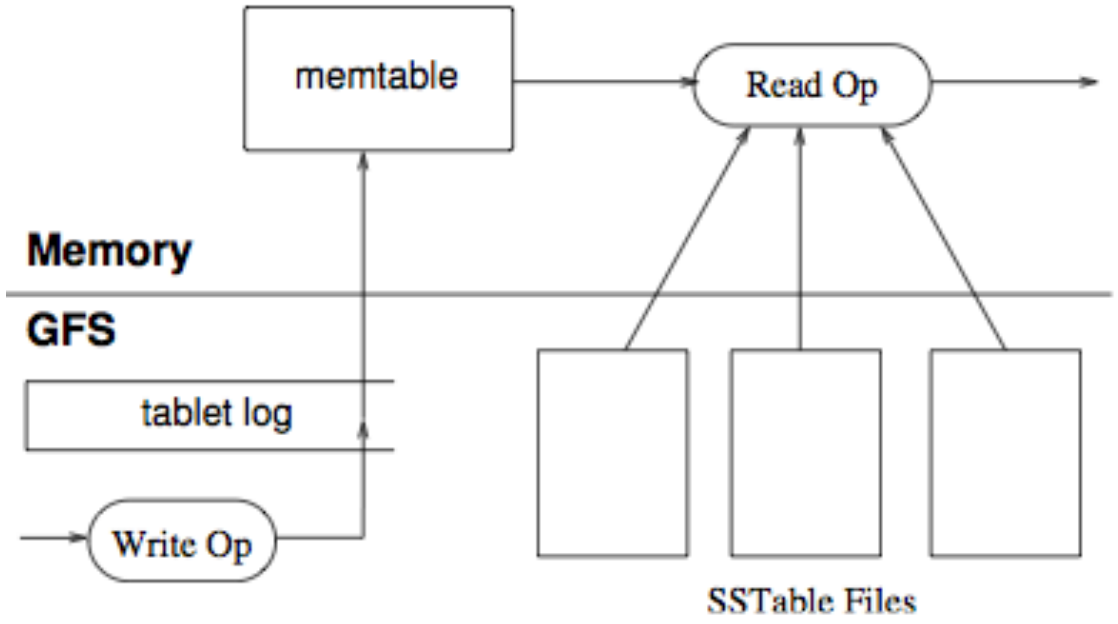
- Consists of client library, one master server and many tablet servers
- Tables start as a single tablet and are automatically split as they grow
- Tablet location information stored in a three-level hierarchy
- Each tablet is assigned to one tablet server at a time
- Master takes care of allocating unassigned tablets to a tablet server with sufficient room
- Master detects when a tablet server is no longer serving its tablets using Chubby

SSTables and memtables

- All data is stored on GFS as SSTables
- SSTables are persistent, ordered, immutable key-value map
- Recently committed updates are held in memory in a sorted buffer called a memtable
- Compactions convert memtables into SSTables.

Reading and Writing data

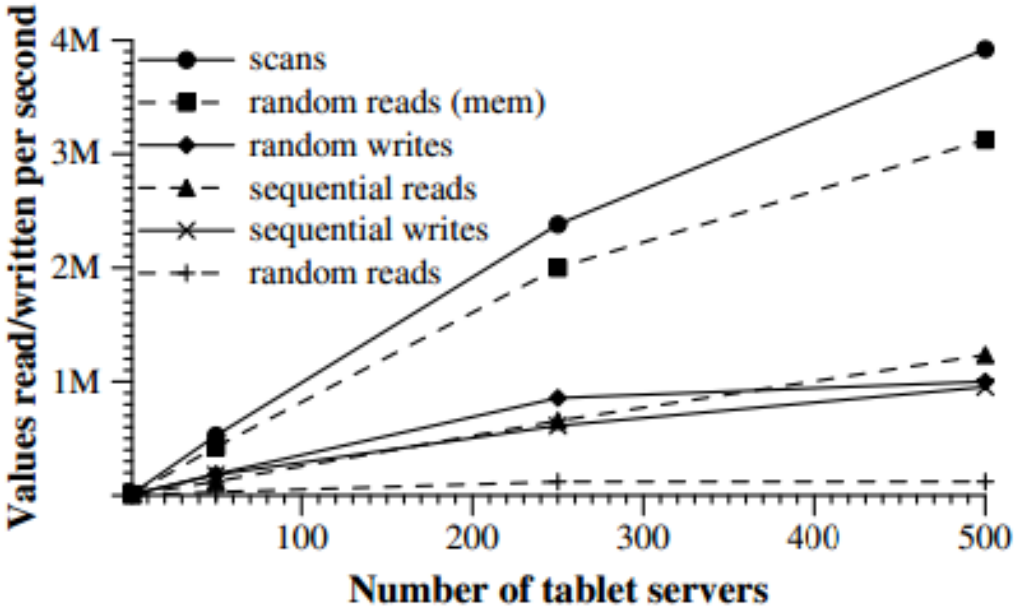
- Reads and writes are atomic.



Refinements

- Locality groups
- Compression
- Tablet Server Caching
- Bloom Filters
- Commit-Log Co-Mingling
- Tablet Recovery through frequent compaction
- Exploiting Immutability

Experiments



Open Source 

Image Source: http://www.siliconindia.com:81/news/newsimages/special/1Qufr00E.jpeg

Image Source: http://www.webresourcesdepot.com/wp-content/uploads/apache-cassandra.gif

Criticisms and Questions

- Depends heavily on Chubby. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable
- Data model is not as flexible as we think: not suited for applications with complex evolving schemas (from the Spanner paper)
- Lacks global consistency for applications that want wide area replication. (I wonder who can solve this problem? Spoiler Alert! It's Spanner)

From Piazza:

- "The onus of forming a locality groups is put on clients, but can’t it be better if done by Master?" *by Mayur Sadavarte*

Introducing Spanner

“As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.”

