Data massage!

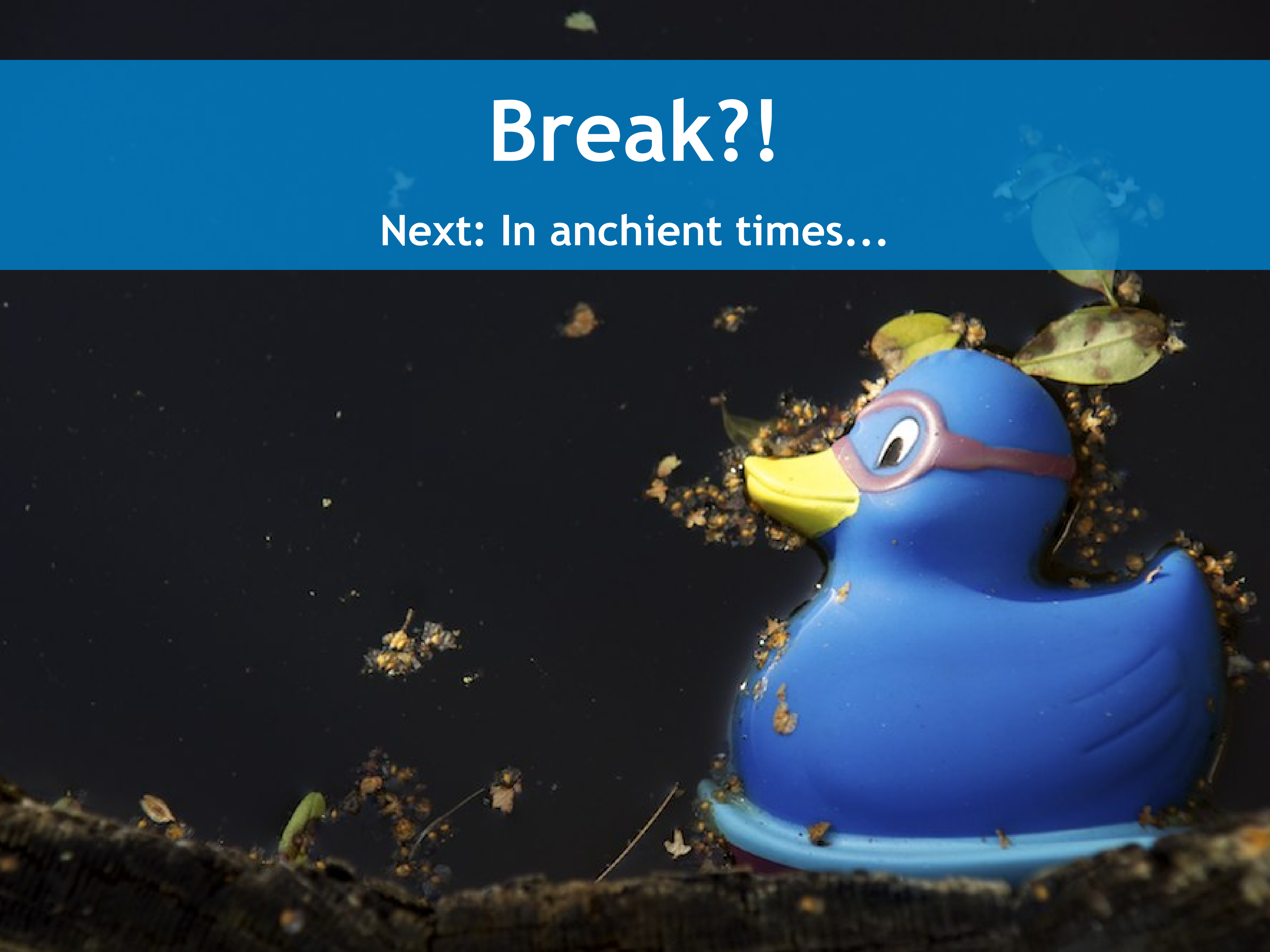
Databases scaled from one to one million nodes

Ulf Wendel/MySQL

The speaker says...

From a workshop at the PHP Summit 2013, Berlin.

Todays database world is confusing. A decade ago there was only a bunch of databases that worked well for web development. In the meantime there are 150+ NoSQL databases in addition to the traditional choices. How come? Join me on a journey from databases made for a single machine to globally distributed databases running on millions of nodes. Developing applications against massive scalable databases is somewhat different. Developing the databases itself is very different. Be warned: we also cover the view of those developing scalable databases.



Break?!

Next: In anchient times...

Have you seen my data?

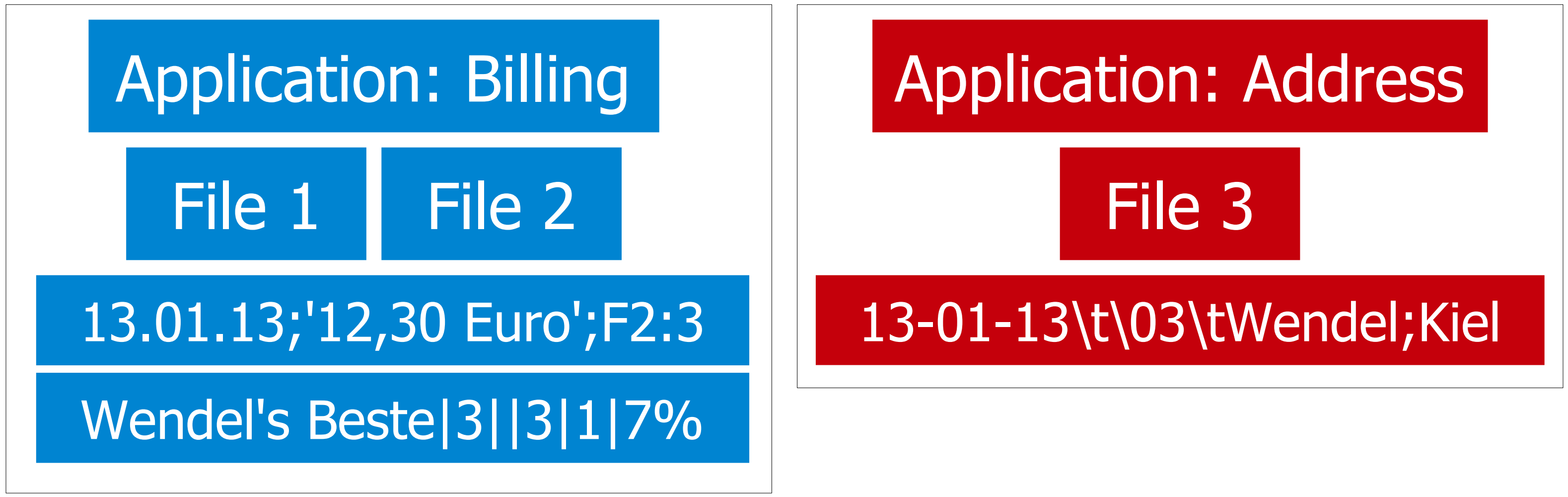
OMG - DBMS' have not been invented yet?

This is Merlin. He travels through time, cupboards and cookware.

Database?

The glory 60th...

- Ever done nigthly data exchange with mainframe hosts – after 2000th?



The speaker says...

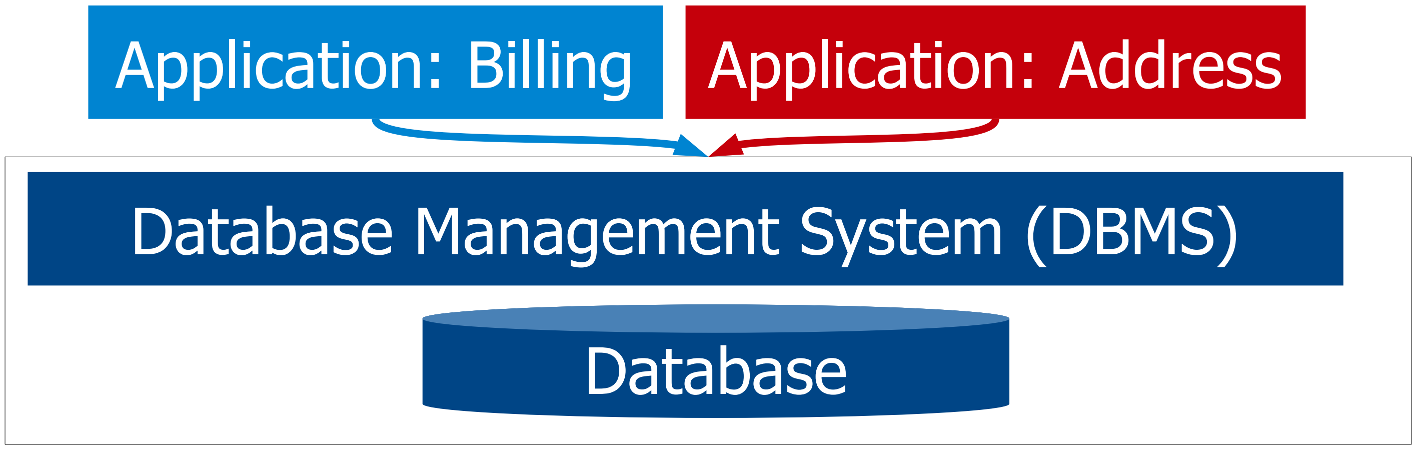
How time flies... I still remember being introduced to the fun of importing data from legacy applications.

The elderly, those that worked already in the 60th, told me about times when applications used to store their data on magnetic tape. Companies had many applications in production use. Each and every of them had their own idea of how to store data. How exciting it was to use hex editors to decipher the data! How creative developers had been on security! How much we earned thanks to all the extra hours spent creating reports with data from multiple applications! How secure our jobs had been: only we knew all the glory details to keep the application running...
Then came the 70th and with them database systems.

Database in the 70th?

General goals

- Internal and (unified) external view on data
- Centralized storage
- Data consistency, efficient data access
- Multi-user support, access control



The speaker says…

In the 70th database systems have been introduced to consolidate the data once kept in individual files spread all over a company.

A database system aims to ensure data consistency and supports use from multiple applications and users in a secure way, for example, by providing access control. Data shall be stored in an efficient manner.

The systems internal view on data is isolated from the users external view. Users do not have to worry about storage details. No matter how the system stores the data, there is an external view on it that does not change.

Data?

Type: string

```
$comment = "What a silly beginning ?!";
$price_eur = 10;
```

Type: currency

Value range (domain)

13.01.13;'12,30 Euro';F2:3

The speaker says…

As mentioned, data consistency is a major topic for database systems. But, what is data, what makes it...?

Obviously, all our data has a type and a value range/domain associated with it. For example, a string is a sequence of alphanumeric characters. Whereas a number contains of digits only.

Operators and data structures

Constructor

```
class seats {
  private $seats;
  private $remaining;
  public function __construct() {
    $this->seats = array('damn, what to put here?');
  }
}
$a = new array();
$a[] = 1;
```

Operator

Operator

Operator

The speaker says…

There can be scalar and non-scalar data types. A scalar (or base) type holds a single data item only. A string or an integer is a scalar type. Whereas a class can be classified as a non-scalar type.

There's a set of basic operators to work the different data types. Sometimes, non-scalar types have constructors that can be used to initialize the members of the type.

Some database systems allow us to define data structures. A data structure is a particular way of storing and organizing data efficiently. For example, an object can be considered a data structure with operators (as opposed to a plain-old data structure without.)

State changes over time

```
class ticket {
  private $seats;
 …
  public function addSeat(Seat $seat) {
    $seats[] = $seat;
  }
}
$t = new ticket();
$t->addSeat(new Seat(1, 1, 9.99));
```

State$_1$ at t$_1$

State$_2$ at t$_2$

The speaker says…

Data in a program is in flux. As the program progresses, the data is modified. The state is changed. Usually state changes follow some rules.

This finally gets us back to databases...

Data model

Data structures

- Basic types with domains and operators
- Rules for building data structures

Operators on data structures

- Set of operators to work on data structures
- Add, delete, modify, search, …

Possibly: additional integrity constraints

- Rules that go beyond implicit data type constrains

Versus database schema:

- Description of a database in accordance to the data model
- Can there be a schema-free data store?

The speaker says…

A data model for a database defines the data types that can be used, the way data is to be structured and what global operators exist to work with the data. Because databases are long running processes, the global operators describe the possible state changes.

Data models are abstract, generic constructs: sets of rules and concepts to describe a database. A database schema is a concrete description of a database in accordance to a certain data model. Hence, there is hardly any data store that is schema-free. But, many NoSQL systems have a flexible schema.

Data models we will touch

Relational data model

- flat
- nested sets

Document model

Key-Value model

Wide columnar

The speaker says…

For the moment we will stay with the relational data model. As we move on, we will dig more into NoSQL data models such as document stores, key value stores and wide column stores.

Relational model, briefly

Despite NoSQL, it is the dominating model

- How could I skip it?
- Perfect match for the workshop's example application?
- What does it mean to use a different model?

NoSQL exists because of the relational model?

- Data-first development style
- Cost of schema changes
- Scalability concerns

The speaker says…

We will discuss the relational model briefly. After all, the existance of some NoSQL stores can be interpreted as an reaction to the shortcomings of the relational model. Besides all the shortcomings, we shall not forget the advantages the relational mode.

Not all new, hot developments that NoSQL brought to us are related to the model itself. There is no to little excuse for a relational database to lack, for example, a HTTP interface or a low-level access method to bypass SQL on demand. There is little excuse for complicated administration and so forth.

Schema design

Gather what information is to be stored

Create Entity-relationship model

- Entity: The subjects, the independent things, the 'nouns'
- Attributes: The 'adjectives', information items and their rules
- Key candidates: Set of attributes to uniquely identify an entity
- Relations: The 'transitive verbs' w. cardinalities (1:1, 1:n, n:m)

Transform ER-model into physical data model

- (Network model, relational model, hierarchical model)
- Relational model: tables, fields, primary keys, …
- Relational model: apply normalization rules

The speaker says…

Developing an application against a traditional database begins with the schema design: what information is to be stored, how is information to be grouped into logical resp. conceptual units? The findings are modeled in an ER-model, for example, using MySQL Workbench, as a design tool. In a second step the ER-model is transformed in a physical data model, for example, the model supported by a concrete relational database. Everybody in the room should be familiar with this, or be able to get his hands on about database modeling.

For today, only the normalization rules matter.

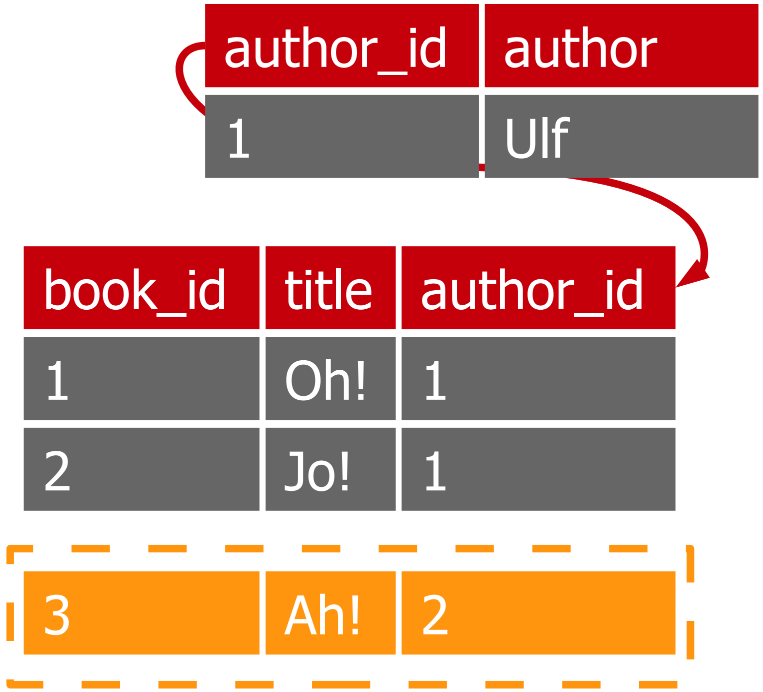
Database normalization

Minimize reundancy and dependency

Modification anomalies

- Update anomaly
- Insertion anomaly
- Deletion anomaly

book_id	title	author
1	Oh!	Ulf
2	Jo!	Ulv
3	Ah!	Shary



author_id	author
1	Ulf

book_id	title	author_id
1	Oh!	1
2	Jo!	1
3	Ah!	2

The speaker says…

One - not the only - objective of database normalization is to free the database of modification anomalies. Data modification anomalies may happen in Key-value NoSQL stores as well. Imagine an author has written two book and thus, his name appears twice in the author column. If the authors name is to be changed and the update is not done carefully, then we end up with multiple, distinct versions of the authors name. An insertion anomaly would arise if we began selling books for which the author has not been born, or just added to our database. A deletion anomaly describes a situation in which we loose all facts on an author once we delete a book record – albeit we still need the author infomation elsewhere.

First normal form vs. N1NF, NF^2

First normal form (1NF)

- All attributes of a relation must contain only atomic values
- Each attribute contains only a single value

Non First normal form (N1NF, NF^2)

- Discussed in all depth very early

blog_id	title	...	comments	
1	Oh!	...	author	comment
			Ulf	This is not in first normal.
			Ulf	s/normal/normal form :-)

The speaker says…

The first normal form basically says that relational table columns can only store atomic values and only exactly one value. It forbids us to create a table similar to the one shown where we have a list of blog comments embedded in a column of a blog posts table.

This has been critizides very early, see also: http://blog.ulf-wendel.de/2013/searching-document-stores-from-the-1980s-to-sql2003-in-a-blink/

SQL:99, SQL:2003 ...

```
CREATE TABLE blogposts (
  id      INTEGER, title   VARCHAR(100),
  content CHARACTER LARGE OBJECT,
  show_comments BOOLEAN,
  comments ROW(
    author VARCHAR(100),
    message CHARACTER LARGE OBJECT
    ) MULTISET
);
SELECT title,
    MULTISET(
      SELECT c
      FROM UNNEST(comments) c
      WHERE c.author = 'Johannes'
    ) FROM blogposts
```

The speaker says…

Only twenty years after researchers had stopped to make proposals how to break out of 1NF and had suggested query languages, SQL:99 and SQL:2003 introduced non atomic data types. SQL:99 adds ROW and ARRAY. SQL:2003 adds MULTISET. The slide shows valid SQL:2003 – I hope its valid, I have not tested it.

If MySQL had this, would this terminate the chatter about joining relations being terribly inefficient compared to key-value/document?

JSON...?

In theory: Use TABLE constructor

- Introduced with SQL:2003, function can return table
- Likely, it would not be fast enough in practice...

In reality: Use functions *sniff*

- MySQL 5.7 adds functions to work on JSON

```
SELECT
  id,
  title
FROM TABLE(json(<blob_column>))

SELECT JSON_MERGE('{"a":"b"}', '{"c":"d"}' )
```

The speaker says...

On an aside. Relational SQL standard compliant databases cannot get quite that far to support JSON in a way that makes there query languages look, say, „appealing". By appealing I mean a query like the first using the SQL:2003 table constructor. One big problem is that SQL is strictly typed whereas JSON is not. Merging the two world is likely to result in poor performance. Thus, we end up with ugly functions to work on JSON.

(I would be surprised to learn that the Postgres way is standards compliant.)

Queries, *very* briefly

Relational algebra

- Selection, projection, rename
- Join, division, aggregation
- Insert, Delete, Replace

SQL

- Query language for the relational algebra
- Standardized – compare to the 60th situation, no nitpicking...
- Declarative: Say what not how

The speaker says…

We are done with the data model and the database schema. Let's very briefly have a look at relational algebra and SQL. The two behave to one another like the data model and the schema. The relational algebra is the abstract, mathematical construct and SQL is one, existing implementation of it.

Two points to remember: relational databases use a standardized query language (SQL). Any two systems offer approximately the same implementation. SQL is a declarative language. You are not supposed to say how data is accessed but only what data.

ACID transactions, briefly

Atomicity

- 'All or nothing': either all work done, or none (no state change)
- Guaranteed in the presence of failures: crashes, ...

Consistency

- Transaction brings database from one valid state to another
- No defined rule may be violated: constrains, trigger, …

Isolation

- Concurrent execution of transactions is equivalent to serial one

Durability

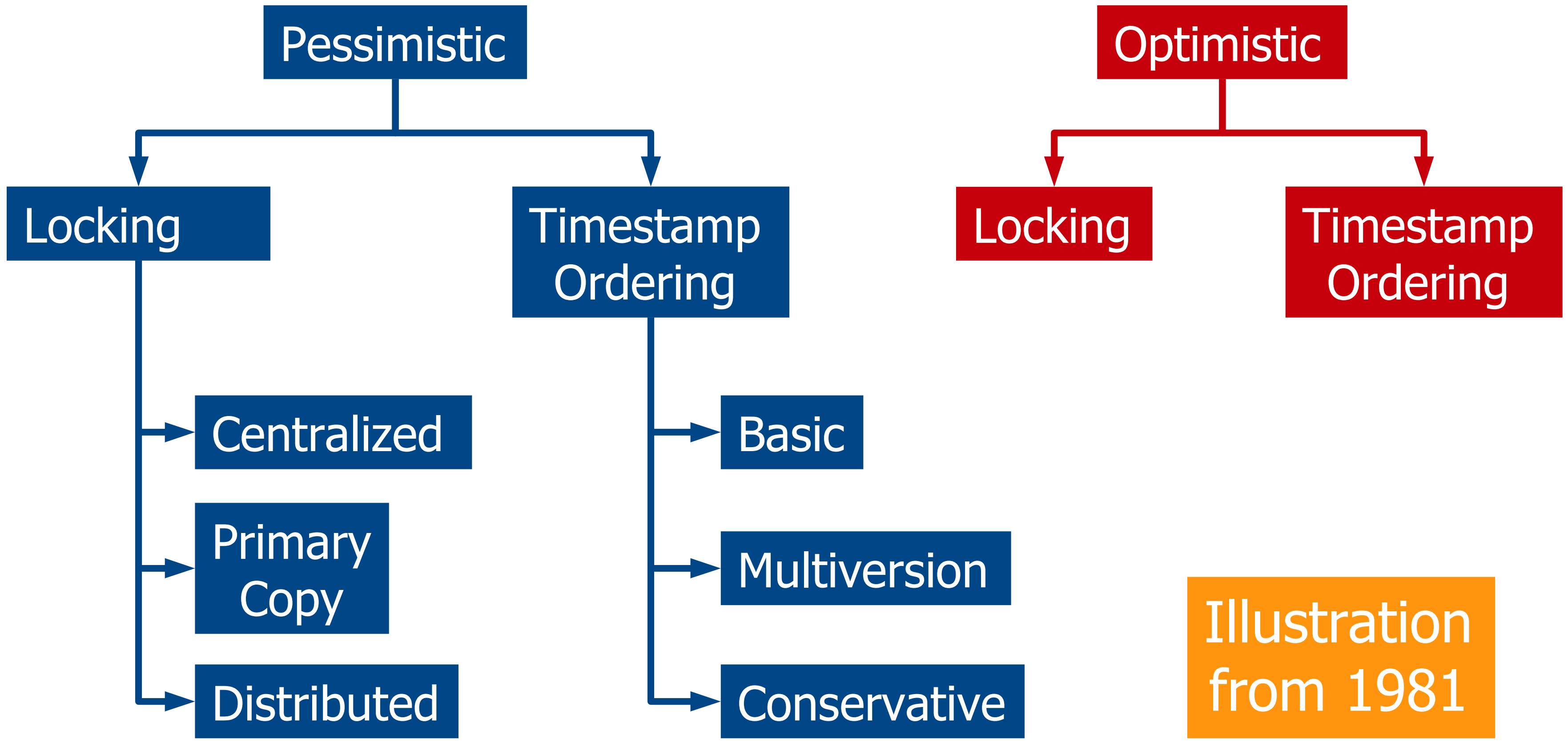
- Once a transaction has been comitted, it will remain so
- Guaranteed in the presence of failure: power outage, ...

The speaker says…

ACID transactions reflect the objectives that lead to the development of database management systems. Data should not only be properly structured (data model, schema) and easily accessible (relational algebra, standardized query language) but also be „safe“ in the context of a multi user environment.

Users either get all of their work comitted or none (atomic). The work does not violate any constraints and leaves the database in a defined, consistent state. Any two users ongoing transactions are isolated from each other until they commit. They do not interfere. And, once work is comitted, users can be sure that it is never lost.

Concurrency Control



The speaker says…

Assume two users and their transactions aim to modify the same data item at a time. Concurrency control algorithms deal with the question how the conflict is handled.

There is extensive, classical literature on transactions and concurrency control. For the sake of this presentation we only need to know that ACID transactions require concurrency control. The two major groups of algorithms in the field are the pessimistic and optimistic groups. The group of pessimistic algorithms is using locks to serialize access. Thus, pessimistic approaches check for conflicts at the beginning of a transaction. Whereas optimistic algorithms delay conflict detection to its end.

Isolation Level

Serializable

- Highest level. As if all transactions executed in serial order

Repeatable read

- Phantom read possible: no range locks
- T1: READ(age > 10), T2: ADD(age = 11), T1: READ(age > 10)
- T1 reads T2's uncomitted age = 11 row

Read comitted

- Non-repeatable read possible

Read uncomitted

- Dirty reads allowed

The speaker says...

ANSI/ISO SQL defines multiple isolation levels which control the degree of transaction concurrency. The lower the isolation level, the less work there is to be done by the concurrency control mechanism. Not only that, the lower the isolation level, the more concurrent accesses are allowed on one data item. Note, if you configure a low level, which is a bit faster, you can get read phantoms. One of them is shown on the slide. Imagine one transaction reads a range of values, say all kids older than 10. Another transaction inserts a new kid of age 11. The first transaction reads again a list of all kids older than 10. The list now includes the new 11 years old kid. This is called a phantom read.

You don't want ACID? MyISAM is still around but...

Physical level, *very* briefly

SQL tables, views and indices

- B[+|*]-tree
- Search, Insert, Delete: O(log n)
- Clustered Index

Disk level

- Pages
- Caching usually handled by specialized block managers

The speaker says...

Another slide that shows keywords I expect anybody in the room to be familiar with.

Traditional record oriented databases often store rows in 4-32KB sized subsequent memory areas called pages. A page holds one or multiple records. If a page cannot hold all the data of a single row, additional overflow pages may be used. To optimize for fast sequential disk read, systems may try to keep pages sorted on disk. Variants of B-trees are used to support search operations. If data is physically stored in the order of an index, one speaks of a clustered index. InnoDB, for example, is using a clustered index to layout a table on disk sorted by primary key. This makes sequential search on the PK particularily fast.

JOINs, *very*, *very* briefly

Different execution strategies

- Nested Loop
- Hash Join
- Assorted mixtures

Execution plan optimization

- EXPLAIN
- Indices

The speaker says...

We will not discuss Join strategies. It is out of scope. Only so much about it: the two basic ones are the nested loop join and the hash join. The nested loop approach iterates over the records of one table to do a record-by-record comparison with a second table using an inner loop. Of course, don't do this on your own in your application code. You don't have the additional search structures, such as indicies, available to make this fast. Hash joins are commonly used when joining a huge and a small table. For one of the tables a hash lookup table is build to thenspeed up the search. Assorted variants exist, there is no common naming for them. For example, MySQL does use hash join style optimizations but does not call them hash join.

Homework

To be studied before the next part begins

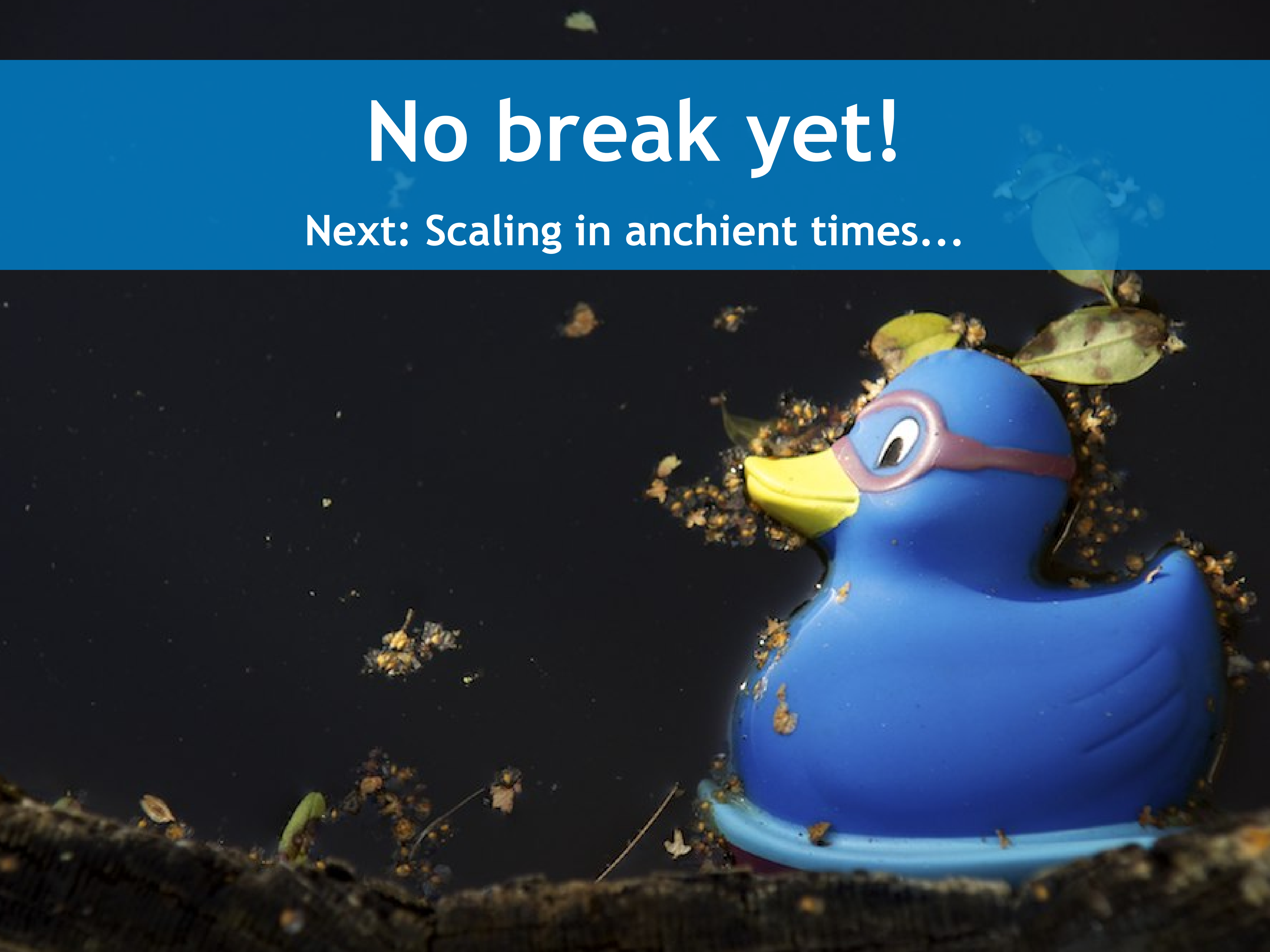
- Datenbanken: Konzepte und Sprachen (Saake, Heuer)
- Datenbanken: Implementierungstechniken (Saake, Heuer)
- SQL-99 Complete, Really (Gulutzan, Pelzer)
- SQL for Smarties (Celko)
- SQL Performance Explained (Wilander)
- Refactoring Databases (Sadalage)
- … any RDBMS vendor's manual
- … whatever you colleguages may recommend to you

The speaker says…

On purpose, we have gone through this introduction at a very high pace. The purpose was to remind you of some ideas and some priciples of traditional, relational database management systems.

As we move on in time and size of the database, we will explore systems that compromise on features. For reasons of breviety, there will be no discussion of what missing features means to you. There is a great amount of classic and new literature that you can study. It will answer your questions easily.

For today, we focus on the scalability vector and architectures.



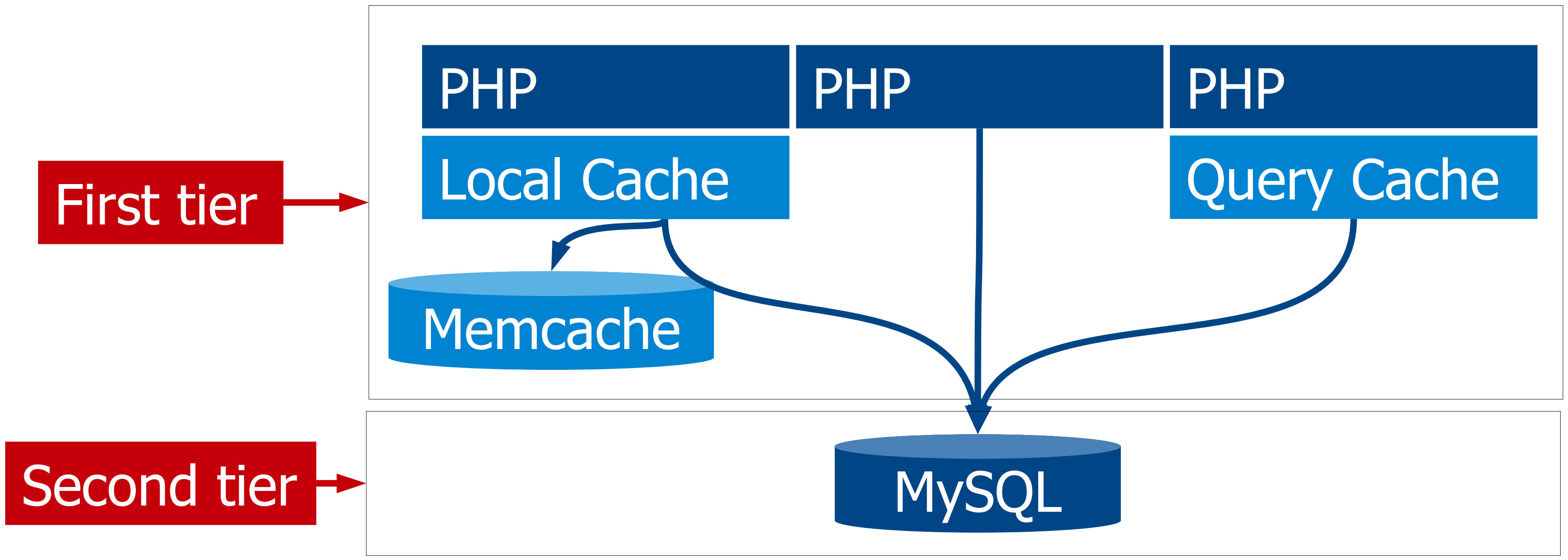
No break yet!

Next: Scaling in anchient times...

Ever since…

Cache as you can!

- Cache: soft-state, inconsistent, stateless
- Database: hard-state, consistent, stateful



The speaker says...

Once a database server is tuned, following all the good literature and tips available, one adds caches to both the pre-Cloud (;-)) and Cloud architecture.

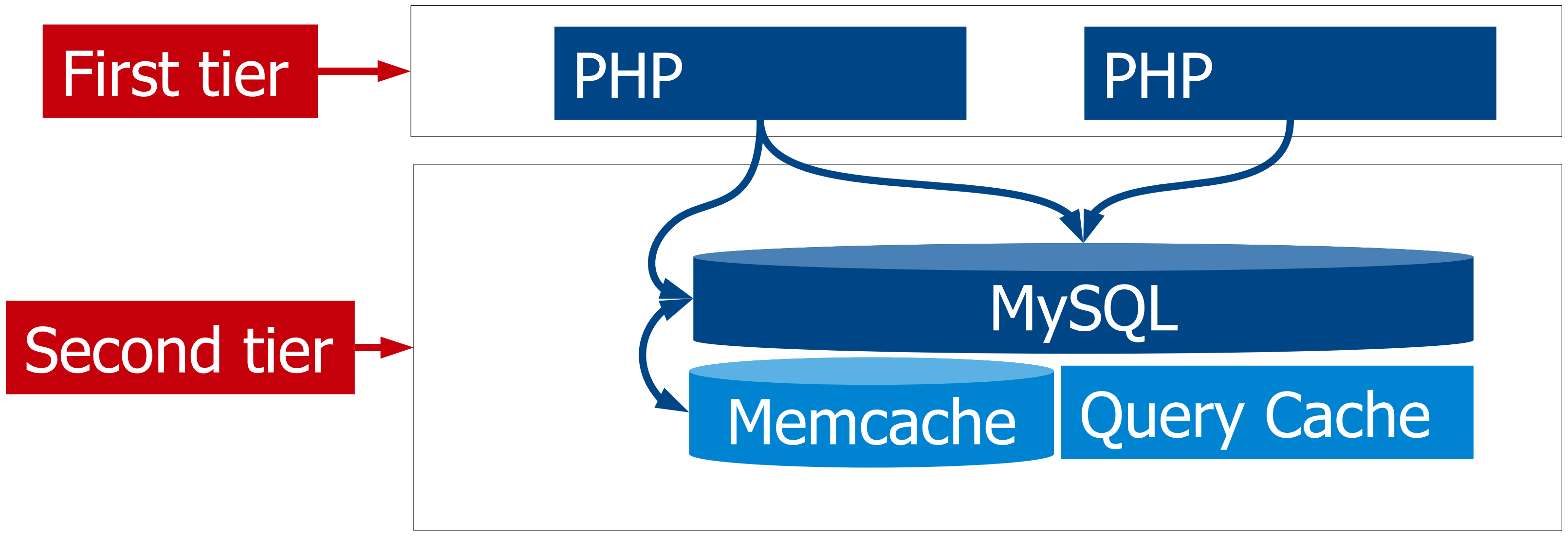
The caches belong in the first tier together with our PHP applications. Everything in the first tier is stateless. This allows us to scale up and down by adding or removing resources. Because of their stateless nature of all components, the caches hold soft-state information only. Information retrieved from a cache may be stale.

The second tier consists of databases hard-state: data returned is always correct.

MySQL? Decade of cosmetics...

The basic rule remains the same

- Database internal query cache: hard-state, consistent
- Fast access: MySQL speaks Memcache protocol



The speaker says...

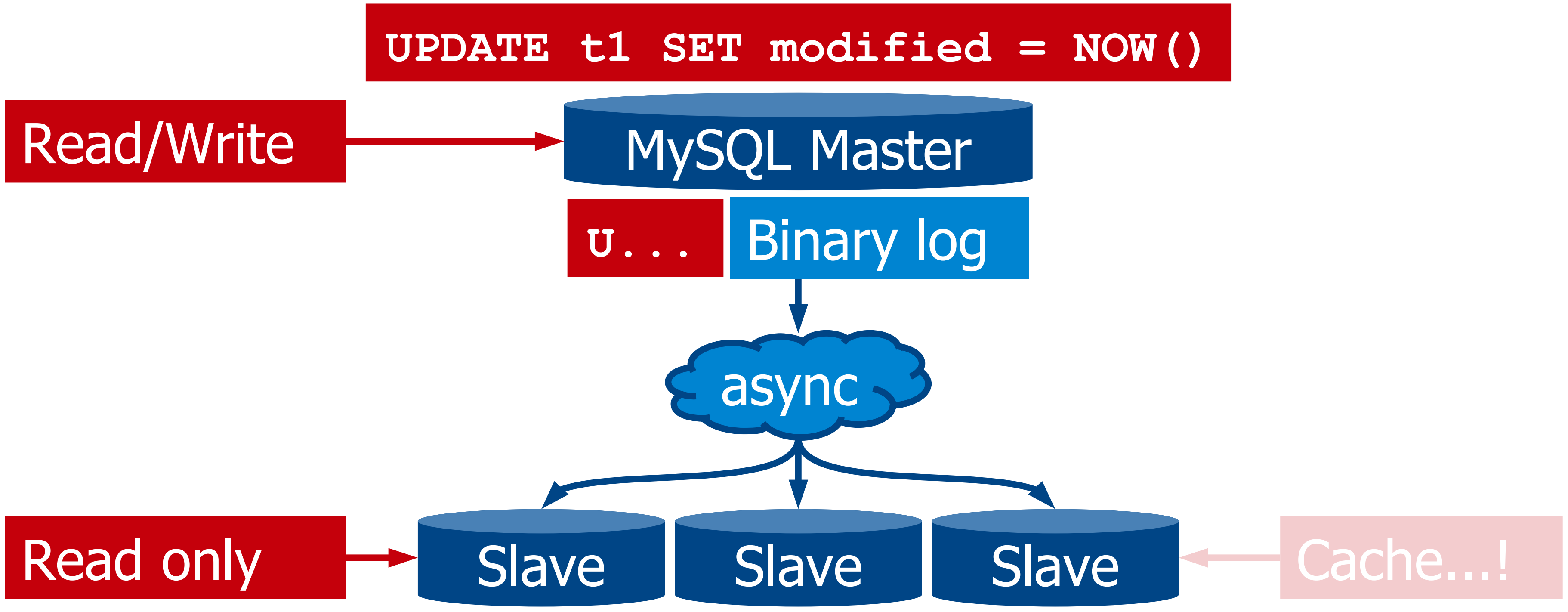
Until today the basic architecture is still the same with two differences. At some point, MySQL got a built-in query cache. The built-in query cache is always consistent.

As time passed, users voted against the HANDLER SQL statements that gave MySQL users access to the low-level record interface through SQL. Community realized that SQL means quite a significant parsing overhead and added new protocols to access the fast low-level record interface. Eventually, MySQL decided to integrate Memcache into the database server to simplify architectures. In MySQL 5.7 the idea behind finally becomes visible: direct access to InnoDB via Memcache protocol is faster than embeded Memcache.

Added --safe-updates options

MySQL 3.23 Replication

- We got not clue, we got no time, BUT we need something !?
- Still, a success story because it came just in time



The speaker says...

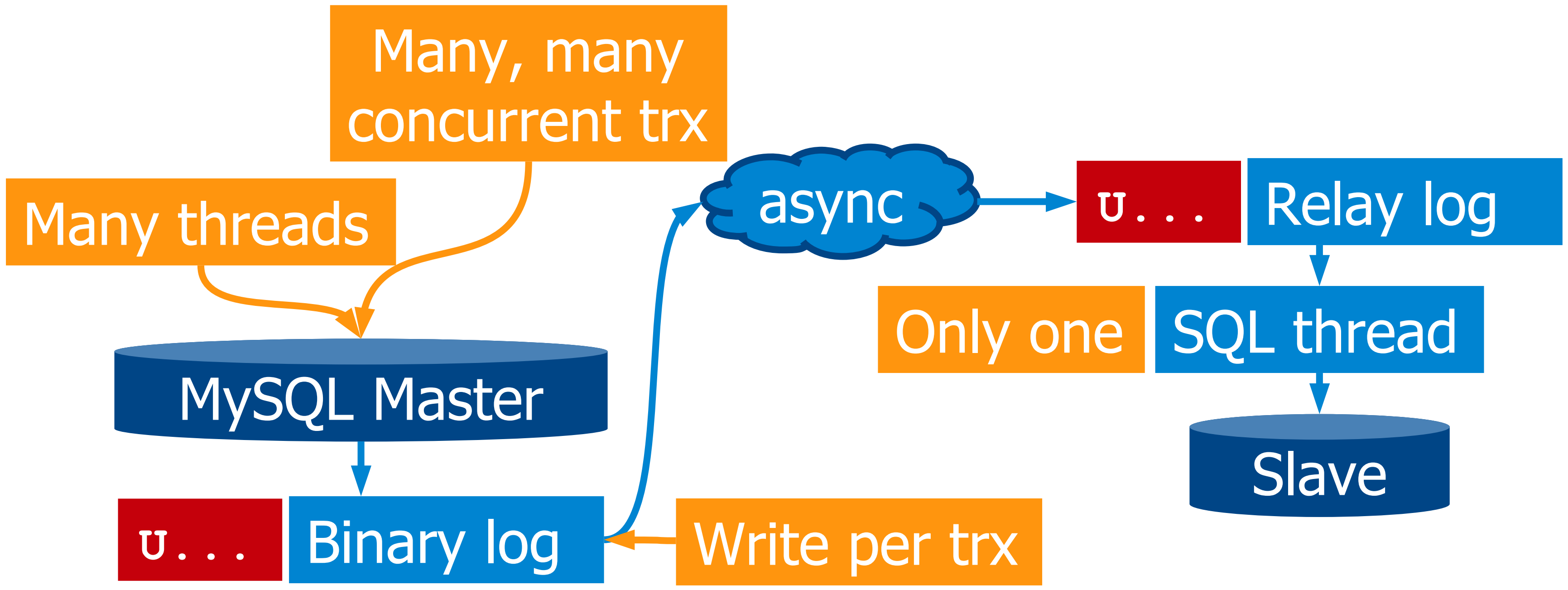
MySQL's reply to scalability was master-slave style replication. Clients direct all their write requests to a master. The master sends the updates asynchronously to the slaves. The clients can now direct their reads to the slaves, given that the application logic allows for slightly outdated data. Assuming that stale data is acceptable is not totally off. We all do: Cache as we can... In an read dominated environment, such as most web applications this is a great scale out strategy. Adding slaves cost you about nothing.

In retrospect there is one or another anecdote to tell about the implementation, tough...

MySQL? Decade of cosmetics...

Implementation anecdotes...

- Statement based, row based and mixed binlog format …
- Master servers never fail, if they do there are 3rd party...
- … as often: good enough for now, fixed in latest releases



The speaker says...

