COMP5349 – Cloud Computing

Week 11: Consistency and Cloud Computing

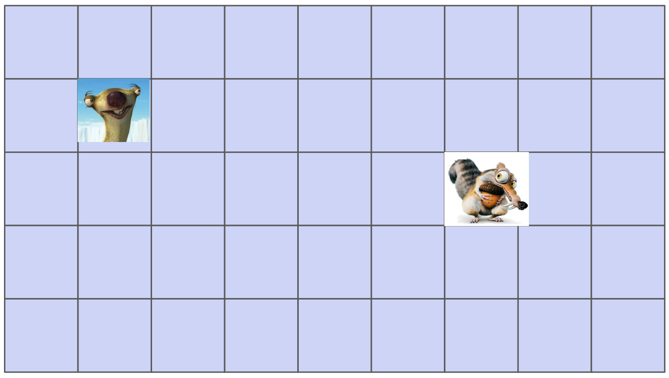
A/Prof Uwe Röhm
School of Information Technologies

Outline

- **Distributed Data Management**
  - **Data Partitioning**
  - **Data Replication**
  - **Distributed Transactions**
  - **2-Phase Commit Protocol**
  - **Paxos Consensus Algorithm**
- Notions of Consistency
  - CAP Theorem
  - Eventual Consistency

Let us Program a Game

- My game idea: ‘Ice Age Farmland’



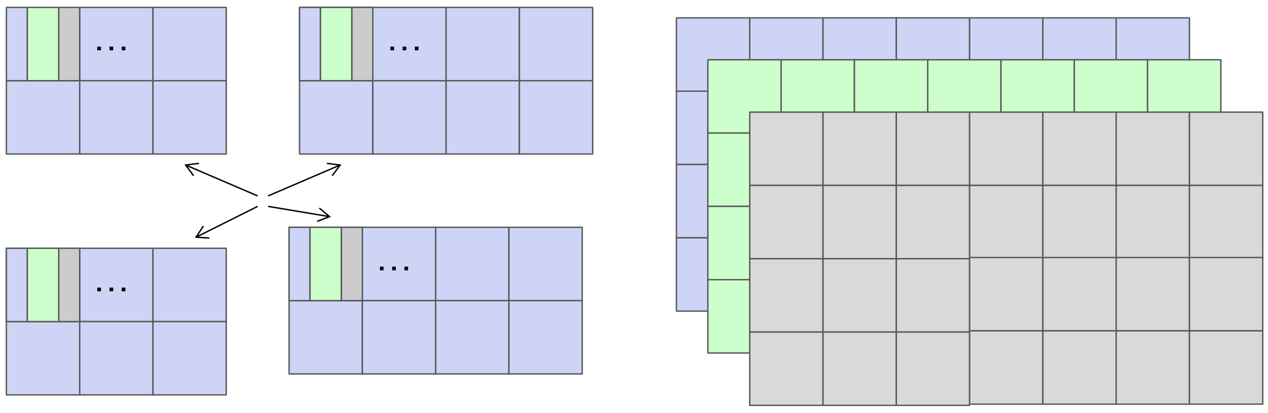
- Players can buy fields of the game area and start ‘farming’ it with ice-age characters
  - The game area is large, say 1 mio x 1 mio fields
  - We hope for millions of subscribing players, playing simultaneously
  - Part of the fun is to see how others develop their ice lands…
  - But who owns which field, and what is on it?

How to achieve this?

- Scalability
  - Want to support growth
    - We might start with just a couple of players, where a single node might be fine, but will it also be fine for thousands of concurrent players?
  - Idea: make problem small enough for single machine to handle, then use multiple machines (sites)
    - **Data Partitioning** or **Sharding**
- Availability / Reliability
  - the larger the system, the higher the probability of failures
    => **Data Replication**
  - Consistency:
    Are distributed transactions with ACID guarantees the solution?

Distributed Data Management: Data Partitioning

- What is partitioned?
  - Horizontal Partitioning vs. Vertical Partitioning



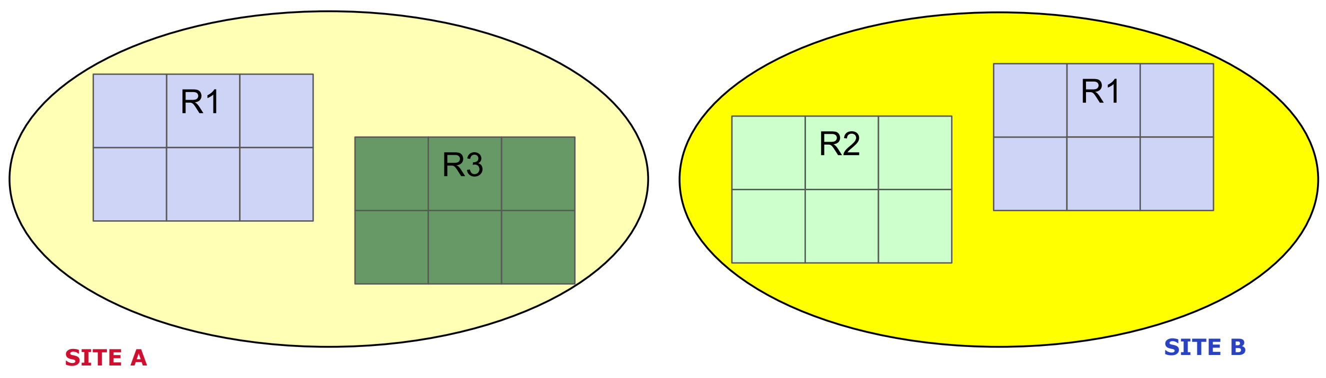
- How do we partition?
  - Round robin vs. Range partitioning vs. Hash partitioning