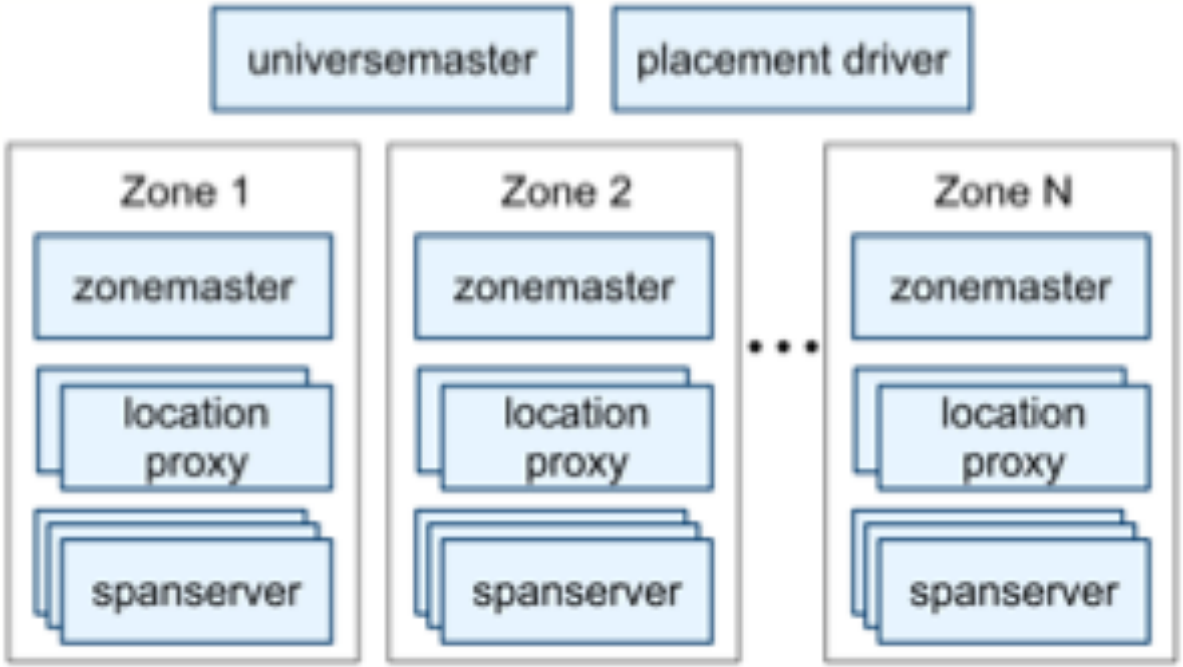
Why Spanner?

- Globally consistent reads and writes
- highly available, even with wide-area natural disasters
- "scalable, multi-version, globally-distributed, and synchronously-replicated database"
- Supports transactions using **2 phase commit and Paxos**

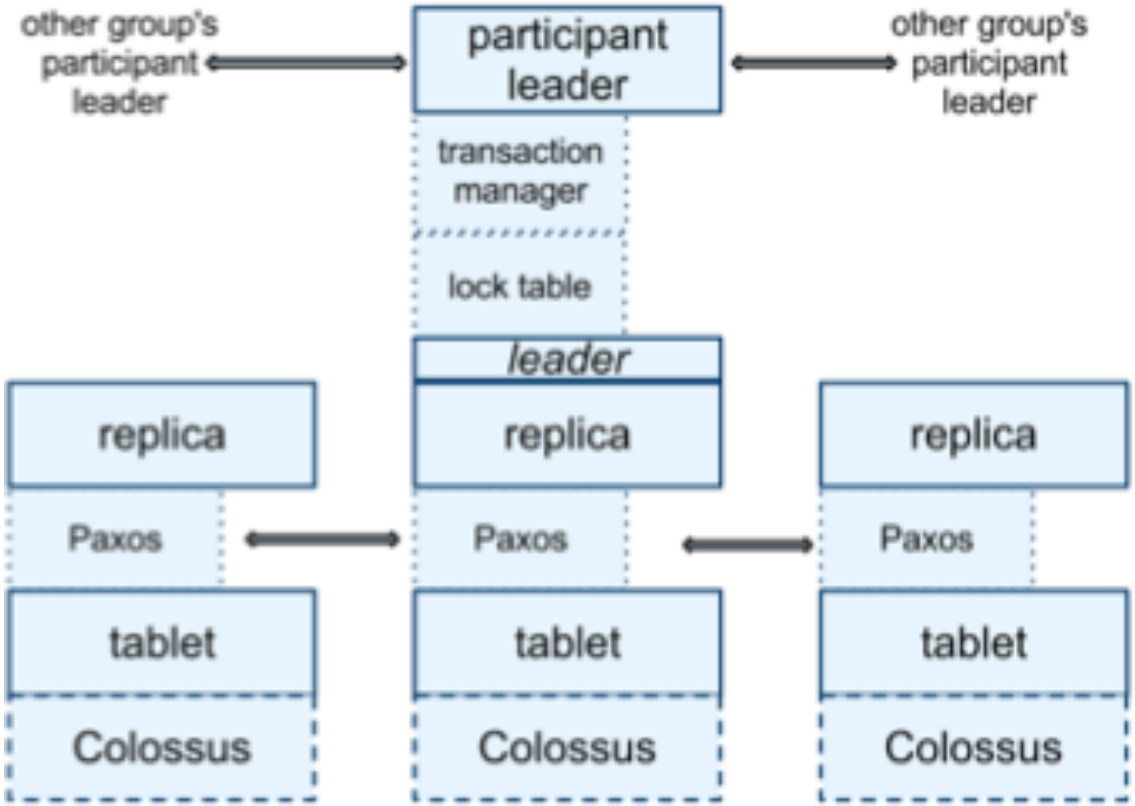
Main focus of this presentation

- True Time
- Transactions

The big players: The Universe



The big players: A Spanserver



Data Model: Tablet Level

- Similar to BigTable tablets
- **(key: string, timestamp: int64) → string mappings**
- Tablets are stored on **Colossus** (the successor to Google File System)
- **Directory**: a bucketing abstraction. It is a set of contiguous keys that share a common prefix. It is a unit of data placement and all data is moved directory by directory **(movedir)**

Data Model: Application Level

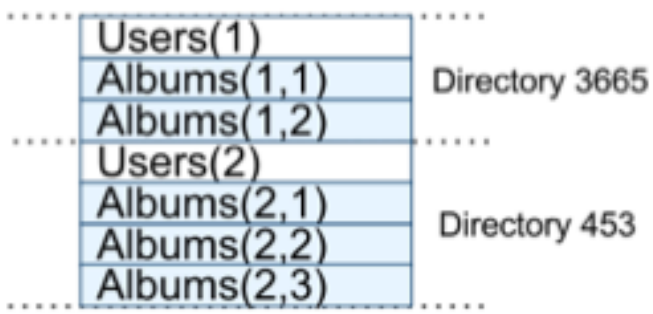
- Familiar notion of databases and tables within a database.
- Tables have rows, columns and versioned values.
- **Databases must be partitioned by clients into hierarchies of tables.** This helps in describing locality relationships which help in boosting performance

Data Model: Application Level

- "Each row in a directory table with key K, together with all of the rows in descendant tables that start with K in lexicographic order, forms a directory."