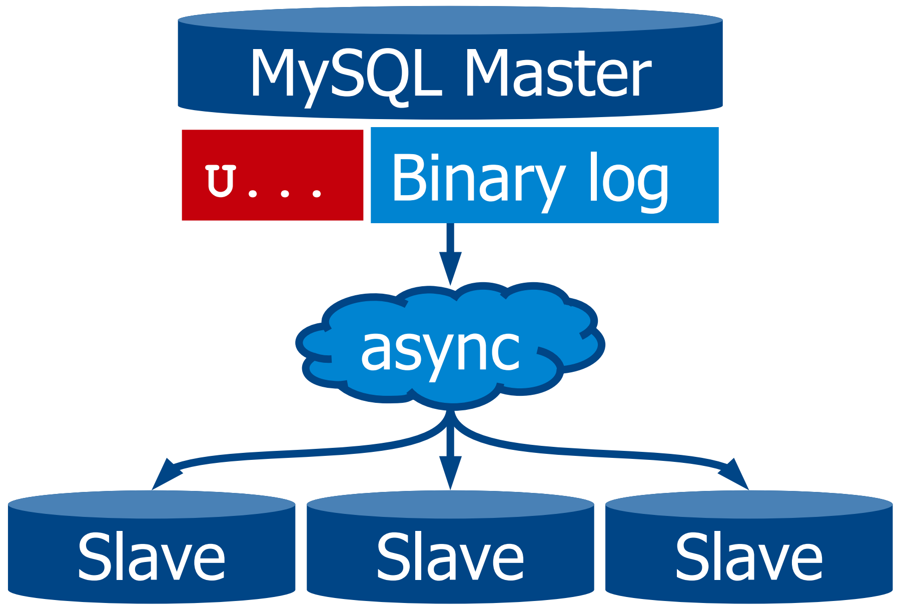
MySQL's reply to scalability was master-slave style replication. Clients direct all their write requests to a master. The master sends the updates asynchronously to the slaves. The clients can now direct their reads to the slaves, given that the application logic allows for slightly outdated data. Assuming that stale data is acceptable is not totally off. We all do: Cache as we can... In an read dominated environment, such as most web applications this is a great scale out strategy. Adding slaves cost you about nothing.

In retrospect there is one or another anecdote to tell about the implementation, tough...

It does not scale...

No write scale out possible

- Master is a single point of failure
- Read/write splitting is annoying – even with PECL/mysqlnd_ms
- MySQL should do their homework ...



The speaker says…

The approach taken by MySQL seems nice on the first look, however, it fails to scale writes.

There is only one master to process all the writes.

Homework? MySQL show !?

Should I join another track?

Your alternatives:
Modernes Frontend mit JavaScript,
Websecurity im Alltag

The speaker says...

No, no.... There is no MySQL show coming.You can find plenty of MySQL presentations on slideshare: http://www.slideshare.net/nixnutz/

We will very soon stop mentioning MySQL at all!

Homework: objectives

Availability

- Cluster as a whole unaffected by loss of nodes

Scalability

- Geographic distribution
- Scale size in terms of users and data
- Database specific: read and/or write load

Distribution Transparency

- Access, Location, Migration, Relocation (while in use)
- Replication
- Concurrency, Failure

The speaker says...

Ok, let's do the homework. For any MySQL engineer to work, it requires a list of objectives to be achieved. Luckily, any book on distributed databases can give that list – see the slide.

What kind of cluster?

When does synchronization happen?		Where are transactions run?	
		Primary Copy	Update Anywhere
	Eager	Not available for MySQL	MySQL Cluster 3rd party
	Lazy	MySQL Replication 3rd party	MySQL Cluster Replication

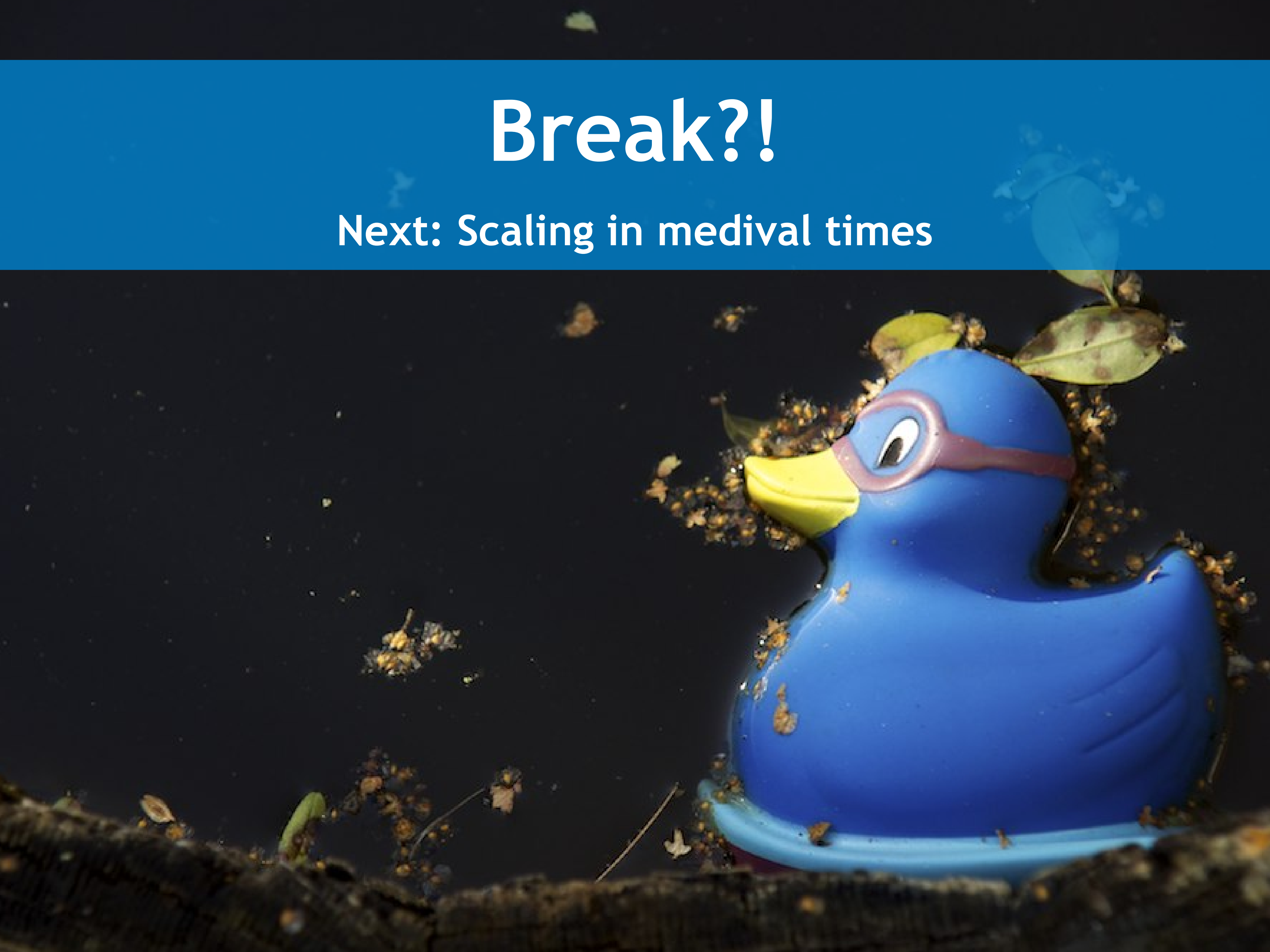
The speaker says…

Once the objectives are clear, we can start to categorize the existing MySQL solutions to analyze the gaps. One easy way to categorize the options is to ask two simple questions:

Where can any (read or write) transaction be executed?
When does synchronization between nodes happen?

Break?!

Next: Scaling in medival times



Cute... But, forget MySQL!

Why? Because! Why? Because!

Traditional RDBMS are dead!

Why? Because! Why? BECAUSE!

MySQL is not Cloud ready!

Why? BECAUSE! Why? BECAUSE!

BECAUSE OF CAP 111!!!!!

W… I WILL LOCK YOU IN THE CABINET!!!

This is Merlin. Since medival times people known one should never mess up with Merlin!

CAP: Brewer's conjecture

It is not possible to create a distributed system which provides all of the three guarantees: Consistency, Availability and Partition Tolerance.

We shall design systems that are always available. This comes at the price of weak consistency.

The speaker says...

In the mid-1990's, when I printed my first hard copy of the PHP 2.0 manual, Eric Brewer was working on cluster-based wide-area services such as search engines, proxy caches and content distribution systems. His experience from that time made him formulate the CAP conjecture in 2000.

What he stated should not surprise anybody who does web development since then! Back then, we all used caches (soft-state) already. We all wanted web application to be fast. We all learned to cope with stale data. We all knew the difference between soft-state and hard-state. We all learned that it is fine for a stateless cache to crash, or become disconnected. Brewer knew that too.

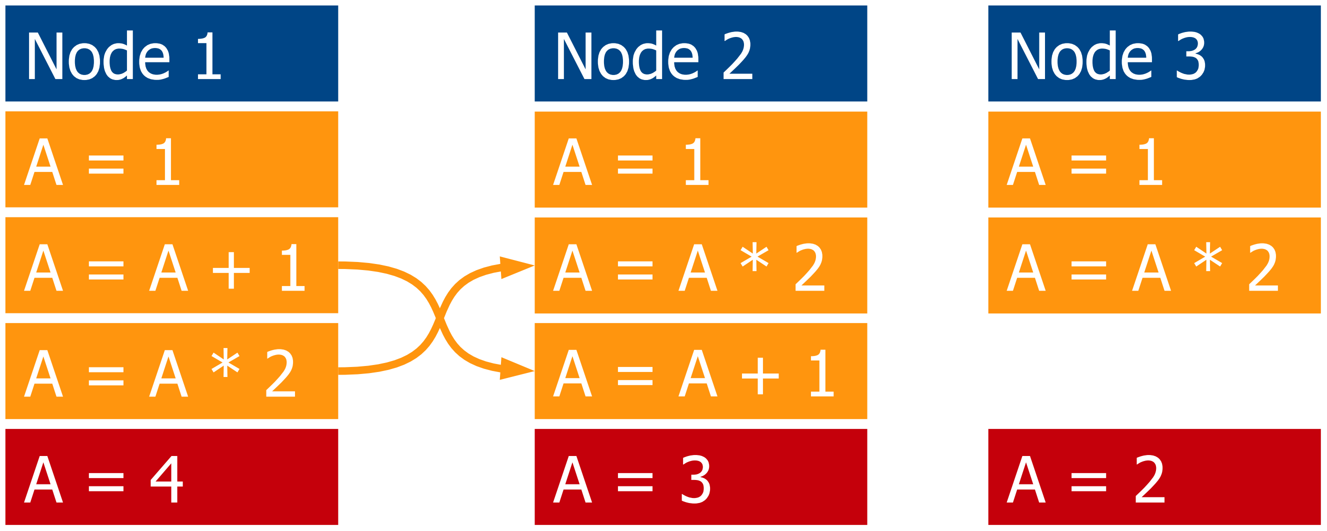
Theory!

CAP

Consistency in CAP

All nodes see the same data at the same time

- Means: all updates are applied to a value in agreed-upon order
- Durability: update confirmed to client must never be lost
- Note: unrelated to C in ACID



The speaker says…

Consistency means that all nodes in a distributed system shall see the same data at the same time.

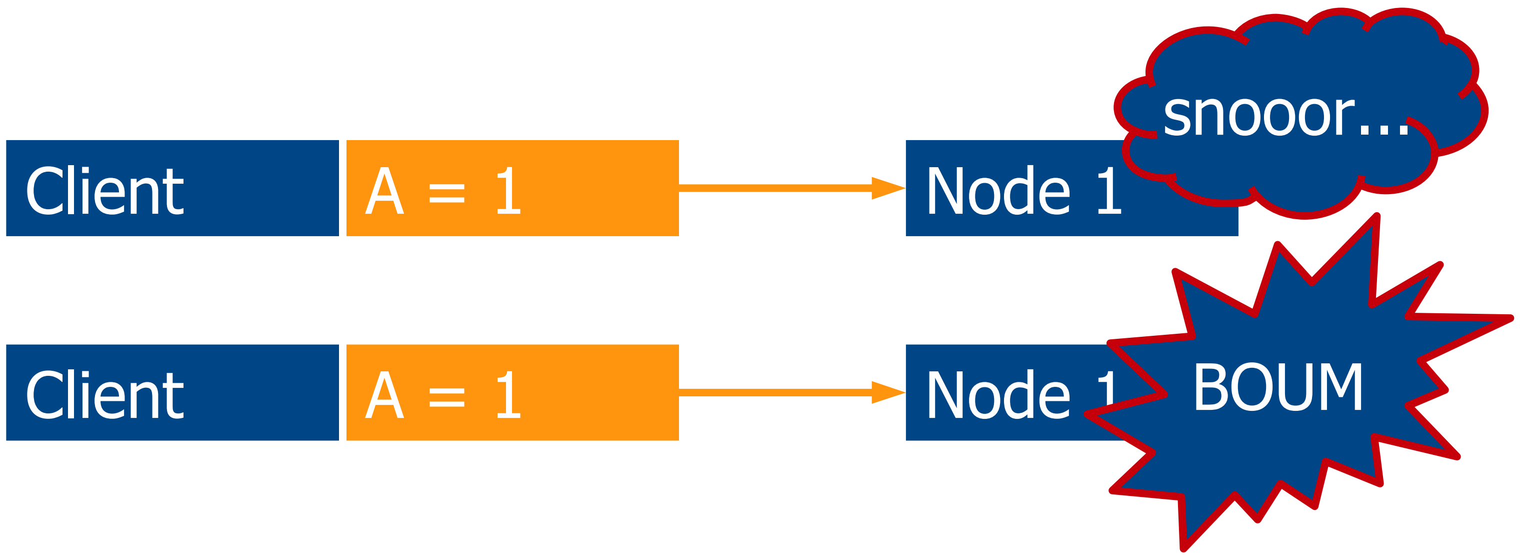
To stay consistent ever node in the distributed system must get the same update commands in the same order. Say you have a variable a = 1. A client sends one node the update a = a + 1, immediately followed by the update a = a * 2. The node now sees a = (1 + 1) * 2 = 4. If another node gets the updates in reverse order, it will compute a = (1 * 2) + 1 = 3. The values have diverged. Thus, updates must be delivered to all nodes in agreed upon order.

Also, none of the updates must get lost.

Availability in CAP

Client always gets response whether request failed or not

- Means: service survives crashed/unresponsive replicas
- Performance: client never waits even if data source is gone
- Brewer wanted *almost* always a response originally



The speaker says…

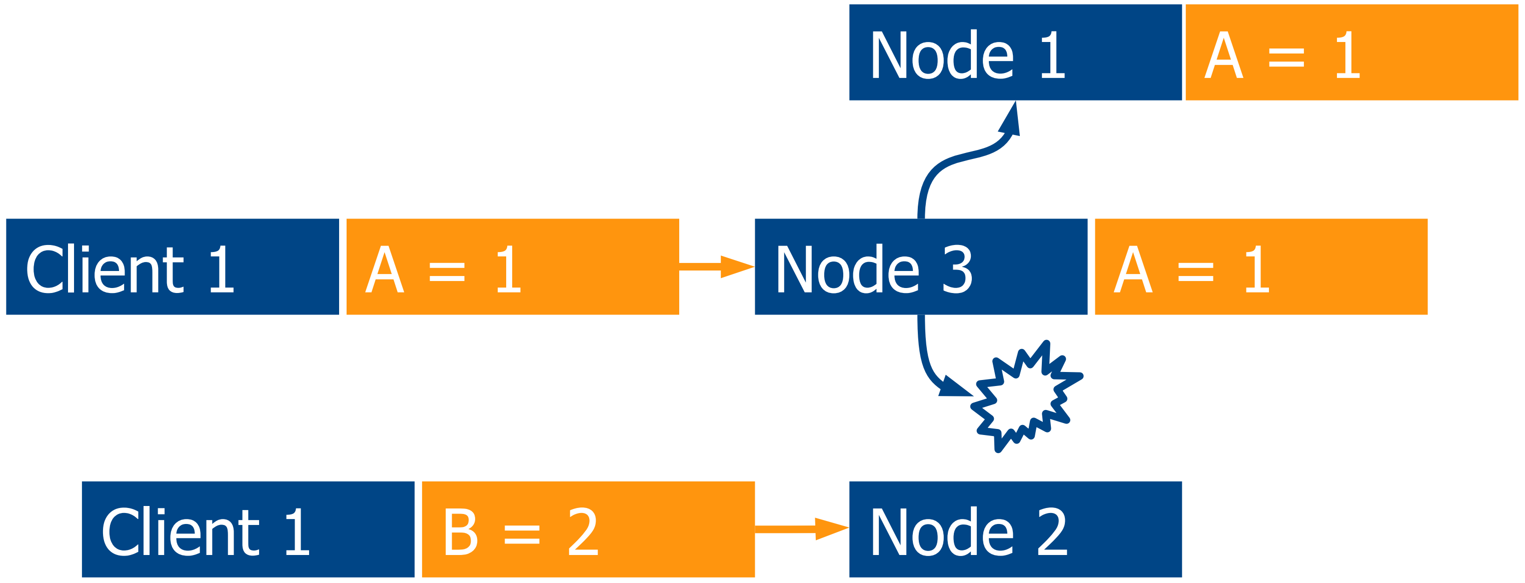
Availability means that a client always gets a reply whether his request has succeeded or failed.

If, for example, the node the client is talking to is crashing in the very moment the client has delivered an update to the node but before the message has reached the others in the group how does the client know whether it succeeded or not? The crashed node cannot sent a reply to the client. Thus, the client could assume the update failed. Did it? A similar situation arises if a node is unresponsive: crashed or snooring?

Partition Tolerance in CAP

Service survives arbitrary message loss or failure of parts

- Means: network failures/cutting off replicas does not block us



The speaker says…

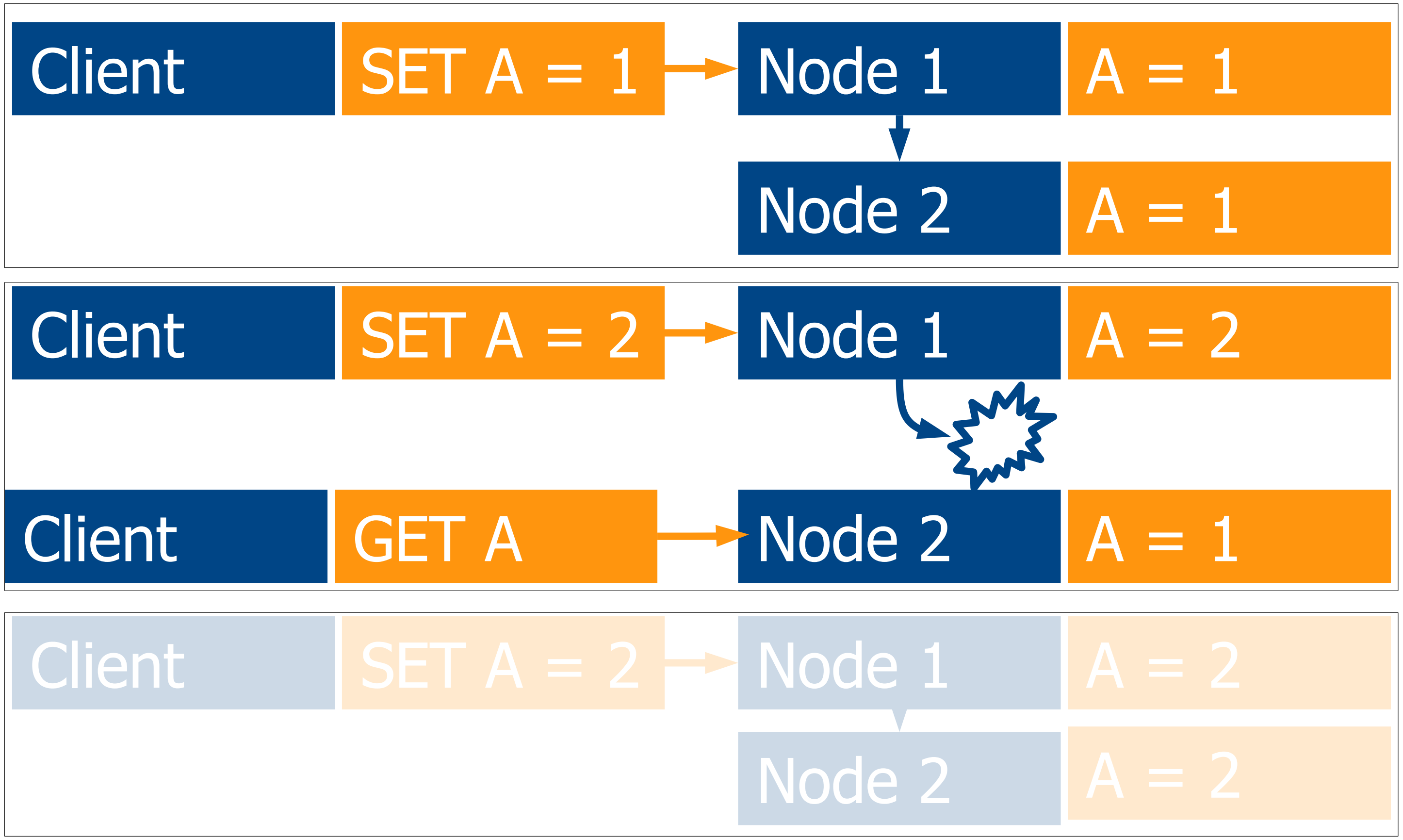
A distributed system is partition tolerant if it survives arbitrary message loss or the loss of parts of its system. No matter whether two nodes loose their network connection or a node crashes, the service must remain operational to be called partition tolerant.

Network partitions may not be frequent in data centers, say within a rack or between two racks, but they can happen in wide area networks, like the Internet.

CAP Theorem

Proof by Gilbert and Lynch (2002)

- Narrower than Brewer had in mind



The speaker says...

Two years after Brewer's conjecture the CAP theorem followed.

The slide might be self-explaining. In the presence of a network partition, we cannot guarantee that one client can observe another clients update from a different partition. The best we can do is buffer updates and resend them after the partitioning is over. Then, under further conditions we touch later, we may be able to consolidate the data eventually.

Use BASE!

Basically Available

- … but not always

Soft-state systems with

- … like a cache, not a hard-state as in ACID

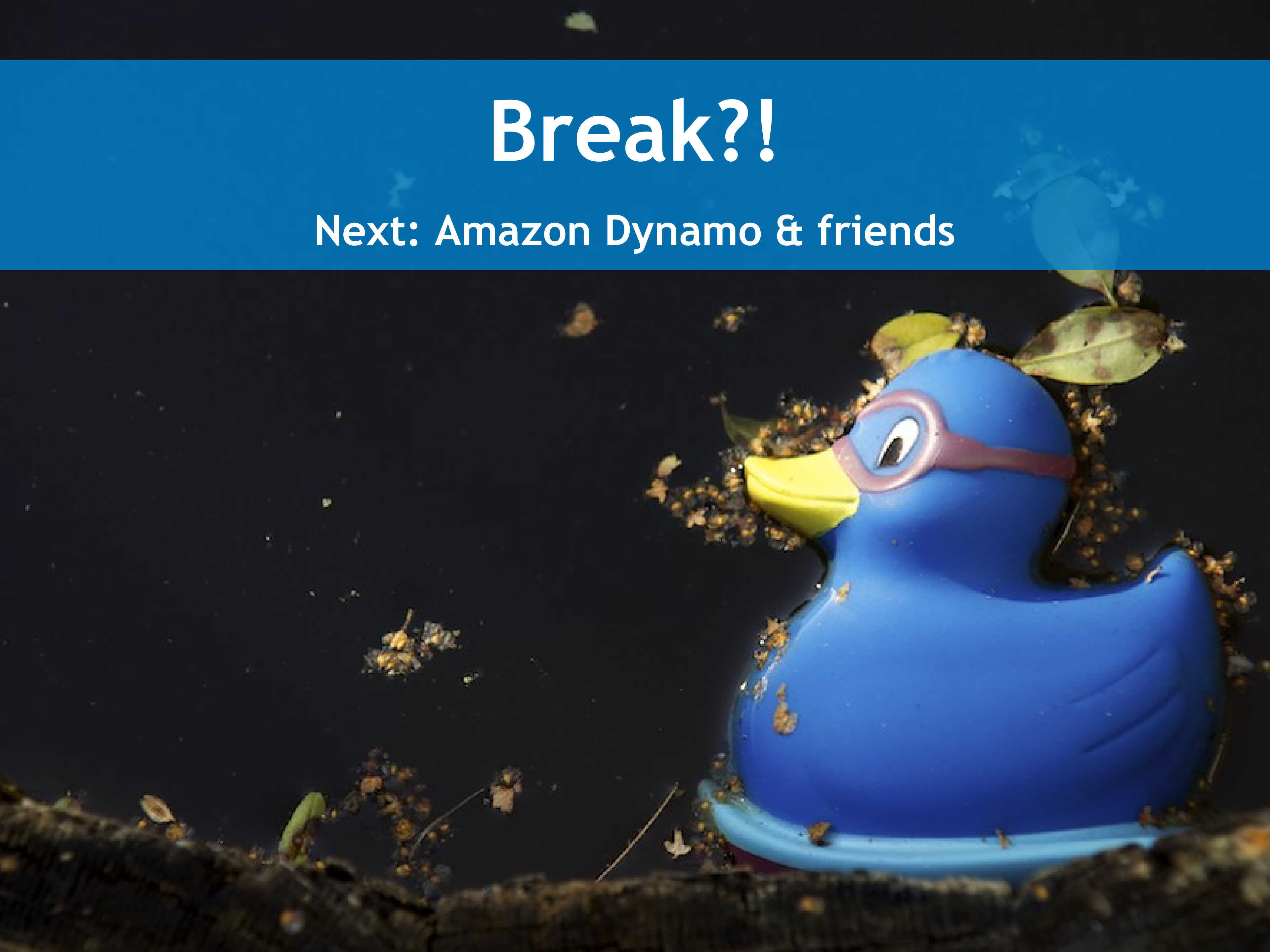
Eventual-consistency

- … we might end up with correct data eventually

The speaker says…

The CAP discussion has lead to BASE. BASE recommends to use weak consistency models.

Does it say more than: use caches? I mean, anything new here for anybody who has developed web apps in 1997 already... Whatever...



Break?!

Next: Amazon Dynamo & friends

Early NoSQL cloud trends

Distributed Hash Tables

- Focus AP in CAP
- Systems: Chord, Amazon Dynamo, Riak

Google Bigtable and clones

- Focus C(A)P in CAP
- BigData
- Systems: GFS, Chubby, Bigtable, Hbase

The speaker says...

Puuuh, given all these issues ranging from CAP to limited speed of light, the totally outdated RBDMs technology from the 70th must be dead! Well, no.

But, let's first analyze two early, massively scalable NoSQL systems for the cloud era. Two systems that seemingly took very different roads with regards to CAP. Amazon Dynamo is clearly an AP system: always available, always!

Whereas Google Bigtable is consistent, partition tolerant and from a practical standpoint also highly available – a C(A)P system?!

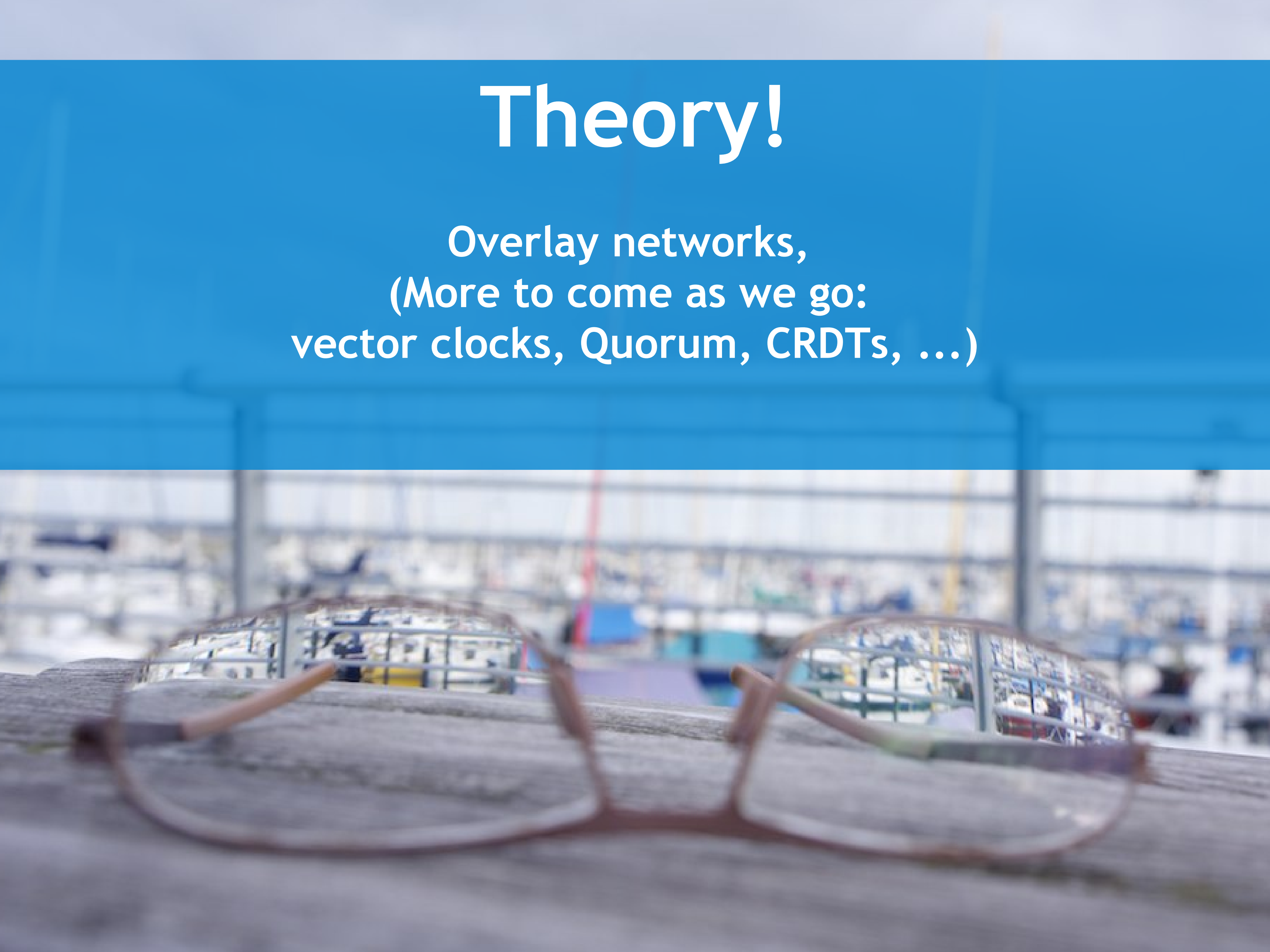
On our way to Dynamo

Distributed Hash Table

- Overlay networks
- Chord

Theory!

Overlay networks,
(More to come as we go:
vector clocks, Quorum, CRDTs, ...)



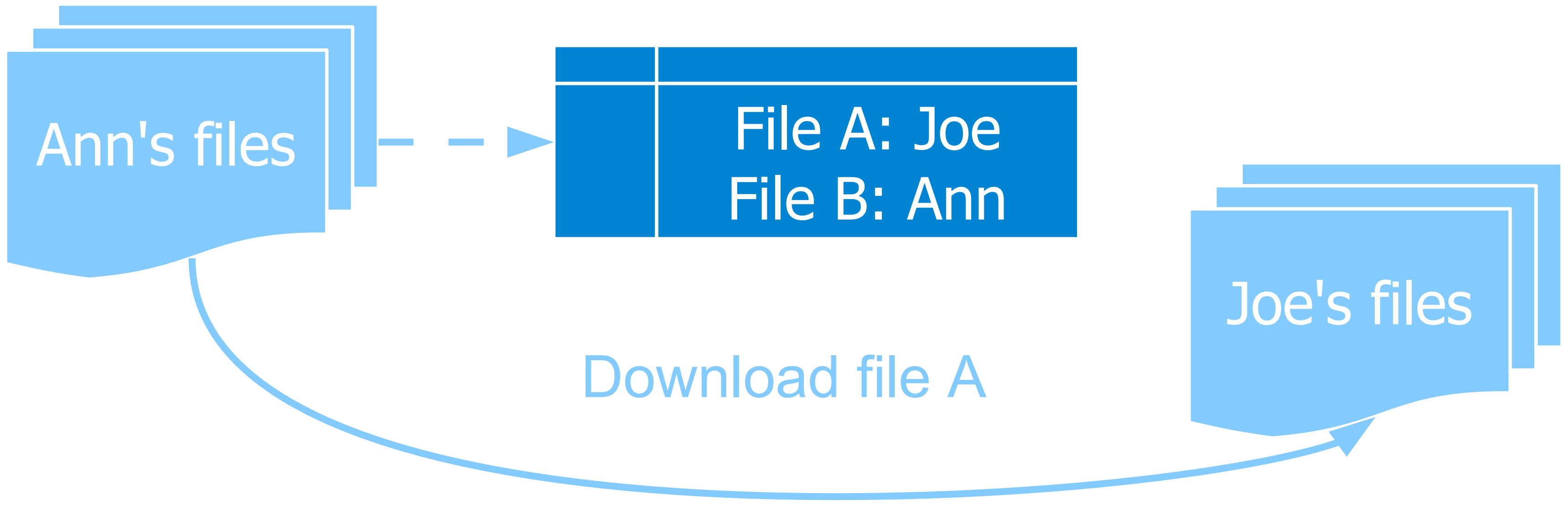
The speaker says…

You are lucky: it is quite a short and simple introduction that leads us to Dynamo. Very much unlike what we will have to master later on when we prepare for understanding Google Bigtable.

Distributed Hash Tables

Origin peer to peer networks

- Overlay networks for file sharing, e.g. Napster, Gnutella, …
- Central service keeps routing information only



The speaker says…

Peer to peer file sharing networks use overlay networks for their services. Distributed Hash Tables have been developed for routing requests in such networks.

Assume Ann and Joe are willing to participate in a file sharding network. Both allow a central service, for example, Napster to track the files they want to share. If Ann searches a file, she asks the central service for a download location of the file. The service looks up locations of the file and tells Ann where to find and download it. Then, Ann can attempt to connect the locations and download the file. The central services holds nothing but routing information.

Chord

P2P overlay network using a DHT

- Stoica et. al. 2001, core API: put(key, value), get(key)

Scalable

- Average search time is O(log n), n = number of nodes

Loadbalancing

- Consistent hashing for an even key distribution

Decentralized

- (Almost) no special nodes, no single point of failure (SPOF)

Availability

- Adapts to network topology changes automatically: node failure, node addition, ..

The speaker says…

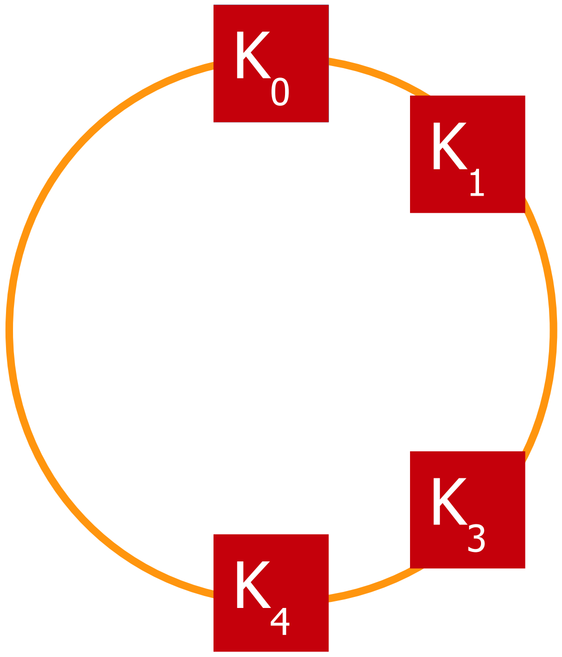
Chord is a P2P overlay network using a Distributed Hash Table. Chord takes a set of machines and organizes them as a cooperative online memory system. Clients can store and retrieve values using the most basic API possible: put() and get(). Strictly speaking there is only one operation: lookup(key) to map key to a network node.

Chord is highly scalable. Search operations require O(log n) time to complete. Consistent hashing ensures even key distribution over the nodes and thus good load balancing. Considering storage, all nodes are equal and there is no single point of failure. And, very nice, the system automatically handles topology changes such as node addition and failure.

The idea, step 1: Virtual ring

Organize keys along a ring

- SHA1, 160 bit long hash key
- Important to avoid collisions



The speaker says…

Let's get the idea, step by step.

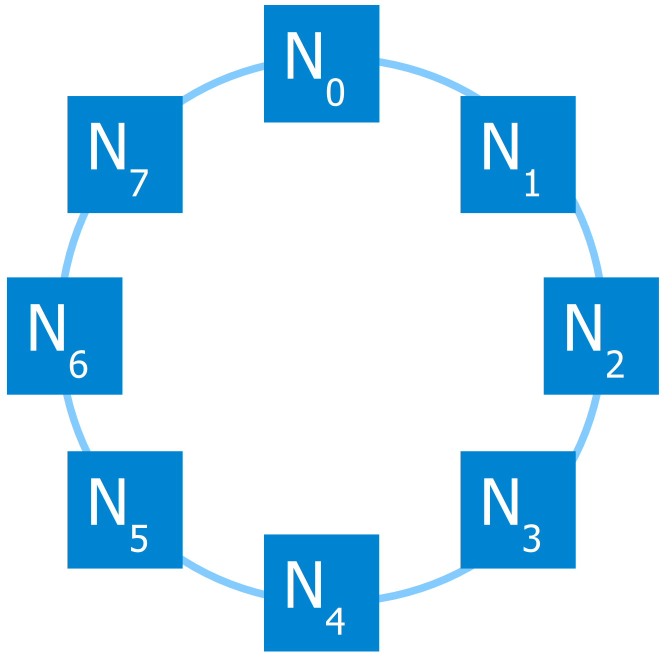
Chords main task is to lookup a network node for a user supplied key: lookup(key).

Chord takes the key and computes a hash key from it. By default SHA1 gets used. SHA1 returns a 160bit long hash key, which means the key space has 2^{160}-1 values. All the possible hash keys are organized along a virtual ring. The hash keys are sorted and we put them clock-wise on the ring. Such a huge key space is required to avoid collisions.

Step 2: map nodes to the ring

Use node IP as key

- A good hashing function will spread the nodes evenly



The speaker says…

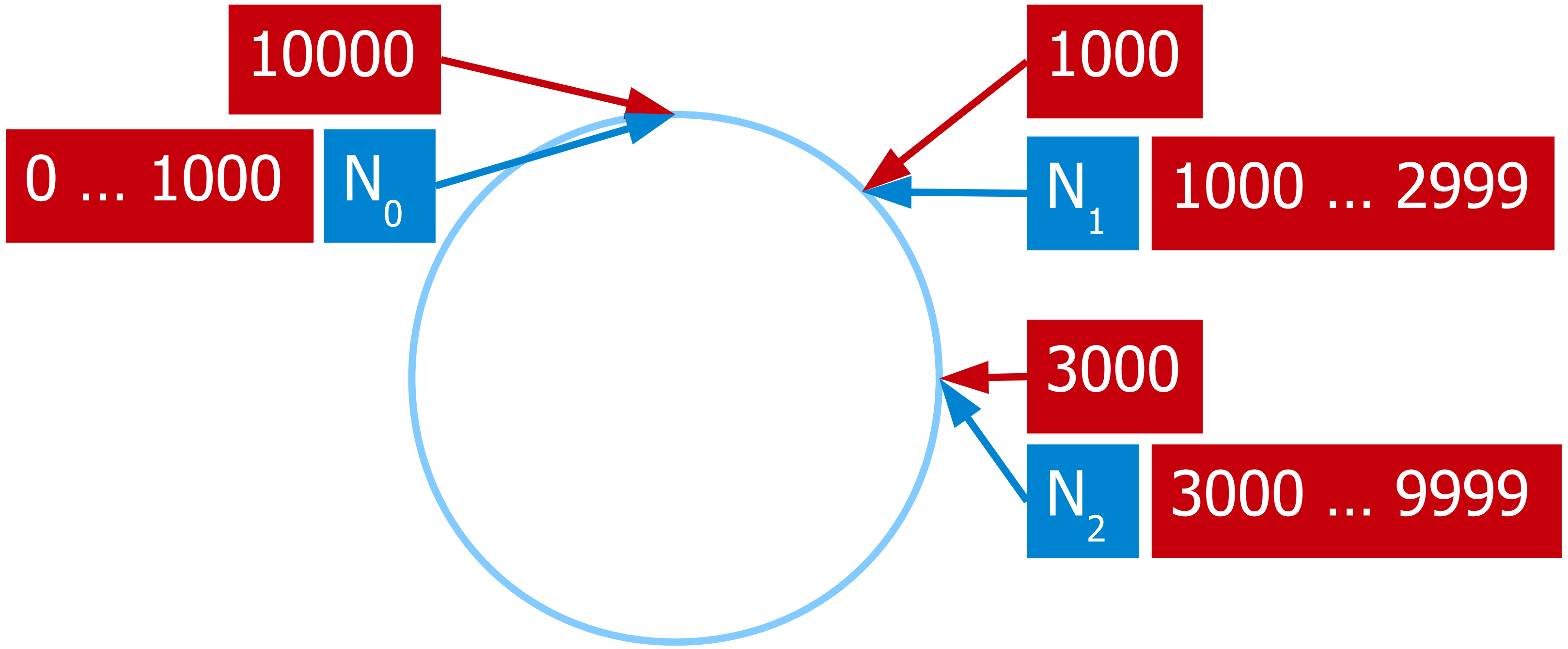
Next, we take a number of nodes and map them to the ring as well. As a key for the mapping we use the IP address of the nodes.

If all goes well, the nodes are evenly distributed on the logical ring, just as in the picture.

Step 3: Map nodes to key ranges

Node N owns keys which belong to his position on the ring and up to the position of the next node

- Size of key ranges per node may differ, a bit



The speaker says…

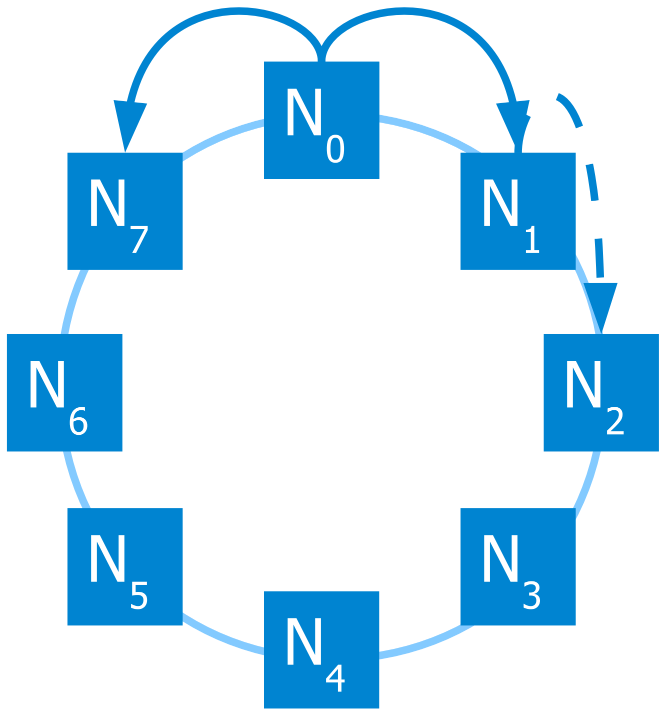
Every node has a position on the ring. Every position on the ring belongs to a certain hash key. The node at a certain position on the ring is responsible for all hash keys from the one which belongs to its position up to and excluding the hash key that belongs to the next nodes position.

(Hash) Key ranges may differ among nodes, however, the illustration shows an extreme example. Chord is using consistent hashing. At average each node handles K/N values. The benefit: when nodes are removed or added, no more than K/N values need to be shuffled.

How search not works…

Each node knows its predecessor and successor

- If you don't own searched key, forward to appropriate neighbour until key is found
- O(n/2)



The speaker says…

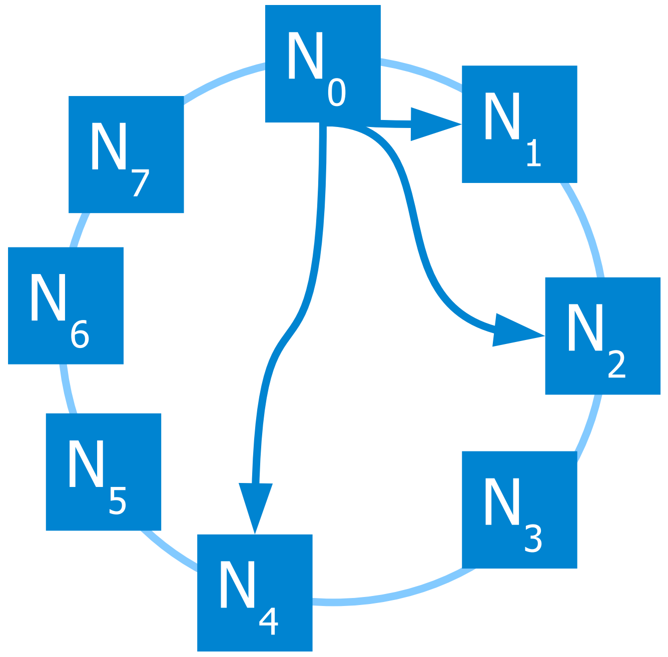
Note that there is no central place where we could simply look up the node that holds a key. The system is decentralized and dynamic.

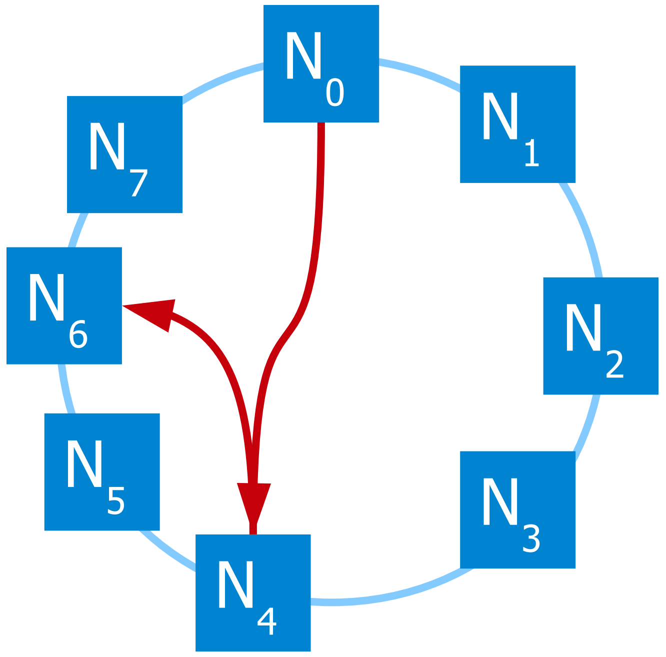
Every node knows its predecessor and successor. If a node searches for a key that it does not hold itself, it could forward the request to the appropriate neighbour. This logic could be applied until the key has been found. If so, a key is found on average after n/2 hops if there are n machines.