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



TrueTime

Method	Returns
$TT.now()$	*TTinterval*: [*earliest*, *latest*]
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

- Shift from concept of time to **time intervals**. e.g. suppose absolute time is t. TT.now() at *t* will give *[t_lower, t_upper]*, an interval which contains *t*. Width of interval is ***epsilon***
- A set of time **masters** per datacenter
- A timeslave **daemon** per machine
- **Atomic Clocks and GPS**
- Daemons poll a variety of masters and synchronize their local clocks to "non liar" masters.
- **epsilon derived from conservatively applied worst-case local clock drift (between synchronizations). Average is 4ms since the current applied drift rate is 200 microseconds/second and poll interval is 30s (Add 1ms for network). Also depends on time-master uncertainty and communication delay.**

TrueTime + Operations

Operation	Concurrency Control	Replica Required
Read-Write Transaction	Pessimistic	Leader
Read-Only Transaction	Lock-free	Leader for timestamp; any* for read
Snapshot Read w/ client-provided timestamp	Lock-free	any*
Snapshot Read w/ client provided bound	Lock-free	any*

*** = should be sufficiently up-to-date**

TrueTime + Operations: Read Write Transactions

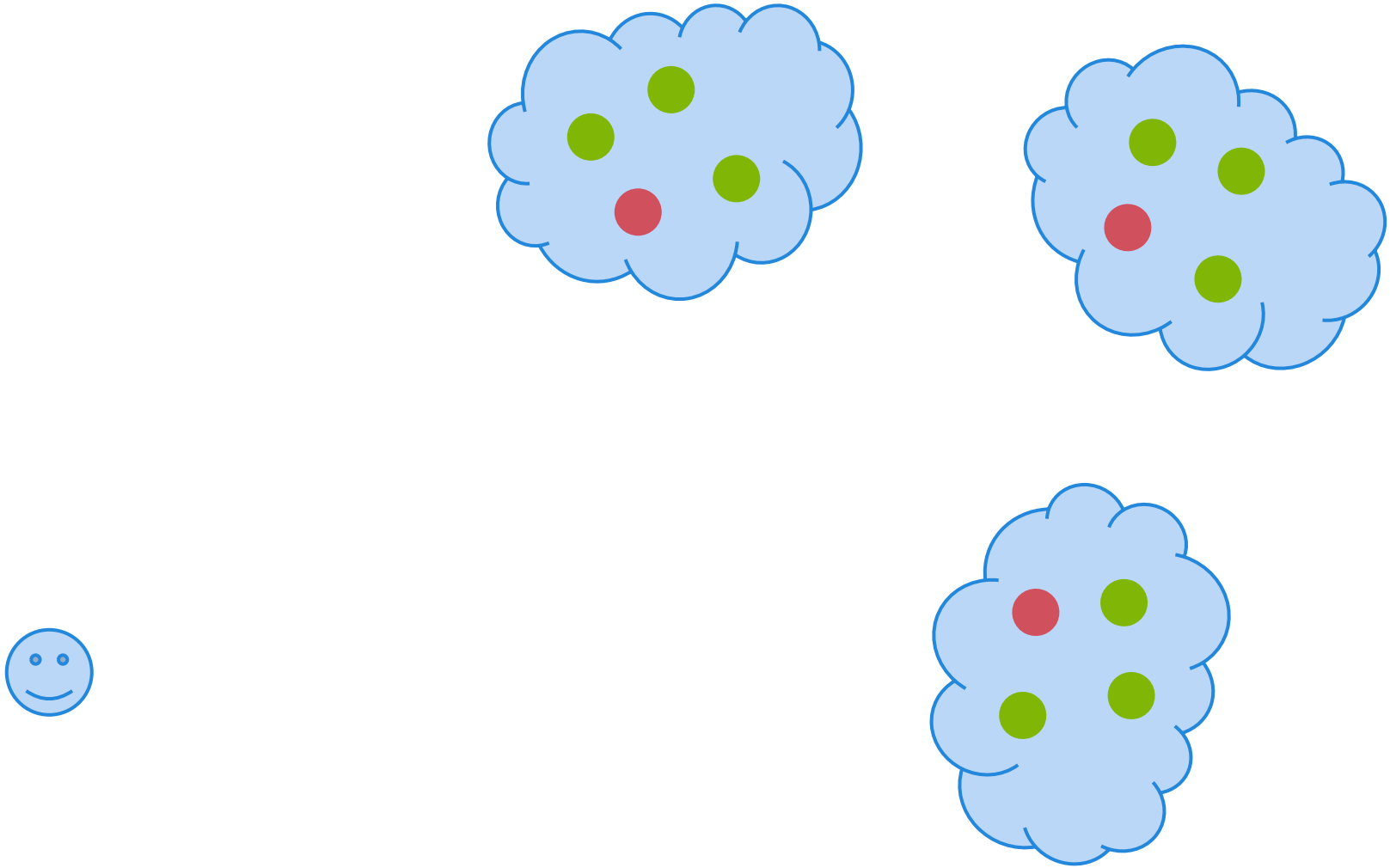
Reads

- Client issues reads to the **leader replica** of the appropriate group
- Leader acquires read locks and reads the most recent data
- All writes are buffered at the client until commit

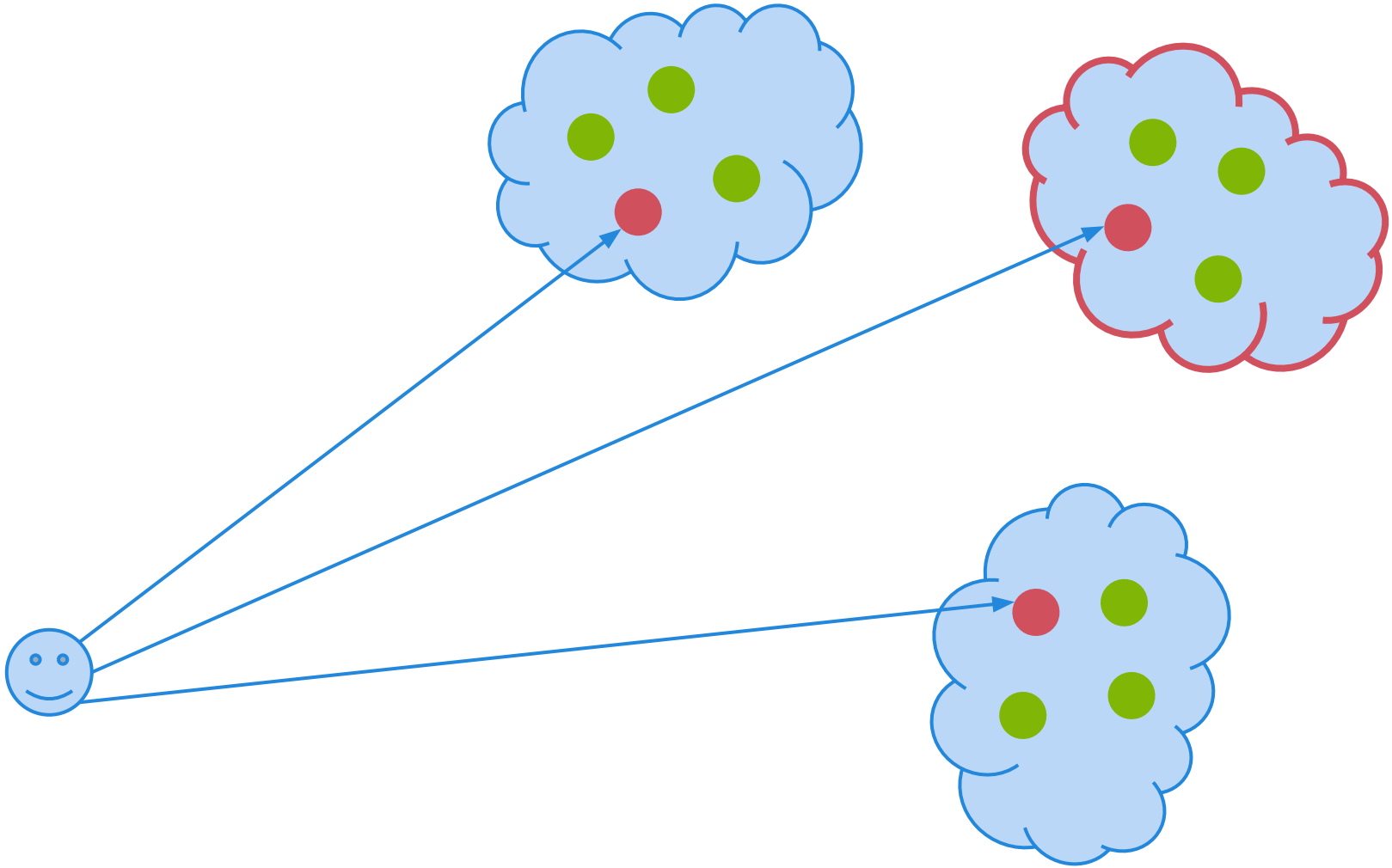
Writes

- **Clients** drive the writes using **2 phase commit**
- **Replicas** maintain consistency using **Paxos**

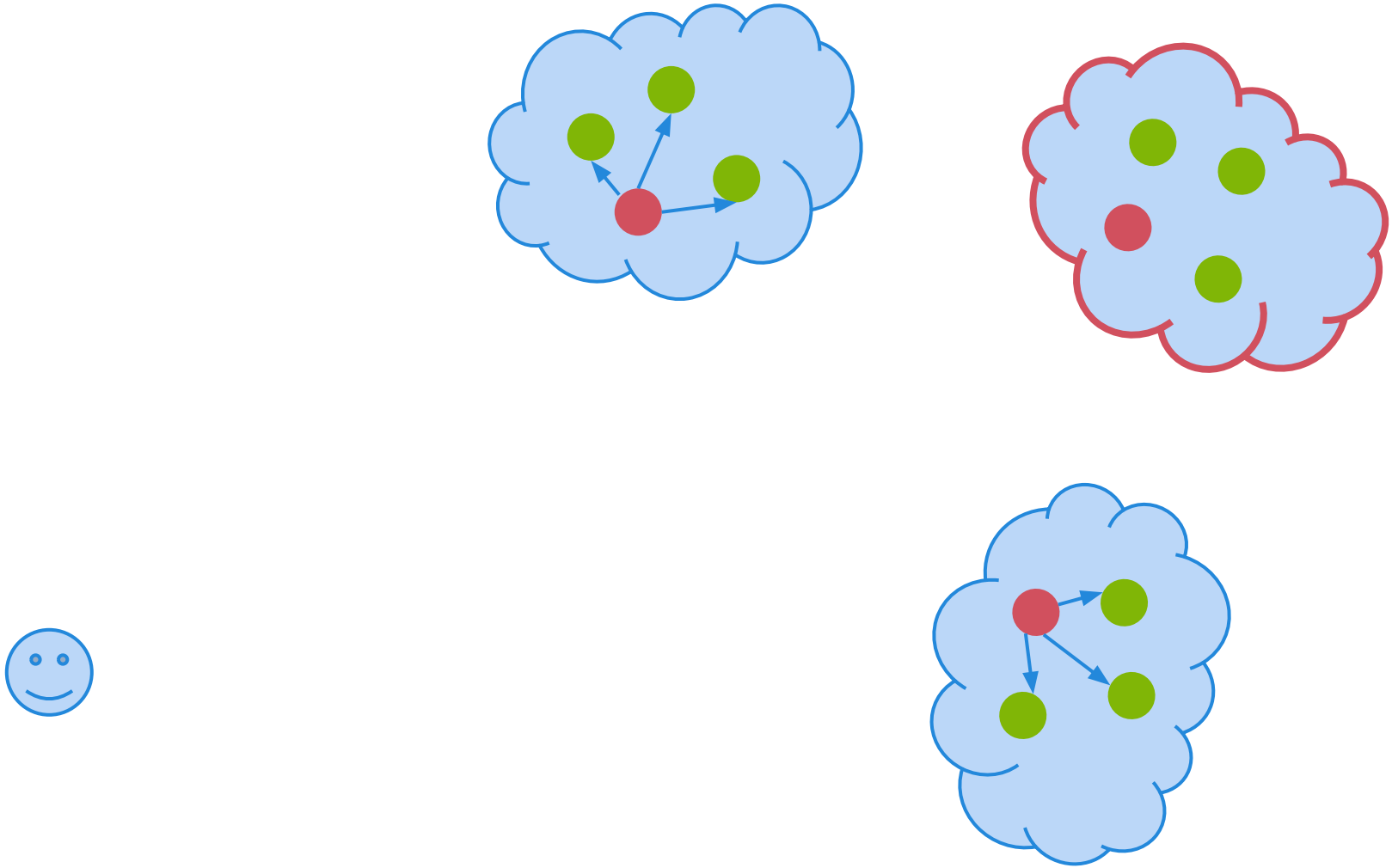
TrueTime + Transactions: Read Write Transactions