Search using finger table

Lookup table at each node

- In a 2^m key space the lookup table has m entries
- Sort of binary search - O(log n)





The speaker says…

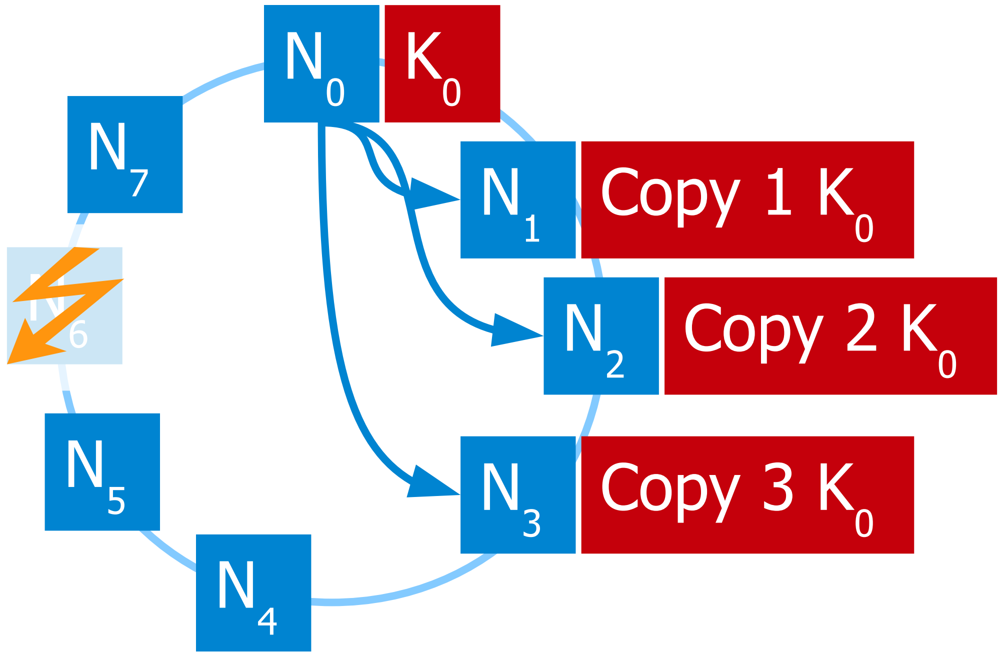
A small lookup table can be used to speed up search. In a 2m key space every node keeps a table of m finger pointer. From the original paper: **The i-th entry in the table at node n contains the identity of the first node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e., s = successor(n + $2^{i -1}$), where 1 <= i <= m (and all arithmetic is modulo 2^m).**

In other words: a node can forward a request to another node in such a way that with every hop the number of remaining nodes that may hold the searched key is halved, very much like in a binary search. To further improve performance nodes will use caches to remember the nodes for certain keys.

Fault and partition tolerance

Replication of data at log n successive locations

- Keep record of log n predecessors and successors
- Failure triggers automatic actions to regain replication factor



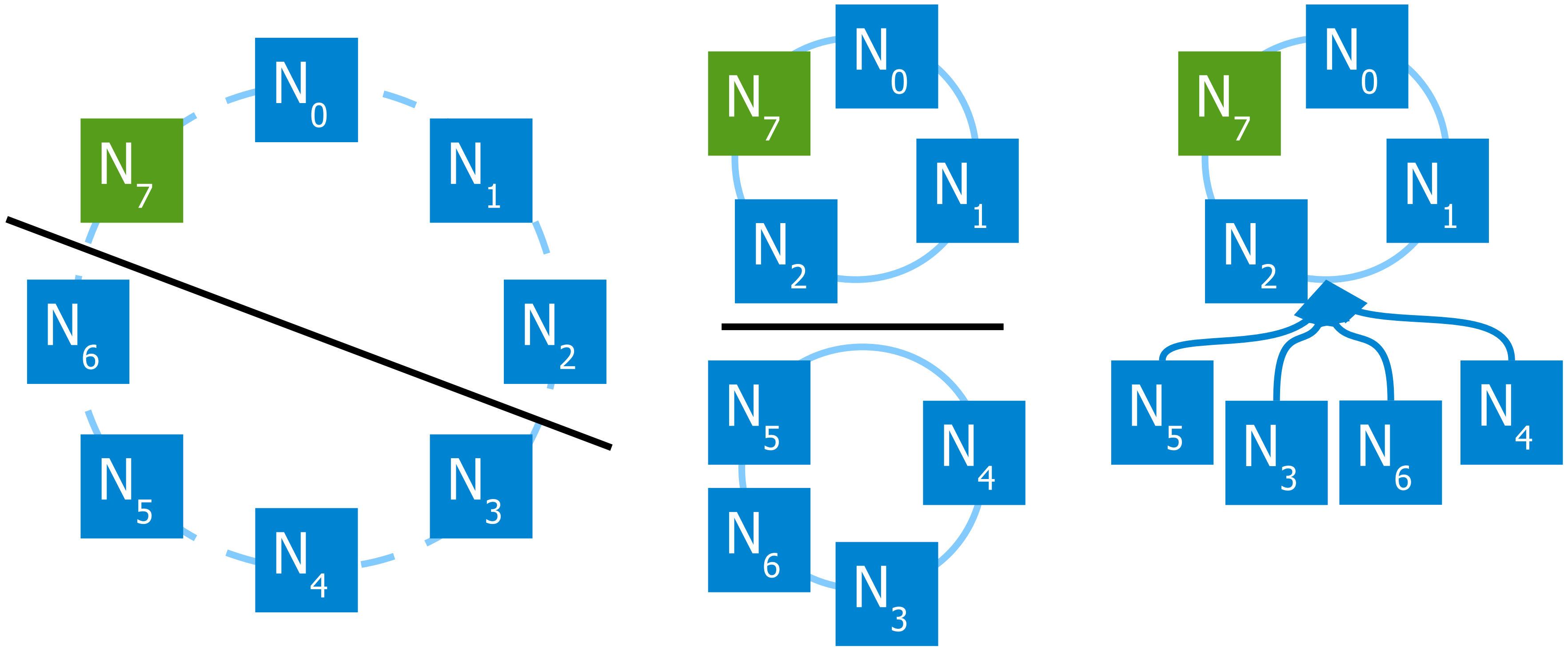
The speaker says…

To improve failure tolerance Chord keeps track of more than its immediate predecessor and immediate successor. It tracks log(n) neigbours.

Also, data is not stored at one machine only but at many. The replication factor is log(n).

Partitioning

Network partitioning may split rings temporarily



The speaker says…

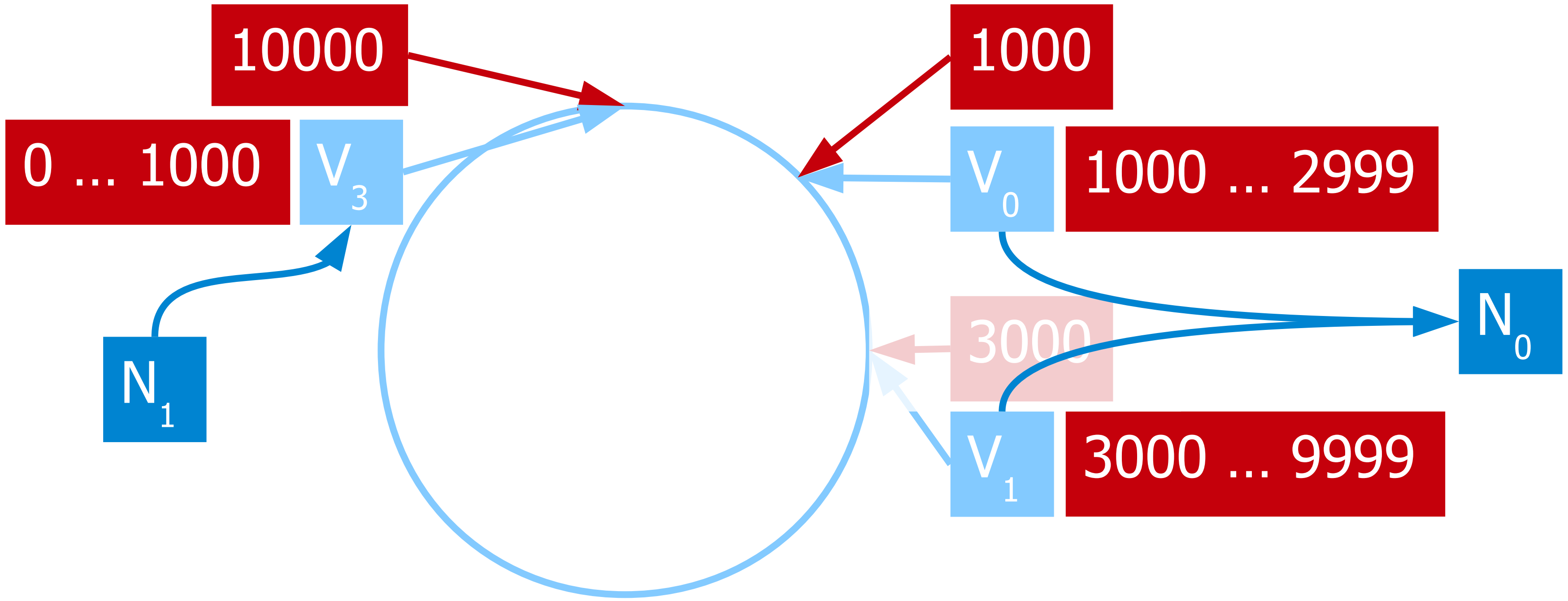
Assume you have a Chord with some nodes in a European data center and others in an Asian data center. If the network connection between the two is interrupted, Chord will form new rings in each of the now disconnected data centers.

Once the outtage is resolved and the nodes can see each other again, the rings will try to merge. There are special landmark servers. Nodes in a ring with no landmark server will leave their ring and join the ring with the landmark server. After some time, the rings will be merged into one.

Amazon Dynamo

Classic use case shopping cart

- DHT, collection of key-value tuples, focus avaiability
- Introduces virtual nodes for fine-grained load balancing



The speaker says…

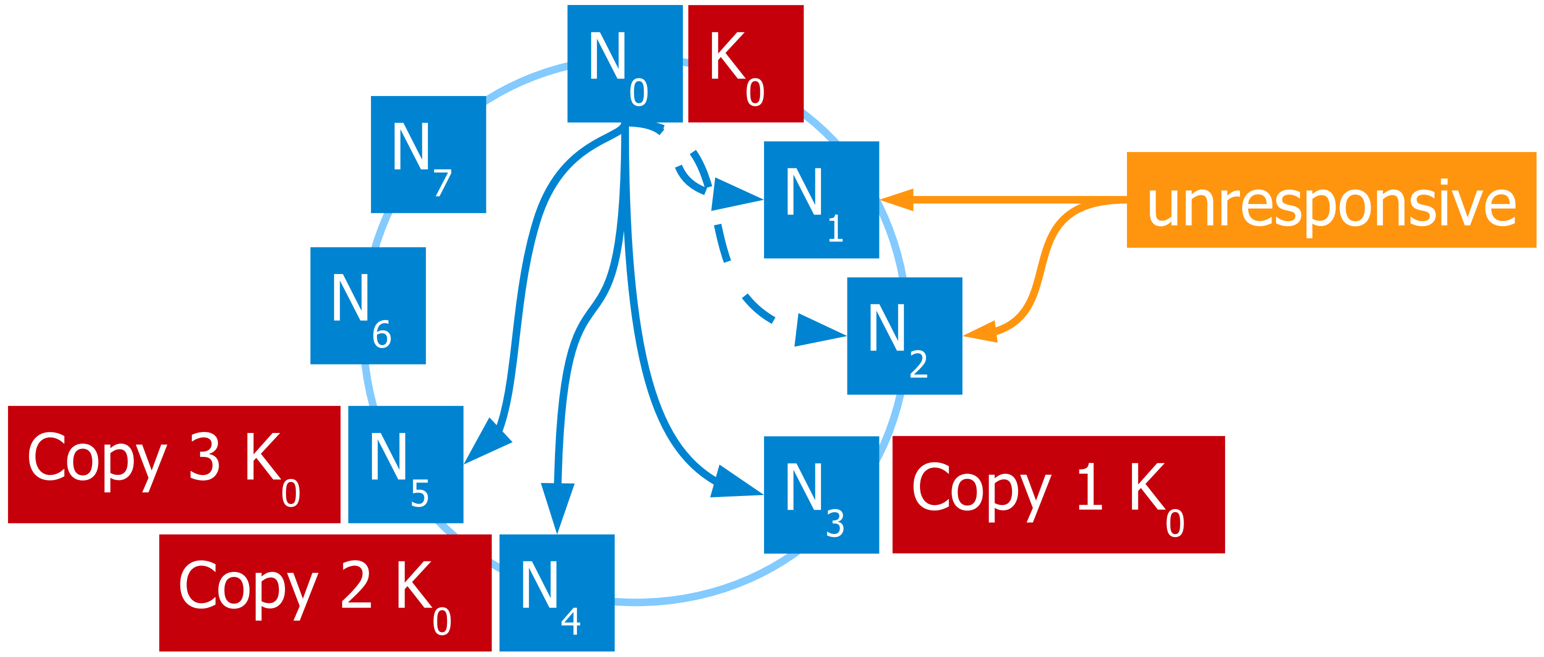
Amazon unveiled the architecture of its Dynamo key-value store in 2007. At its heart it is a DHT similar to Chord. The primary use case is large-scale e-commerce and namely the shopping cart of Amazon. It is designed for maximum availability and best possible response times even in the presence of failures (AP in CAP) but offers low consistency only by default (C in CAP).

Load balancing in heterogenous environments is improved by assigning a variable number of virtual nodes to a physical node. This way, more powerful physical nodes can be assigned more virtual nodes than less powerful ones.

Fast response, sloppy quorum

Replication of data at n successive, responsive nodes

- Users can observe „lost" cart items
- Self-repair process (gossip style) moves keys to their appropriate nodes



The speaker says…

Chord stores a value on log(n) successive nodes to ensure fault tolerance. Dynamo developers have been worried that if a successive node is unresponsive, it delays a put() operation in an unacceptable manner. Thus, put() will skip unresponsive successors.

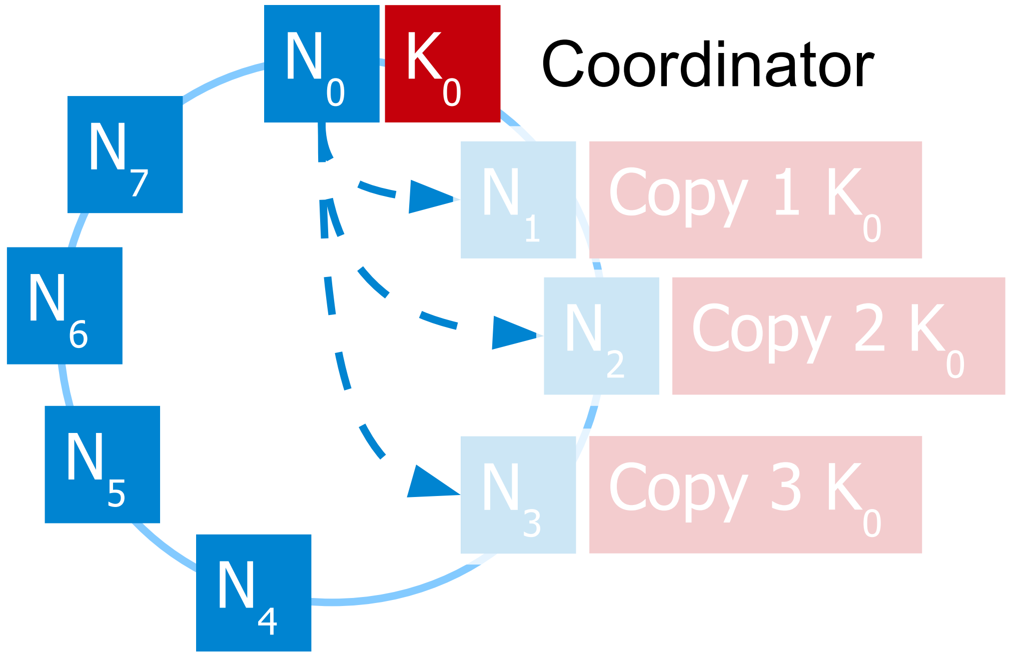
Imagine the skipped node N_1 becomes responsive again. A user queries her shopping cart. The query may happen to be serviced by N_1 . The node lacks a copy of K_0, which may be an item in the shoping cart. It seems the system lost the item and the user adds it again. Amazon accepts this risk.

A continious background self-repair process shuffles keys until they are hosted on the right nodes to reduce the risk.

Eventual Consistency

Replication of a put() is asynchronous

- Eventual Consistency



The speaker says...

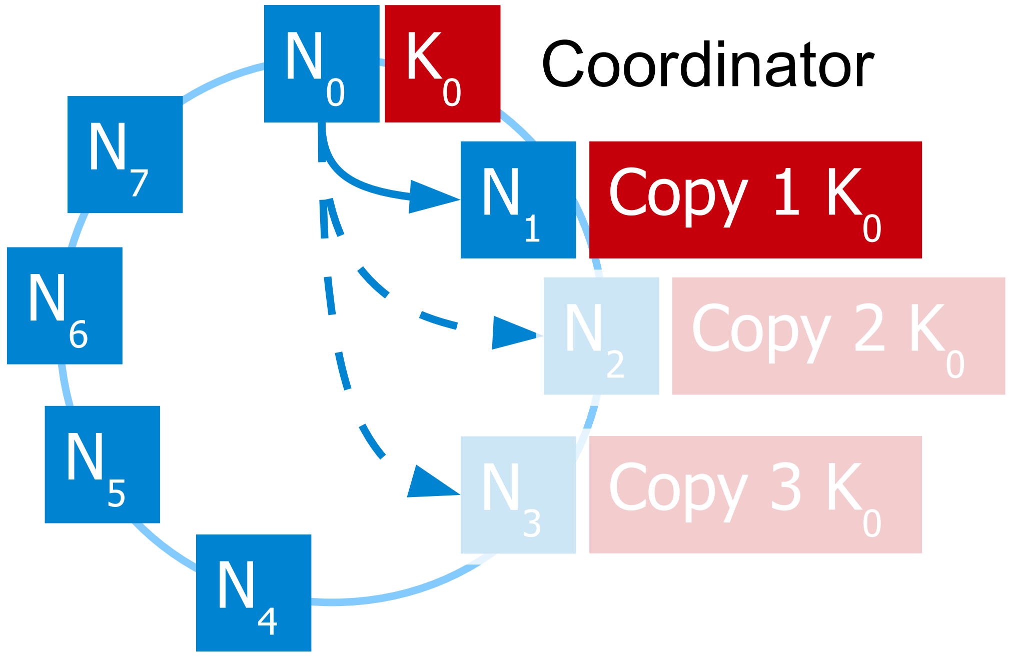
Similar to Chord, Dynamo can store every key on multiple nodes for fault tolerance. The number of copies can be configured.

In order to reply as fast as possible to a put() operation the so-called coordinator uses asynchronous replication. Eventually, after some time, all replicas hold the latest version of the key. Like with MySQL Replication it may happen that a client either does not find a copy of the key at a node that is supposed to hold a copy, or the client sees an old stale version of the key.

Write quorum

Coordinator can wait for W nodes to acknowledge put()

- W is configurable
- Motivation: fault tolerance



The speaker says…

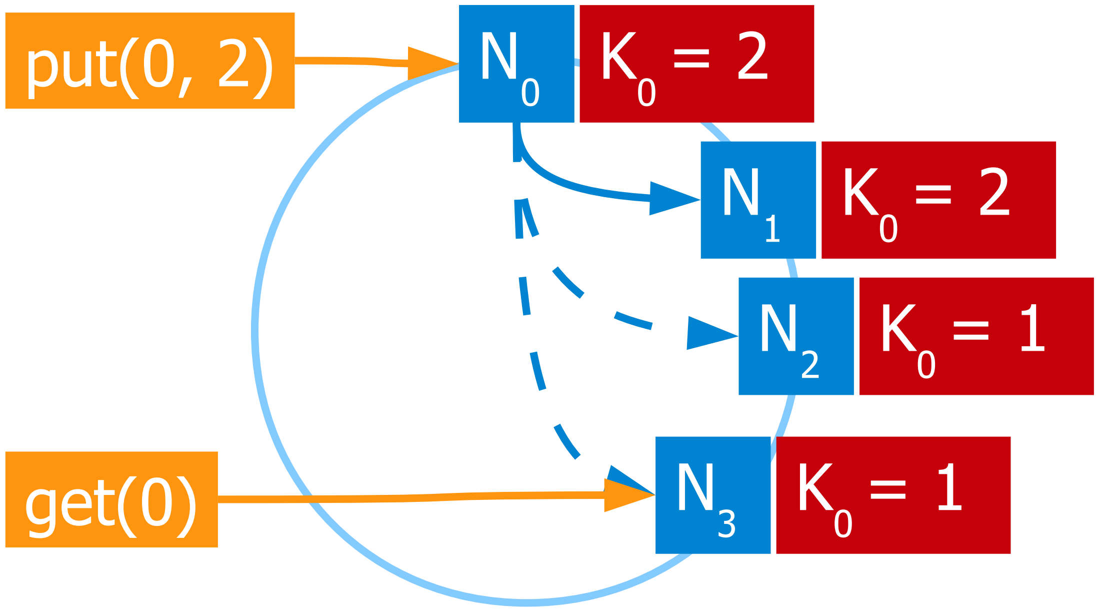
For fault tolerance, the coordinator can wait for a quorum of W nodes to acknowledge a put() before it considers it successful.

Obviously, the lower W is, the faster the write operation will be.

Quorum flexibility

No intersection of read and write quorum required

- N = 4, W = 2, R = 1



The speaker says…

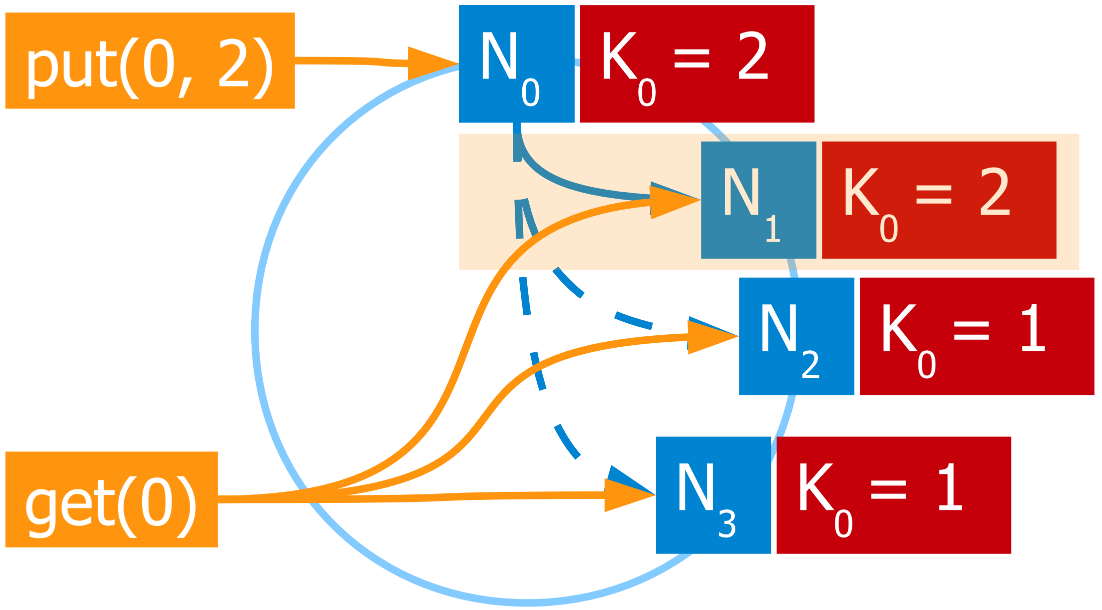
Assume Dynamo is configured to hold N = 4 copies of every key. Every write is replicated asynchronously to four copies. The write quorum is set to W = 2: two nodes must acknowledge a put before it is considered successful. For super fast reads, a read quorum is set to R = 1. Any node holding the desired value can immediately reply to a client.

Let there be a put() and shortly thereafter a get(). The get() happens to be run against a node that has not yet replicated the latest changes. Obviously, there is no way to know for the client whether it works on stale data. The situation is not any better with R = 2, as we may read from N_2, N_3.

Which is current?

Intersection of read and write quorum: R + W > N

- N = 4, W = 2, R = 3



The speaker says...

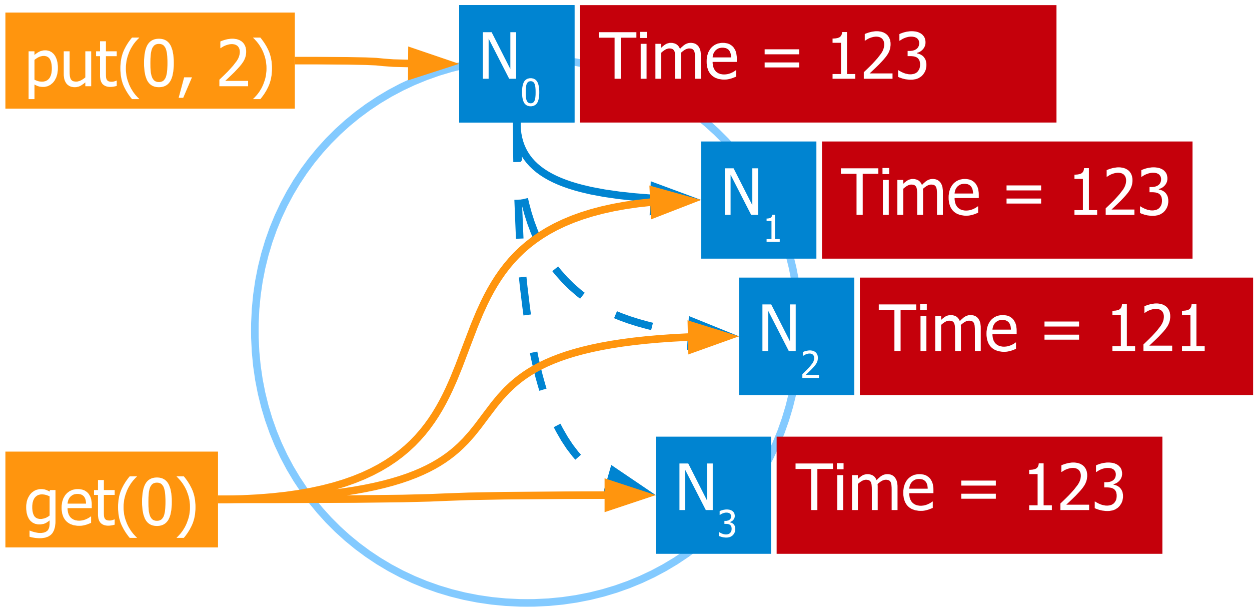
A quorum for which R + W > N is true ensures that the read and write quorums intersect. Thus, in a classical system, strong consistency can be achieved without having to wait for all replicas to apply the latest changes (ROWA[A]).

In the example, the put() and get() operations intersect at node N_1. But how would the client that has issued the get() know whether the value returned from N_1, N_2 or N_3 is the current one?

Using clock time?

Requires synchronized clocks on all machines

- … yes, we cover Google later on



The speaker says...

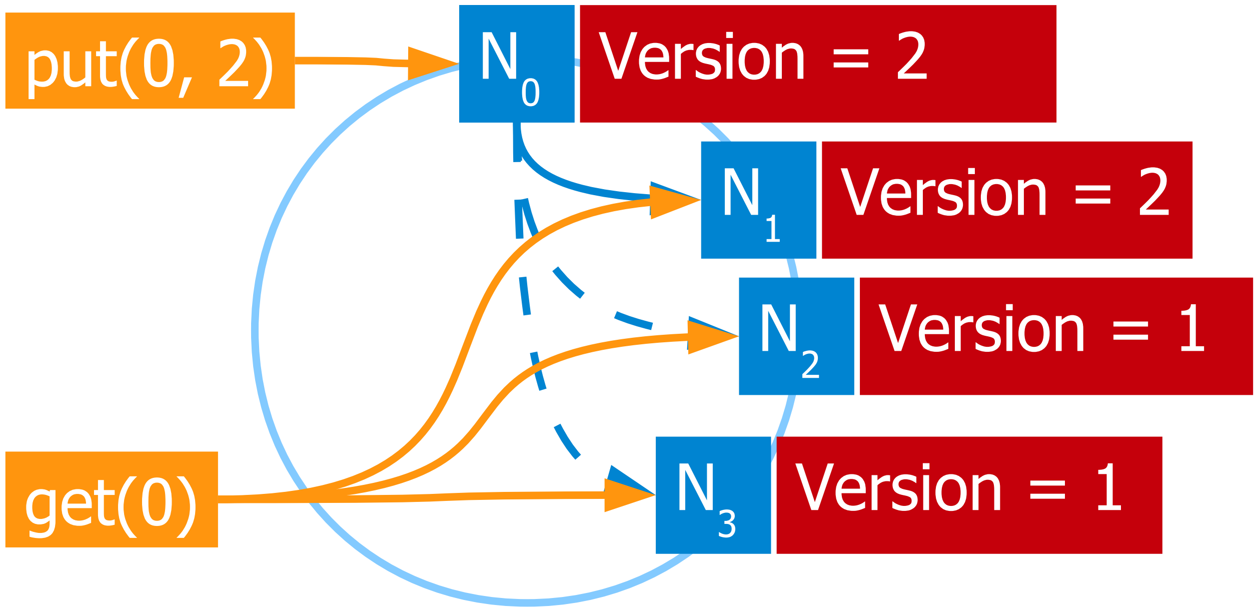
Using a timestamp to determine the most recent version is not much of an option for most. In a typical cloud setting with a mixture of virtual and physical machines spread around the globe, one cannot expect all clocks on all machines to be perfectly synchronized. Thus, comparing timestamps can lead to random results.

If you are like Google, you can try using a combination of an atomic clock and GPS. This is affordable and very precise. The error that remains may be small enough to be of no practical relevance. But, if you are not like Google...

Using version stamp?

Introduces single point of failure

- Either all updates come from the same source
- Or we introduce a single, global sequencer

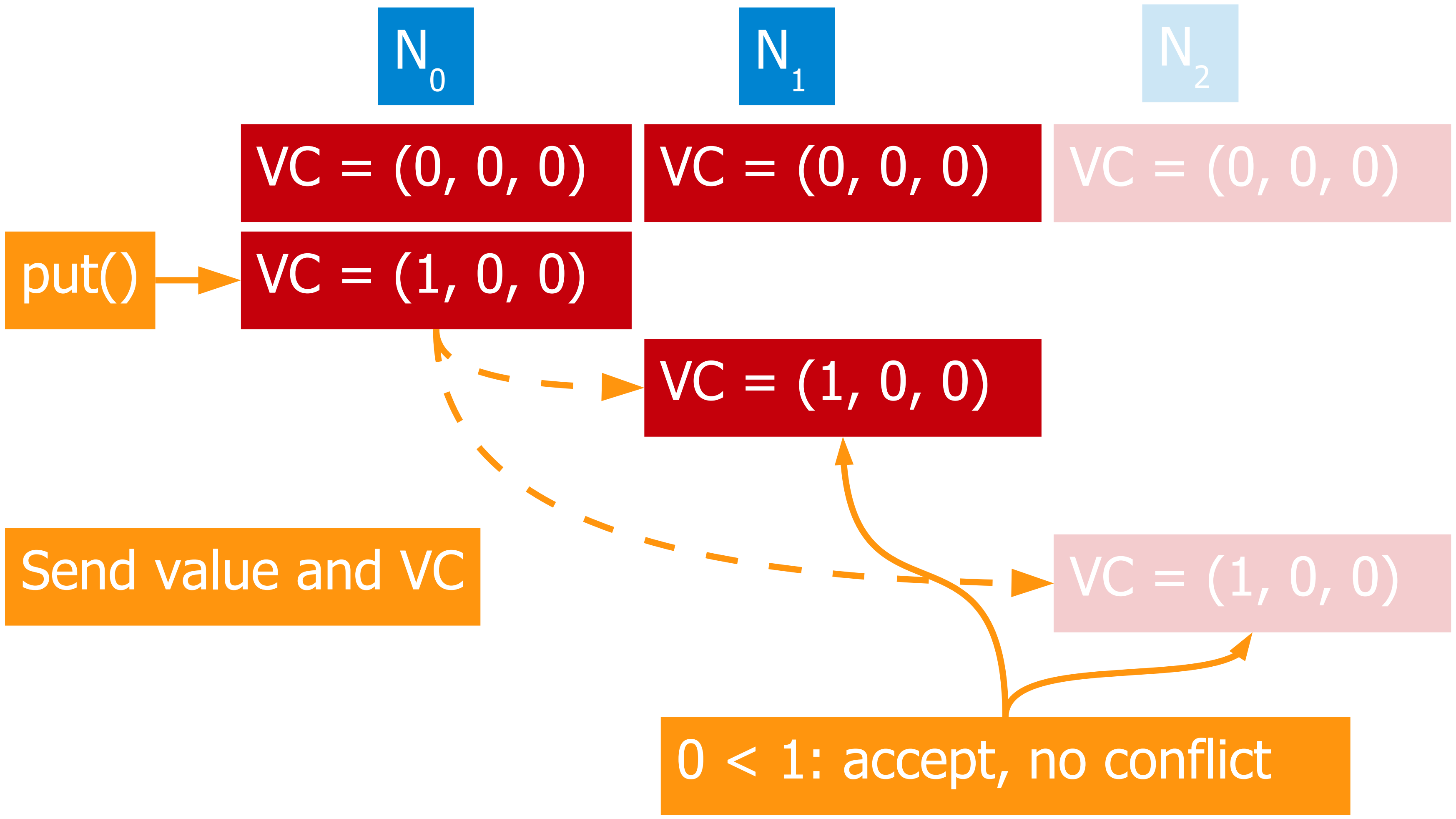


The speaker says…

Version stamps would make it easy to detect the latest version. Each time an update is performed, the version is incremented. Unfortunately, this requires the use of one authoritative source incrementing the version number.

This complicates matters in a peer-to-peer system where updates may origin from many nodes and introduces a single point of failure, which is against the goals of the decentralized system.

Use vector clocks!



The speaker says…

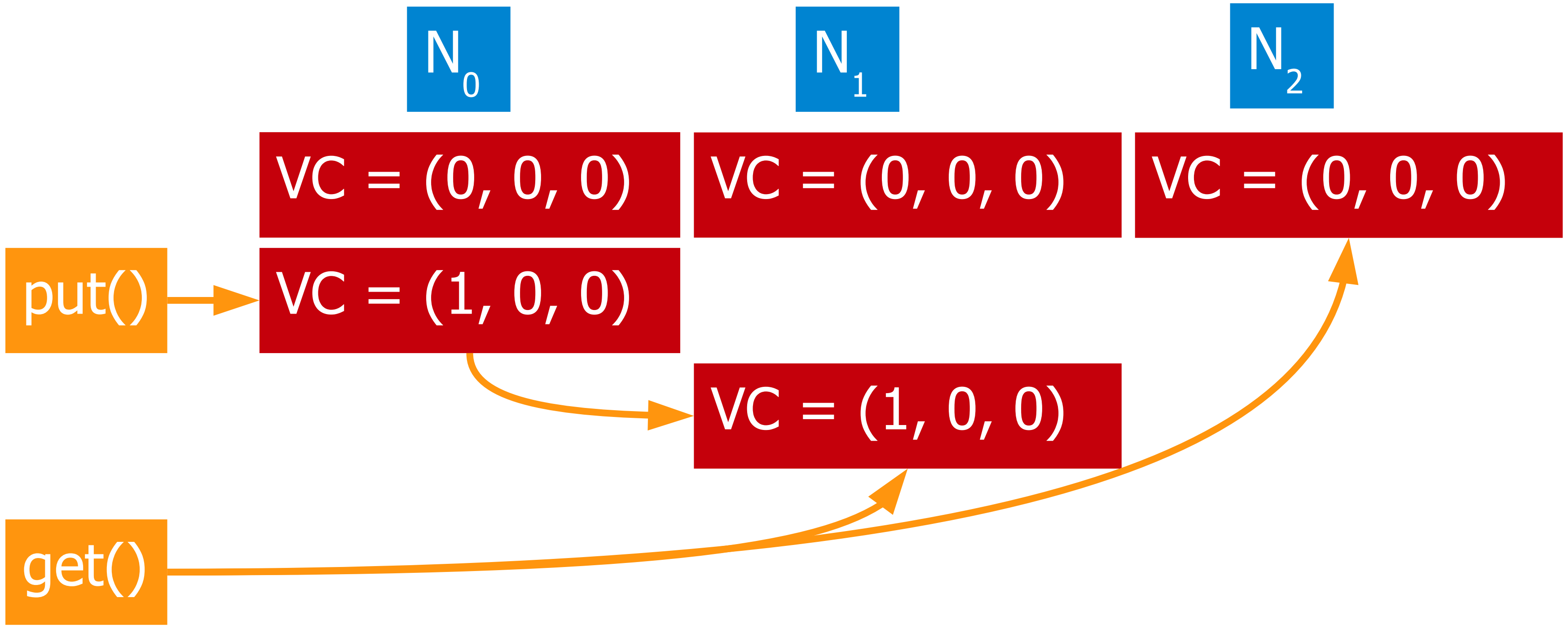
What works is using a vector clock. A vector clock holds the version stamp of every node that has a copy of a data item.

Say, we have a data item replicated at three nodes, N=3. We do an update on node N_0 and the update is replicated to $N_{1.}$ When N_0 does the update, it increments its version stamp in the vector and sends it to the other nodes together with the modified data item. When the other nodes receive the update, they compare their local vector clock with the received one entry by entry. If all version stamps are lower than the local ones, there's no conflict. Then, the new value is accepted and the local vector clock is updated.

Getting the latest version

Vector clocks record casual order

- Client can detect which value/VC came „first" despite async
- Precondition: R + W > N (Dynamo default)

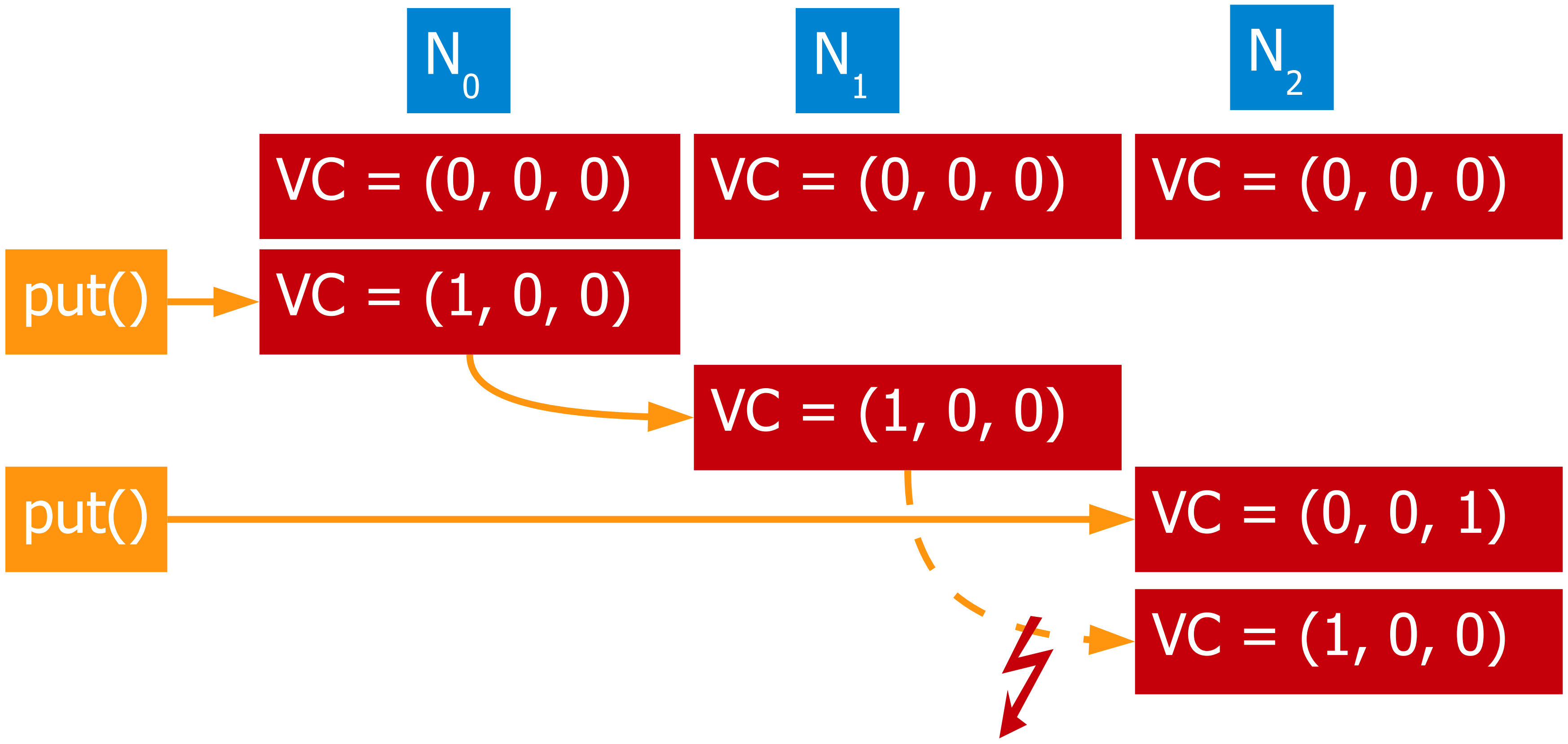


The speaker says…

Vector clocks let us detect casual order: which message, which event, which update has preceeded another one? You can easily grasp this at the quorum example on the slide which is using N = 3, W = 2, R = 2 (Dynamo defaults).

Get() returns two vector clocks VS_{N1} = (1, 0, 0) and VC_{N2} = (0, 0, 0). Then, the client compares the entries one by one. The „older" vector clock is the one which has a lower version stamps at all positions in the vector. Result: although asynchronous messaging was used a client can find the latest version even without the existance of a master/primary or the like because R + W > N (2 + 2 > 3).

Detecting write conflicts



The speaker says…

Write conflics can be detected in the same way.

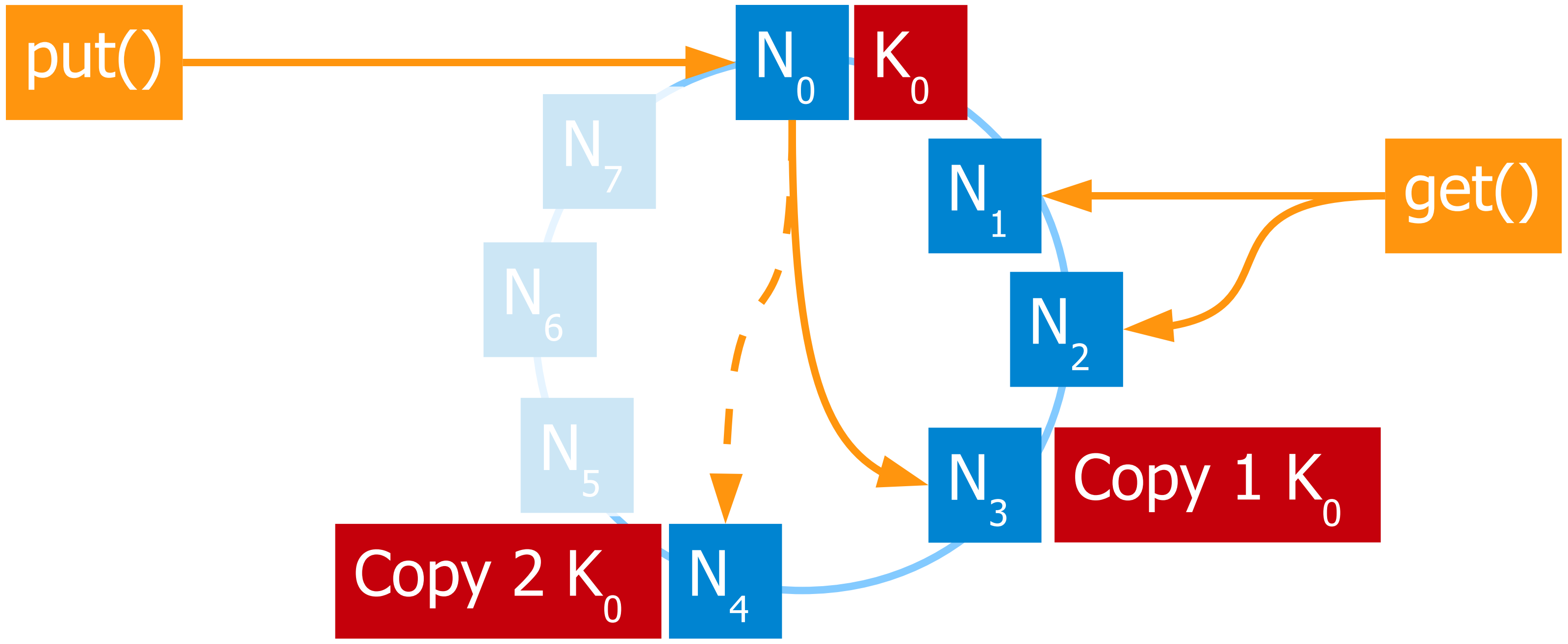
It is the clients responsibility to solve them. Dynamo will keep both versions of the data in conflict and wait for a client to decide.

In other words: your problem! This is the drawback of a system designed for maximum availability.

Sloppy quorum implications

With Dynamo: R + W > N != strong consistency

- Quorum size: N = 3, write quorum: W = 2 (N_0, N_3)
- Read quorum: R = 2 (N_1, N_2)



The speaker says…

The extreme focus on performance and availability bares another challenge. As noted at the beginning, Dynamo may decide to skip unresponsive nodes temporarily and leave it to a background process to move data items to their appropriate slots. N_0 should write to its immediate successors N_1 and N_2 because of N = 3 (three copies per data item requested). Because N_1 and N_2 are unresponsive the data is put on N_3 and N_4. Put() confirms the wite as soon as W = 2 nodes have the data, for example, as soon as N_0 and N_3 have it. Some time later, N_1 and N_2 return. A get() runs against them and fails to see the latest update although $R + W > N$.

Welcome to riak!