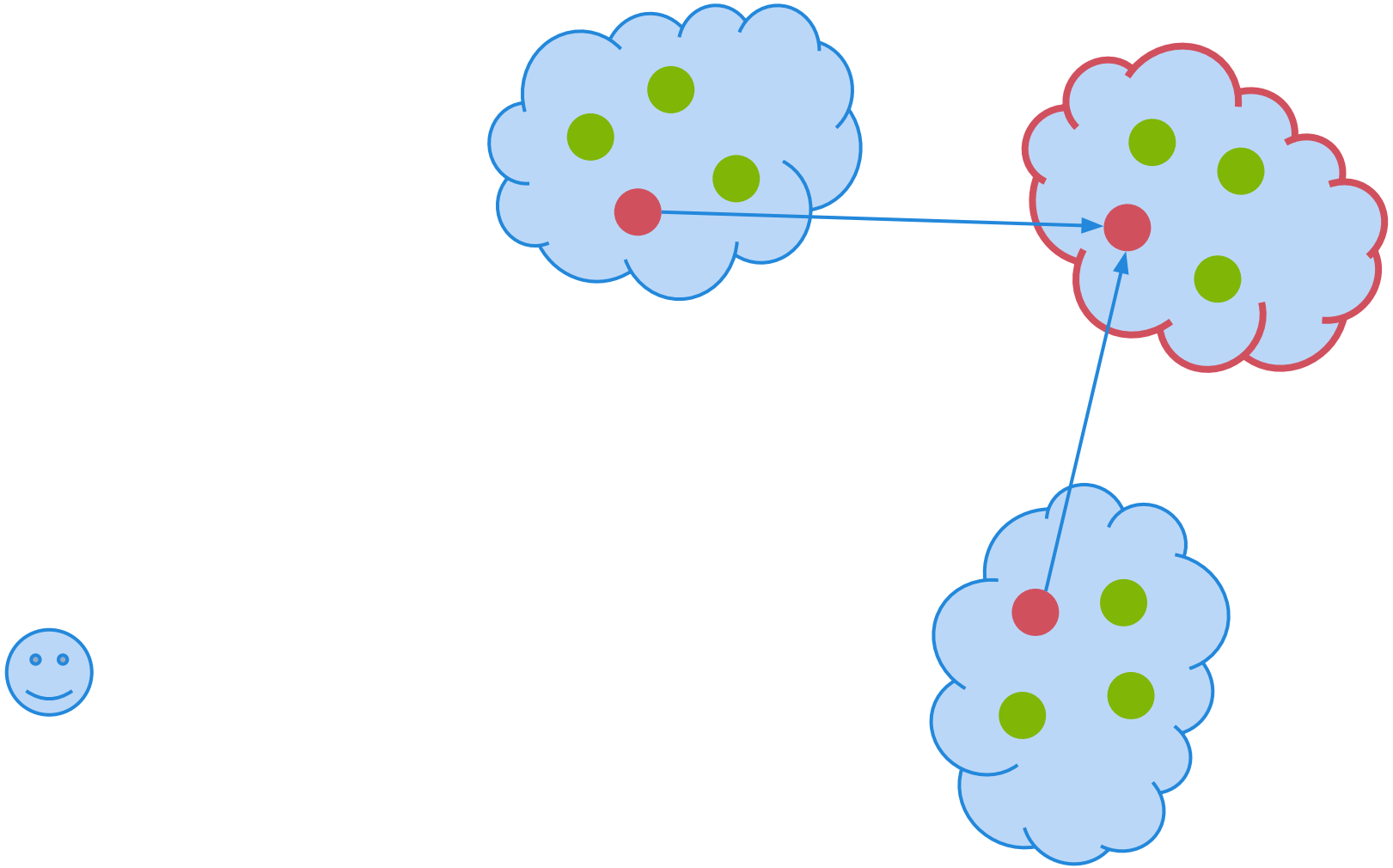
TrueTime + Transactions: Read Write Transactions