Key-Value store, follows Dynamo paper

- Excellent REST-ful API, Google Protobuf interface
- Assorted client libraries: Java, Python, Perl, PHP, .NET, ...
- Stores anything: JSON, XML, text, images, ...
- 2.0: Optional strong consistency

Cluster-wide search

- MapReduce, Riak Search (full-text, since 2.0 Solr)
- Secondary Indexing (Memory, LevelDB)

Pluggable durable storage

- Bitcast, Memory, LevelDB

The speaker says...

Riak is an open source (Apache License 2.0) distributed database that follows the ideas of the Dynamo paper closely. Individual riak nodes are said to achieve tens of thousands requests per second. Considering that the original Dynamo paper already showed results for clusters consisting hundrets of servers, scaling a cluster to this size or beyond should be possible.

Riak is backed by Basho Technologies (basho), a private company. Guess who's a member of the board? Correct: Dr. Eric Brewer. If you think I'm speaking too nice about Riak, please, enjoy http://aphyr.com/posts ...

Another system following Dynamo ideas is BigCouch.

Riak 2.0: CRDT's

Conflict-free Replicated Data Type

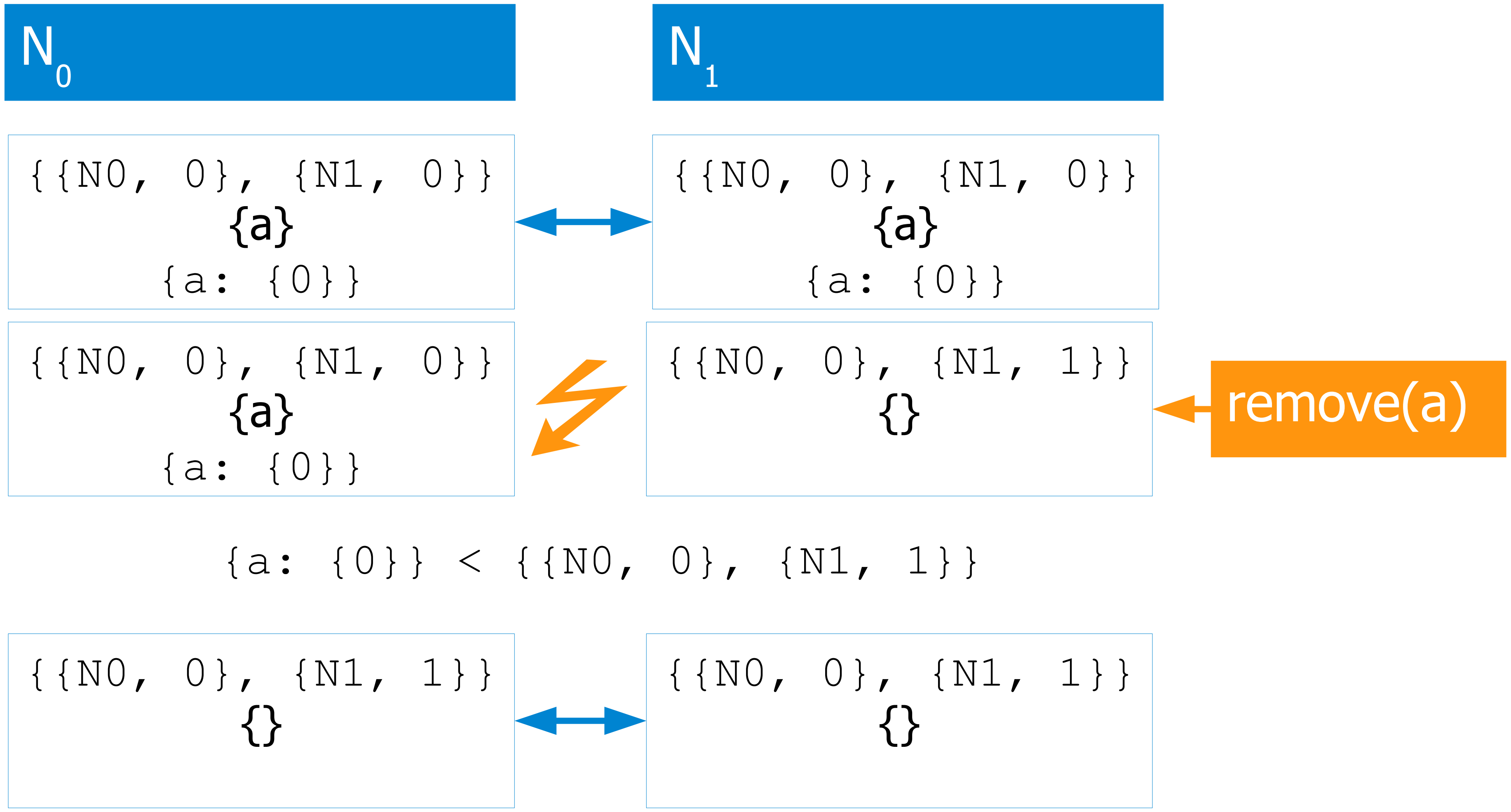
- AKA Convergent RDT, AKA Commutative RDT
- Abstract Data Types: Counter , Set, Map, Registers, Flags
- Strong Eventual Consistency

The speaker says…

Conflict-free Replicated Data Types are abstract data types. Easy. They are special abstract types in the context of eventual consistency: eventually, they converge to a consistent state even in the presence of a partition! In other words: you can perform any operation allowed on the ADT, at any time, without ever having to think about conflicts. The node/replica takes care of resolving the conflict (eventually).

The trick behind: operations either commute, or there is an implicit rule which conflicting operation wins. For example, Riak chooses an „add wins" strategy if in doubt.

Riak 2.0: CRDT's



The speaker says...

Imagine you have a set of {a} on two nodes/replicas. Partitioning happens *and* the system is aware of the partitioning. After the partitioning Riak finds two distinct sets: {a}, {}. How to merge: what has caused this? Either the sets have been empty before the partitioning, or a was remove from the set during the partitioning.

Vector clocks and casuality will tell! Riak keeps a vector clock for the set (that dominates) and a vector clock for each member in the set (see documentation). This enables Riak to find out whether a was removed or not part of the set before partitioning. CAP? Solved – sort of...: eventually strong consistent.

CAP: Dynamo & frieds

In the context of CAP

- Consistency – poor and eventual: developers to solve the mess
- Consistency – poor but: high hopes on CRDT's
- Availability – high: quorum, decentralized, shared nothing
- Partition tolerance – extreme: continues running, always

The speaker says…

We have looked at Dynamo because it is an extreme example of an AP system in the context of CAP.

It is the ultimate data store for a web shop that assumes unlimited stock and operates on very unreliable hardware including very unreliable network connections ;-). The system is decentralized, all data is stored multiple times and no time is spent upfront on ensuring consistency to optimize for fast responses even in the presence of network paritions. It scales virtually indefinetly, at least for key value accesses. The downside is that you have to fix the inconsistency.

Next up is Bigtable – the C(A)P system.

Summary: Dynamo & frieds

Data model

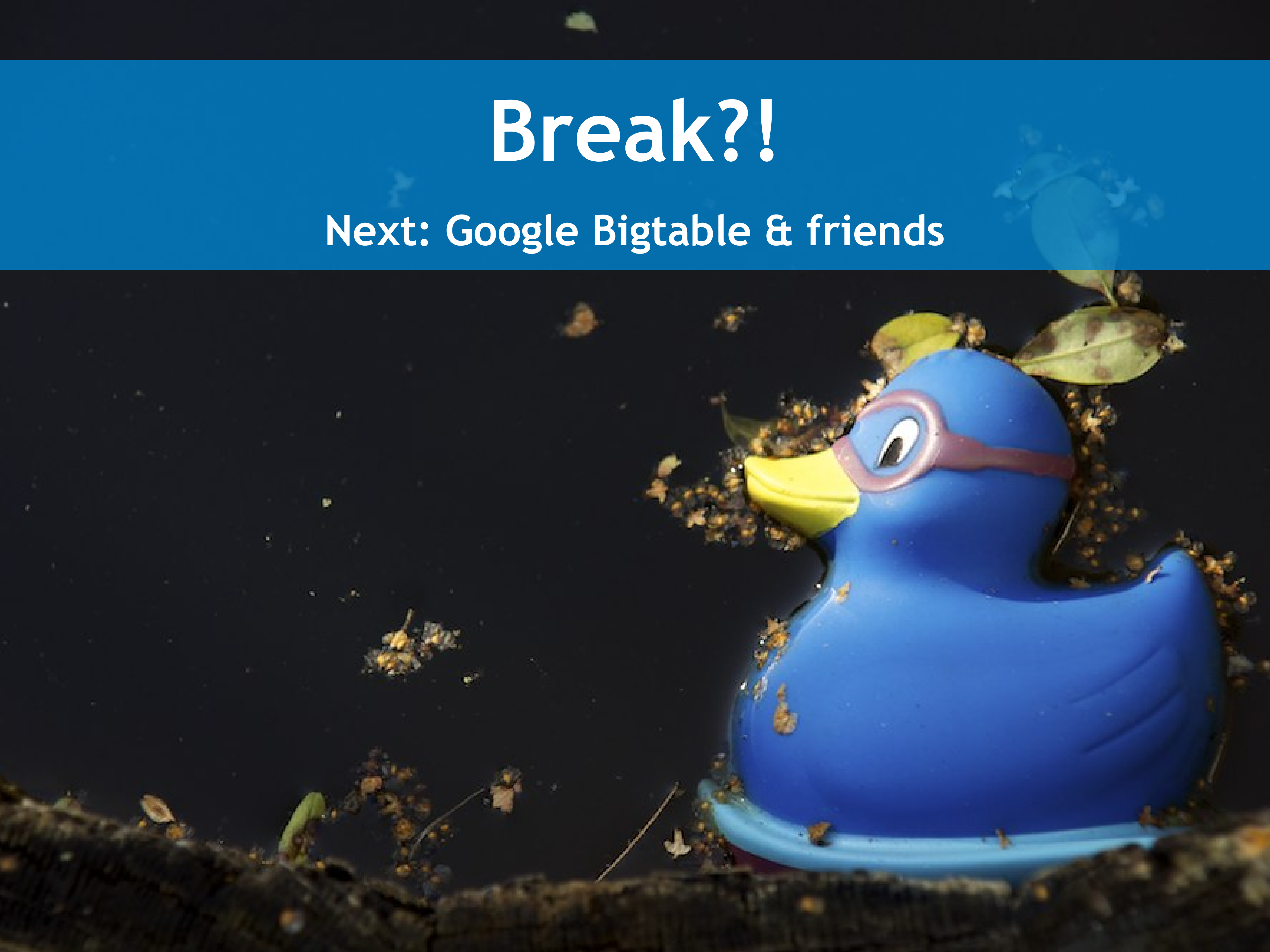
- Key Value: (string) → (binary)
- Few abstract data types, otherwise uninterpreted BLOB
- Weak links from one key to another
- Search limited to key lookup and basic batch processing (MR)
- Sharding, vertical partitioning,

When?

- Simplistic schema is enough
- Simplistic ad-hoc querying is enough
- Elasticity and „always on" are a must

When not?

- Transactions! Range scan!

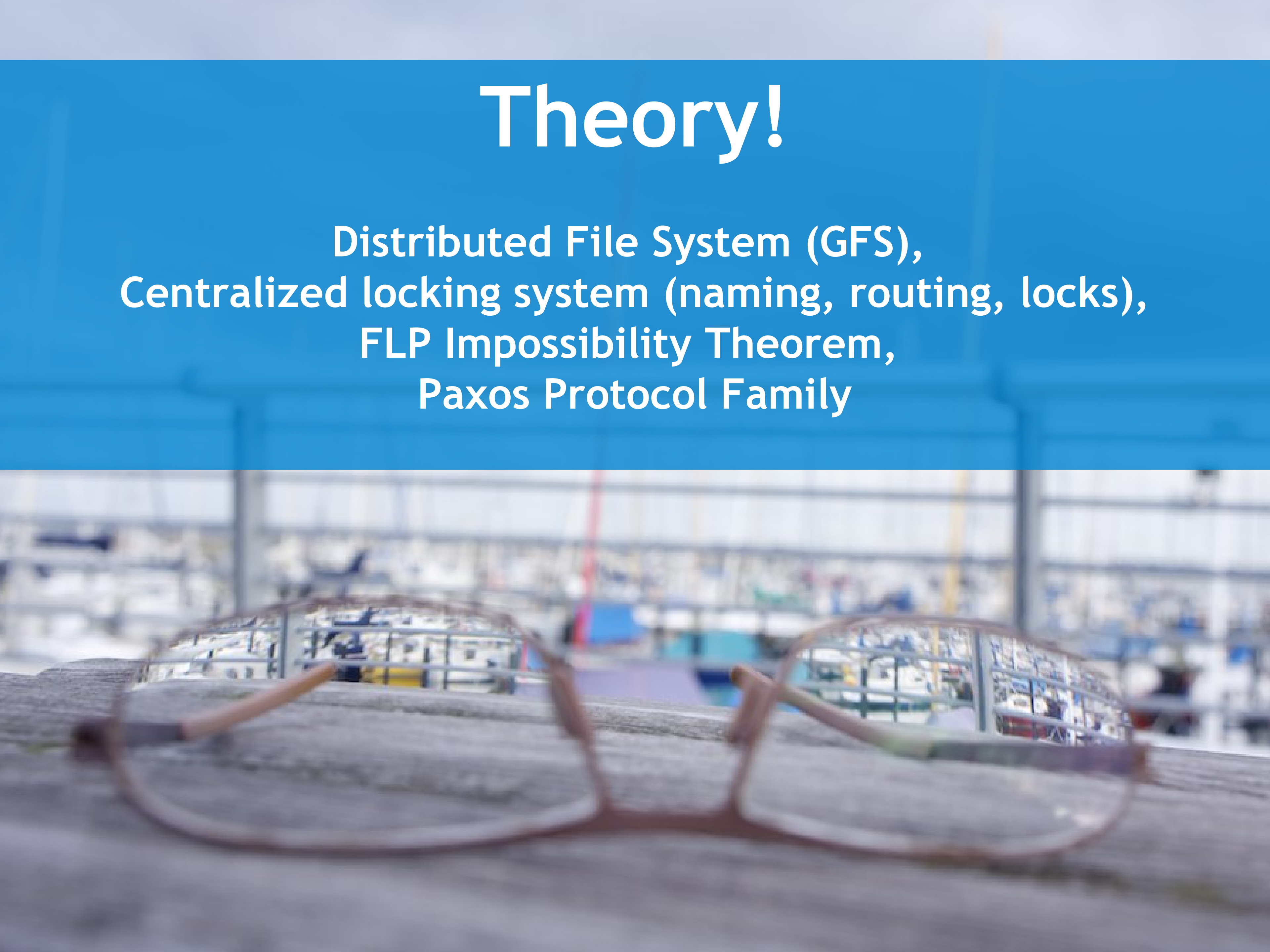


Break?!

Next: Google Bigtable & friends

The speaker says…

Anyone in need for a break?



Theory!

Distributed File System (GFS),
Centralized locking system (naming, routing, locks),
FLP Impossibility Theorem,
Paxos Protocol Family

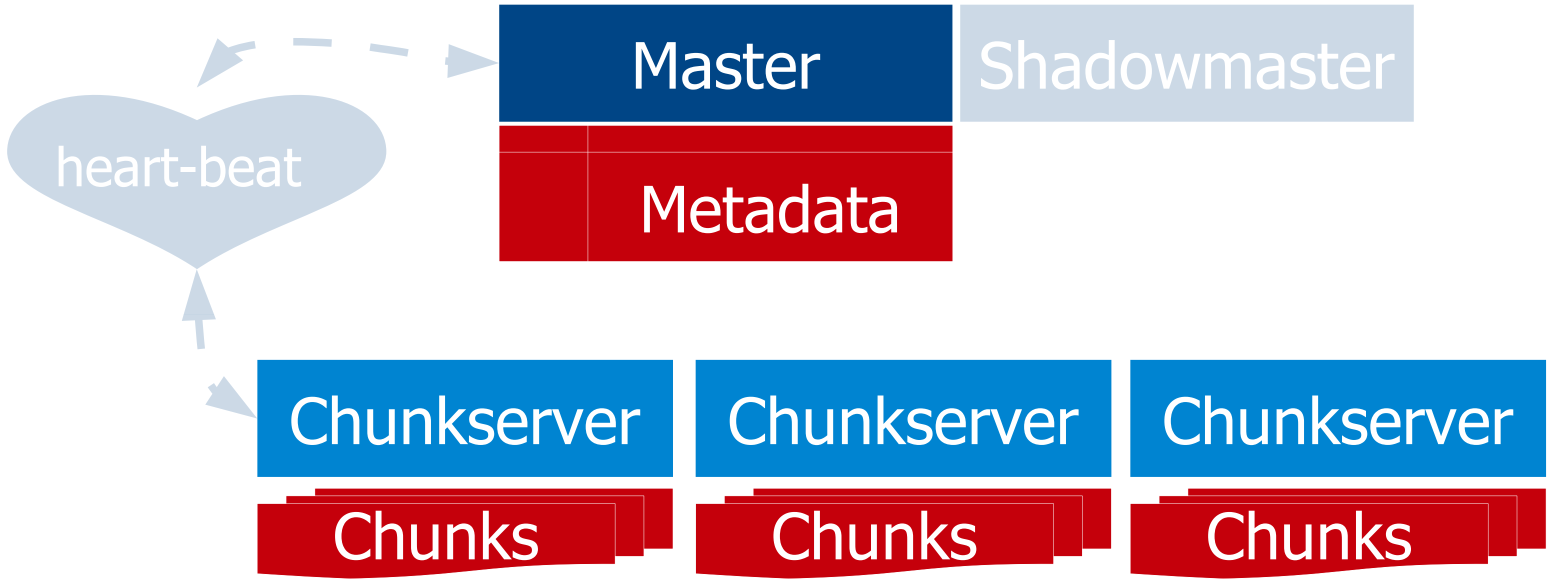
The speaker says…

Before we can understand the design of Google Bigtable we have to master a huge amount of building blocks for this distributed system. Some of the stuff is pretty theoretical in nature such as Paxos. Usually, you will only read that Paxos are used but never be told what they actually are. Paxos are all over if you open your eyes...

Google File System

Distributed File System

- Designed for big, if not huge files, files split in chunks
- Concurrent read and atomic append, rarely delete
- High latency, high throughput
- Replication factor 3, stateless master



The speaker says…

The Google File System is a building block for BigTable, which is why we touch it. GFS is built atop of commodity hardware and Linux.

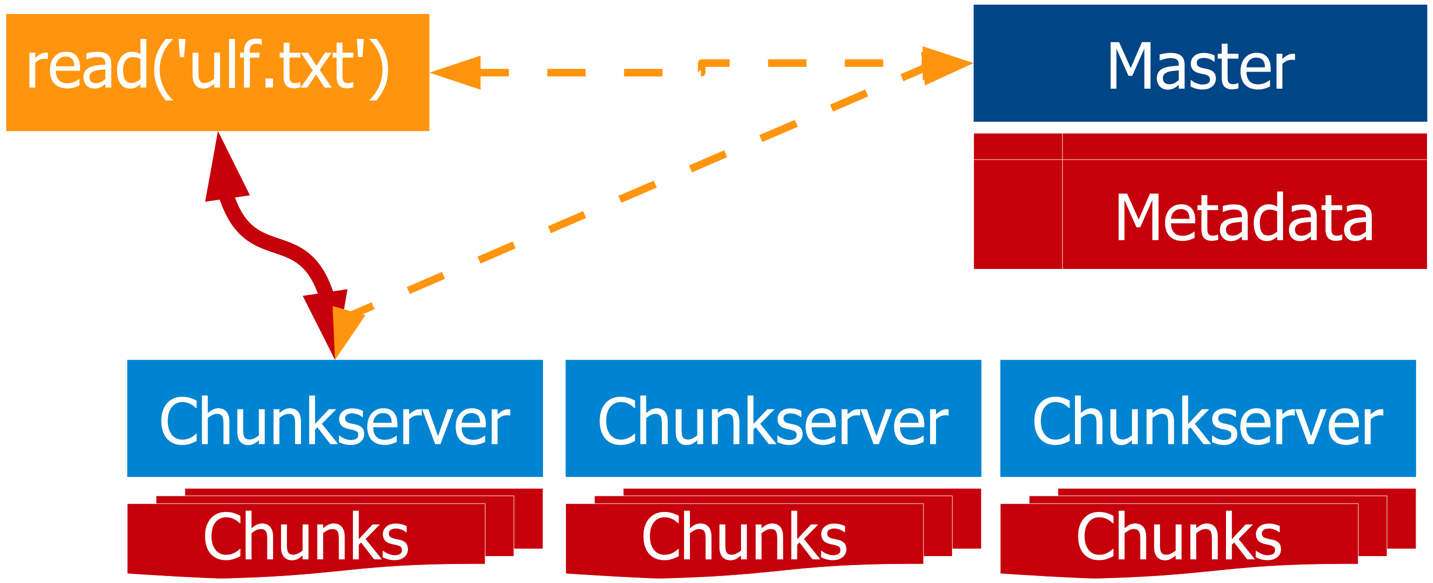
Files stored in GFS are managed in chunks. Chunks are stored on chunkservers. By default each chunk has a replication factor of three.

The master does not store chunks but keeps metadata information where to find the chunks of a file. Clients connect to the master to learn where to find files. Because metadata information is relatively small, it can be kept in main memory and the master can handle many clients. The master is using heart-beats to monitor chunkservers and learn about chunk distribution.

GFS client interaction

Basic read protocol

- Ask master for lease and where to find file
- Cache info, consistently
- Read from chunkserver



The speaker says…

To read a file or portions of a file, a client asks the master where to find the desired data. The master replies with a lease and the information where the data can be found. The client will cache this information.

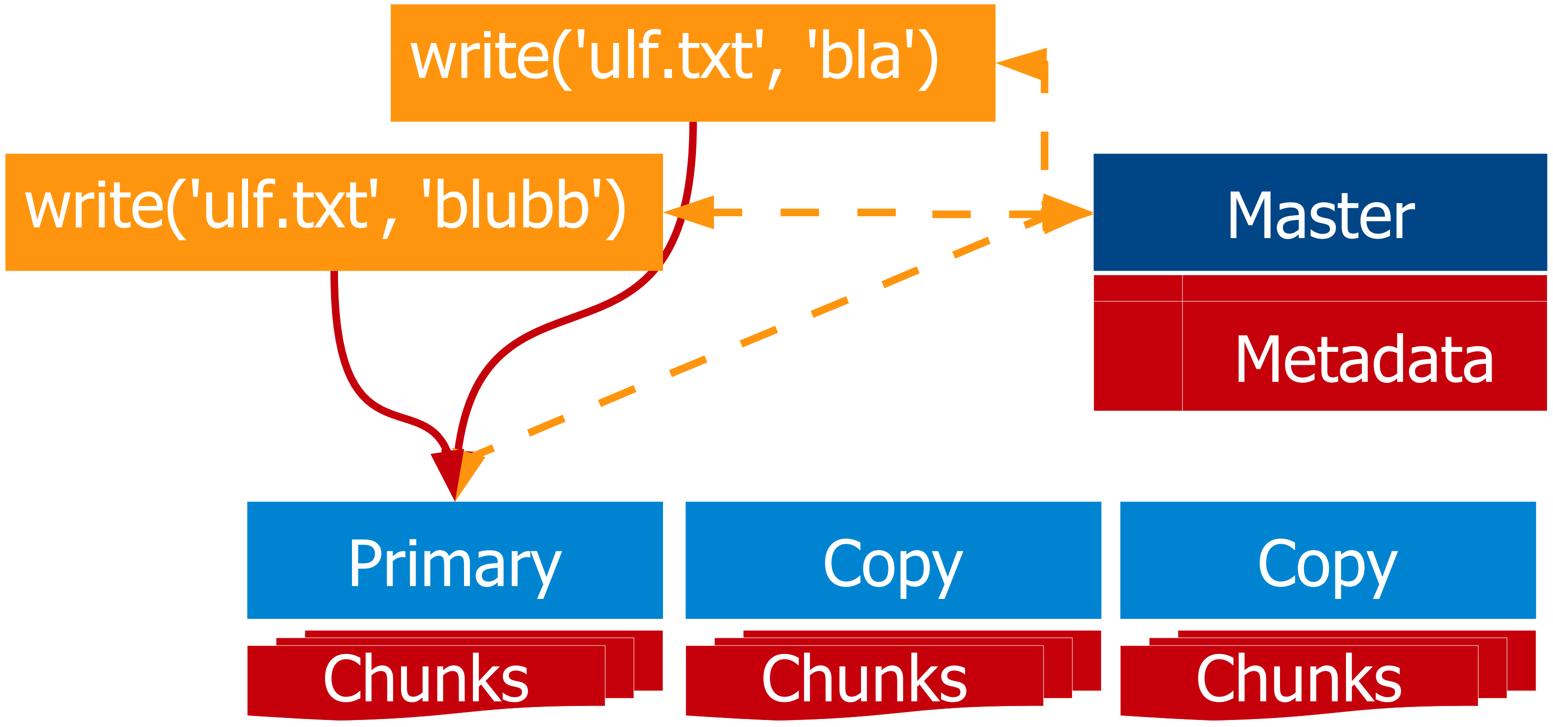
Then, the client contacts the corresponding chunkservers providing them the lease. If the lease is valid, the chunkservers will send the requested file to the client.

The client may read consistent or inconsistent data!

GFS write

Weak consistency model

- No distributed lock manager
- Writes may interleave, 'blblubbla' may be the outcome!



The speaker says…

When GFS performs writes, the master first picks a primary for a chunk. The primary ensures that all copies receive update operations in the same order. This is a common way to simplify replication: recall the Dynamo casual ordering issues?

GFS has a weak consistency model: concurrent writes to one file are not isolated from each other. There is no distributed lock manager which could stop clients from writing to the same file at the same time in a way that the writes interfere. Thus, inconsistencies may arise. GFS favours concurrency and performance over consistency.

GFS implications

Applications must adapt

- Atomic: create, rename
- Rename trick: write to new, rename after write
- Write workarounds: checksums, checkpoints, ...
- But 'atomic at least once append'

Applications did adapt

The speaker says…

GFS offers one stronger operation which is the equivalent to O_APPEND. Appending data to a chunk at an arbitrary position (as opposed to a write at a specific offset) is atomic at least once. The protocol used for the operation does not require a global lock manager. However, the data appended may end up at different byte offsets on different replicas.

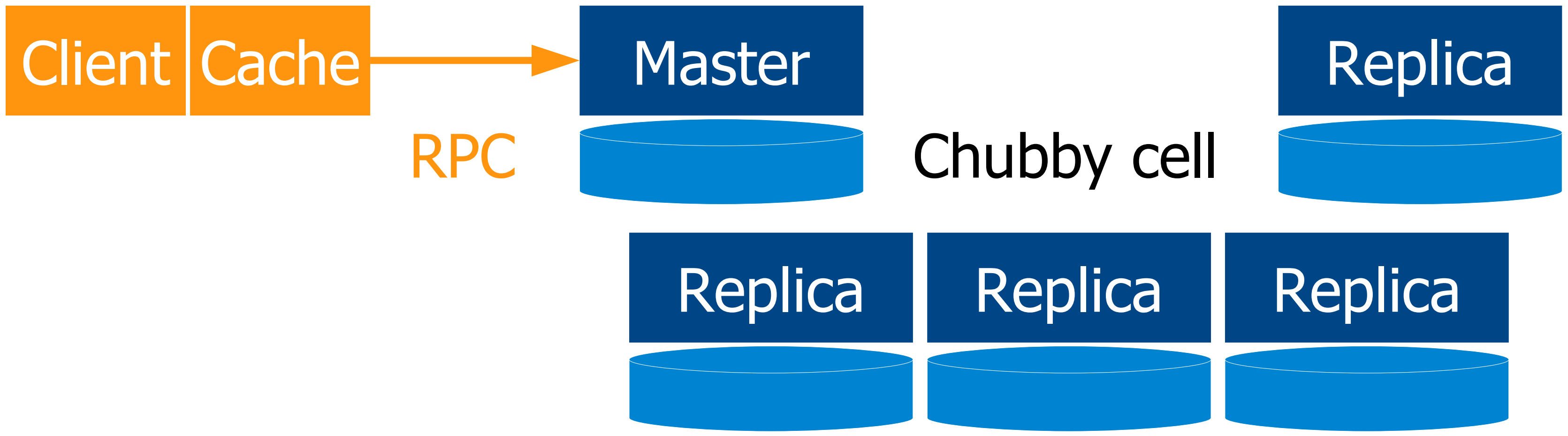
Whatever, append was one of the most important operations and it worked.

But how to build a big table, a database like structure on top of this?

Google Chubby

Distributed locking service

- Paxos for distributed consensus
- Coarse-grained advisory shared and exclusive locks
- Stores small files, e.g meta-data to locate master/primary
- Some publish/subscribe messaging



The speaker says…

Chubby is a distributed locking service. It is designed to give coarse-grained advisory shared and exclusive locks to clients. Coarse means, the lock is supposed to be held for several minutes at least. Advisory means, the lock does not block access to an actual resource. Hostile clients could ignore the lock and access the resource directly.

For scalability, Google deploys many Chubby cells. There can be one or more cells per data center. Each cell typically consists of five replicas. Replica failures turned out to be rare and resolved within seconds. Consistent caches make it possible for a cell to handle tens of thousands clients in a timely manner.

Centralized lock service

Developers have no hard time...

- Looks ike local mutual exclusion but is RPC!
- Few messages, few API calls
- No need for applications to link library for distributed consens
- No large quorum, progress in the presence of failures

```
/* pseudo code */
LocalOperations();
Acquire();
/* critical section */
UpdateSharedFile();
AccessSharedResource();
Release();
```

The speaker says...

A central locking service is a very simple yet elegant way to coordinate concurrent accesses to shared resources. Using a lock for mutual exclusion should look familiar to any developer immediately. Use of RPC ain't complicated either as long as you never forget it is a remote call.

Using a central locking service free's applications from linking a library for distributed consens, which can be challenging if the application is written in many programming languages. Also, if each client participates in distributed consens, quorums get huge quickly.

Not made for transactions

Two-Phase Locking

- Phase 1: acquire all required locks
- Phase 2: release locks

Housekeeping

- Redo and undo logs, deadlock detection

```
/* pseudo code to illustrate 2PL */
Acquire('src');
if (src.balance > amount) {
  src.balance -= amount;
  Acquire('dest');
  dest.balance += amount;
Release('dest', 'src');
```

The speaker says...

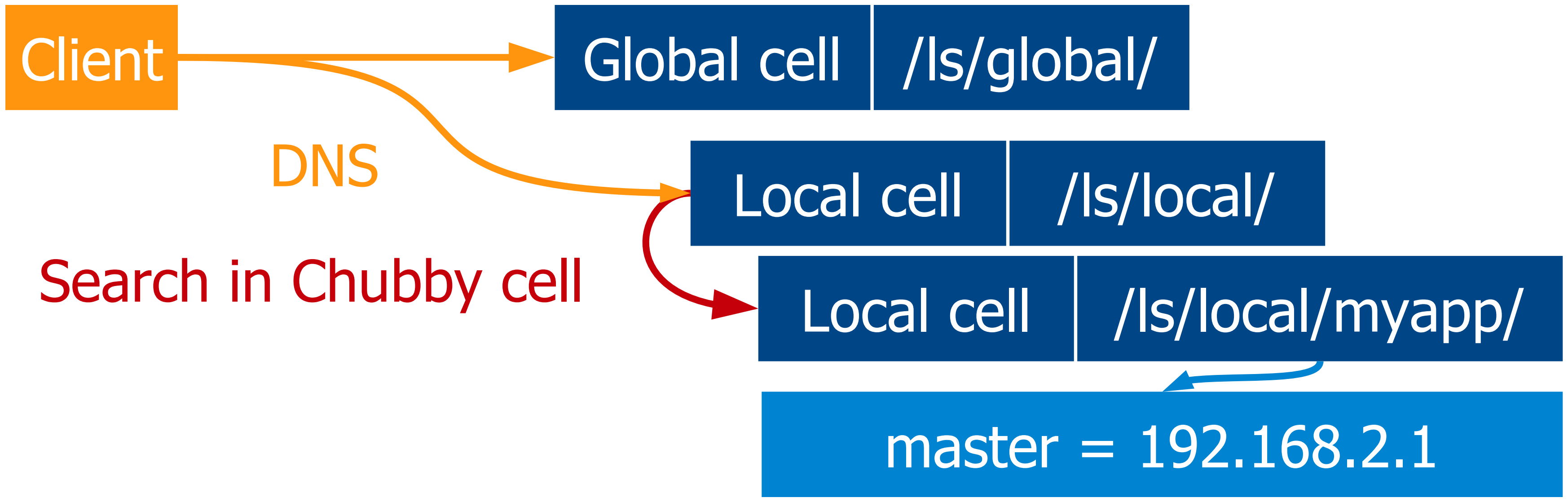
Having a distributed locking service does not imply you can easily implement transaction logic. Not only is Chubby not designed for fine grained locks but also the application would have to take care of many things required.

For example, the application must acquire locks following two-phase locking. Two phase locking says that you have to acquire all locks and then, afterwards, release them. Once you have released a lock you must not acquire a new lock. Here's why, ask yourself what state the other process will observe and base his decisions on: `Acquire('src'); Release('src'); /* other process scheduled to run now */; Acquire('dest'); Release('dest');`

Chubby as a naming service

Chubby can store meta-data of application (e.g. config)

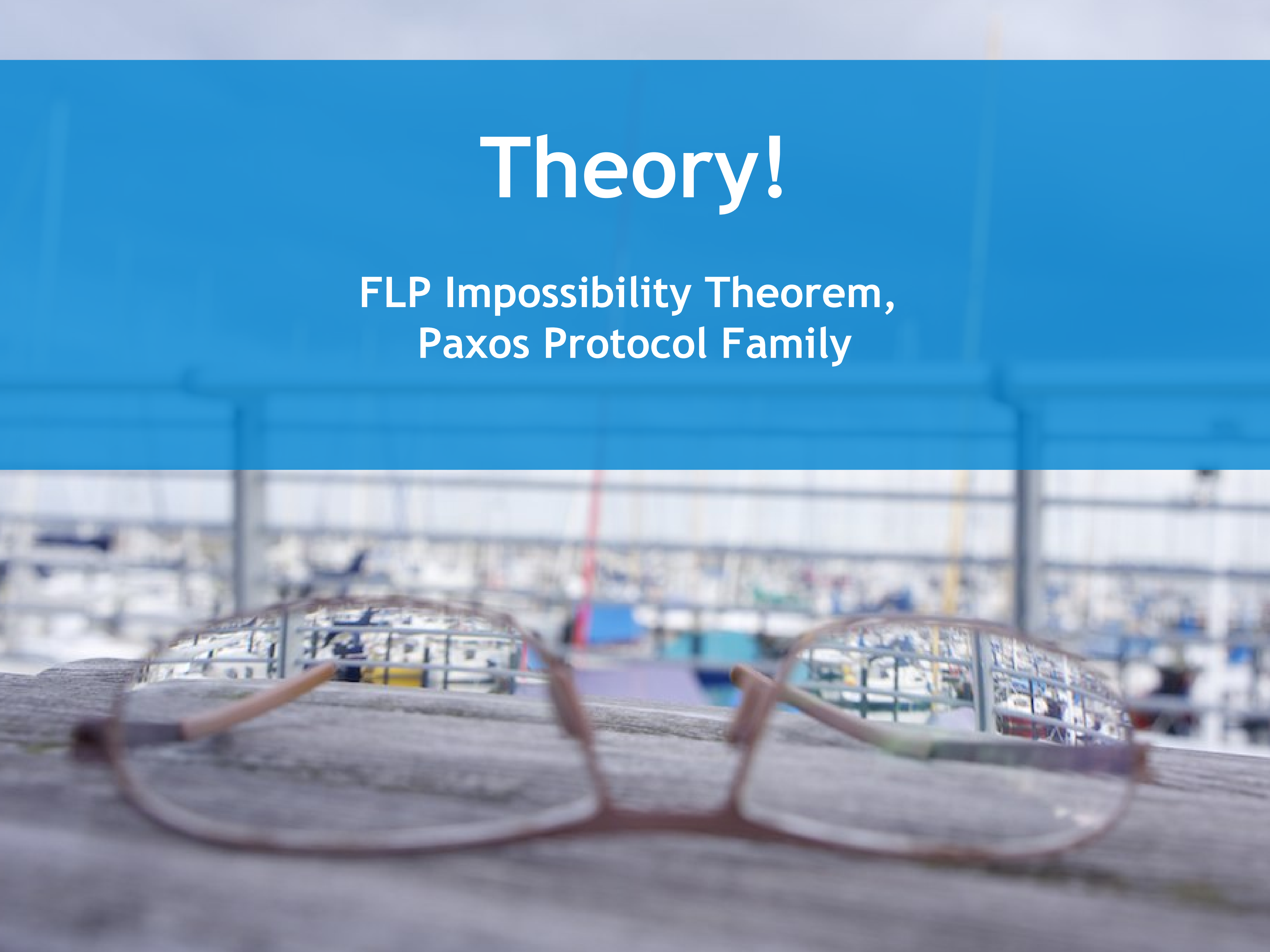
- Meta-data is stored under a path /ls/local/myapp/...
- Global cell spanning data centers is found via DNS
- Local cell within data center is found via DNS



The speaker says…

A distributed application needs to know where to find an entity, for example, a master server of a certain service. An application using Chubby can look up a world wide global Chubby cell and its local data center chubby cell using DNS. The global cell is available under the name /ls/global and the local one under /ls/local/. If Chubby replicas fail DNS entries are updated accordingly. Pretty much like you might move a virtual IP from a failed MySQL Replication master to a newly promoted MySQL Replication master.

Once the local cell is located an application can contact the cell and search for meta-data files stored in the chubby cell.



Theory!

FLP Impossibility Theorem,
Paxos Protocol Family

The Consensus problem

It is impossible for a set of processors in an asynchronous sytem to agree on a binary value, even if only a single process is subject to an unannounced failure.

Impossibility of Distributed Consensus with One Faulty Process, Fischer-Lynch-Paterson, 1985:

„We have shown that a natural and important problem of fault-tolerant cooperative computing cannot be solved in a totally asynchronous model of computation. These results do not show that such problems cannot be “solved” in practice;[...]“

The speaker says...

There's not only CAP that plagues architects of distributed systems but also the famous Fischer-Lynch-Patersion Impossiblity Result from 1985. FLP proved that asynchronous distributed systems cannot achieve consens. A system is asynchronous if message transfer times are not bound and processor speed is not know upfront. Under these circumstances, it is impossible to distinguish between a machine that has crashed and a machine that is slow. The crashed machine would require to run a recovery protocol whereas one should simply wait for the slow machine to reply. Luckily, like with CAP, there's the theory and the practice. One solution to the problem is the Paxos family of protocols.

Paxos Properties

(Useful together with Replicated State Machines)

Safety

- Non-triviality: Only proposed values can be learned
- Uniform Agreement: Two processes cannot decide differently
- Termination: Every correct process eventually decides

Fault tolerance

- If less than half the nodes fail,
  the rest nodes reach agreement eventually

The speaker says...

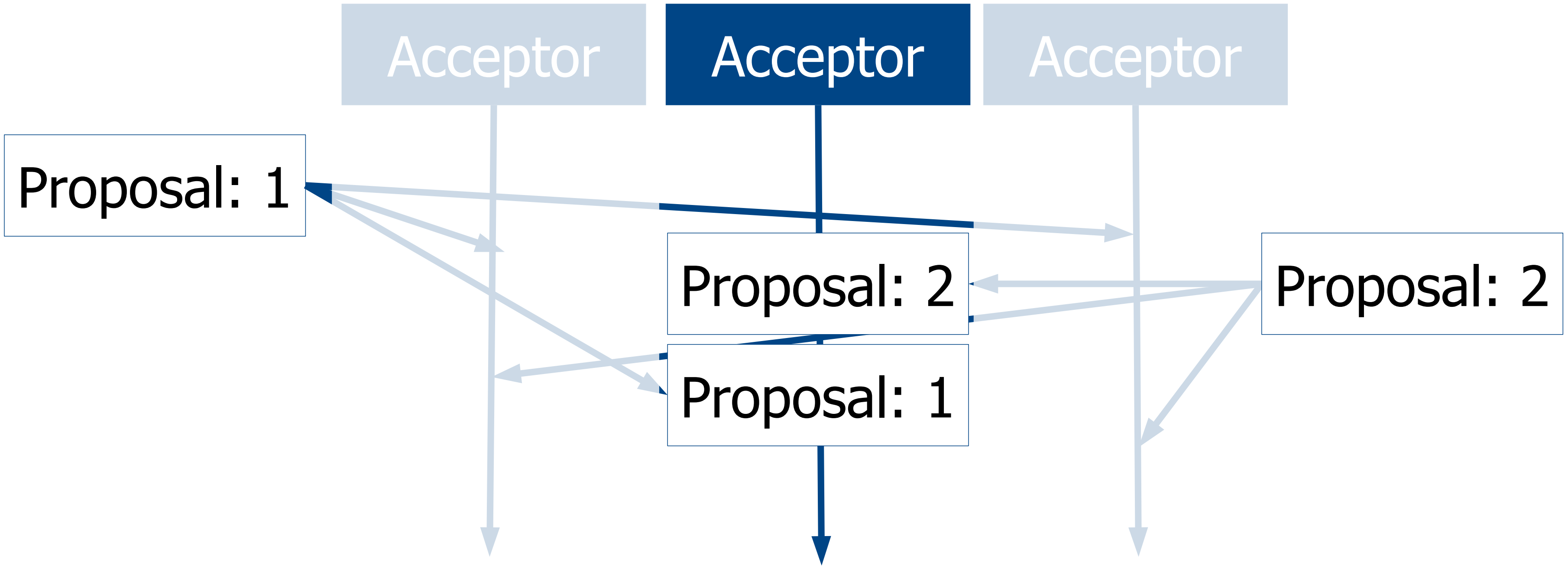
A detailed discussion of the Paxos algorithms family is out of scope. Only the very basics are covered ;-).

A common approach to replication is using a replicated state machine. A state machine has a defined starting point. Each input received is passed through the transition and output function to produce a new state and output. Multiple copies of the same state machine will produce the same outputs given that they receive the same inputs in the same order. But how can be have a distributed set of state machines receive the same inputs in a fault tolerant way? How to cope with asynchronous messages? And, are databases state machines?

Sequence numbers (slots)

Asynchronous messages may arrive in arbitrary order

- Acceptors need a way to detect which proposal came first



The speaker says…

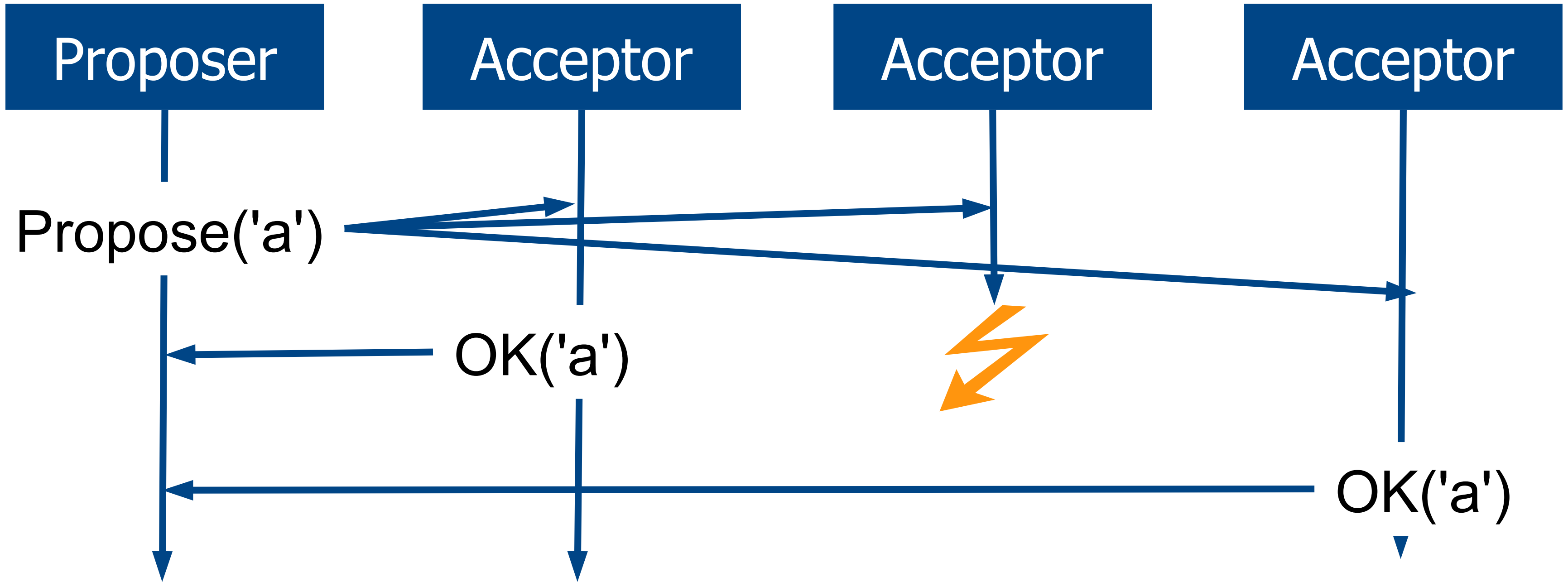
In an asynchronous system message delivery times can vary. Two messages sent one after another are not guaranteed to arrive in the order sent. This is not compatible with the replicated state machine approach. Messages or inputs must be processed in the same order on all state machines to ensure that all state machines go through the same state sequence.

Thus, Paxos require a sequence number to be added to proposals. This way, all acceptors agree which proposal came before and after.