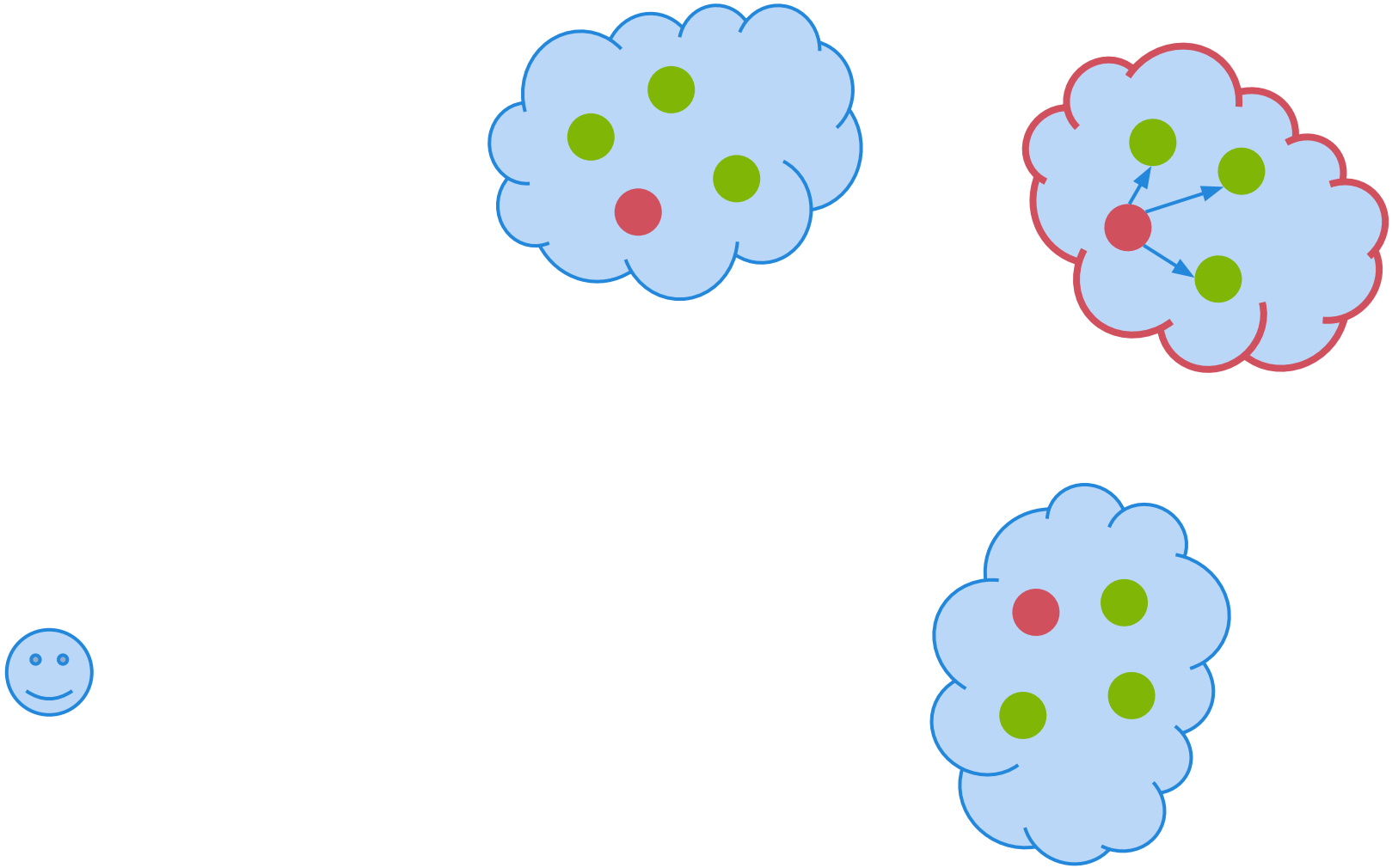
TrueTime + Transactions: Read Write Transactions



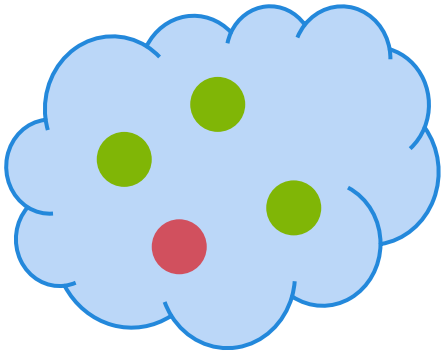
TrueTime + Transactions: Read Write Transactions

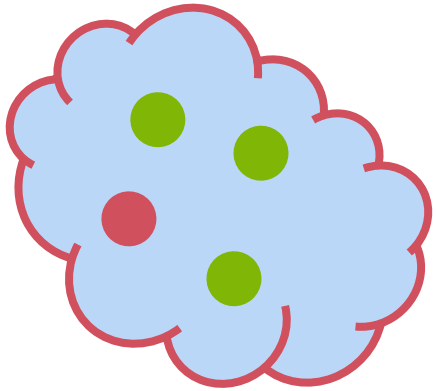


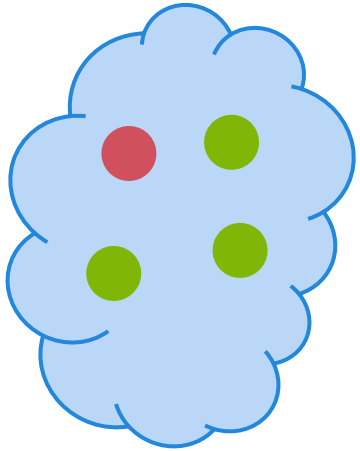
TrueTime + Transactions: Read Write Transactions



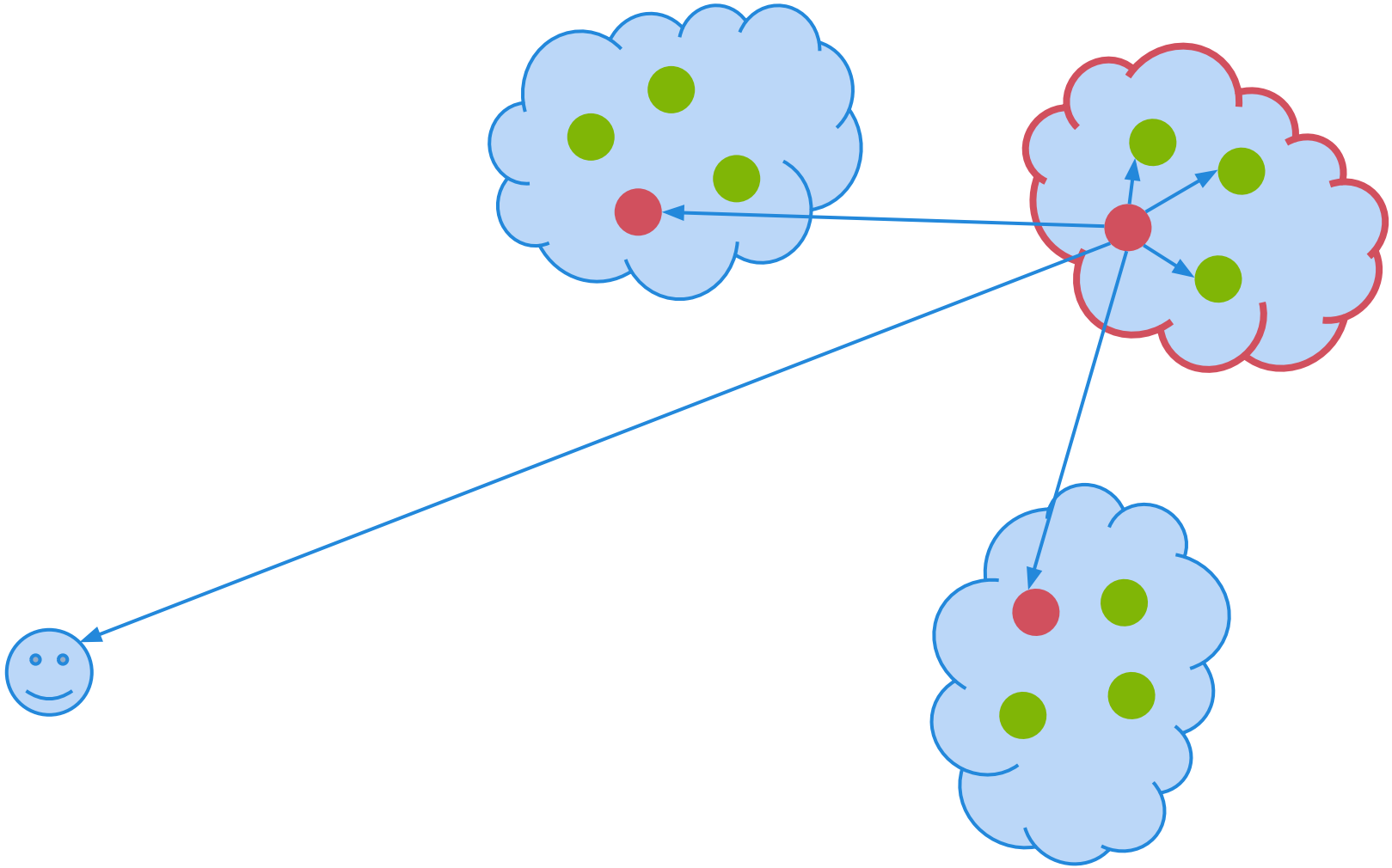
TrueTime + Transactions: Read Write Transactions



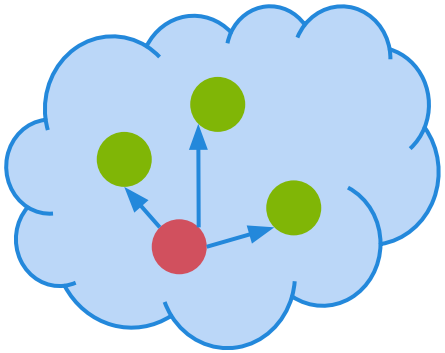


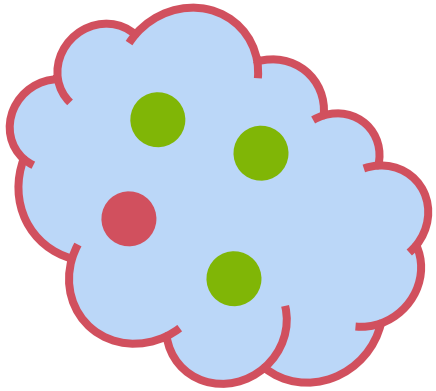


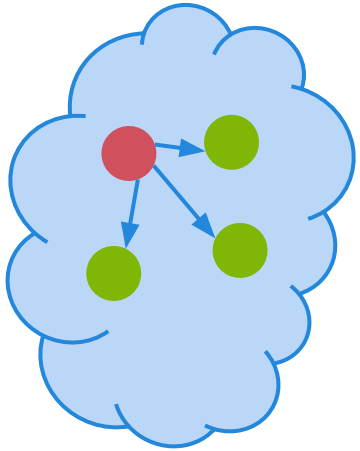
TrueTime + Transactions: Read Write Transactions



TrueTime + Transactions: Read Write Transactions







TrueTime + Transactions: Reads at a timestamp

- **Reads can be served at any sufficiently up-to-date replica**
- Uses the concept of **"safe-time"** to determine how up-to-date a replica is
- ***t_safe* = min(*t_Paxos_safe*, *t_TM_safe*)**. Per replica basis
- Can serve a read at timestamp *t* at a replica *r* iff *t* <= *t_safe*
- *t_Paxos_safe* = timestamp of the highest applied Paxos write
- ***t_TM_safe* = min(*prepare_i*) - 1** over all the transactions involving this group
- *t_TM_safe* is infinity if there are zero prepared but not committed transactions

TrueTime + Transactions: Generating a read timestamp

We need to generate a timestamp for Read-Only Transactions (clients supply timestamps/bounds for Snapshot reads)

- 1 Paxos group: timestamp = timestamp of the **last committed write** at a Paxos group
- Multiple Paxos groups: timestamp = **TT.now().latest**. This is simple though it might wait for the safe time to advance.