Naive approach

Single proposer and multiple acceptors

- Related: 2-Phase-Commit protocol



The speaker says…

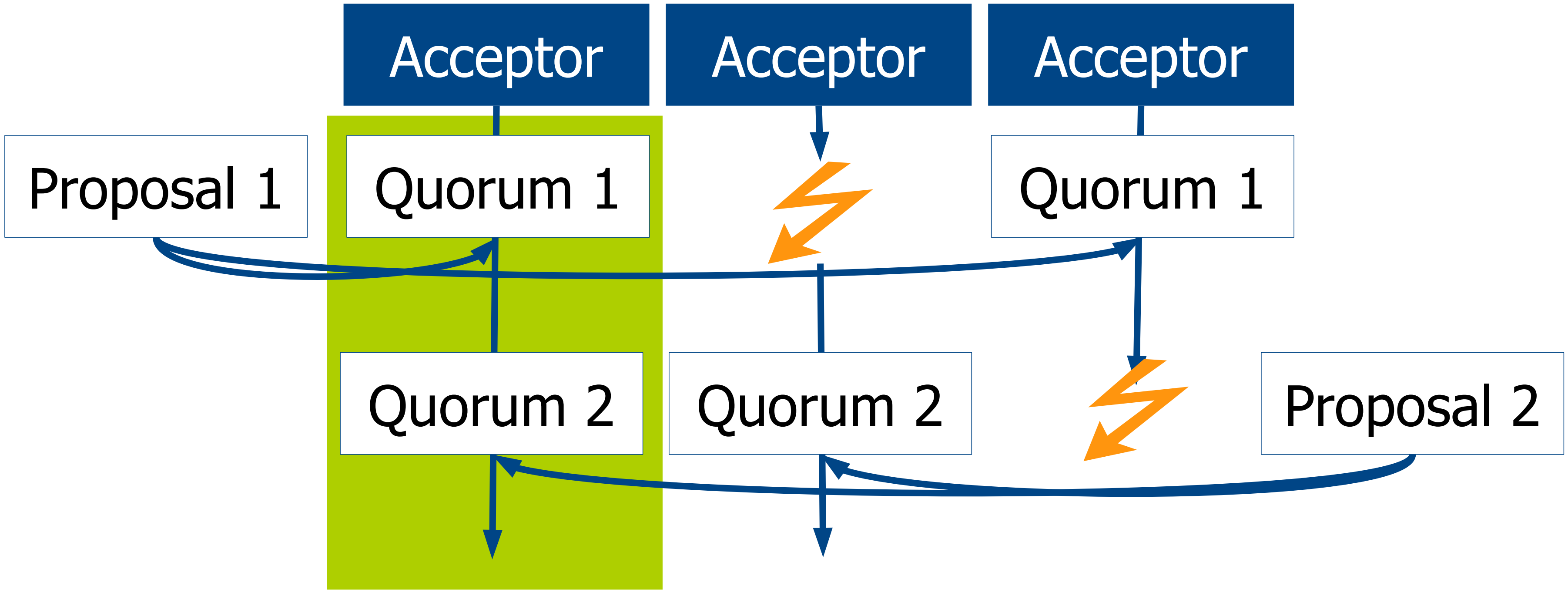
In an naive approach one processor may propose a value and sent its proposal to some acceptors. Then, the acceptors accept or reject the value. They inform the proposer of their decision sending it a message. If the proposer gets an OK from everybody, then the value is accepted and all acceptors can be instructed to commit the value in a second phase.

However, what if an acceptor fails to reply? According to FLP we cannot proceed as we do not know for sure whether the acceptor crashed or is slow. The protocol stalls. Thus, we need to find a minium set of acceptors that still guarantees correctness (consistency).

Majorities

Using (strict) quorum: n/2 + 1 acceptor votes required

- Any two quorums of acceptors will have at least one in common.

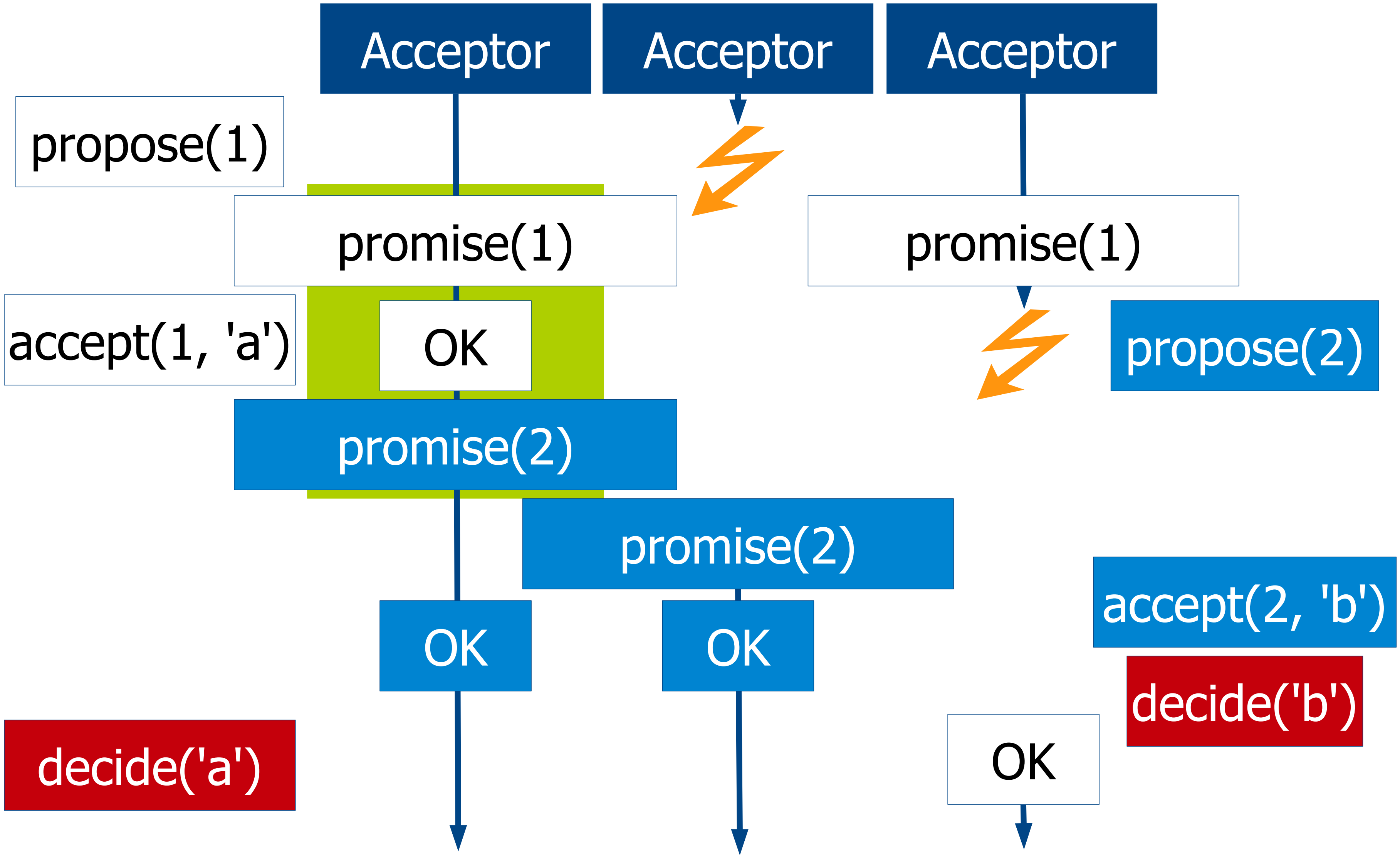


The speaker says…

The failure of an acceptor can be overcome by requiring only a majority of acceptors to agree on a proposal. No matter how many proposals from subsequent quorums there are, any majority of acceptors will have complete information and can ensure that only legitimate proposals are accepted.

Please note, it can happen that there are multiple proposals during the execution of the protocol.

Majority alone is not enough

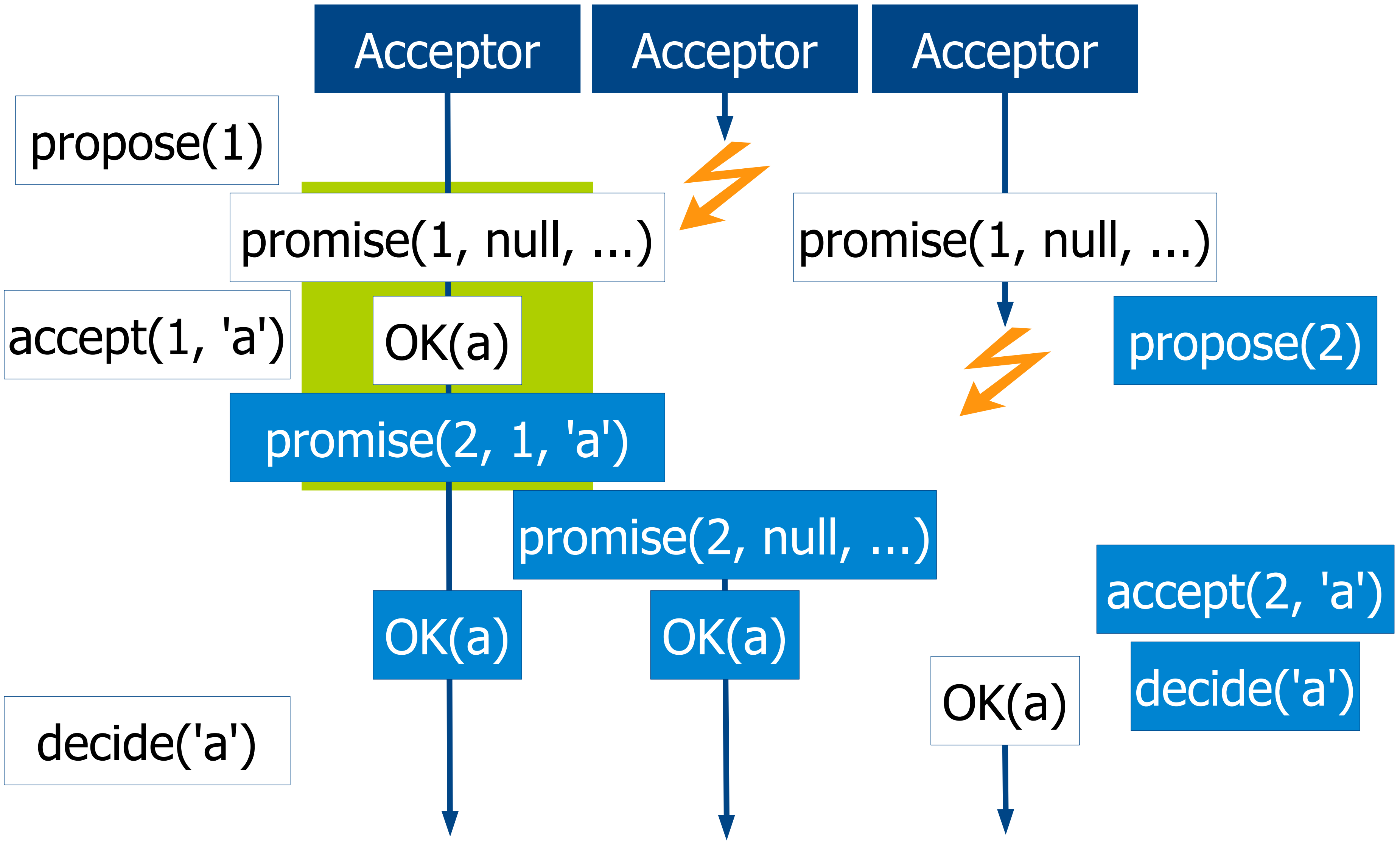


The speaker says…

Majority alone is not enough. Imagine a first proposer reaches a minimum quorum to get promises from the acceptors and sends out an accept('a') message. Then, one acceptor fails. Thus, no decision can be made.

Meanwhile a second proposer achieves a different quorum and gets promises from acceptors because he has provided a higher sequence number. The second proposer then tells the acceptors to accept('b'). They reply OK and the second proposer considers 'b' decided/commited. At this point the first proposer gets the outstanding OK for accept('a') and considers 'a' decided. Obviously, the result is incorrect: two proposers decided on two distinct values.

Legitimate proposals



The speaker says…

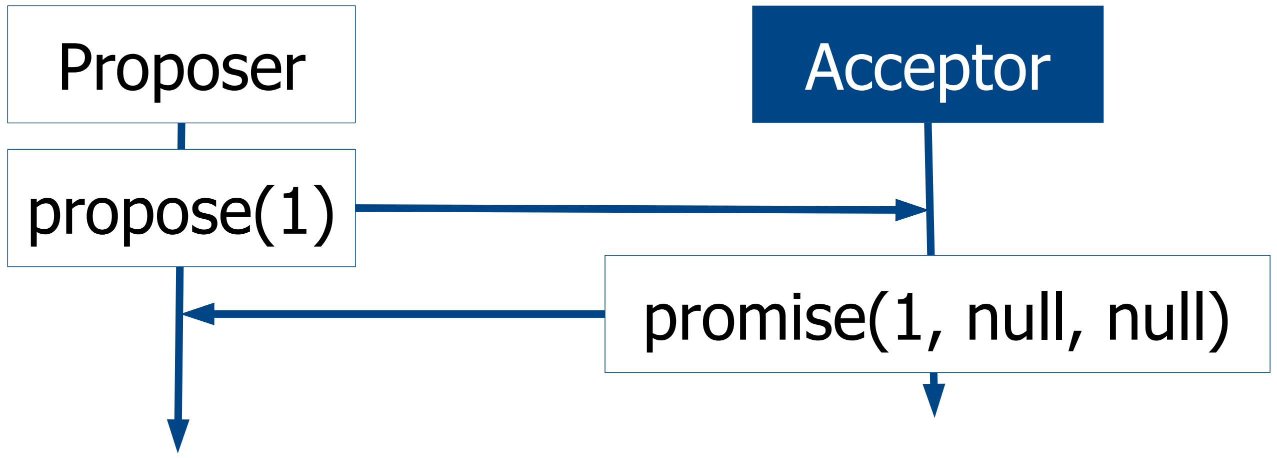
The Paxos algorithm ensures that the two processors decide on the the same value by exploiting the fact that there is an intersection between any two quorums. The acceptor that took part in both quorums can hint the proposer that it must not suggest a new value.

In order words: a proposers suggestion may be rejected until agreement on the last proposal has been reached.

The protocol: phase 1

Propose

- Once process decides to be proposer (leader)
- Proposer suggests sequence number N higher than before
- Acceptor rejects H if less than highest it has seen, else Acceptor promises to accept no proposals with lower sequence number, sets his high= N and replies OK with highest previously *accepted* sequence number and value



The speaker says…

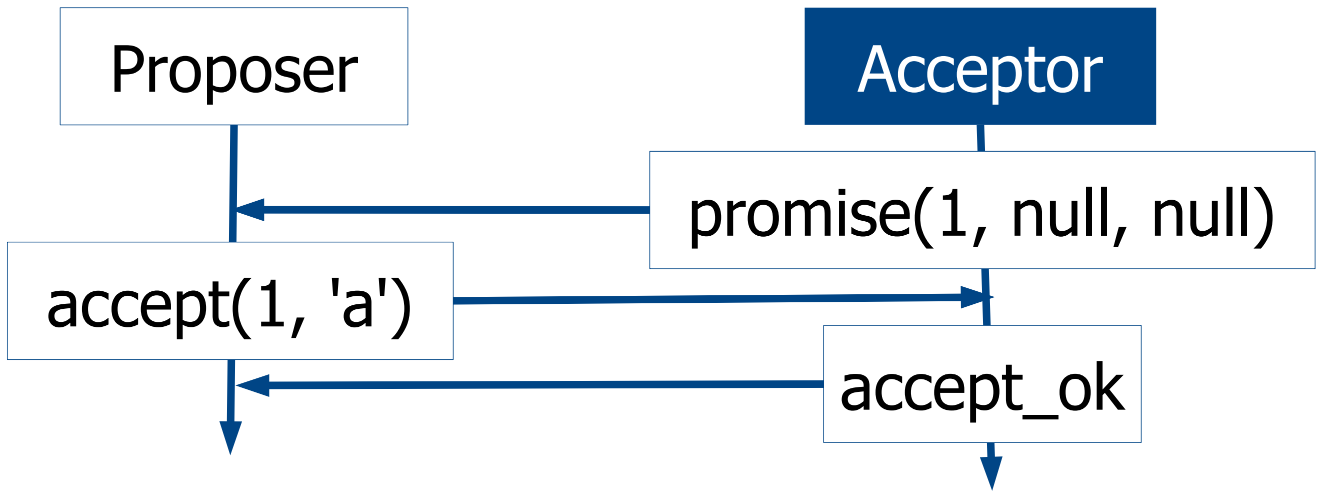
The Paxos protocol is presented in three phases, taking the proposers perspective. In the first phase one of the processors that seek distributed consensus decide to become a so-locally leader to propose a value. For simplicity, we will call it the proposer always. For simplicity, also assume the proposer is the same always.

The proposer generates a new sequence number and asks the others to accept it. Assume all acceptors have not seen any higher sequence number from any more recent proposal. Then, they promise to reject proposals with a lower number and return the highest previously accepted sequence number and value, if any.

The protocol: phase 2

Accept

- Proposer checks majority replies for the value V associated with the highest N. If V = NULL proposer may pick a value, else proposer must pick that value.
- Proposer sends accept(sequence number, V)
- Acceptor rejects if his high > N, else
  set high = N, last accepted v = V, last accepted n = N



The speaker says…

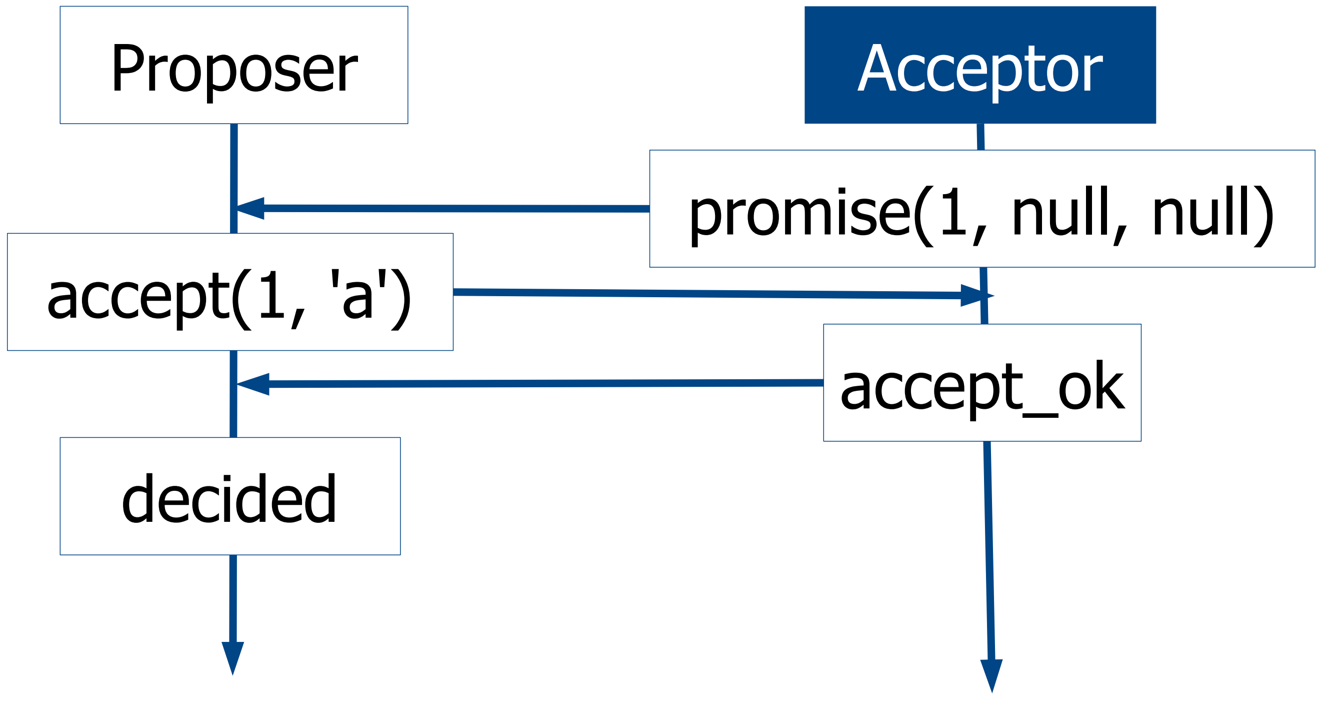
Given that a majority of acceptors agree on the proposal, the proposer may continue. If they do not, the proposer stops.

The proposer checks the replies to determine a value everybody shall accept. If no acceptor has reported any previously accepted value, then the proposer may choose a value. Otherwise, the proposer must pick the latest value accepted reported by the majority of acceptors. Assume the case of being allowed to suggest a value, then the proposer sends an accept() message with his sequence number and his own value. Upon receipt, acceptors check the sequence number again. If it is still the most recent proposal, they reply OK and remember their decision.

The protocol: phase 3

Decide

- If no majority sends OK to proposer, delay and restart
- Else, proposer considers protocol terminated



The speaker says…

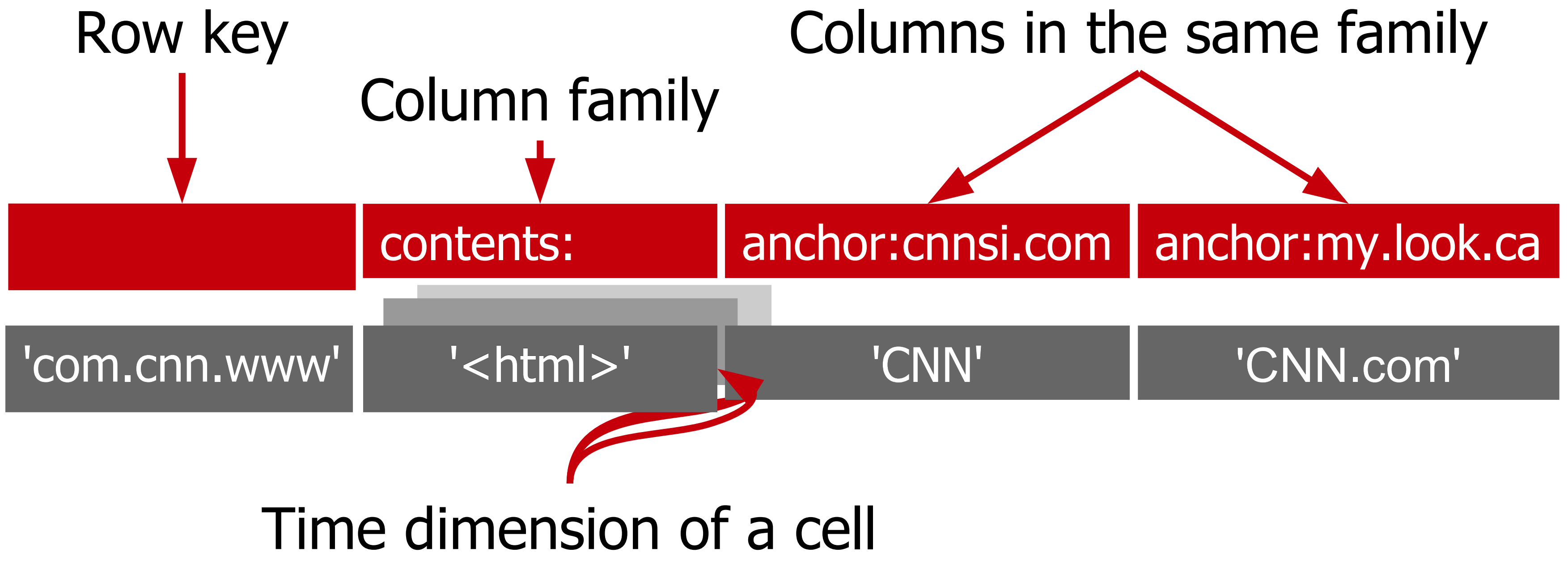
Given that a majority of acceptors sends accept_ok, the proposer considers the protocol terminated and a decision has been made.

Don't consider implementing a Paxo yourself. There are many optimizations to make them faster under these or that circumstances. It is not simple to get them right. Rely on experts work and use one of the existing implementations. Paxos have strong similarieties with a bounch of other algorithms – each having its own pitfalls.

Google Bigtable

Key value store

- Web indexing, Google Analytics, Google Earth, …
- Sparse, distributed, persistent multi-dimensional sorted map
- (row key, column, time) → value, atomic row access



The speaker says…

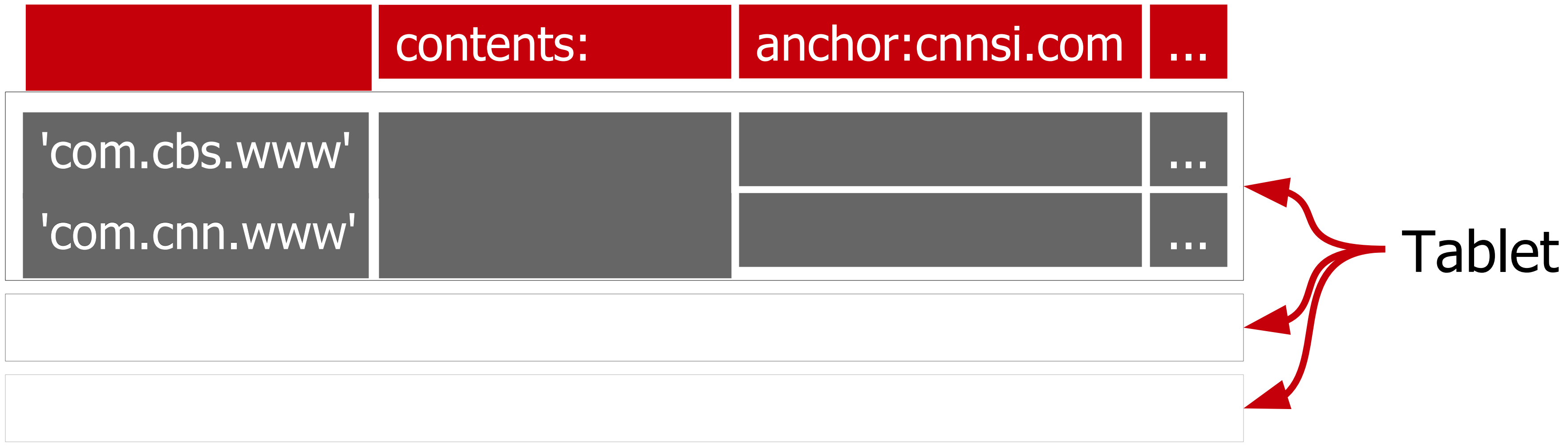
Bigtable was the first distributed storage system for structured data Google discussed in public. It is a huge key value store. The data model is that of a multi-dimentional sorted map. The map is indexed by a row key, a column (family) and a timestamp. The value is a binary string.

The row keys are strings. The map is sorted lexiographically by the row keys. Key can have a length of up to 64K bytes, but typically the size is between 10 and 100 bytes. A row is sparse: before data can be inserted, column families must be defined. However, a column family can contain any number of columns, which can be created at any time. Each cell can contain multiple versions of its data indexed by a timestamp.

Distribution logic: sharding

Tablets are unit of distribution and load balancing

- Map is sorted by keys
- Split into 100 – 200 MB Tablets assigned to tablet servers
- Row oriented storage



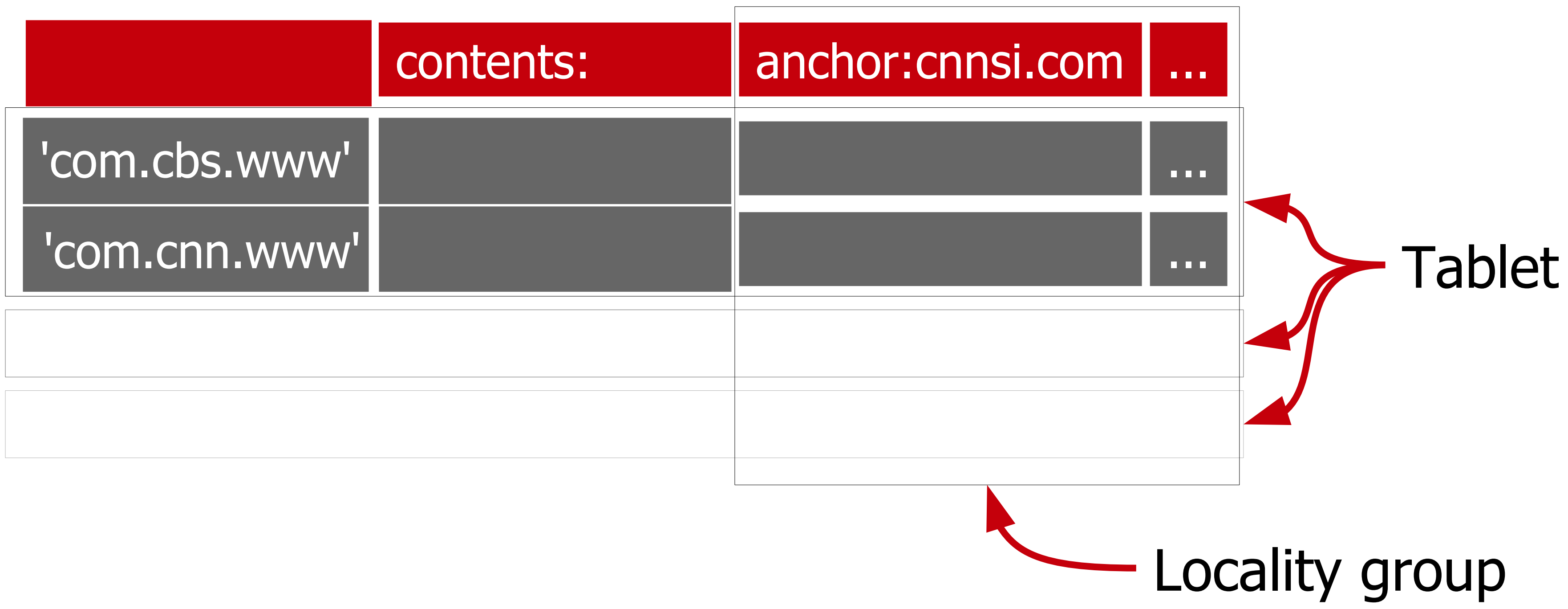
The speaker says…

Bigtable dynamically partitions the map by row key into so-called tablets. Tables are the unit of distribution and load balancing. By carefully planning and choosing their row keys, client can achive good locality for their queries. Short row range scans are likely to require communication between a small number of machines only.

Locality groups

Optional segregation of tablets into locality groups

- Store commonly used column families together



The speaker says…

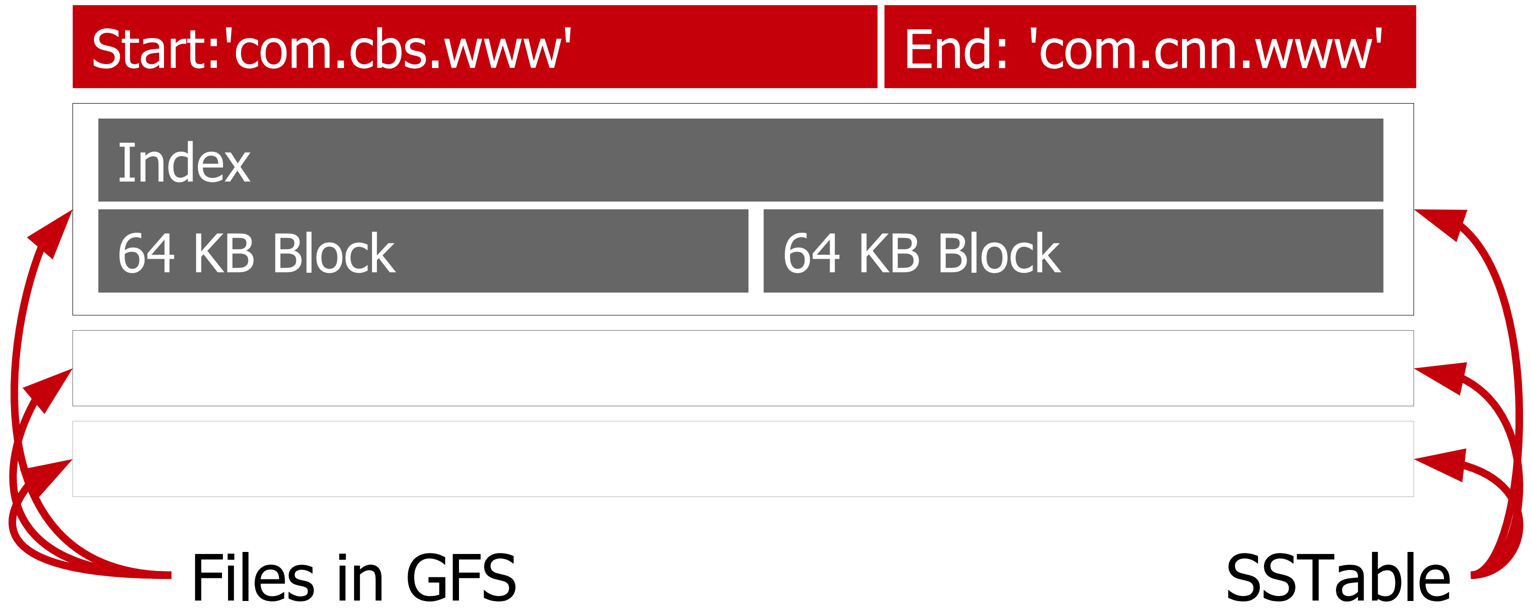
Users can further influence the physical data layout by defining locality groups. Hence, we get a mixture of horizontal and vertial partitioning.

If, for example, a client frequently needs to access the anchor: column family independent of the contents: family, then the anchor: column family can be assigned to be stored together in one locality group. Queries affecting anchor: columns only are not burdened to read any of the contents: columns. The query becomes faster as less data needs to be read.

Physical layout

Tablets are made of SSTables

- Using GFS for strongly consistent replication
- SSTables are an immutable, sorted file of key-value pairs
- Clients can request (lazy) in-memory storage



The speaker says…

Tablets are made of a set of GFS files called SSTables. An SSTable is an immutable, sorted file of key-value pairs. Data can be deleted but not updated in place. New records cause new files to be written. Recall the GFS properties...

Old files are garbage collected on demand. GFS provides replication with strong consistency guarantees to Bigtable.

Clients can request SSTables to be lazily loaded into memory to improve performance. Things like block sizes are configurable – for the same reasons as in classical RDBMs.

Compression

Many clients use two stage approach

- First pass: compression of large windows
- Second pass: eliminate repititions in small 16 KB windows
- 2006: encode 100-200 MB/s, decode: 400-1000 MB/s

Project	Table size (TB)	**Compression ratio**	# Cells (billions)	#Column Families	#Location Groups	% in memory	Latency sensitive?
Crawl	800	**11%**	1000	16	8	0	No
Analytics	20	**29%**	10	1	1	0	Yes
Analytics	200	**14%**	80	1	1	0	Yes
Earth	0,5	**64%**	8	7	2	33%	Yes

The speaker says…

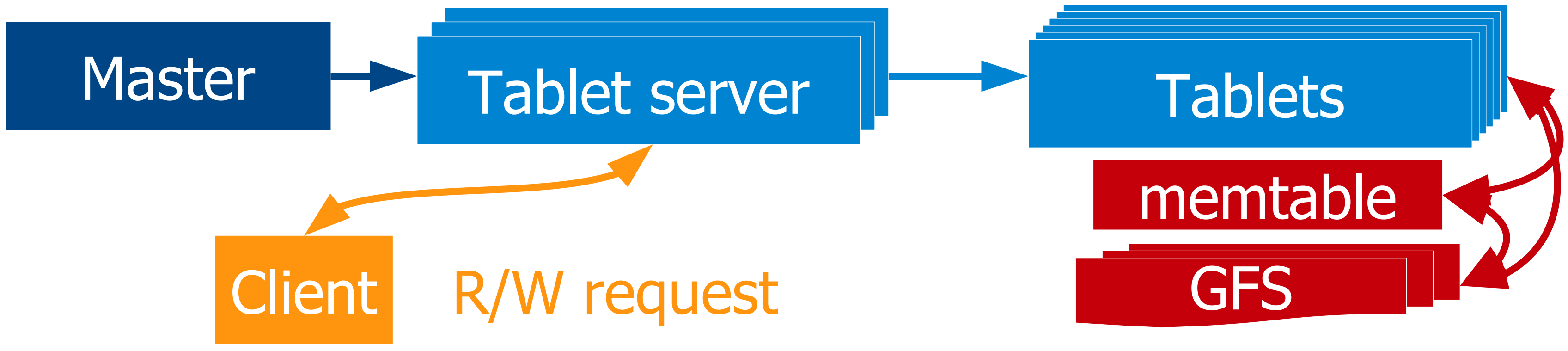
BigTable is using a denormalized storage pattern. Given the resulting data duplication and – often – storage of plain text, one could expect gzip or the like to be a perfect compression algorithm to reduce disk footage (size and thus read performance). However, users found hat a two stage approach could compress HTML up to a factor of 1:10 whereas gzip achieved only about 1:3.

The two stage approach first uses applies compression on a big chunk of data hoping that repetition is more frequent in the large chunk than in smaller ones. The large chunk is likely to contain, for example, multiple version of the same HTML document each differing only little. Then, in a second stage similarities in smaller chunks are compressed.

System view

Master for administrative tasks only

- Monitoring and management of tablet servers
- Assigns tablets to tablet servers (load balancing, failures)
- Garbage collection of GFS files
- Coordinates schema changes



The speaker says…

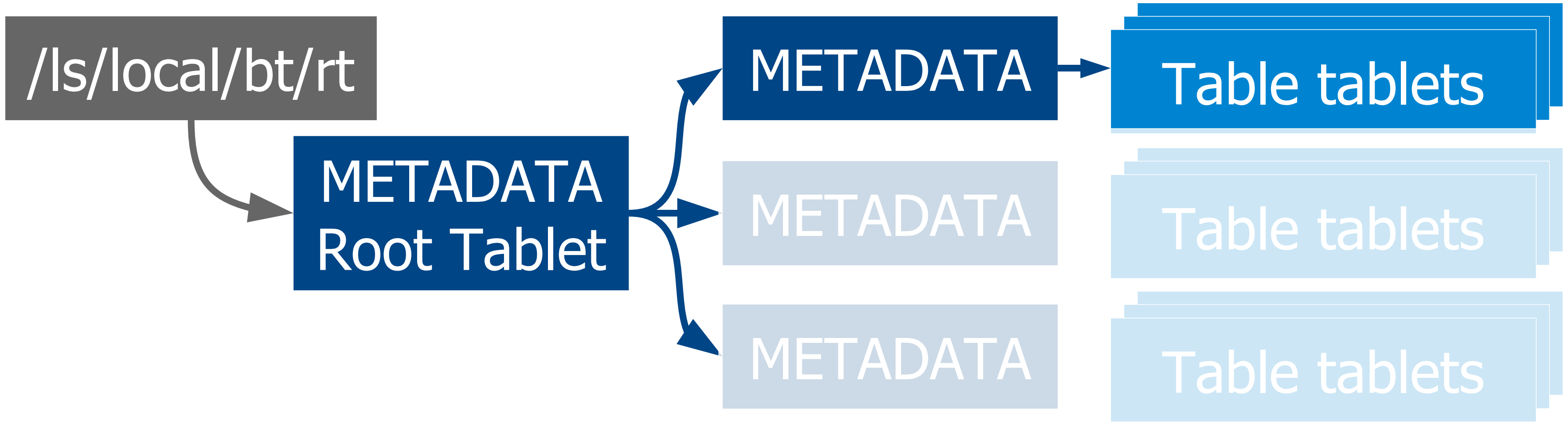
A BigTable cluster is made of a single master, many tablet servers, a GFS cluster and Chubby (see below). Clients hardly ever talk to the master but access tablet servers directly for reading and writing data.

The master performs administrative tasks only. It monitors tablet servers, assigns tablets to tablet servers, takes care of garbage collection of no longer used GFS files (SSTablets) and coordinates schema changes such as creating a table or adding and removing a column family. A management software (not shown) takes care of the master itself.

Tablet location (B+-tree like)

Clients cache tablet locations

- Recursive design using METADATA table
- METADATA table maps (table_id, end_row) → location
- Chubby file stores location of metadata table root tablet



The speaker says…

Tablet locations are stored in a special BigTable system table called METADATA. The METADATA table contains the location of every user table and is indexed by the tuple (table_id, end_row). Clients query the METADATA table to learn about the tablets and tablet servers. Clients cache the locations. To find the first tablet of the METADATA table, clients query Chubby for a file that holds the location of the entry point.

Note the recursive design of the catalog: table locations are stored in a table.

Bigtable and CAP

Large distributed system vulnerable in many ways

- Memory and network corruption
- Large clock skew
- Hung machines
- Software bugs in all layers, administrative failures (GFS quota)
- … and network partitioning

In the context of CAP

- Consistency – strong: using GFS and atomic row access
- Availability – no figures available: Chubby availability 99.995%
- Partition tolerance – focus one data center, Chubby sessions

The speaker says…

Bigtable is a proof that even in the presence of CAP one can design very large and scalable distributed systems. Google reports more causes of system vulnerability in large distributed systems than just network partitioning. I am not aware of any recent and extensive survey on the cause of failures – most information is dated. However, as a rule of thumb and general principle CAP also applies to Bigtable. BigTable is strongly consistent. Availability is bound to Chubby. Chubby availability is reported to be between 99.99% (low) and 99.995% (avg). Chubby sessions help detecting partitions quickly, Paxos ensure progress. State is in Chubby and GFS. This helps restarting the system (components) quickly „from scratch“. Tendency to CP.

Welcome to HBase!

Key-Value store, follows Bigtable paper

- Direct Java API, Binary Thrift interface, HTTP/REST interface
- Stores anything: BLOB (anything: XML, JSON, image...)
- Strong consistency for single cluster
- Eventual consistency for replicated clusters

Cluster-wide search

- Hadoop MapReduce, Scans, Bloom Filter

Durable storage

- HDFS distributed file system

Auxiliary

- ZooKeeper (Chubby), Chukwa (Monitoring), Hive, Pig

The speaker says…

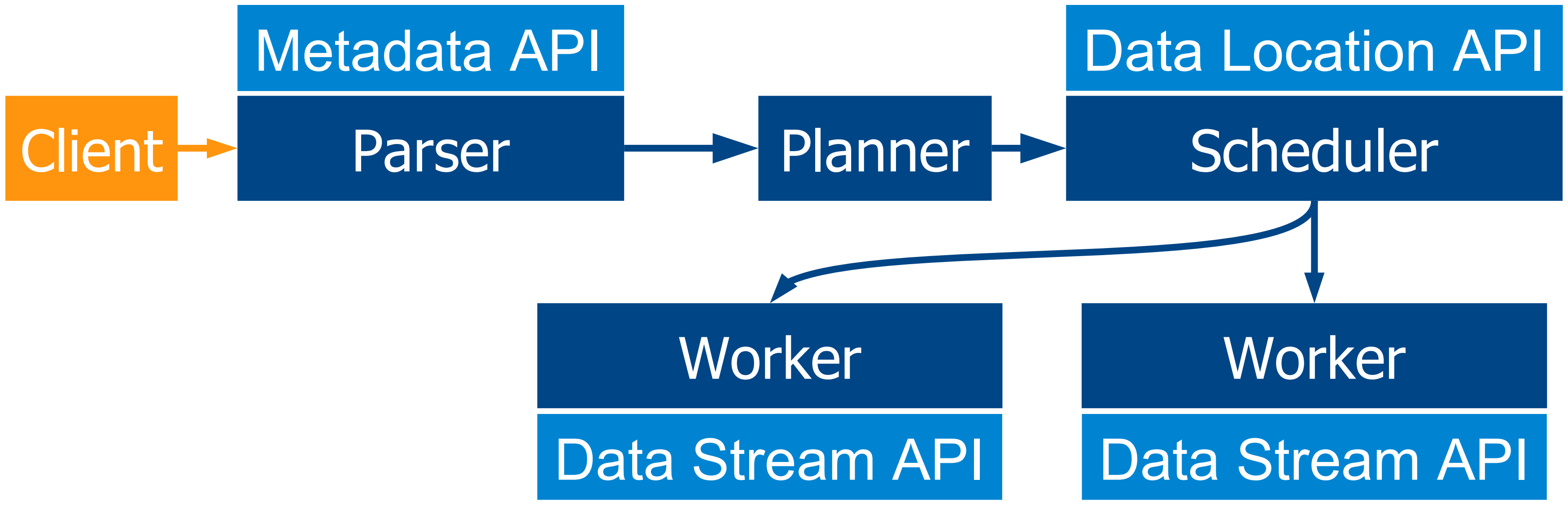
The Open Source counterpart of Bigtable is HBase. HBase builts on top of Hadoop. Hadoop provides MapReduce capabilities and the HDFS distributed file system. ZooKeeper takes the role of Chubby: distributed configuration management, consens and a publish/subscribe messaging system. Real time monitoring is done with Chukwa. Hive and Pig are additional components for cluster-wide search. Hive offers a SQL-like query language called HiveQL, which implements a subset of SQL-92.

If you have just five servers, forget about Hadoop and HBase. Facebook has some 100 Petabytes stored in a single HDFS cluster (figures from August 2012).

Facebook's Presto

Distributed SQL query engine

- Hive comparison: 4-7x more CPU efficient, 8-10x faster
- Approximate queries based on samples: 10-100x faster
- Pluggable storage backends: HBase, Hive, Scribe, ...



The speaker says…

Facebook has recently open sourced a distributed SQL query engine for ad-hoc data analysis. The SQL query engine can work with assorted data stores among them HBase. It promises significant better CPU efficiency and performance than Hive. Hive also was initially developed by Facebook.

Hive uses MapReduce jobs. MapReduce batch jobs are executed sequentially and persist intermediate results on disk. Presto creates Java JVM bytecode that executes through multiple workers using pipelining and in-memory processing where possible.

Cloudera's Impala

Distributed query engine

- Based on Google Dremel paper
- HiveQL: SELECT, JOIN, aggregate functions... - what else ;-)
- Hive I/O bound comparison: 3-4x faster
- Hive with joins comparison: 7-45x faster
- Hive with cache hit: 20-90x faster

The speaker says...

Probably, the #1 to ask for commercial Hadoop support is Cloudera. At about the same time like Facebook, Cloudera has also unveiled a distributed query engine for HDFS and Hbase called Impala. Impala is Open Source and available under the terms of the Apache License.

What's interesting here is to see how traditional RDBMS parallel database optimizations find their way into the Hadoop stack. Add better secondary indexing and benchmark results from commercial parallel RDBMS systems may no longer show the „elephants" (traditional parallel RDBMS) lightyears ahead of Hadoop & friends?