Experiments

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Experiments

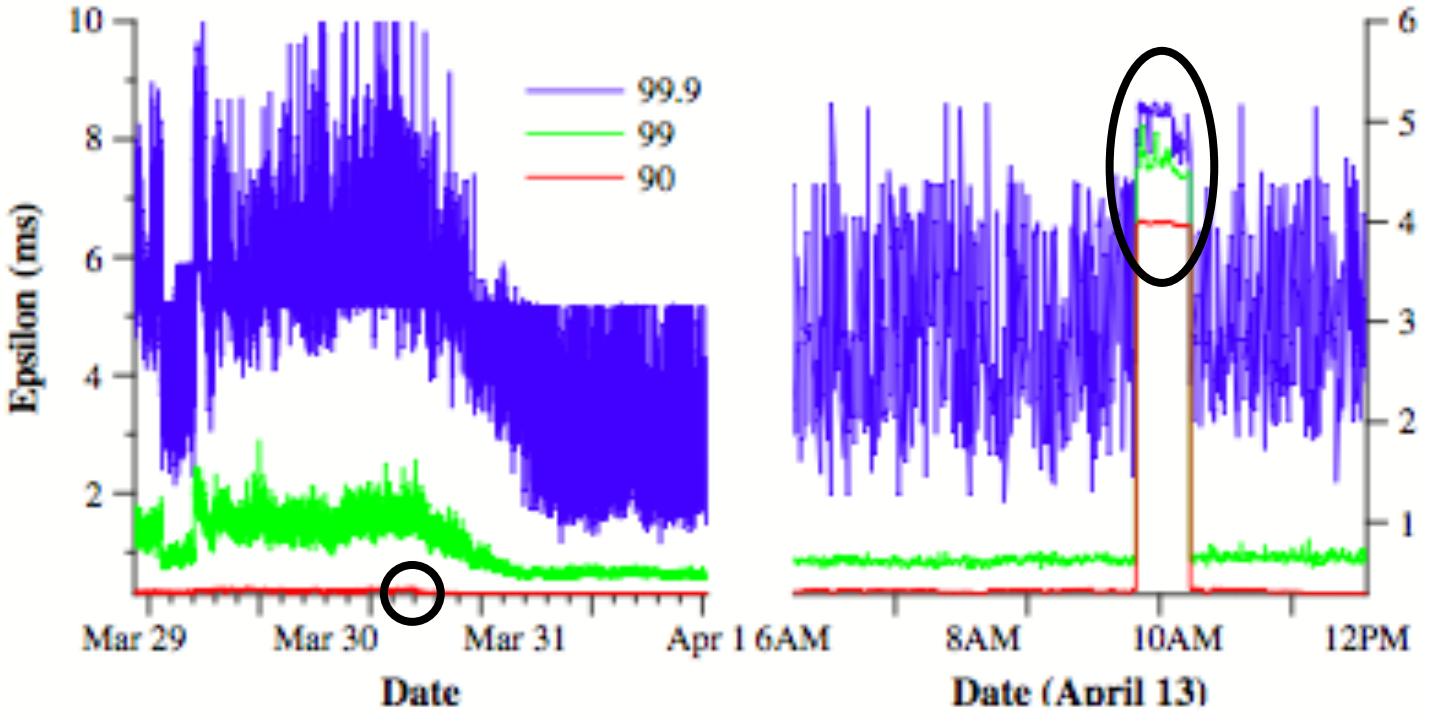


Figure 6: Distribution of TrueTime ϵ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

Case Study: F1

F1 is Google's advertising backend. It has 2 replicas on the west coast and 3 on the east coast. Data measured from East coast servers.

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

Open Source 

Yet.

Questions and Criticisms from Piazza

- "Overhead of Paxos on each tablet has not been evaluated much." *by Mainak Ghosh*
- "It is not clear for me how the TrueTime error bound is computed. How does it take into account of local clock drift and network latency. How sensitive it is to the network latency, since a client has to pull the clock from multiple masters, including master from outside datacenter, so the network latency should not be non-negligible" *by Cuong Pham*
- "Whether Spanner disproves CAP? Is Spanner an actually distributed ACID RDBMS?" *by Cuong Pham*
- "This paper is only a part of Spanner and doesn't include too much technical details of TrueTime and how time synchronization is being performed across the whole Spanner deployment. It will be interesting to read the design of TrueTime service as well." *by Lionel Li*

Introducing Flat Datacenter Storage

"FDS' main goal is to expose all of a cluster's disk bandwidth to applications"

Why FDS?

- "a high-performance, fault-tolerant, large-scale, locality-oblivious blob store."
- We don't need to move computation to the data anymore
- datacenter bandwidth is now abundant
- "flat": drops the constraint of locality based processing
- dynamic work allocation

Data Model

- Blobs
- Tracts

API

Getting access to a blob
`CreateBlob(UINT128 blobGuid)`
`OpenBlob(UINT128 blobGuid)`
`CloseBlob(UINT128 blobGuid)`
`DeleteBlob(UINT128 blobGuid)`
Interacting with a blob
`GetBlobSize()`
`ExtendBlobSize(UINT64 numberOfTracts)`
`WriteTract(UINT64 tractNumber, BYTE *buf)`
`ReadTract(UINT64 tractNumber, BYTE *buf)`
`GetSimultaneousLimit()`

Figure 1: FDS API

- Non-blocking async API
- Weak consistency guarantees

Implementation

- Tractservers
- Metadata server
- Tract Locator Table (TLT):
  Tract_locator = (Hash(g) + i) mod TLT_Length

Networking

- datacenter bandwidth is abundant
- full bisection bandwidth
- high disk-to-disk bandwidth

Experiments

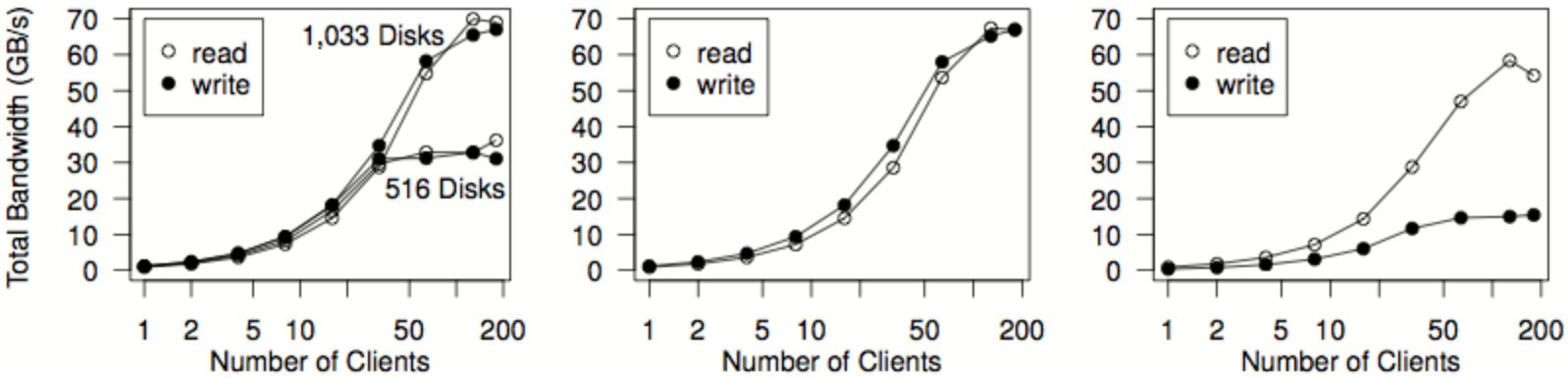


Figure 4: Mean aggregate throughput of 1 to 180 clients reading and writing 8MB tracts on a 1,033-disk cluster. Standard deviation is less than 1% of the mean of each point. The x axes use logarithmic scales. *(a)* Sequential reading and writing in a single-replicated cluster. Results for a 516-disk cluster are also shown. *(b)* Random reading and writing in a single-replicated cluster. *(c)* Sequential reading and writing in a triple-replicated cluster.

Questions and Criticisms from Piazza

- "Cluster growth can lead to lot of data transfer as balancing is done again. They have not given any experimental evaluation of this part of the work. Feature like variable replication also complicates this process." *by Mainak Ghosh*

References

- All information and graphs about Bigtable is from http://research.google.com/archive/bigtable.html
- All information and graphs about Spanner is from https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf
- All information and graphs about Flat Datacenter Storage is from https://www.usenix.org/system/files/conference/osdi12/osdi12-final-75.pdf