Summary: Bigtable & frieds

Data model

- Key Value: (string, column, timestamp) → (binary)
- Uninterpreted BLOB
- Basic scan, key lookup and MapReduce
- Sharding, vertical partitioning
- Locality can be influenced: choice of key, locality groups

When?

- BigData, preferrably with a temporal aspect

When not?

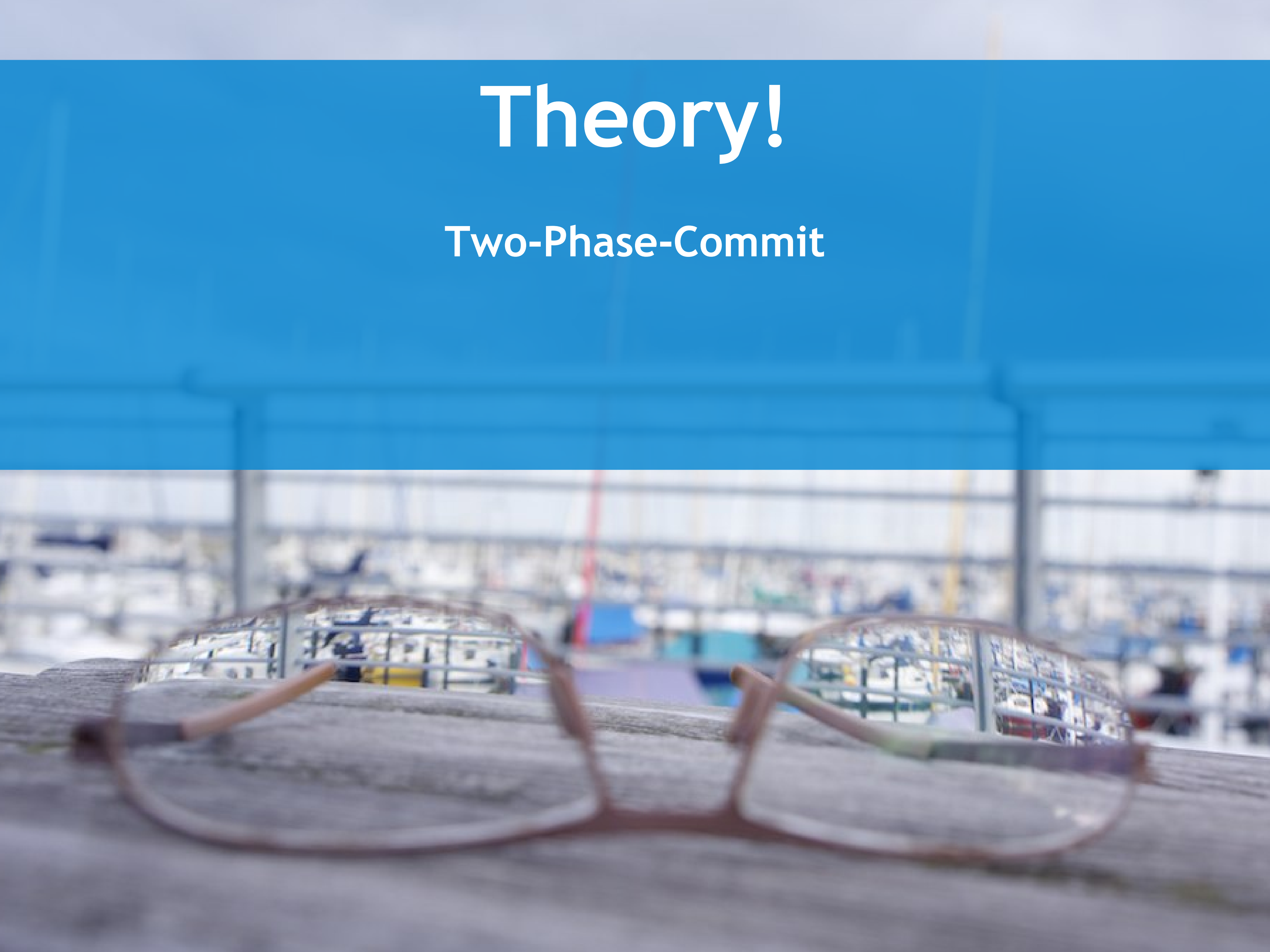
- Transactions!

Break?!

Next: Google Spanner – millions of nodes

The speaker says…

Anyone in need for a break?



Theory!

Two-Phase-Commit

On our way to Spanner

2-Phase Commit

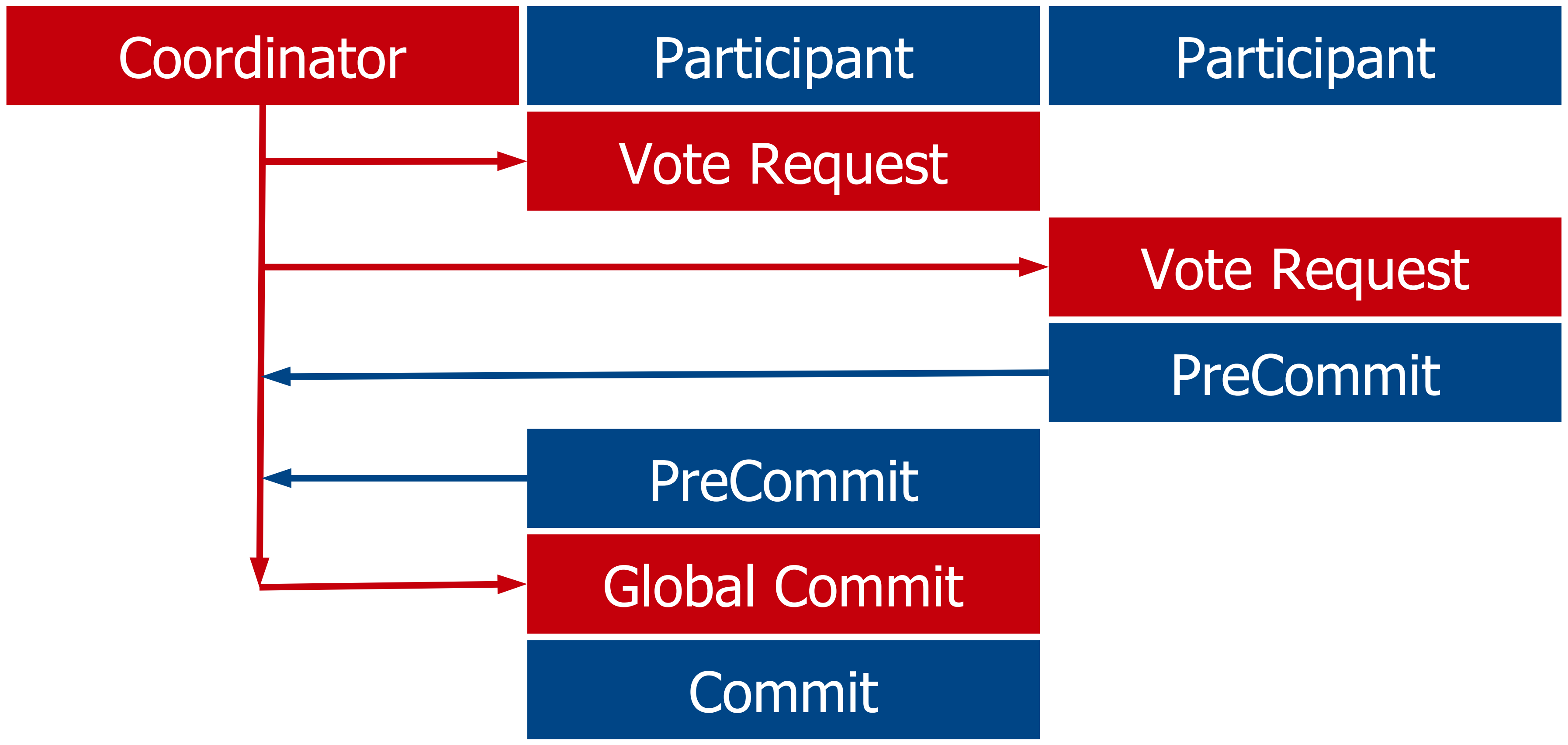
The speaker says…

Lucky you! You have seen almost of the major building blocks required to understand Google Spanner. And, this is the second last theory block for today.

Two-Phase Commit

Complex failure handling required

- Later evolution: Three-Phase Commit (3PC)



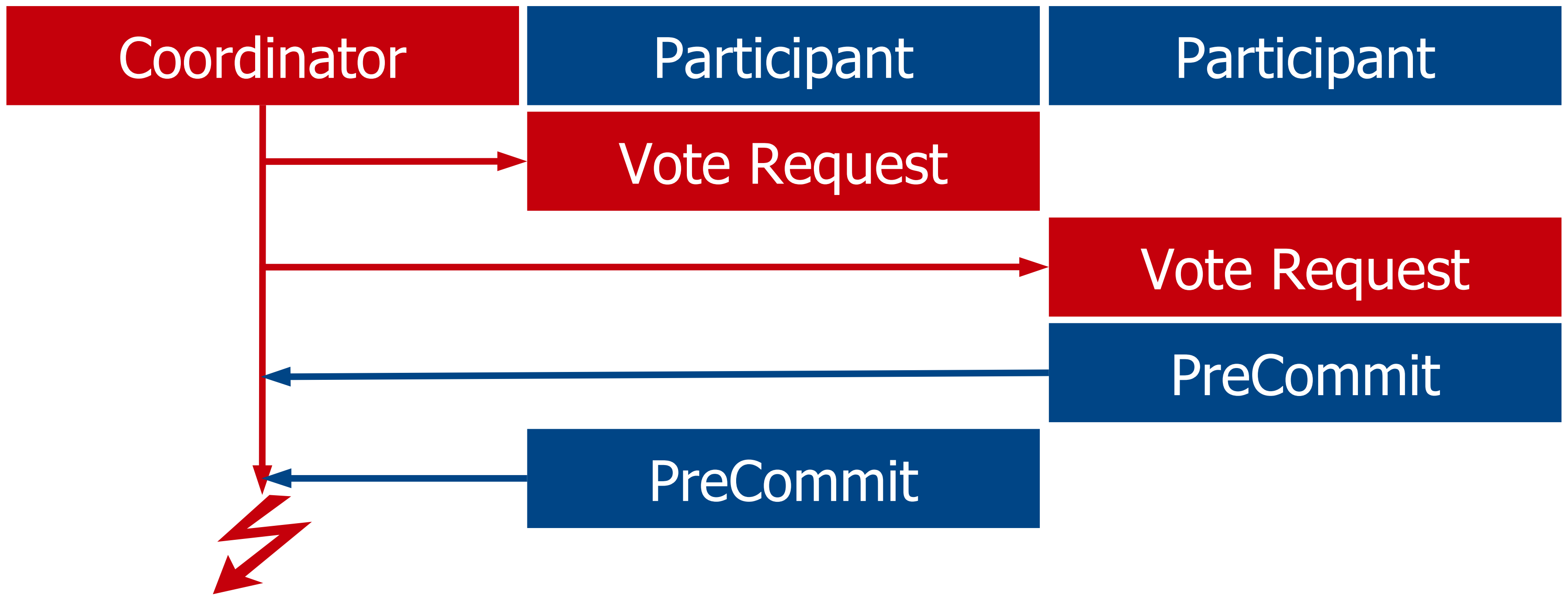
The speaker says…

Two-Phase Commit is a protocol for distributed commit of transactions. It consists of two phases: 1) prepare, 2) commit (or abort).

One process looks for a decision on a transaction on which multiple processes took part. To decide on the outcome of the transaction, the so-called coordinator, sends a prepare request to all participants. Then, participants checks if they could commit, or want to abort the transaction. A participant that replies to the coordinator that he could commit makes a firm statement that must not be reverted. In an ideal world all participants reply to the coordinator that they could commit and the coordinator sends out a global commit message. Otherwise, the coordinator sends global abort.

Fault Tolerance: 2PC

Two-Phase Commit is a blocking protocol



The speaker says…

Various issues can occur in an asynchronous environment.

The worst case scenario is a crash of the coordinator after all participants have voted to precommit. The participants cannot leave the precommit state before the coordinator has recovered. They do not know whether all of them have voted to commit or not. Thus, they do not know whether a global commit or global abort has to be performed.
As none of them has received a message about the outcome of the voting, the participants cannot contact one another and ask for the outcome. Spanner has a clever yet simple solution for this...

Google Spanner

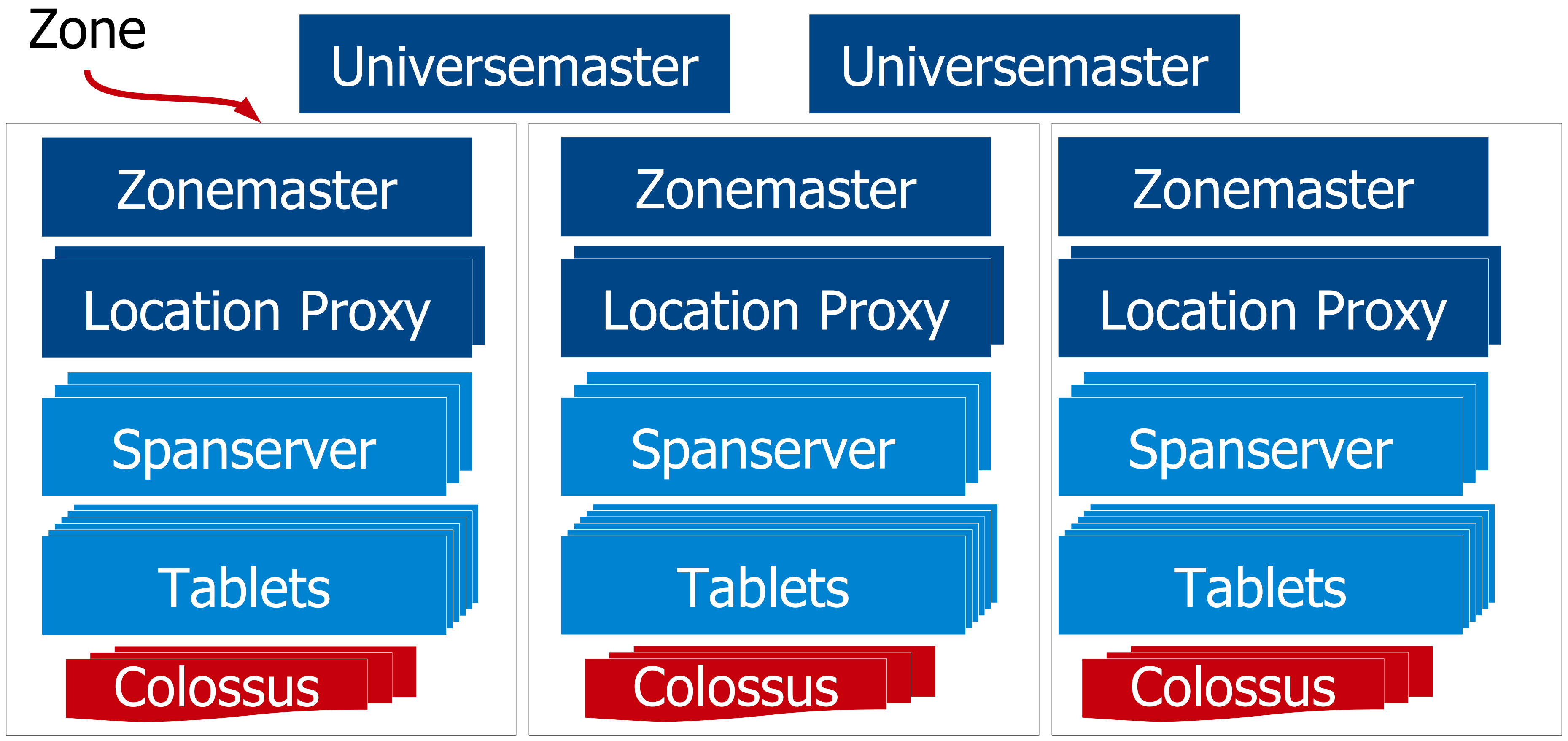
Structured key value store

- Distributed multi-dimensional map:
  (key [, timestamp]) → string
- F1 advertising backend (goodbye MySQL!), ...
- Schematized semi-relational tables
- ACID transactions
- Declarative SQL-based query language
- MapReduce
- Unique TrueTime API
- Synchronous replication among data centers

The speaker says…

Google Spanner is a globally distributed database that succeeds Bigtable. Bigtable was followed by a system called MegaStore which was built a-top of Bigtable and offered synchronous, strongly consistent replication between data centers. Furthermore it had support for transactions and a SQL-based query language. MegaStore quickly became very popular as developers prefered the stronger guarantees it offered albei replication among data centers was not very fast. This resulted in the development of Spanner. Spanner was first used by F1 advertising app. F1 used to be deployed on a sharded MySQL setup. However, resharding was too complex (2 years for resharding!) and replication was not good enough: asynchronous, no easy failover.

System view


Zone
Universemaster
Universemaster
Zonemaster
Location Proxy
Spanserver
Tablets
Colossus
Zonemaster
Location Proxy
Spanserver
Tablets
Colossus
Zonemaster
Location Proxy
Spanserver
Tablets
Colossus

The speaker says…

A Spanner deployment is called a universe. The universemasters are for monitoring and debugging.

The universe has many zones. Zones are the unit of physical isolation: there can be one or more zones in a data center. Replication is between zones.

Each zone has one zonemaster and between one hundret and one thousand of spanservers. There's a hot standby for the zonemaster, similar to Bigtable. Clients ask the location proxies where to find data. The GFS distributed file system has been replaced with Colossus. A placement driver rebalances data.

Hierarchical data model

Application can impact locality

- Related data is stored together physically: reduce join costs

```
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 CASCADE DELETE
```

The speaker says…

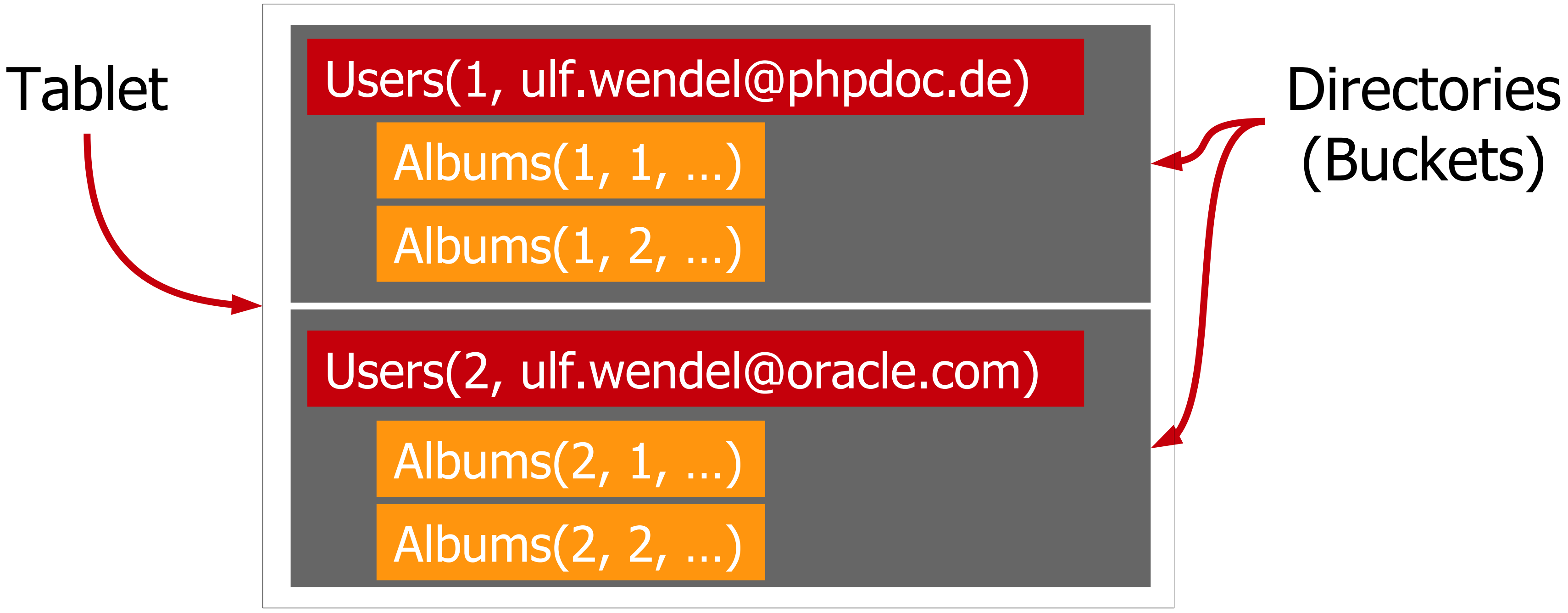
Joins are very expensive in any distributed database. It is the laws of physics that make them expensive. Even sophisticated query optimizers cannot avoid sending data from one node to another for computing a join. However, in a globally distributed database sending large amounts of data around can be prohibitively expensive. Thus, clever co-location on nodes is a must. What we do here is store two tables physically together to avoid a join.

To do so, one defines a parent table and a derived table. The derived table – albums – must prefix its primary key with the primary key of the parent.

Tablets and directories

Directories are the unit of distribution and load balancing

- A tablet holds any directories frequently accessed together
- Automatic balancing



The speaker says…

The „prefixing“ will ensure that records from the two tables interleave. On the storage level, one row from the users table with uid = 1 is followed by all rows from the albums table where uid = 1 and so forth. This makes a join between the two tables cheap as it translates to a disk scan.

Spanner partitions data with respect to hierarchy in the data model. The partitions are called directories. Load balancing and data distribution is based on directories.

A tablet is a container for directories. Please note, Spanner tablets are quite different from Bigtable tablets.

Full ACID Transactions

All Transactions

- Using TrueTime API for linearizability

Read-Write Transaction

- Pessimistic, 2 phase locking
- Replica leader involved

Read-Only Transaction

- Optimistic, lock-free
- Replication leader to get timestamp, otherwise run anywhere

Non-transactional, consistent snapshot read

- Optimistic, lock-free
- Client to provide timestamp or timestamp bound

The speaker says...

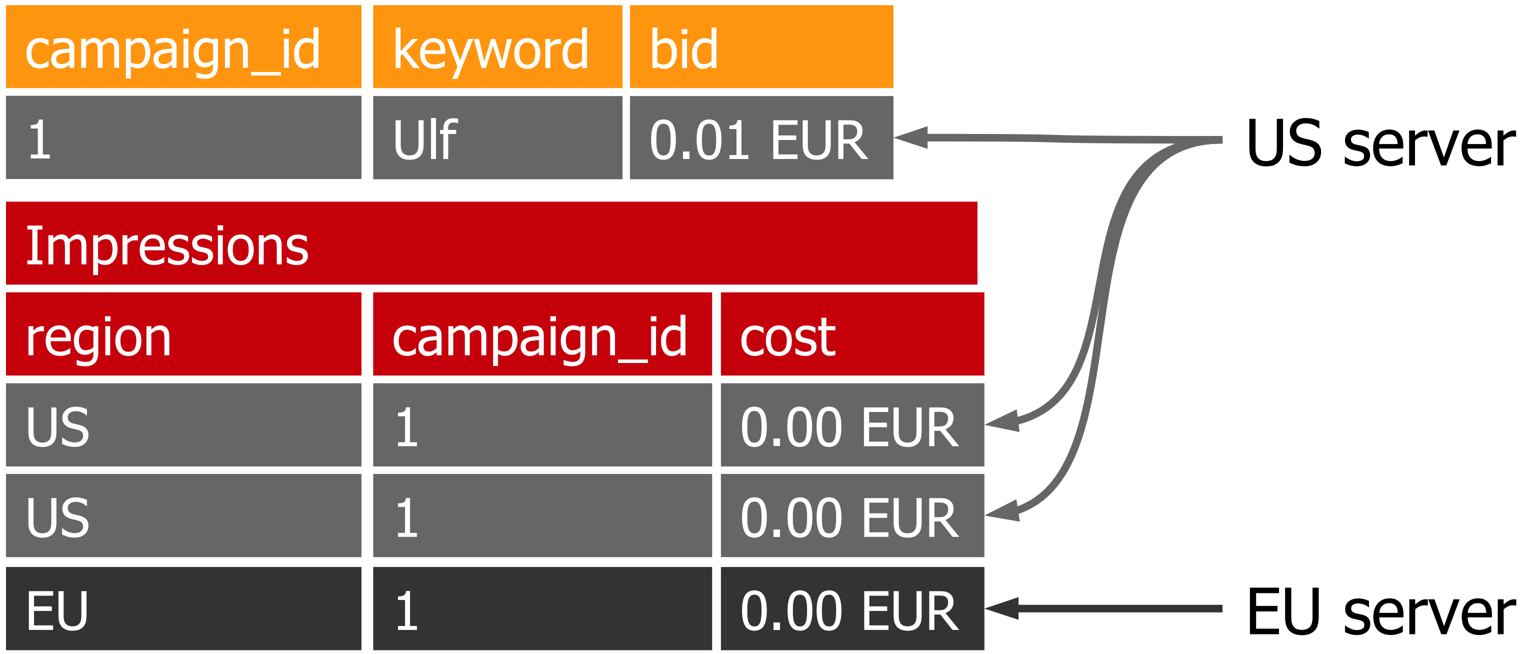
Spanner supports (ACID) transactional reads and writes as well as snapshot reads. All reads are non-blocking and lock-free. Reads use a multi-versioning approach. The way data is versioned is unique: Spanner is using clock time for versioning. More on their TrueTime to come.

Writes use two-phase locking. This choice has been made with respect to long running MapReduce jobs. Pure multi-versioning concurrency control is known to be tricky with respect to long running transactions: validation happens at the end of the transaction. Imagine you run a 10 minute MR job and don't see a conflict before it's done...

Example: why TrueTime rules

Ad system with two tables: campaigns and impressions

- Impressions (click) table sharded among US/EU for locality



campaign_id	keyword	bid
1	Ulf	0.01 EUR

Impressions

region	campaign_id	cost
US	1	0.00 EUR
US	1	0.00 EUR
EU	1	0.00 EUR

The speaker says…

Assume you are running an ad system which has one table with compaign details and one that records the impressions/clicks. To improve performance, you shard the impressions table based on the region. Clicks from EU customers shall be recorded in the EU to avoid long network latency. The same is true for US customers: they shall not wait for an remote EU server to confirm the recording of a click.

Example: why TrueTime rules

Standard transaction ordering scenario...

- Campaign gets created on US server with transaction T1
- Ad appears, user clicks on ad in Europe
- EU server writes impression using transaction T2
- If anybody, anywhere in the world reads the result of T2 then he must also read T1's result, because it happened before T2.

```
T1: INSERT campaign(id, ...) VALUES (1)
```

```
T2: INSERT impressions(campaign_id, region, ...) VALUES (1, 'EU', ...)
```

```
T3: BEGIN; SELECT * from campaign; SELECT sum(costs) FROM
        impressions WHERE campaign_id = 1; COMMIT
```

The speaker says...

Let's create a simple scenario where two transactions need to be ordered correctly. The transactions execute independently from each other on different servers, even on different continents.

The first creates a campaign, the second records a click belonging to the campaign. If, after some time, a third transaction is run that reads the result of the second transaction, it is clear that it also needs to see the result of the first one. There's some order between the transactions, which we have to record. As we learned from Dynamo, we can't simply use wall clock. Clocks are not perfectly synchronized: the clicks timestamp could be lower than the campain creation timestamp... what do to? Vector clocks?!

Vector clocks are tricky

Vector clocks must be passed around with messages

- Campaign gets created on US server with transaction T1
- Ad appears, user clicks on ad in Europe
- **CAUTION:** Ad in browser must contain vector clock of T1
- **CAUTION:** JavaScript click recorder must send VC to DB
- EU server writes impression using transaction T2
- **CAUTION:** DB stores T1's VC together with T2 and its own VC
- If anybody, anywhere in the world reads the result of T2 then he must also read T1's result, because it happened before T2.
- **CAUTION:** DB must detect casuality between a good number of VC's. Remember vector has entry for each node!

The speaker says…

The problem with vector clocks is that they must be passed around with every message to be able to detect a casual dependencies. In order for the European server to know that its impression record must serialize after the creation of the campaign on the US server, the EU server needs to get the vector clock of the US server. To compare the clocks and sort events, one needs the clocks...

In practice this means, the VC must find its way from the US server via the ad in the browser, the JavaScript that records the click to the EU server. This is too error prone!

TrueTime rules!

Wall clock with a bounded deviation on every node

- $Time_{now}$ = TrueTime.Now($Time_{past,}$ $Time_{future}$)
- Deviation between 0 and 6 ms, 90% < 2 ms
- The bound is important not the exact time!

t = 1.234567	T1: INSERT campaign(id, ...) VALUES (1)
t = 1.234568	T2: INSERT impressions(campaign_id, region, ...)
t = 1.345678	T3: BEGIN; … ; COMMIT

The speaker says…

Given the problems with vector clocks, Google decided to bite the bullet of synchronizing wall clock time. Wall clock time is monothonically increasing (no funny leap seconds allowed...) and available around the globe. Thus, it can be used to serialize transactions reliably around the globe. Using a combination of GPS and atomic clocks they managed to develop a TrueTime service which can give a precise estimation of the exact time.

The TrueTime API does not return the exact time but returns a promise that the time is beween a lower and upper value. The interval is some, few milliseconds.

Replicated write

Protocol for write transactions

- Acquire locks
- Execute reads (and shadow writes)
- Commit $\text{ctime}_{\text{now}}$ = TrueTime.Now($\text{Time}_{\text{earliest,}}$ $\text{Time}_{\text{latest}}$)
- Replicate writes using Paxos
- Wait until $\text{ctime}_{\text{latest}}$ < TrueTime.Now()
- Acknowledge commit
- Perform writes
- Release locks

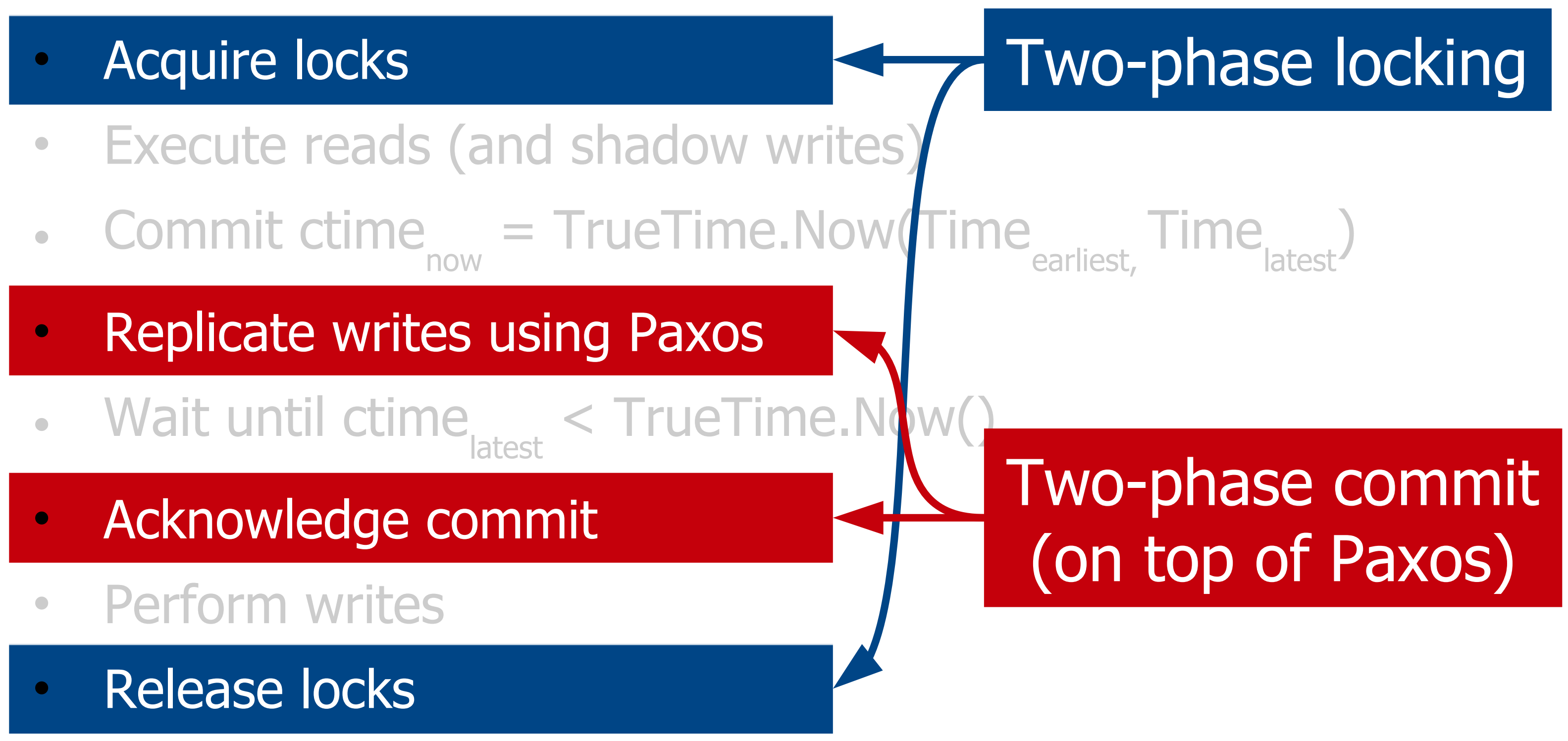
The speaker says…

The transaction protocol explains why it is sufficient to know an interval of the exact time. When a write transaction starts, it acquires some locks and performs it reads and writes. The client sends a commit message and Spanner fetches the timestamp to be used for the transaction.

Writes get replicated using Paxos. If all goes well, transaction processing continues by checking the time again. Spanner waits until TrueTime confirms that its estimated start time is definetly in the past. This is usually the case. Replicating changes often takes longer than the interval of our time estimation was. Hence, no wait in practice.

The two two-phase protocols

Protocol for write transactions



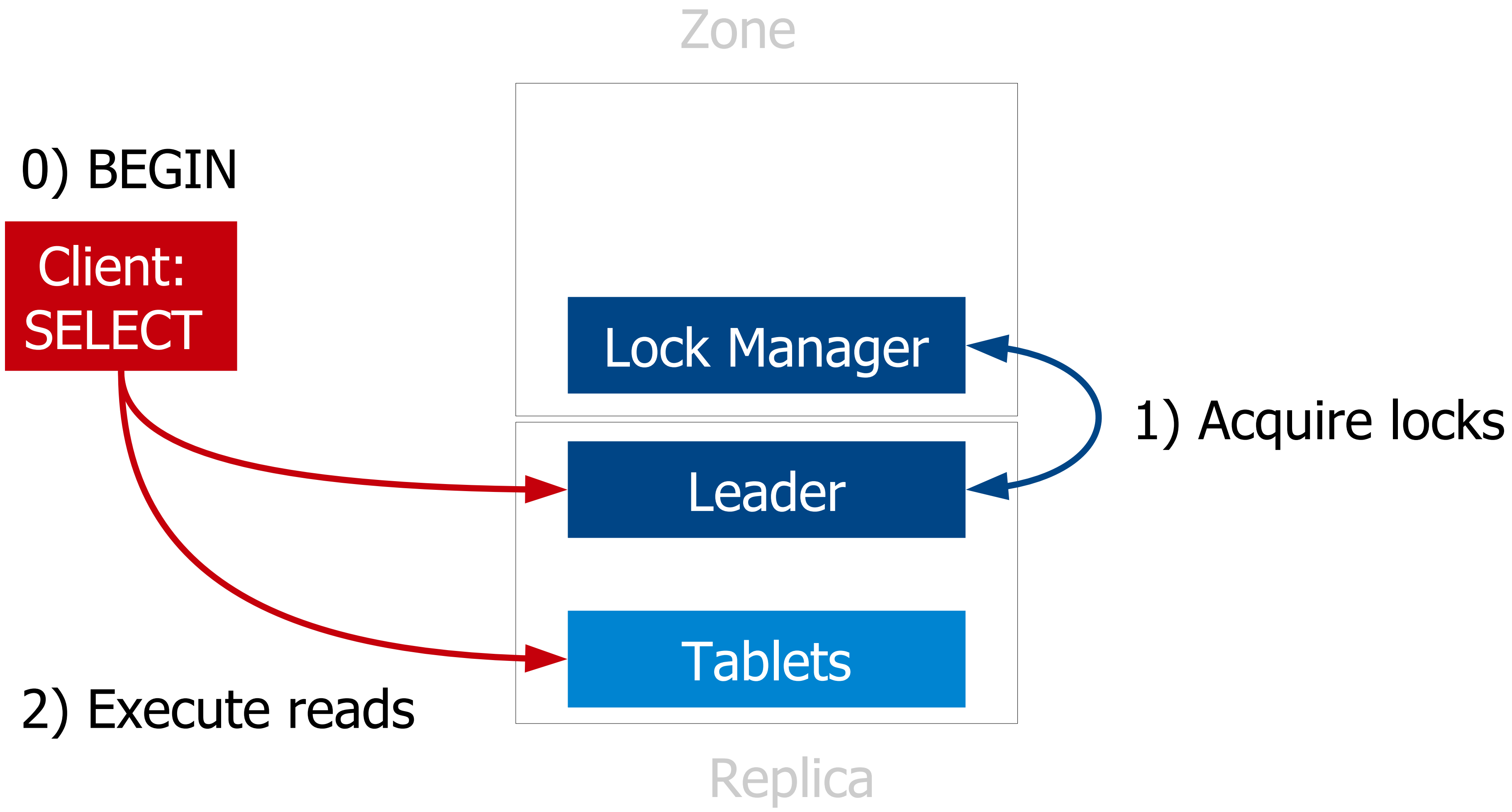
- Acquire locks
- Execute reads (and shadow writes)
- Commit $\text{ctime}_{\text{now}}$ = TrueTime.Now($\text{Time}_{\text{earliest,}}$ $\text{Time}_{\text{latest}}$)
- Replicate writes using Paxos
- Wait until $\text{ctime}_{\text{latest}}$ < TrueTime.Now()
- Acknowledge commit
- Perform writes
- Release locks

The speaker says…

Not only the TrueTime API is interesting. It is also noteworthy to realize how Spanner deals with the blocking nature of the two-phase commit protocol.

2PC is run on top of Paxos. As it takes at least three replicas to gain anything from using Paxos, it can be assumed that Google is using in fact at least three replicas. The paxos will survive the failure of n - (n / 2 + 1) nodes, means one out of three may fail. As 2PC is stacked over paxos the failure of a participant will not require a secondary protocol to ensure progress. Even if one out of three assumed replicas would fails to send an answer to the prepare message, the 2PC protocol would not block.

Transaction system view

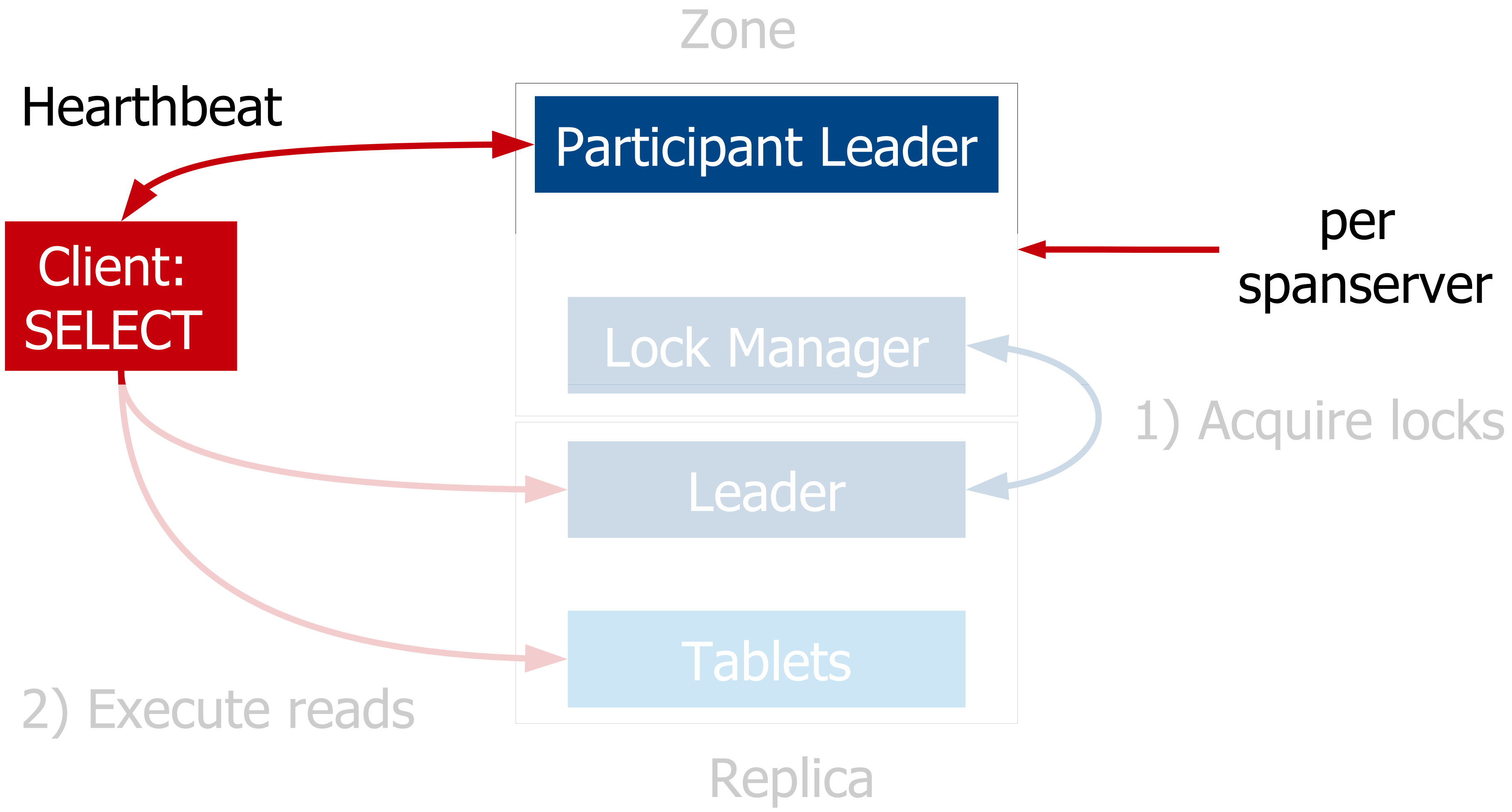


The speaker says…

The most critial issue with 2PC is the failure of the coordinator. In case of Spanner, the client takes the role of the coordinator.

Upon start of a transaction, the client issues read requests to the (paxos) leader of a tablet. Each tablet is replicated using a paxos group. The leader acquires the necessary locks and the client starts reading.

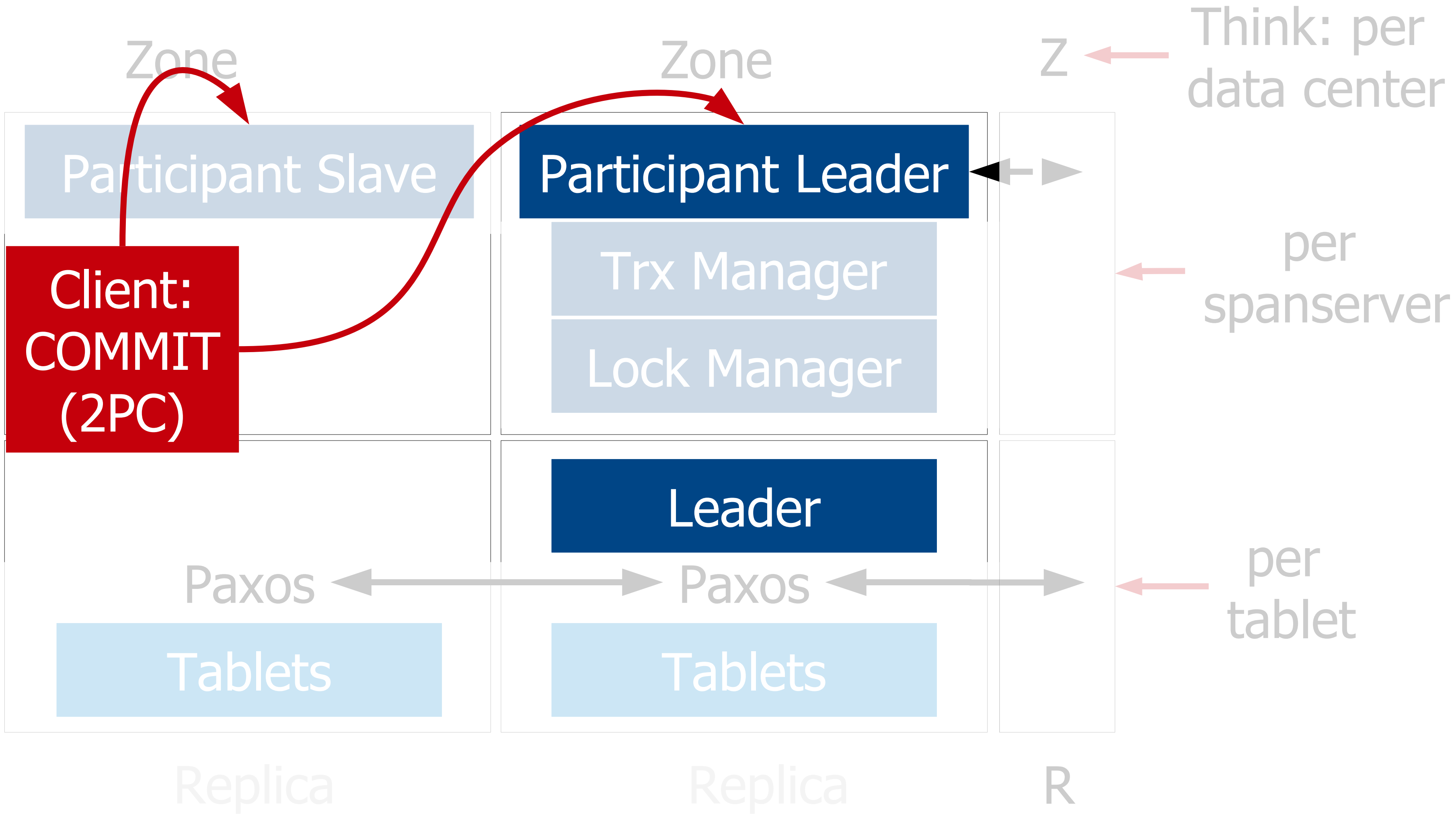
Transaction system view



The speaker says…

Every spanserver has a component called participant leader.
The participant leader and the client exchange heartbeats.

Transaction replication

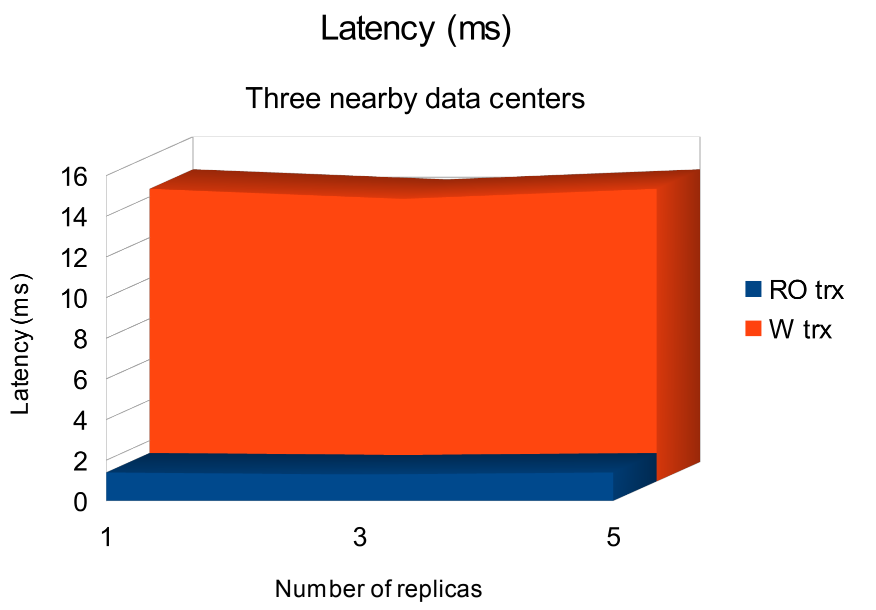


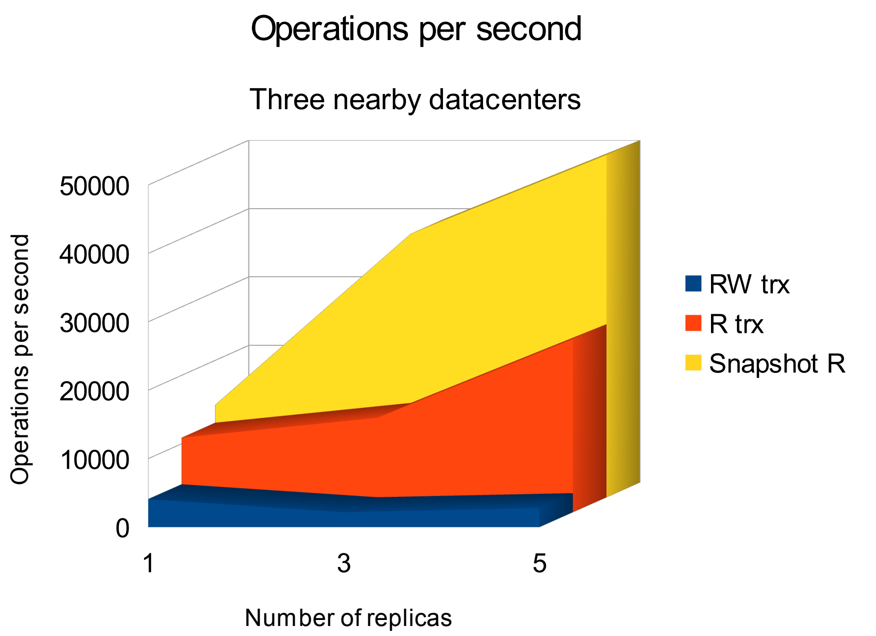
The speaker says…

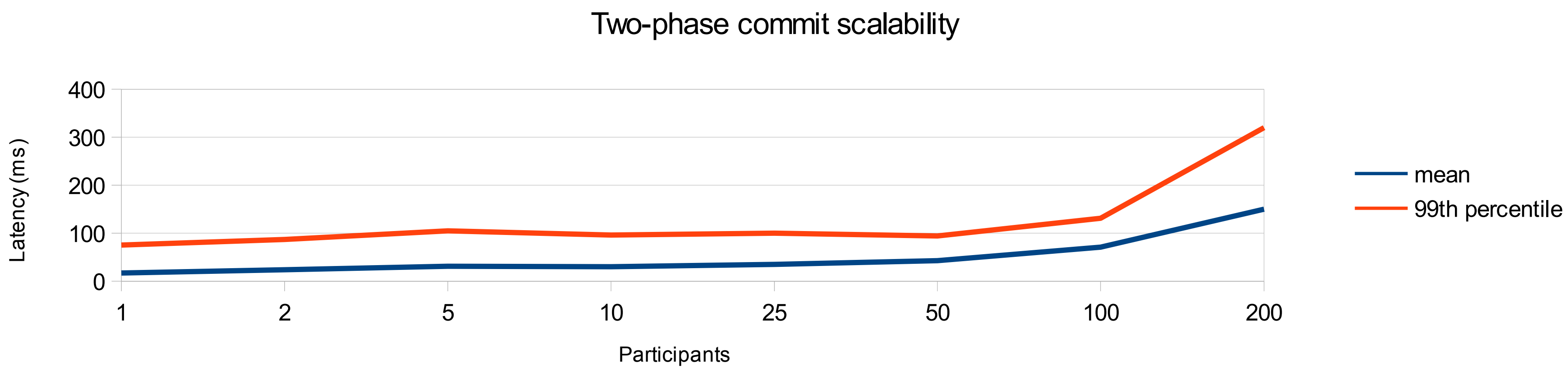
When the client is ready to commit, it begins the two-phase commit protocol. The client becomes the coordinator and takes various actions. We are not interested in the details of who does what and thus no more details are given. The Spanner paper has the details, for example, on how the locks on the other replicas are set.

If the client fails to send keepalive messages, the 2PC coordinator can be assumed to be dead. Then, the participant leaders can step in to solve the blocking case of 2PC: the failed coordinator.

Microbenchmarks







The speaker says…

Google provided only some Microbenchmarks for a test setup among three nearby data centers (1ms latency). Using nearby data centers is a sensible choice as otherwise mother nature dictates the run times. Base/disk overhead for 4 KB write seemed around 5ms, Paxos added some 9ms for one replica. Latency remains stable up to 5 replicas it is around 14ms. Write throughput seems not to drop between 3 and 5 replicas, read throughput goes up. Snapshot read scales proportional by the number of replicas. Two phase commit latency increases significantly beyond about 50 participants. Availability, not shown, is virtually unaffected by Paxos crashes – in the worst case slight degeneration for 10 seconds (leader lease time)

Spanner and CAP

In the context of CAP

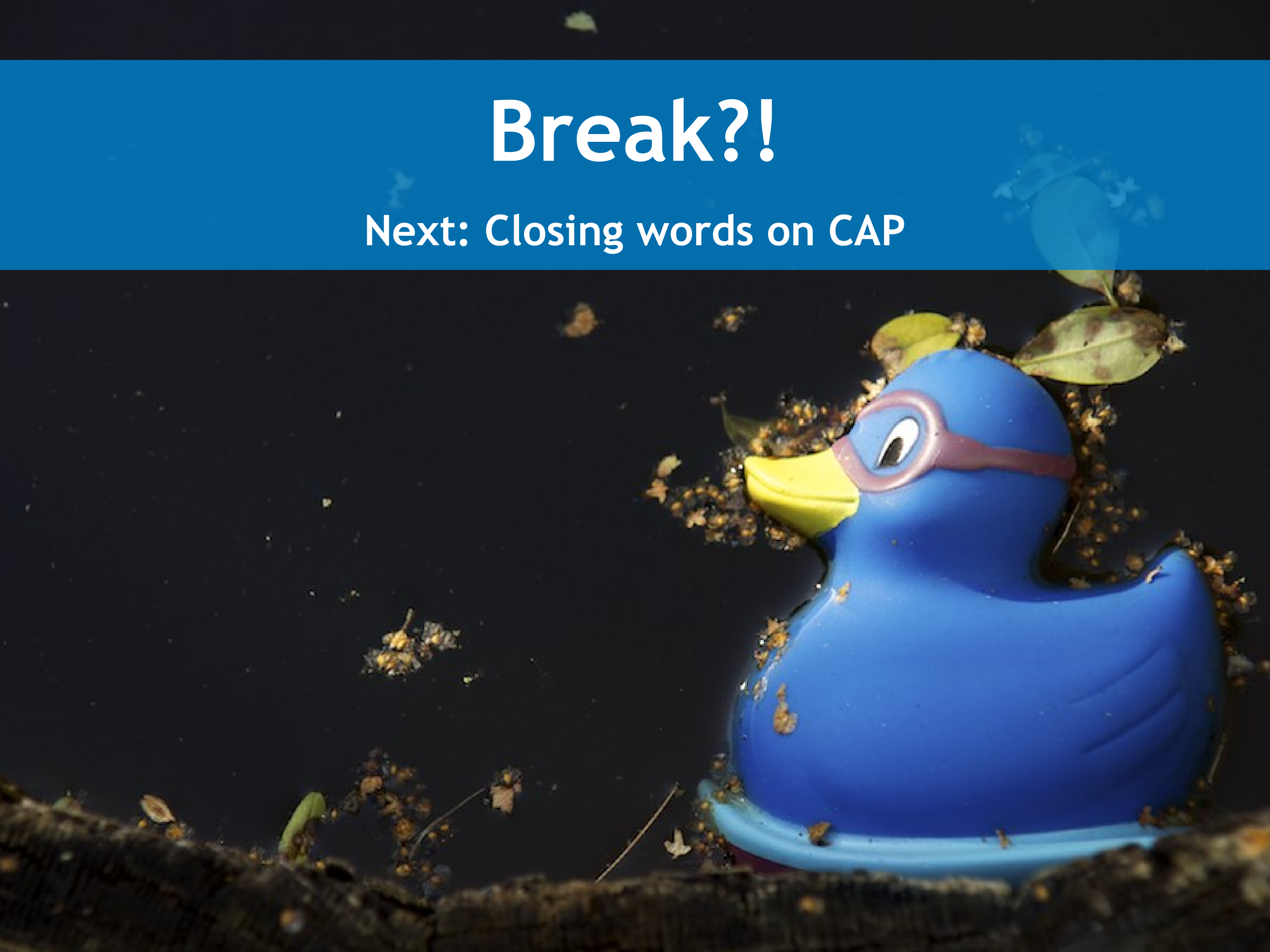
- Consistency – strong: explicit synchronous replication
- Consistency – strong: ACID transactions
- Consistency – strong: using Colossus
- Availability – high: replicated among data centers
- Availability – high: option to replicate world-wide
- Availability – high: single partition replica latency great
- Availability – good: acceptable latency up to 50 partitions
- Partition tolerance – high: Paxos
- Partition tolerance – high: session sub protocols

… Hmm ?!

The speaker says…

Spanner seems to prove CAP wrong. It does not. However, proper engineering can push the boudaries significantly towards a system that is consistent, available and sufficiently partition tolerant.

Join me on a journey towards a distributed systems professors take on CAP. The professor is Ken Birman.



Break?!

Next: Closing words on CAP

The speaker says…

Anyone in need for a break?

Theory!

Group Communication Systems,
Virtual Synchrony (Isis, Isis2),
Atomic Broadcast

The speaker says…

Know what? This is the last bit of theory for today...

Group Communication System

Distributed inter-process communication

- Message exchange in a group of processes
- Has an understanding of who is in the group
- Abstracts networking and routing details
- Provides communication primitives with different guarantees

Simplifies the design of distributed software

The speaker says...

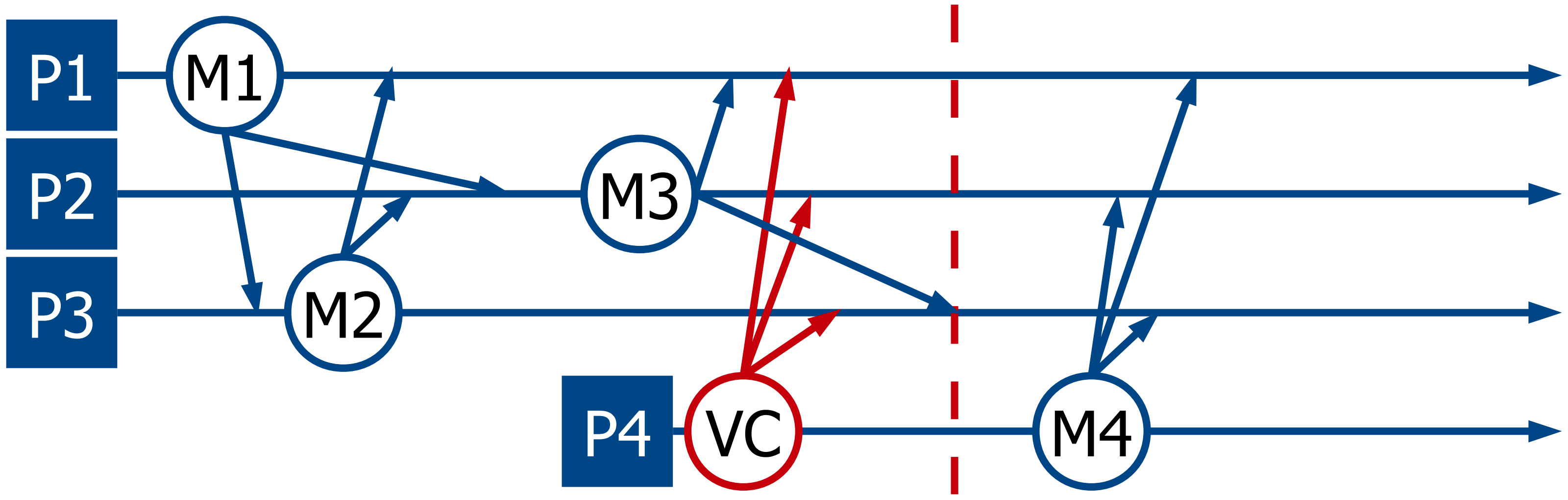
Imagine, you are asked to create a new replication system for MySQL. How do you make a group of MySQL servers communicate with each other? It should be clear that anything along the lines of „I use XML-RPC... Apache... HTTP... PHP“ is prohibitiely slow. Thus, you would have to deal with networking protocols directly. As we will see, this is not trivial. And, how would you know which MySQL servers are currently in your group?

A group communication system can greatly simplify the task. It provides you with simple to understand messaging methods with different guarantees. We will discuss the Isis/Isis^2 GCS briefly.

Virtual Synchrony

Groups and views

- Reliable multicast



G1 = {P1, P2, P3}

G2 = {P1, P2, P3, P4}

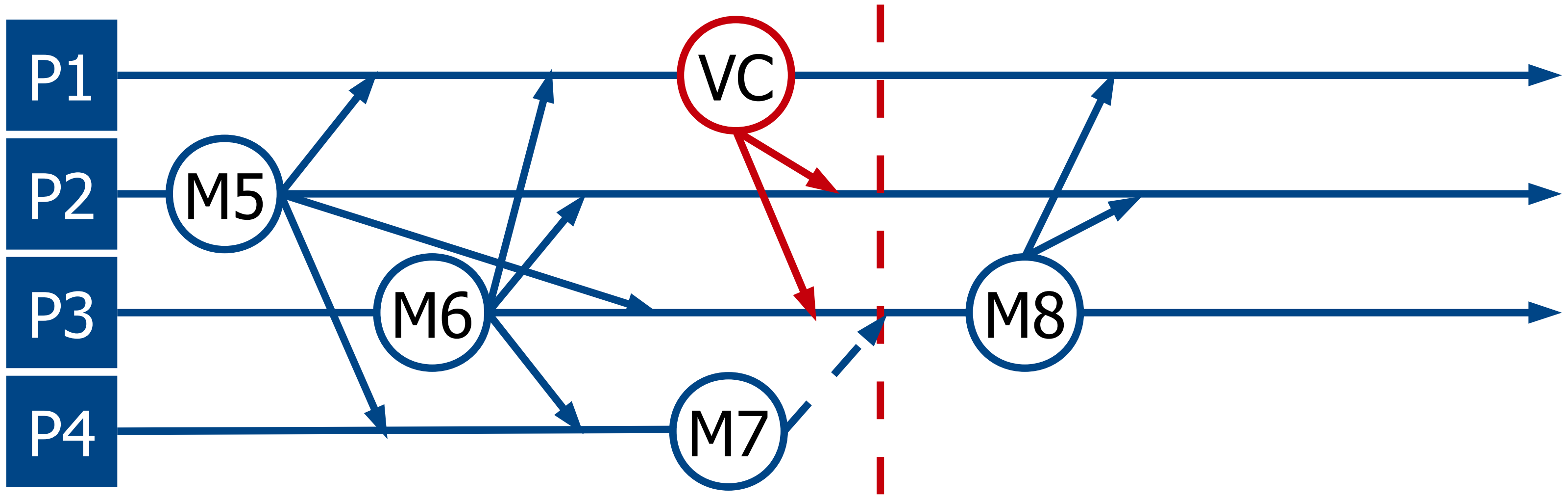
The speaker says…

Virtual Synchrony is build around the idea of associating multicast messages with the notion of a group. A message is delivered to all members of a group but no other processes. Either the message is delivered to all members of a group or to none of them. All members of the group agree that they are part of the group before the message is multicasted (**group view**). In the example, M1...3 are associated with the group G1 = {P1, P2, P3}. If a process wants to join or leave a group a **view change** message is multicated. In the example, P4 wants to join the group and a VC message is send while M3 is still being delivered. Virtual Synchrony requires that either M3 is delivered to all of G1 before the view change takes place or to none.

Virtual Synchrony

View changes act as a message barrier

- Remember the issues with 2PC …?



G2 = {P1, P2, P3, P4}

G3 = {P1, P2, P3}

The speaker says…

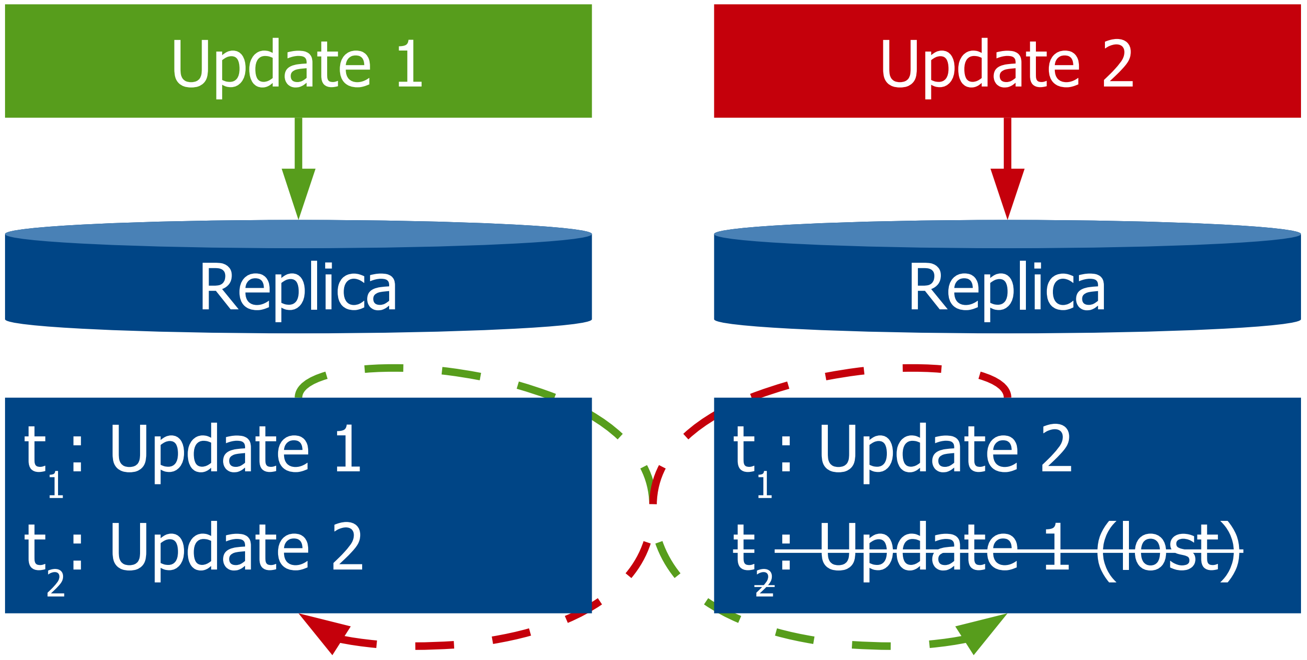
There is only one condition under which a multicast message is allowed not to be delivered: if the sender crashed. Assume the processes continue working and multicast messages M5, M6, M7 to group G2 = {P1, P2, P3, P4}. While P4 sends M7 it crashes. P4 has managed to deliver its message to {P3}. The crash of P4 is noticed and a view change is triggered. Because Virtual Synchrony requires a message to be delivered to all members of the group associated with it but the sender crashed, P3 is free to drop M7 and the view change can take place.

A new group view G3 is established and messages can be exchanged again.

Reliable, delivered vs. received

Message ordering and fault tolerance

- Fast protocols are unreliable – magic required



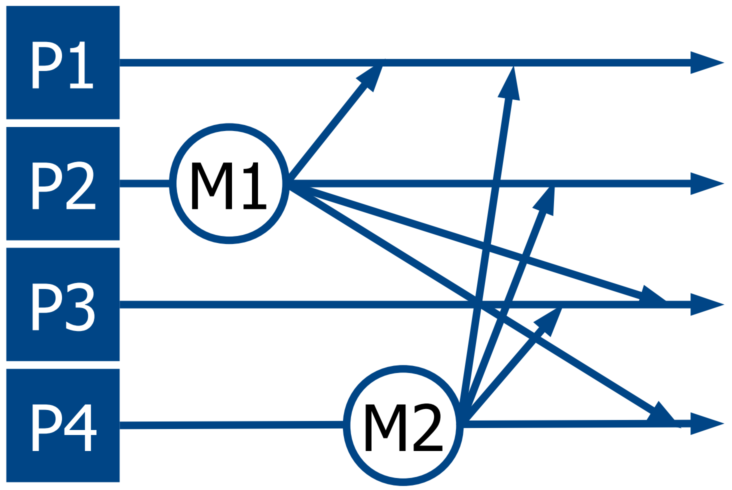
The speaker says…

Virtual Synchrony offers reliable multicast. Reliability can be best achieved using a protocol higher up on the OSI model. Isis, an early framework implementing Virtual Synchrony, has used TCP point to point connections if reliable service was requested. TCP is a connection oriented protocol (endpoint failures can be deteted easily) with error handling and message delivery in the order sent. However, **using TCP only there are no ordering constraints between messages from any two senders.** Those ordering constraints have to be implemented at the application layer. We say a message can be **recieved on the network** layer in a different order than its **delivered to the application** by the model discussed.

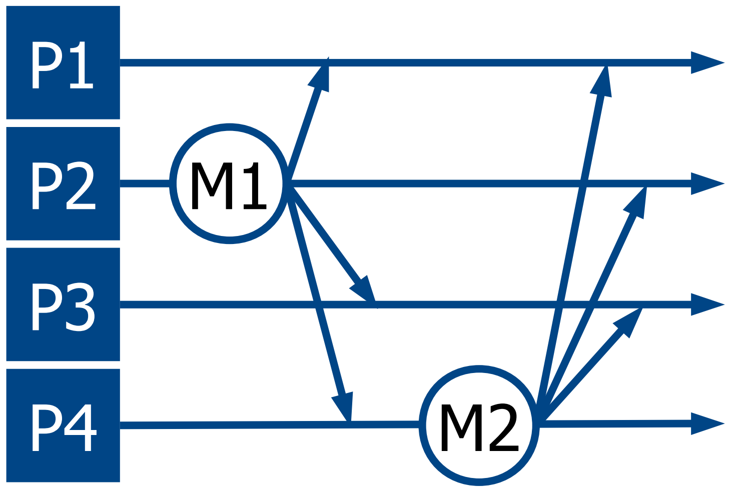
Atomic broadcast definition

AB = Virtual Synchrony offering total-order delivery

- „Synchrony" does not refer to temporal aspects



Unordered delivery



Ordered delivery

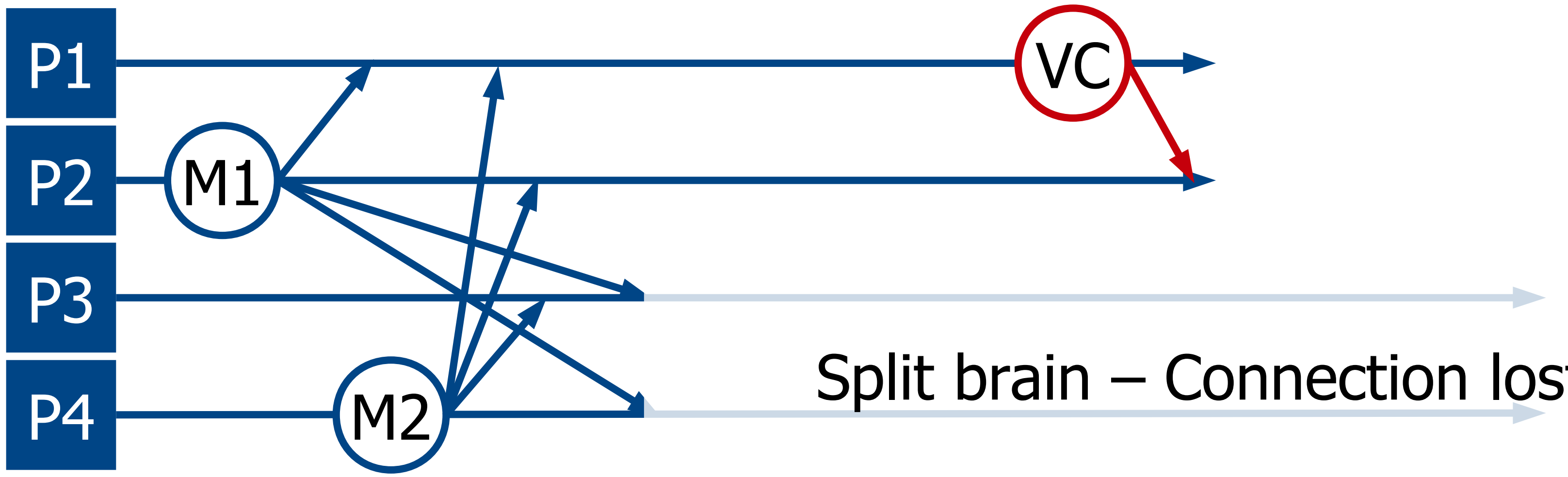
The speaker says…

Atomic broadcast means Virtual Synchrony used with total-order message ordering. When Virtual Synchrony was introduced back in the mid 80s, it was explicitly designed to allow other message orderings. For example, it should be able to support distributed applications that have a notion of finding messages that commute, and thus may be applied in an order different from the order sent to improve performace. If events are applied in different order on different processes, the system cannot be called synchronous any more – the inventors called it virtually synchronous.

How to cook brains

Wash the brain without marketing fluff, split brain, done!

- System dependent... E.g. Isis failure detector was very basic



n1({P1, P2, P3, P4]) = 4 n2({P1, P2}) = 2 < (n1/2)

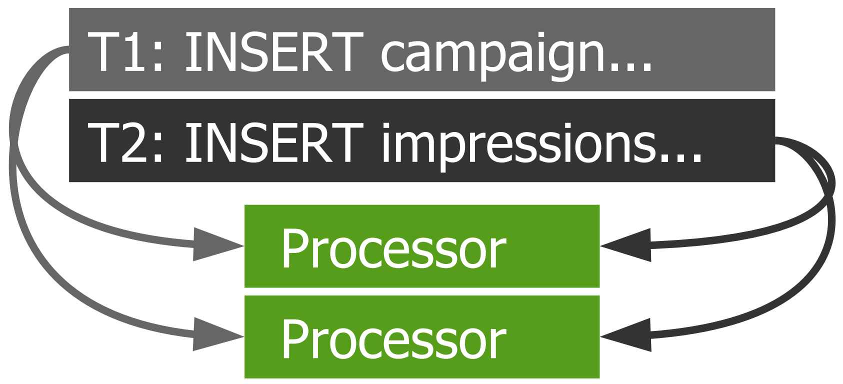
The speaker says…

The failure of individual processes – or database replicas – has been discussed. The model has measures to handle them using a fail stop approach.
To conclude the discussion of fault tolerance we look at a situation **called split brain: one half of the cluster lost connection to another half. Which shall survive? The answer is often implementation dependent.** For example, the early Virtual Synchrony framework Isis has a rule that a new group view can only be installed if it contains n / 2 + 1 members with n being the number of members in the current group. In the example both halves would shut down.

Overcoming the „D“ in CAP

CAP implies D(urability) from ACID

- Fast, consistent *local* reads desired
- C: Every processor has to get all updates
- C: Every processor must execute all of them in the same order
- C -> D: Hence, nobody may forget updates!
- D: We need durability in order to never forget...



The speaker says...

ACID and CAP are related – every NoSQL marketing whitepaper proves that :-) ! Assume, CAP lovers want fast local reads. Does this clash with ACID? Let's listen what the author of Isis and Isis^2 has to say about it.

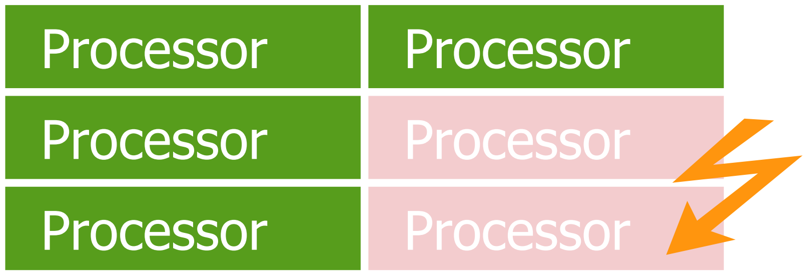
C in CAP requires that all processors get all the updates. As we saw, transactional updates must be ordered to achieve consistency among processors. Thus, the processors must get all updates in a defined order.

Processors must never forget an update. Updates must be durable.

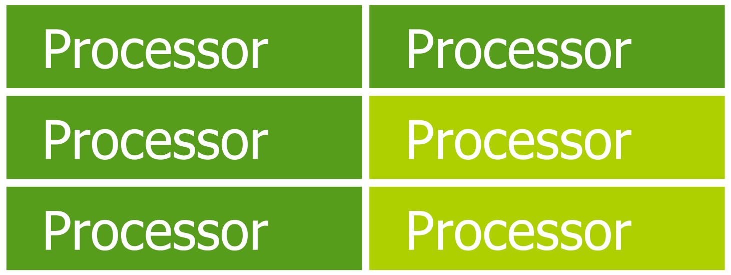
From D to A in ACID

CAP says we have an issue with D in ACID

- C -> D: Durability is needed not to forget updates
- Recall: For CP we need to use a quorum
- Recall: Read and write must overlap – read becomes slow!
- CP with quorum violates: Atomicy means „all or nothing"
- If nobody „forgot" updates, atomicy would not be violated
- Hence, the lack of durability limits atomicy



F = 2, N = 6, R + W > N



W = 4, R = 3

The speaker says…

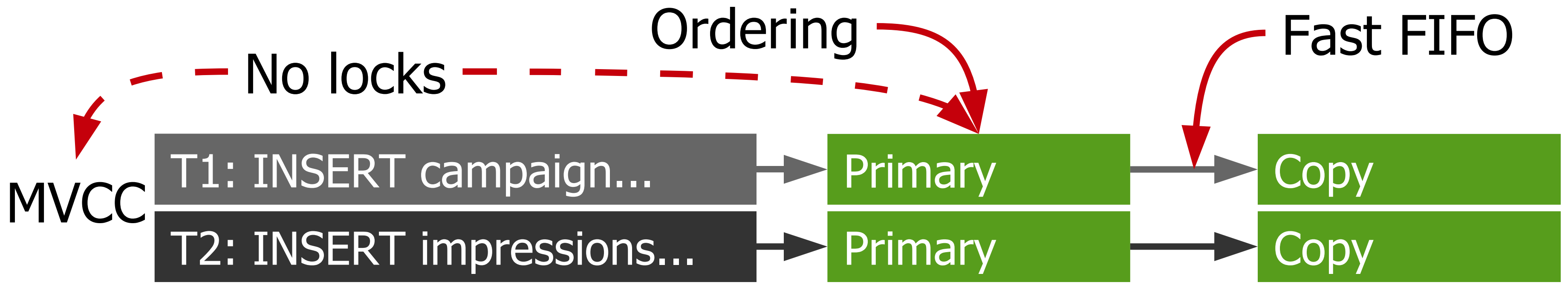
CAP says we cannot have consistency, availability and partition tolerance together. To be partition tolerant and consistent, we need to use a quorum (CP). A quorum makes things slow. We cannot use super-fast local asynchronous messages and read from just one processor, but we have to read from multiple processors instead! This is ultimately what availability in CAP is about. Eventually, there is a consistent reply, it is just slow.

In a CP system that is using a quorum, Atomicy in ACID is violated. Atomicy means „all or nothing“. Not all processors have the update. If the update was durable – if it was never „forgotten“, we could get Atomicy. Hence, D limits A.

C in CAP is a non-issue

Consistency requires ordering and slow locks?

- ACID: Durability limits atomicy, ACI no problem
- Consistency in CAP needs update ordering. Expensive?
- Use primary copy! Primary orders updates
- Use primary copy! Fast FIFO between primary and copy, no implicit impact on availability in CAP
- Does Consistency in CAP need locks for concurrent access?
- Use primary, use multi version concurrency control



The speaker says…

Consistency in CAP is not a show stopper for ACID per se.

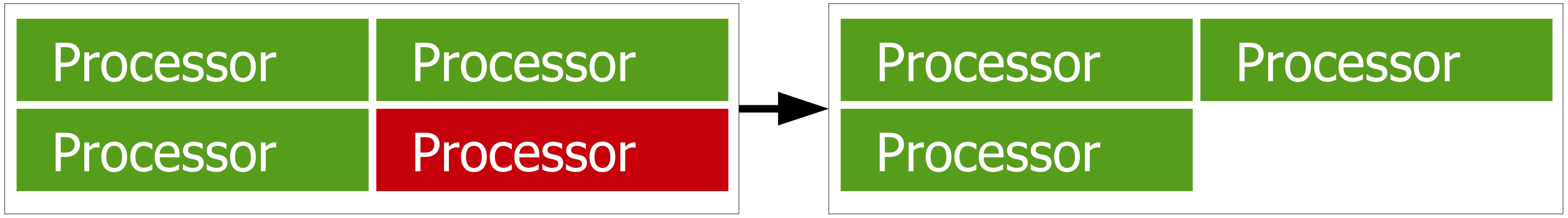
Consistency means we have to apply updates in a certain order on all processors. That is easily done using a primary copy scheme. All updates are routed to the primary. The primary sets the order and forwards the updates to the copies. This is fast as we only need FIFO messaging between the primary and copy (no negative impact on latency in availability). Do we need slow locks? No, first we have a primary. Second, we can use MVCC (no negative impact on latency in availability).

So, ACI are no problem at all. All we need is D!

Avoiding slow read quorums

Use virtual synchrony group abstraction

- Faulty processors are removed from the group
- The remaining group of N - F members is consistent!
- Consistent read possible from any group member



The speaker says…

We would not have to bother with slow read quorums, if we would remove faulty processors immediately. Faulty processors could not irritate us with inconsistent, outdated replies.

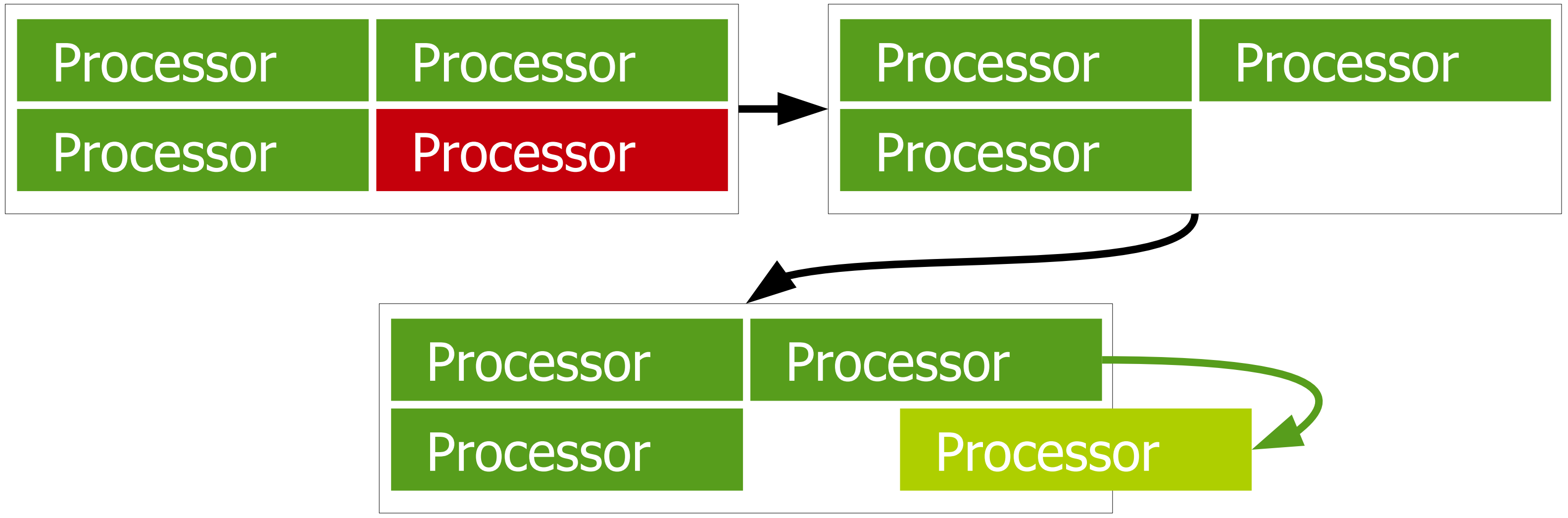
With only up-to date and consistent processors in the group, a read could be served from any group member. Reads would be fast: faster than quorum reads!

The virtual synchrony group abstraction removes faulty processors from a group, it makes us faster.

View changes

Use virtual synchrony group abstraction

- VC view change waits for majority to exclude faulty process
- Updates blocked during VC view change
- VC view change as expensive as BASE quorum write wait
- Processors can rejoin a group



The speaker says…

If virtual synchrony group members consider a processor faulty, they form a quorum to vote about excluding the processor. Any update is stalled during the view change. This is not an excessive wait. BASE does a similar wait all the time. BASE waits for a write quorum…

The FLP impossibility theorem tells us we cannot distinguish between slow and failed processors in an asynchronous network. Thus, in practice, we have to choose a timeout after which we consider a processor failed (the timeout makes the asynchronous network synchronous). This timeout determines for how long VC stalls updates. As we wait for the timeout, updates are buffered and not applied. Note: we are not inconsistent, we wait in consistent state.

Durability costs are negliable

The cost of durability depends on the network protocol

- Updates must not be lost, they must be durable
- Forces a GCS to use Paxos over unreliable network protocols
- Early Isis^2 results for 800 processors, 78 nodes, 1GBit ethernet: 25ms with peaks to 100 ms

- Would map to 800 tablet / 5 replicas = 160 spanserver respectively 800 tablets / 3 replicas ~ 266 spanserver on Spanner. Google reported 2PC latency of 70ms with peak 150ms for 100 spanserver (=participants) and 150ms with peak 350ms for 200 spanserver (= participants)
- Recall: 2PC requires many messages, not just one

The speaker says...

CAP does not say one cannot build a consistent, available (= fast) and partition tolerant ACID compliant data store! Clever engineering can get you very far if you stay within one data center or we talk data centers nearby each other. Both Spanner and the Isis2 group communication library point this way.

Do not give up on consistency before you tried!

Relaxing consistency

Relaxing the C in CAP

- ICMP and tricks (e.g. gossiping) is very fast but unreliable
- Isis^2 unreliable send: 2ms latency (800 processors, 78 nodes)
- Soft-state can accept weak consistency
- Hard-state requires strong consistency
- Time constrained weak consistency
- Epsilon/deviation constrained weak consistency

The speaker says…

In many cases, consistency can and must be relaxed. The famous example is the ATM. You can disposit money although the ATM is disconnected from the bank and your balance is unknown. But, only you have a limit. Weak consistency is allowed but limited by a deviation/epsilon – search for such cases.

Assume you can accept that only 99,999% of the ATMs show your exact balance and two ATMs may show different values for one minute before they are consistent again. This allows a distributed system to use super-fast, unreliable ICMP for one minute. Then a leading primary sends all processors the correct value using method that is a magnitude slower but 100% reliable.

How fast can ACID be?

Is not knowing the success of a transaction OK?

- Try asynchronous API
- Client sends update
- Client does not block until update is durable/delivered
- Client continues not knowing the outcome
- Server acknowledges update to client, or server crashes

- … guess how many transactions a 30 node MySQL Cluster can achieve this way

The speaker says...

There may also be cases when a client can send updates and does not have to wait for a confirmation from everybody that the update has been received. Note that the client does not know yet whether its transaction has succeeded. The client must be prepared to find out, some time later, that it failed.

Guess how fast MySQL Cluster gets if you use an asynchronous client? 1.17BN writes/min resp. 4.3 BN reads/min on a 30 node cluster.

Famous words…

[…] In the decase since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. **The NoSQL movement also has applied as an argument against traditional databases** […] The „2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. **CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.** […] **Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order.** Thus strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition. [...]

The speaker says…

From „CAP Twelve Years Later: How the 'Rules' Have Changed“, Eric Brewer (May 2012)

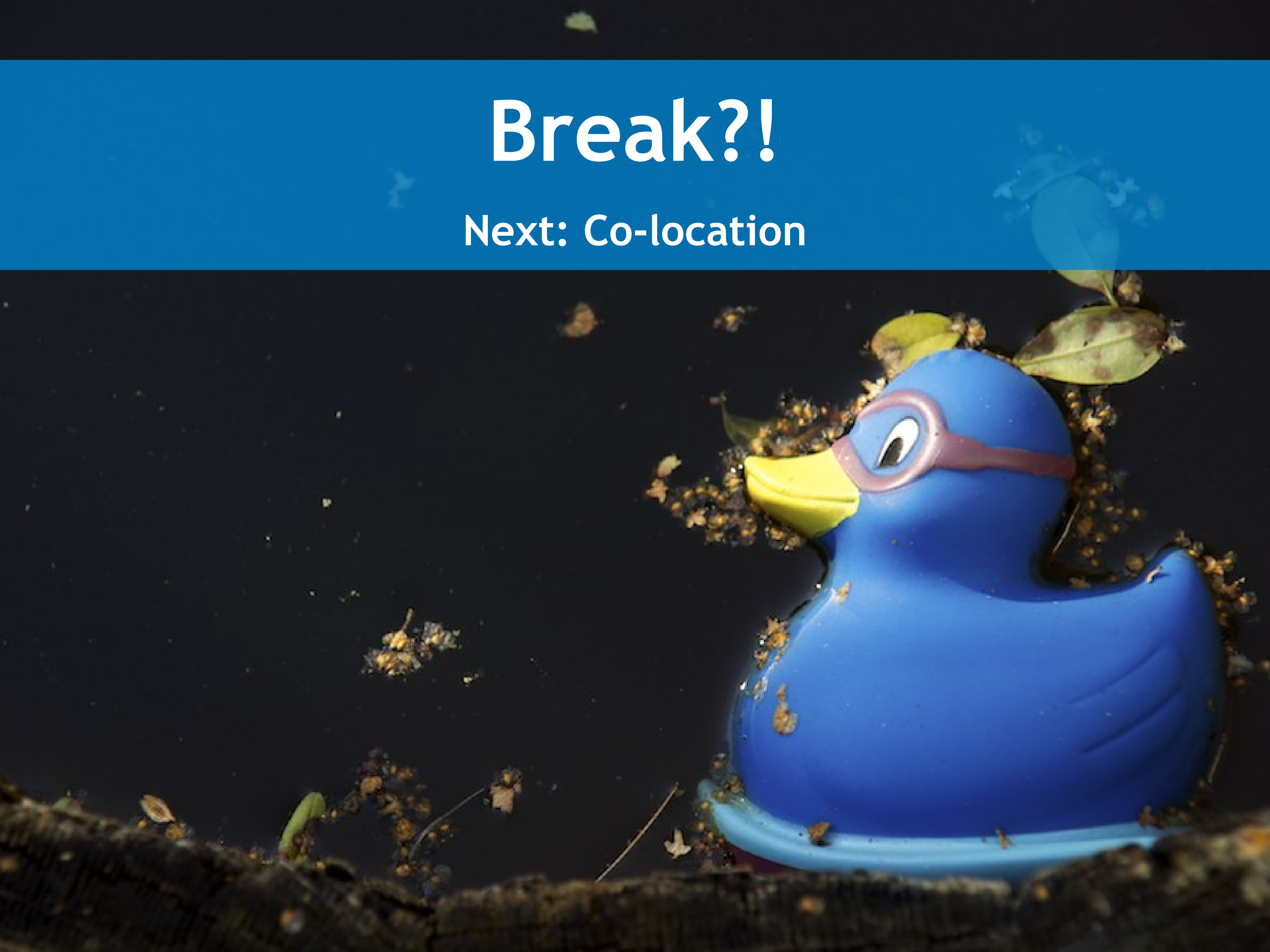
CAP is true – for a small part of the design space. In practice partitions are rare. If they happen, they prohibit consistency. During a partitions, systems shall have a defined behaviour. Ideally, they remain operational. Likely with limited capabilities. Once the partitioning is over, consistency shall be regained.

In practice, would you be fine with an ACID system that restricts a minority partition to reads and tells you it is eventual consistent? Would you be willing re-route a client?

NEVER FOOL MERLIN!

Take your cap off to greet him!

This is Merlin. He is soooo cute.
But, Merlin, next time you visit us...
Please, eat all the lentils you put in
the very corner of the cupboard.
The vacuum cleaner cannot get them.



Break?!

Next: Co-location

The speaker says…

Congratulations: you made it. No more theory...

So we can scale?

Distributed Transactions do not scale in the Cloud?

- Latency but not throughput is an issue
- Assuming Ethernet only – SCI/Infiniband is out
- Assuming law of physics does not apply: within data center, among nearby data centers

Strong consistency? Paxos latencies reported recently

- Spanner: 4K write ~ 14ms @ 3 data centers
- Gaios: 4K write: <10 ms with >9ms for disk logging
- Spinnaker: Write just 5-10% slower than Cassandra quorum
- (Isis^2: ~10 ms with 3 firm ACKs, 78 nodes, 800 participants)

The speaker says...

We have explored the costs of distributed transactions and explored different takes on the C in CAP. Strong consistency has a price, distributed transactions have one. As Google's 2PC latencies show it is a significant one.

BUT: on Ethernet, within a data center or among some nearby data centers we could use atomic broadcast for strong consistent replication. The Paxos research system Spinnaker was just a tad slower than Cassandra (Dynamo follower) for strong consistency writes than Cassandra's eventual consistent write. Birman's combination of VC + Paxos points into the same direction. If operational times >= Paxos latencies, this could be it! In other words: if your average SQL query takes longer than 10 to 15ms.

Co-location is imperative

Partitioning is a must

- Data size
- ROWA(A) scalability limit

When co-location happens

- Static: based on schema information
- Dynamic: adaptive to actual query load

The speaker says…

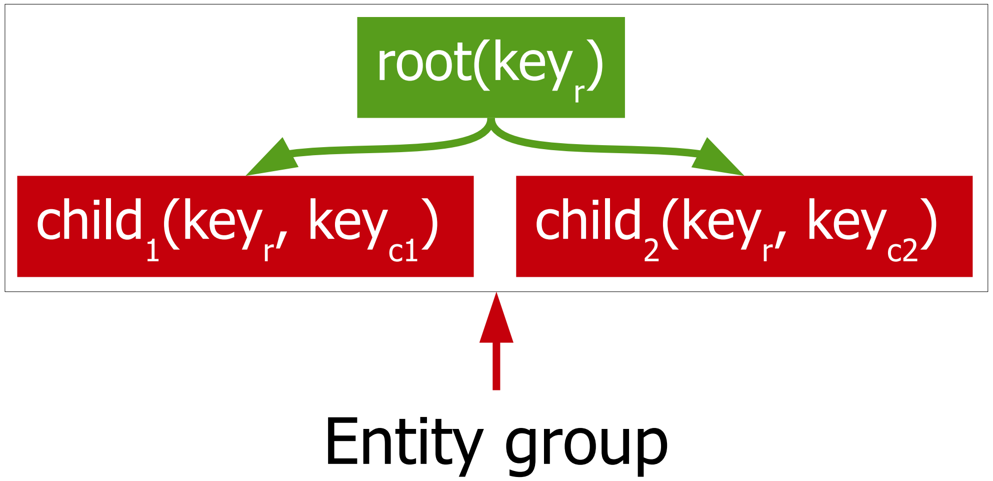
Judging from Paxos made code, Paxos average latency for small groups should be ~1ms but they must persist their state when running, which slows them down. Even in the presence of fast Paxos, ROWA(A) could evolve into a natural scalability limit.

Hence, eventually we must partition our data. Once we partition it, we are back to the problem of distributed transations – this time at a larger scale. This leads us to: we must think about co-location of data in the presence of transactions to minimize distributed transactions.

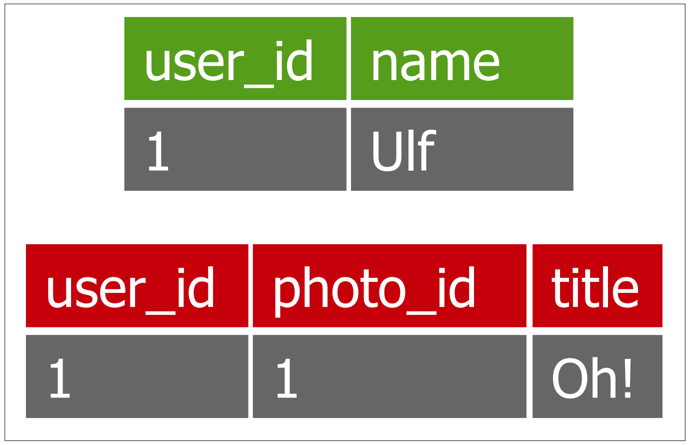
Static: Entity Groups

Hierarchical model

- Google Megastore, physical co-location
- Entity maps to a single row in a relation table
- Re-paritioning: sorted storage minimizes data movement



$\text{root}(\text{key}_r)$
$\text{child}_1(\text{key}_r, \text{key}_{c1})$
$\text{child}_2(\text{key}_r, \text{key}_{c2})$
Entity group



user_id	name
1	Ulf

user_id	photo_id	title
1	1	Oh!

The speaker says…

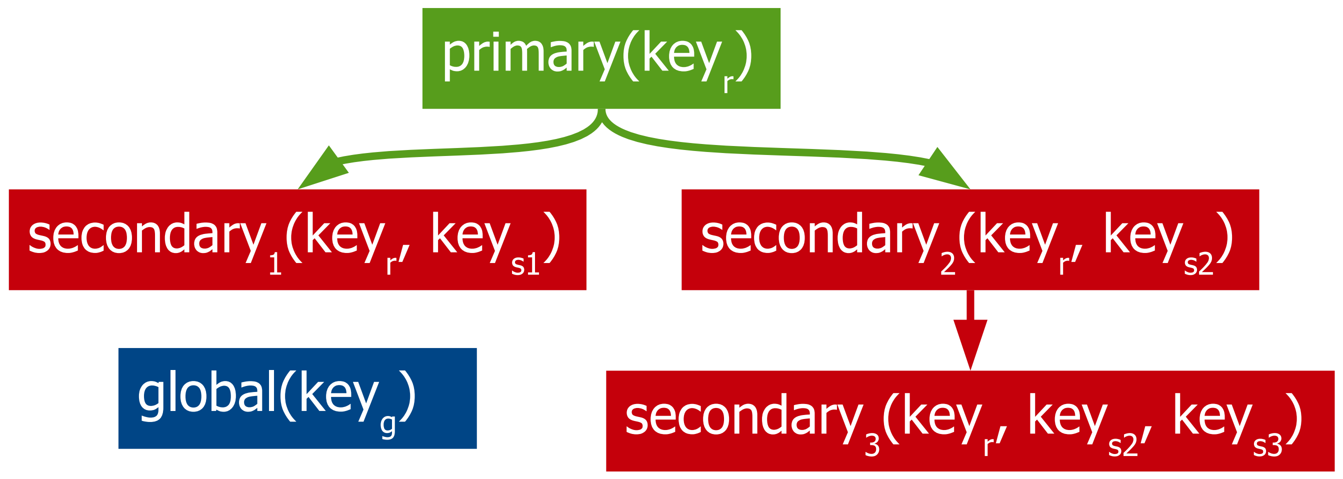
Bigtable pointed the way data is co-located in Megastore using explicit schema information. Like Spanner, Megastore lets users define tables (entities) with named and typed values (columns). A set of columns together forms the primary key. A hierarchical relationship can be formulated between entities to makes row interleave on the physical storage: a user record is followed by all photos of the user. Quite a good pattern for email accounts, blogs and the like.

Entity groups are a good granule for partitioning. Partitioning becomes particularily easy if data is stored in key order: split and merge become cheap if data is to be moved.

Static: Tree schema

Hierarchical model

- ElasTraS, H-Store


primary(key_r)
secondary$_1$(key_r, key_{s1})
secondary$_2$(key_r, key_{s2})
global(key_g)
secondary$_3$(key_r, key_{s2}, key_{s3})

The speaker says…

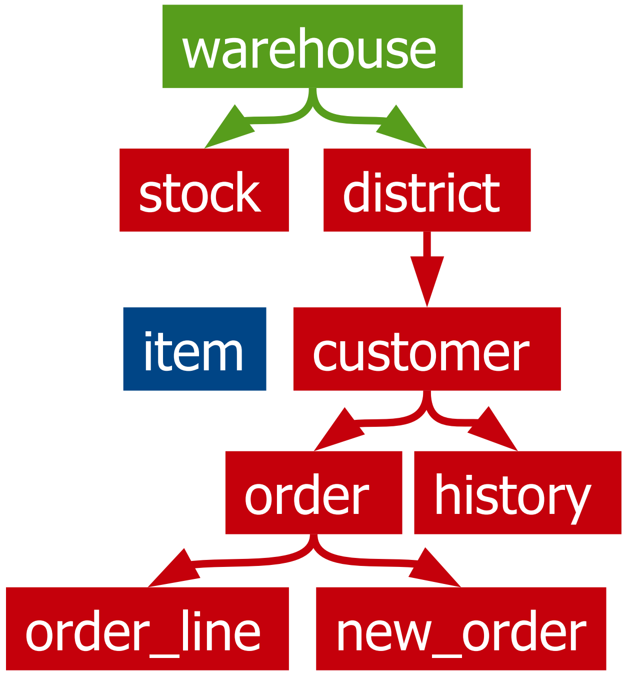
A standard tree schema leverages foreign key relations to identify data that should be co-located in a partition. There is a primary table and a set of secondary tables which contain the primary key of the primary as a foreign key. The depth of the tree is not limited.

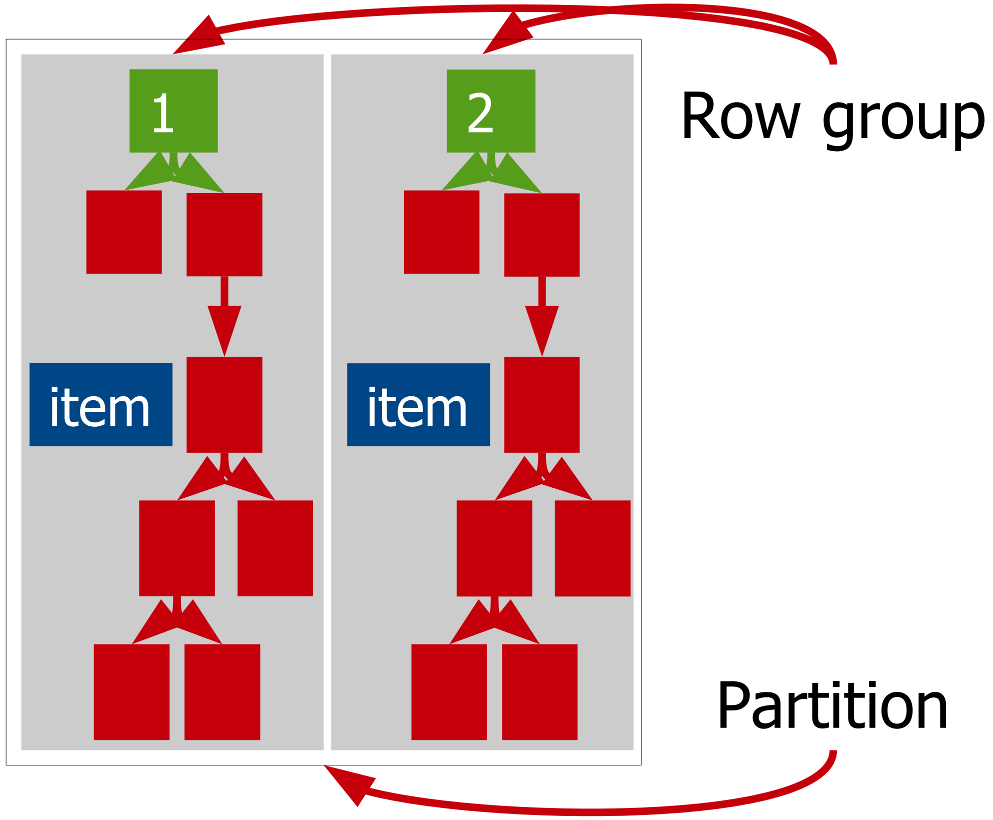
Tables related to each other this way are accomplished by global tables. Global tables are assumed to be used as lookup tables mostly. Their data changes rarely. Replicating global tables in all partitions is inexpensive.

Static: Tree schema

Example TPC-C benchmark

- 85% of the queries are on one warehouse





The speaker says…

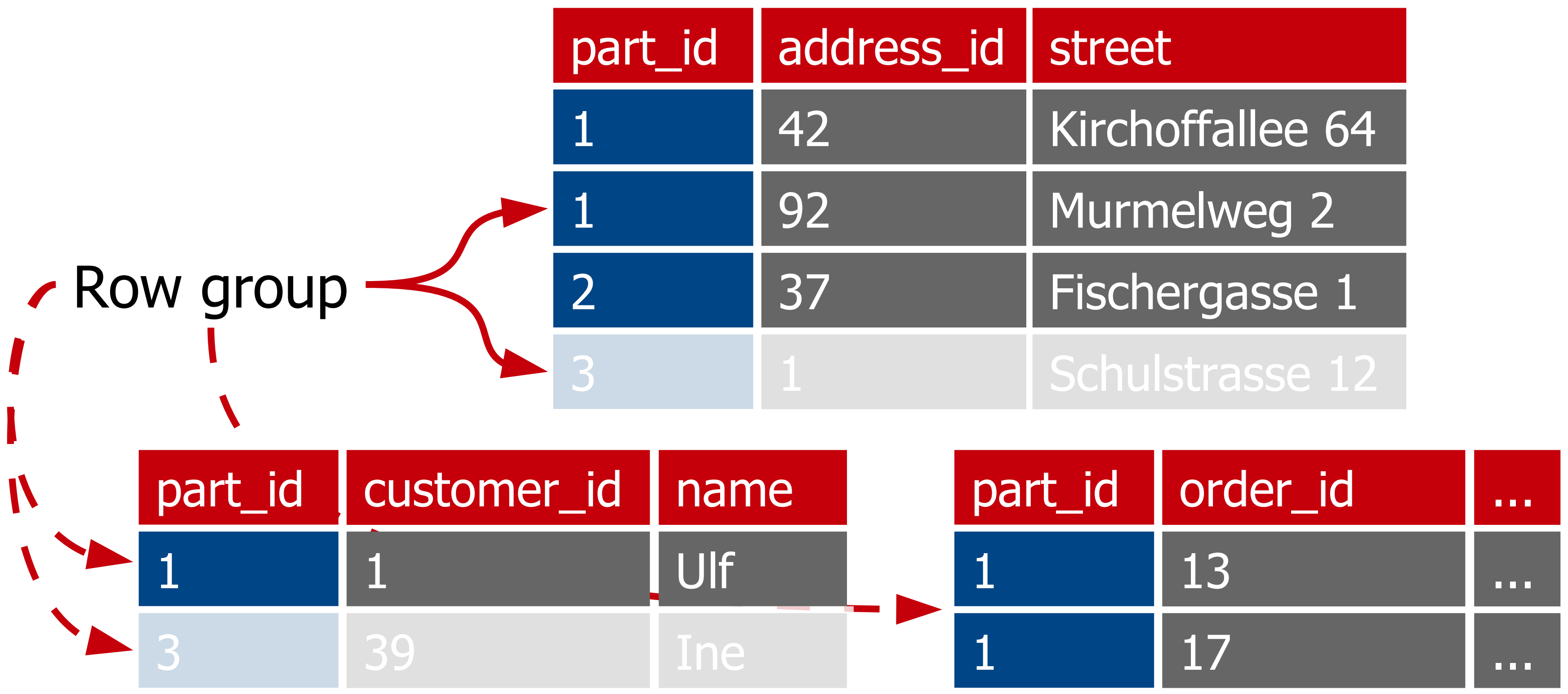
The slide shows an example of the TPC-C standard benchmark modeled as a tree. Some 85% of the benchmarks queries cover no more than one warehouse. Therefore, the primary key of the warehouse is a good partitioning key.

All rows that belong to a primary key in the primary/root table are called a row group. A partition may hold one or more row groups.

Static: Keyed table group

Table group

- Cloud SQL Server (Microsoft Azure)



part_id	address_id	street
1	42	Kirchoffallee 64
1	92	Murmelweg 2
2	37	Fischergasse 1
3	1	Schulstrasse 12

part_id	customer_id	name
1	1	Ulf
3	39	Ine

part_id	order_id	...
1	13	...
1	17	...

The speaker says…

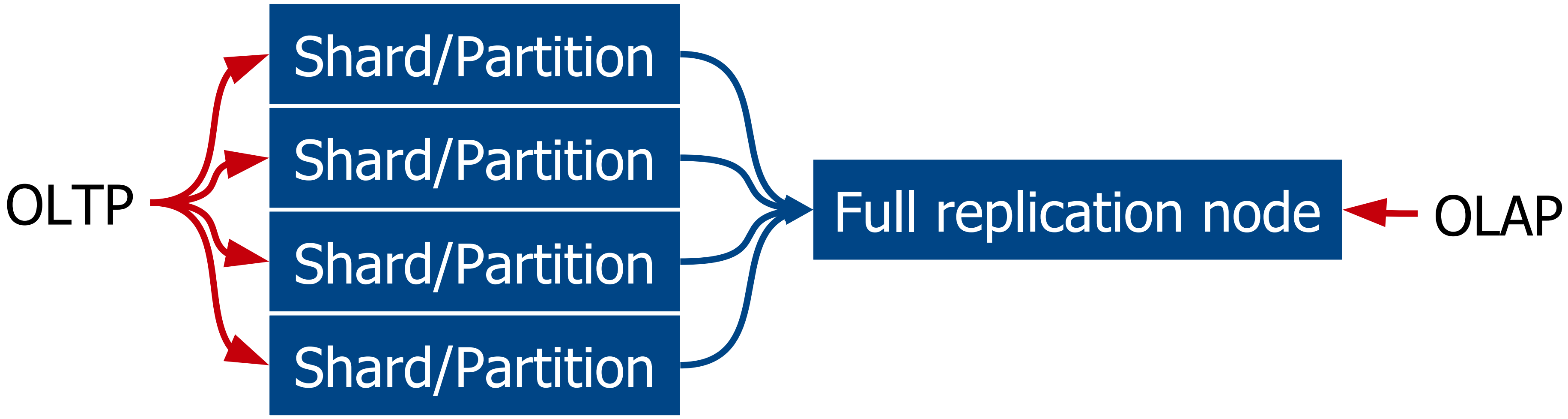
The keyed table group is a generalization that allows co-locating arbitrary tables. The tables to be kept together in a partition share the same partition key. The partition key is a column in the tables. It does not need to be a primary key nor does it need to be a foreign key. However, they could be primary or secondary keys. Row groups exist, just as explained in the previous slide.

This model is used by Cloud SQL, the parallel and distributed SQL Server that runs Microsoft Azure. BTW and AFAIK, Microsoft Azure is uing Infiniband for high-speed interconnects.

Static: full replication nodes

Recomposing split data

- Shards handle the myriad of OLTP clients
- Full replication node build from shard for OLAP/joins



The speaker says…

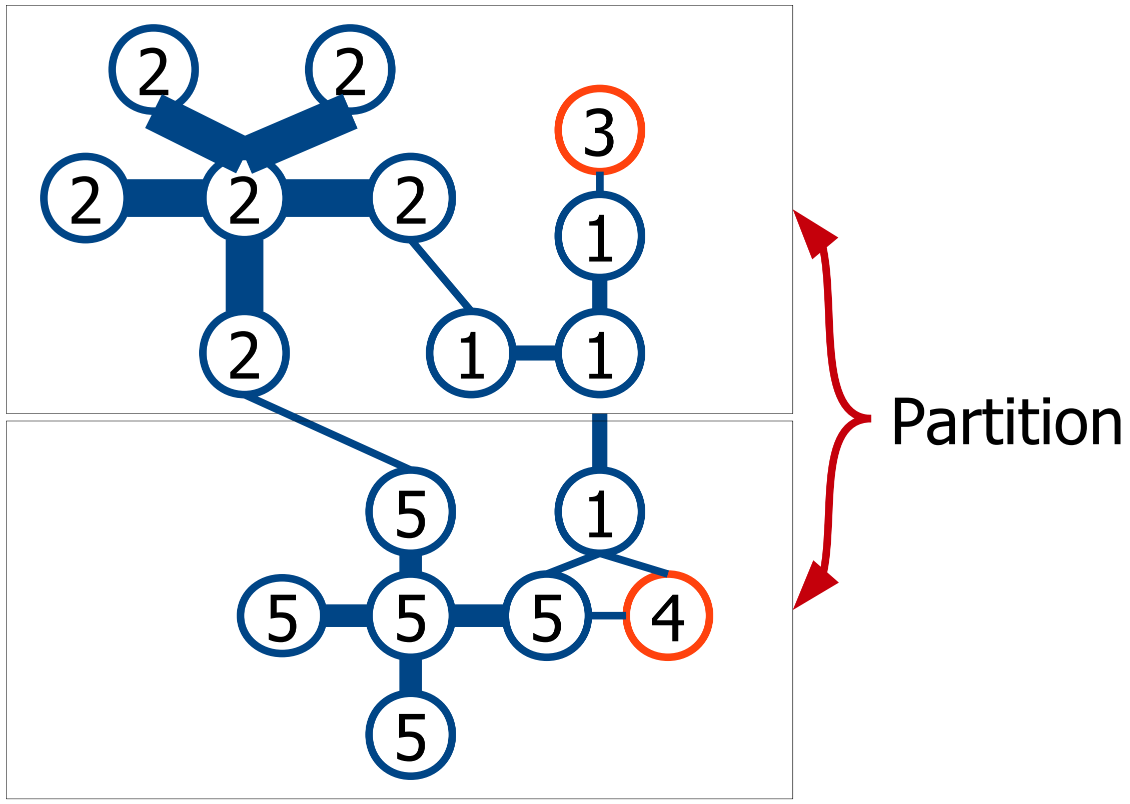
If data sizes are not prohibitively large, co-location can be done at a much larger granule. Full replication nodes can be introduced that replicate from multiple partial replication nodes. Queries that span over multiple partitions can be efficiently executed on the full replication nodes. In a way, this is similar to a materialized view.

BTW, the new MySQL multi-source replication should be able to support this pattern.

Dynamic: access-driven

Application's data access modeled as a graph

- Use well-known graph partitioning
- Results in very complex routing



The speaker says…

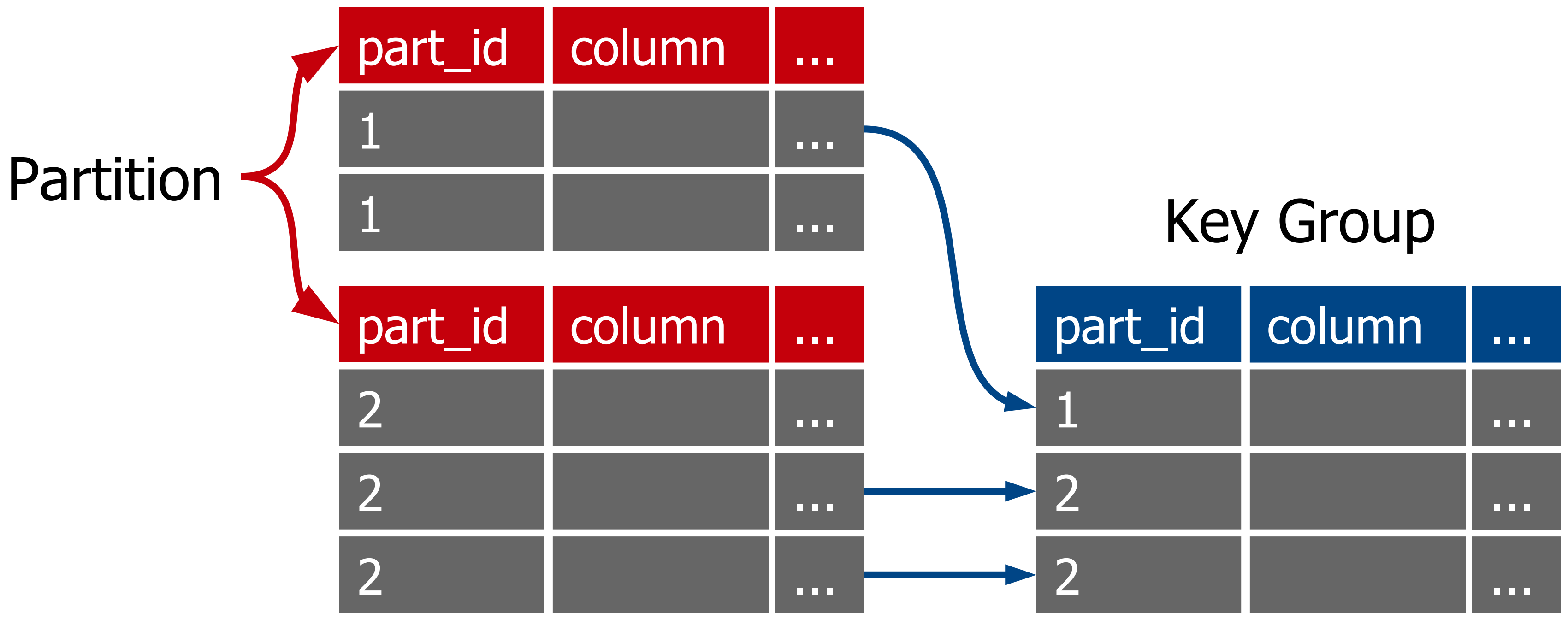
Curino et. al. made an (academic) proposal to model the accesses of an application as a graph. Then the graph is partitioned so that reads profit from replicating a frequently accessed row multiple times but updates require as few distributed transactions as possible.

This may lead to very good partitioning but complicates client routing. There is no simple lookup function a client can use to find a server that holds a partition. The same is true for any routing server a client could consult to find the appropriate partitions. Multiple proposals to solve this exist, none is easy.

Dynamic: application driven

Key Group abstraction

- Application requests group to be co-located on node
- Might not pay off if group initialization is costy



The speaker says…

Assume you have an online games application. Players can choose between many games. Usually, once they picked a game, they play it for hours. As long as the game is being played the player's data and the game's data should be co-located on one node. Thus, the application asks to create a group that holds the required data. From now on, consistent manipulation of the data in the group is only guaranteed within the scope of the group. When the player ends its game, the application releases the group.

Having pre-partioned data, as in the slide, is not precondition for this pattern. The protocol details are similar to those we explored for distributed transaction and replication.

DONE!

Next: Selected News

Moral? Looks like an ACID systems could scale full replication to 5..50 synchronous nodes on plain Ethernet.
Consistency, availability and partition tolerance would be given when taking a practical perspective.

Beyond that, partial replication (partitioning) must be used. We've seen some proposals how to optimize a relation schema for it.

Raft

RAFT - In Search of an Understandable Consensus Algorithm

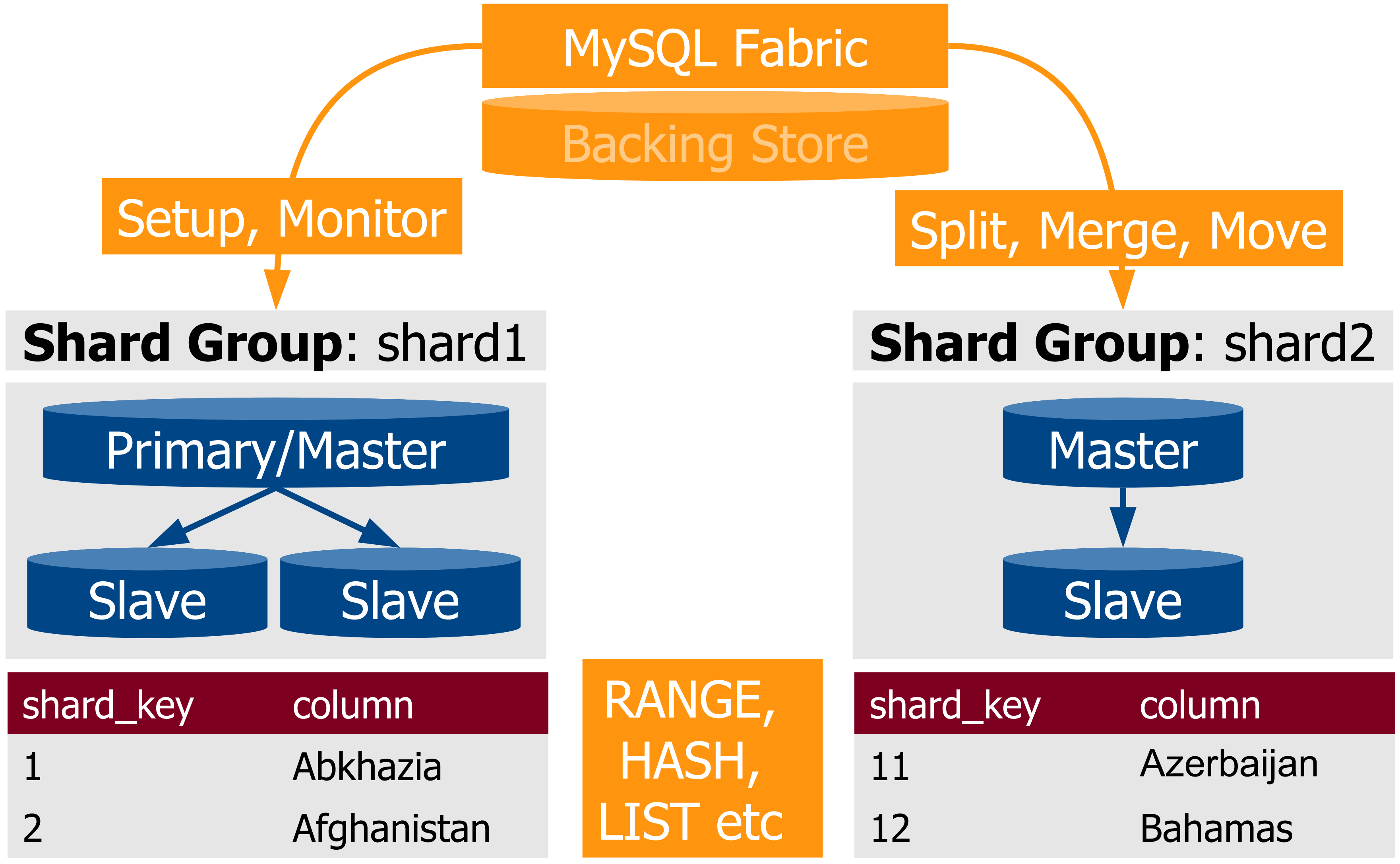
- The paper to read instead of the classic „Paxos made simple“
- They claim most of their readers found it simple, really

No more monolithic kernels

Decoupling of modules in the database kernel

- Paper: Unbundling Transaction Services in the Cloud
- Paper: MoSQL: An Elastic Storage Engine for MySQL
- New kid on the block, acting agressive on MySQL: NuoDB

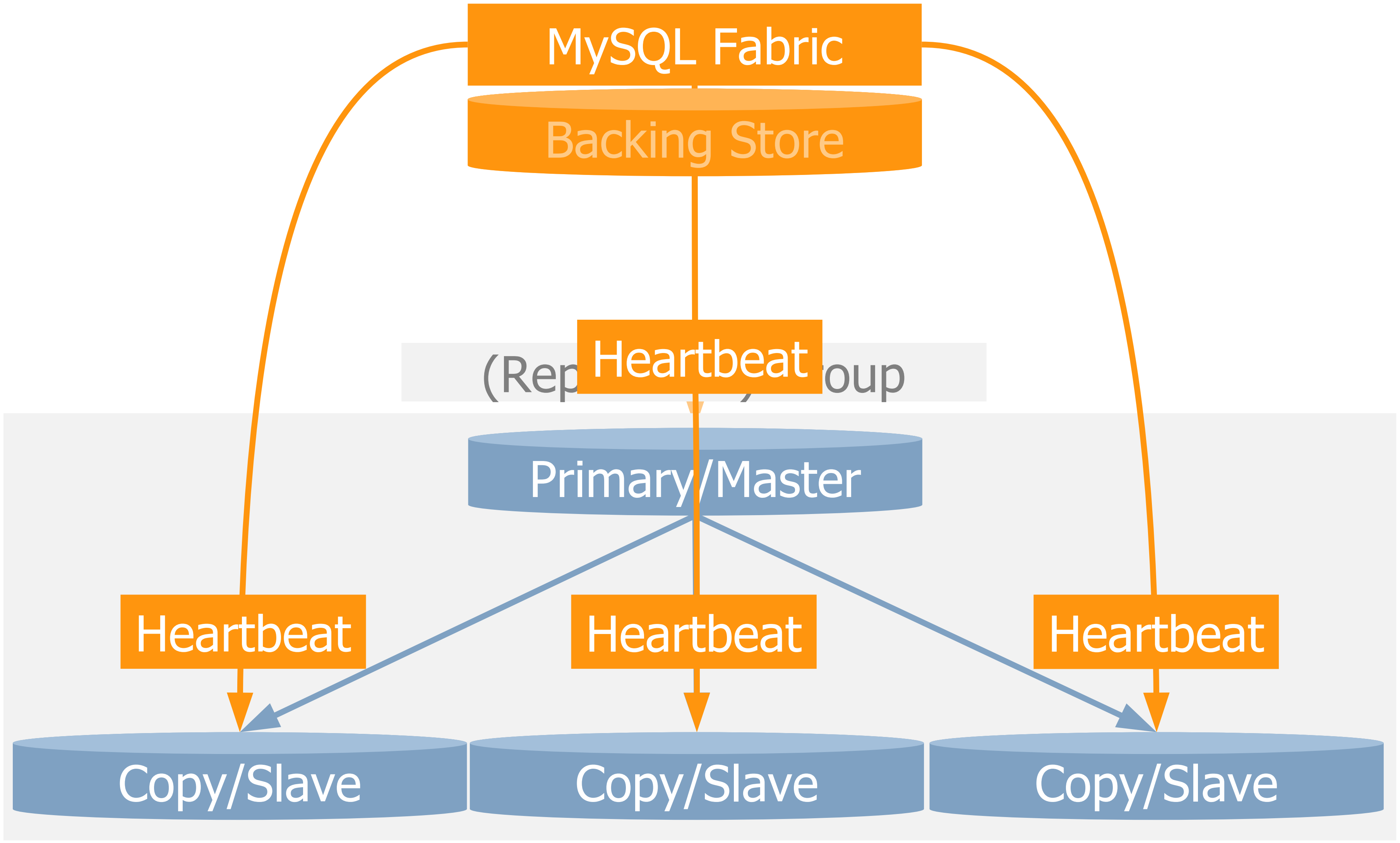
BTW, MySQL 5.7: Sharding



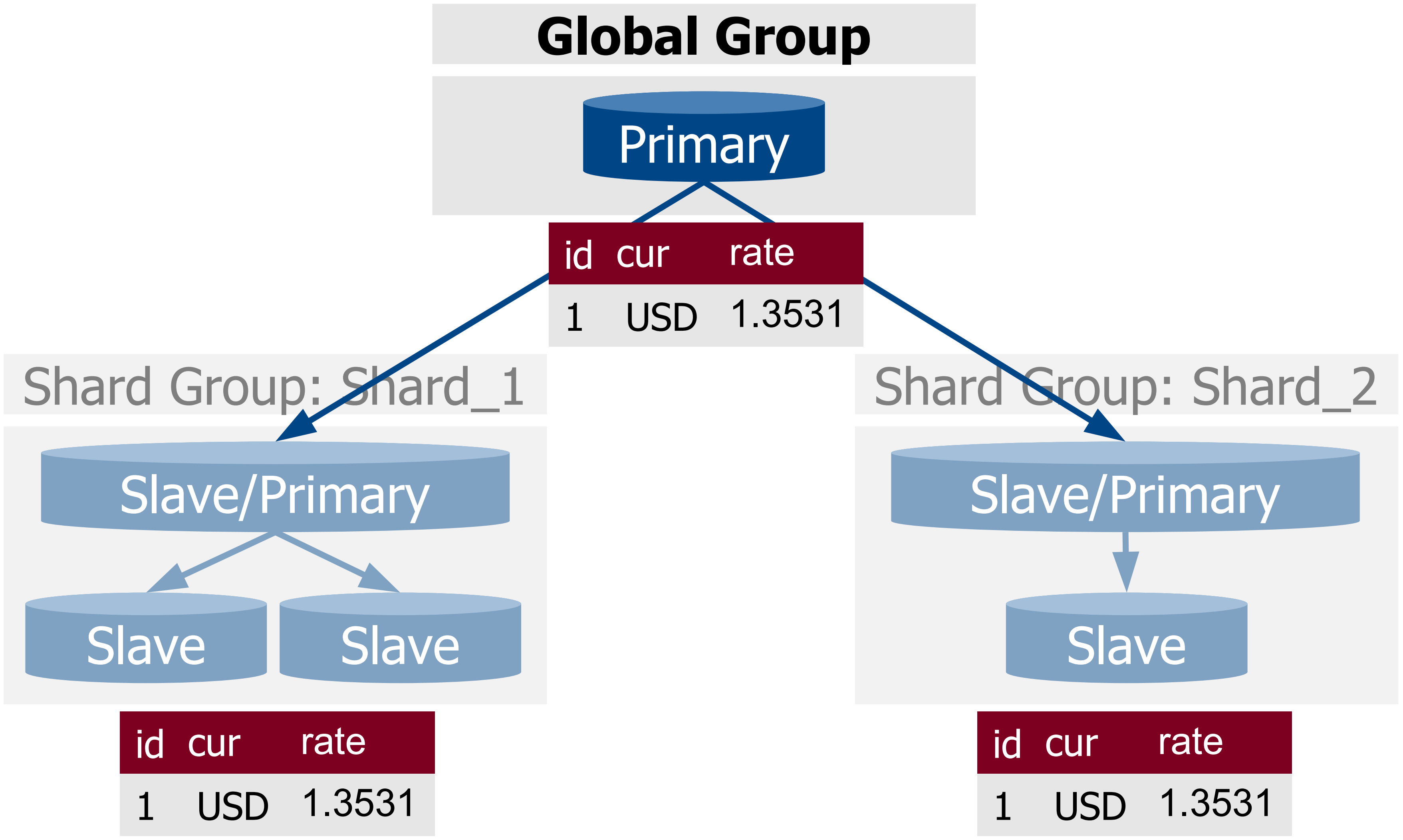
shard_key	column
1	Abkhazia
2	Afghanistan

shard_key	column
11	Azerbaijan
12	Bahamas

Fabric: Availability



Fabric: Co-location/model

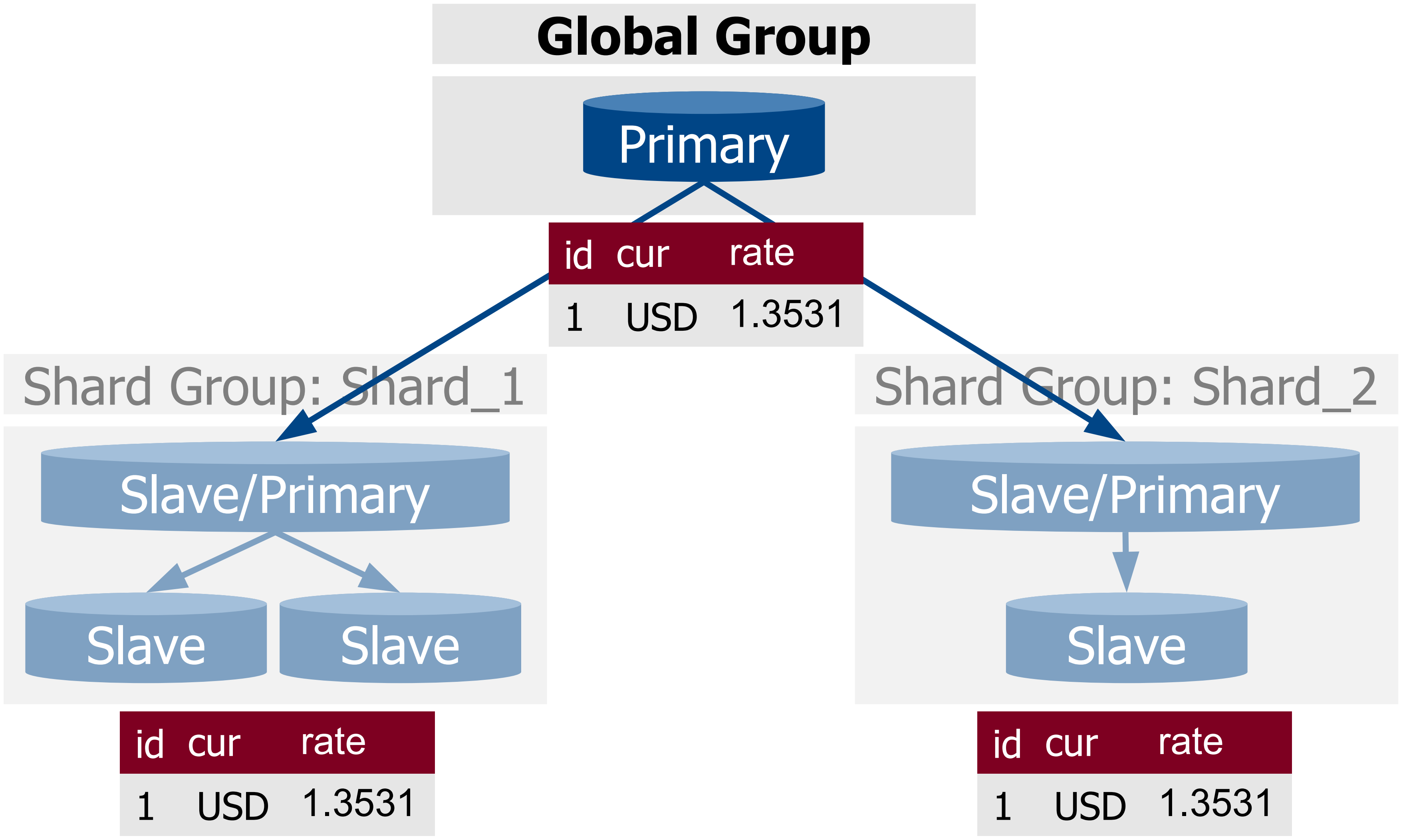


id	cur	rate
1	USD	1.3531

id	cur	rate
1	USD	1.3531

id	cur	rate
1	USD	1.3531

Fabric: Co-location/model

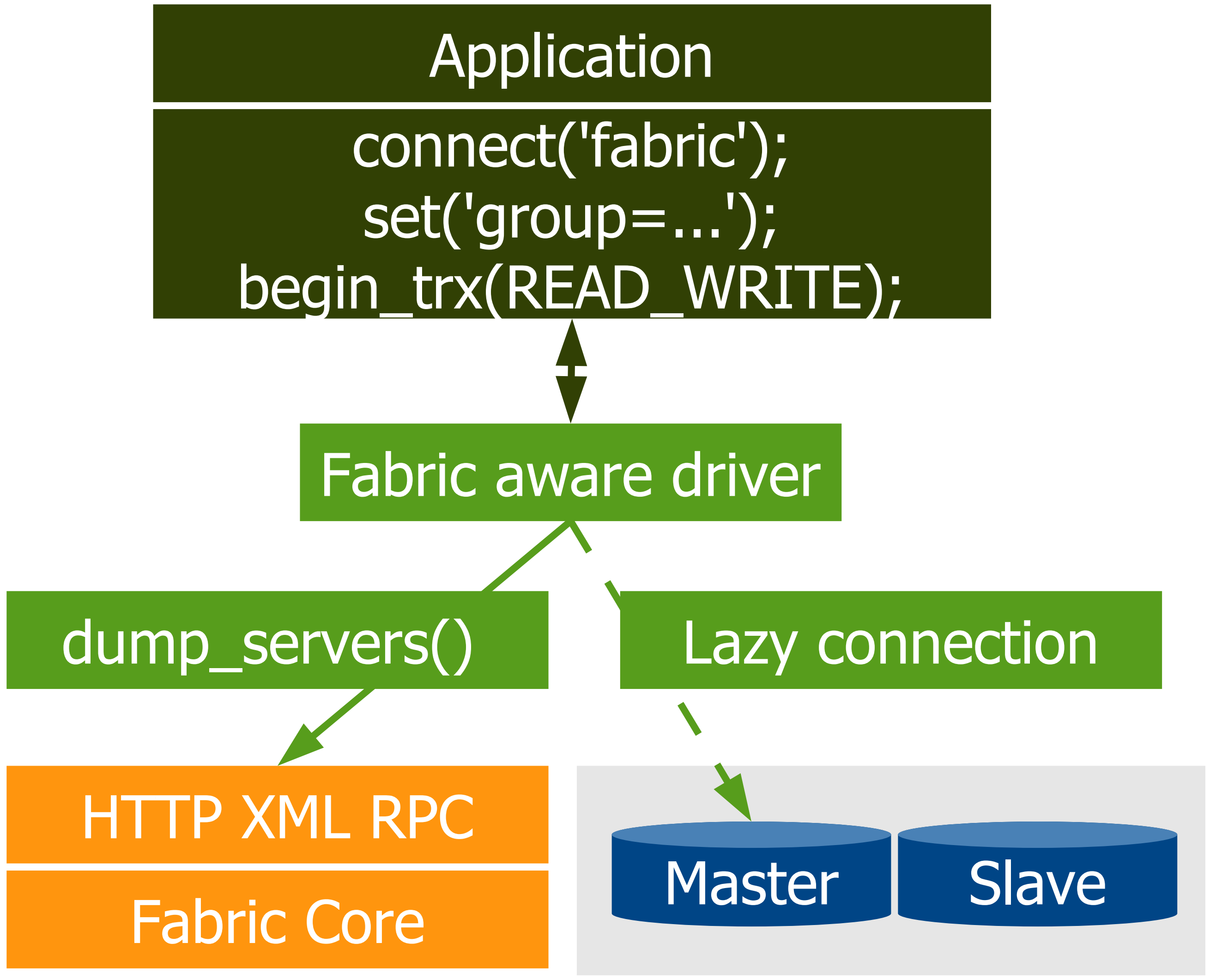


id	cur	rate
1	USD	1.3531

id	cur	rate
1	USD	1.3531

id	cur	rate
1	USD	1.3531

Fabric: Client view



FABRIC - AT THE END ???

Yeah, it is a preview only. See my other presentation.

THE END

Contact: ulf.wendel@oracle.com

The speaker says…

Thank you for your attendance!

Upcoming shows:

Talk&Show! - YourPlace, any time...

.

… seriously, if you like any of the presentations,
I can repeat them in your company, given it's nearby.