Data Sharding

- Traditionally, data partitioning (vertically and horizontally) is defined in the context of *parallel database systems*
  - This means: within a single schema / site
- For *distributed database systems*: need to fragment (shard) data among several sites
- **Data Sharing**: Distributing data partitions over several nodes
  - Shard = Data Partition
    - Some textbook refer to this as horizontal or vertical *data fragmentation*
  - All the reasons from above, but in particular to help with scalability and parallel query processing
    - assumes queries need access to a single shard only!
    - Otherwise can be counter-productive, e.g. if we need to join two partitioned tables which are no *co-located* per site…
      - Co-located: join tuples are on the same node, e.g. by partitioning both tables on the join attributes

Distributed Data Management: Data Replication

- **Replication**
  - when the value of some data item is stored in more than one place
    - Similar issues arise with caches
  - Gives increased availability
  - Faster query evaluation (parallel query execution on different replica)



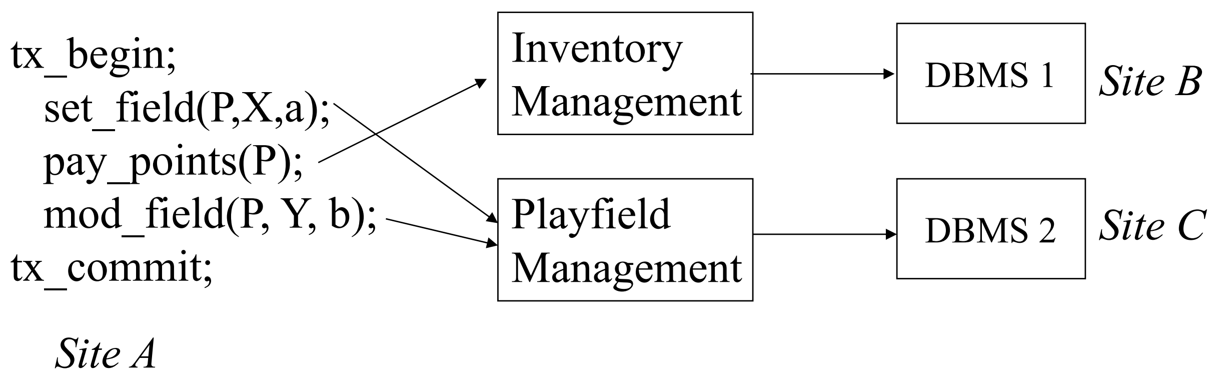
How to Keep Replicas Consistent?

	Update Location	
Update Propagation	Eager Primary Copy	Eager Update Anywhere
	Lazy Primary Copy	Lazy Update Anywhere

- “Always” consistent
  - At least, apps shouldn’t observe difference from using single dbms
    - “transparent replication”
      - Formal definition for “1 copy serializable (abbreviated as 1-SR)”
  - Some systems propose “1-copy SI”
- Eventually consistent
  - “convergent”
  - If updates cease for long enough, all copies will reach a common value
- Intermediate approach: limited divergence

Distributed Transactions

- One approach to always consistent: **distributed transactions**
- Incorporate transactions at multiple servers into a single (distributed) transaction
  - Not all distributed applications are legacy systems; some are built from scratch as distributed systems



Distributed Database Systems

- Operations at each site are grouped together as a subtransaction and the site is referred to as a ***participant*** of the distributed transaction
  - Each subtransaction is treated as a transaction at its site
- ***Coordinator*** module (typically part of TP monitor) supports ACID properties of distributed transaction
  - Transaction manager acts as coordinator

Correctness of Distributed Transactions

- **Goal**: distributed transaction should be ACID
- Each *local* DBMS
  - supports ACID properties locally for each subtransaction
    - Just like any other transaction that executes there
  - eliminates local deadlocks
- In addition the transaction should be *globally* ACID
  - **A**: Either *all* subtransactions commit or all abort
    - two-phase commit protocol
  - **C**: *Global* integrity constraints are maintained
  - **I**: Concurrent distributed transactions are *globally* serializable
    - Even if local sites are serializable, subtransactions of two distributed transactions might be serialized in different orders at different sites:
      At site A, T_{1A} is serialized before T_{2A}
      At site B, T_{2B} is serialized before T_{1B}
  - **D**: Each subtransaction is durable

Global Isolation

- **Problem:** local serializability at each site does not guarantee global serializability
  - *Shipping* subtransaction is serialized after *audit* subtransaction in DBMS at warehouse 1 and before *audit* in DBMS at warehouse2 (local isolation), *but*
  - there is no global order

time ↓	*shipping tx*	*audit tx*
		sum items at warehouse 1;
	ship item1 from warehouse1;	
	ship item2 from warehouse2;	
		sum items at warehouse 2;

Global Atomicity

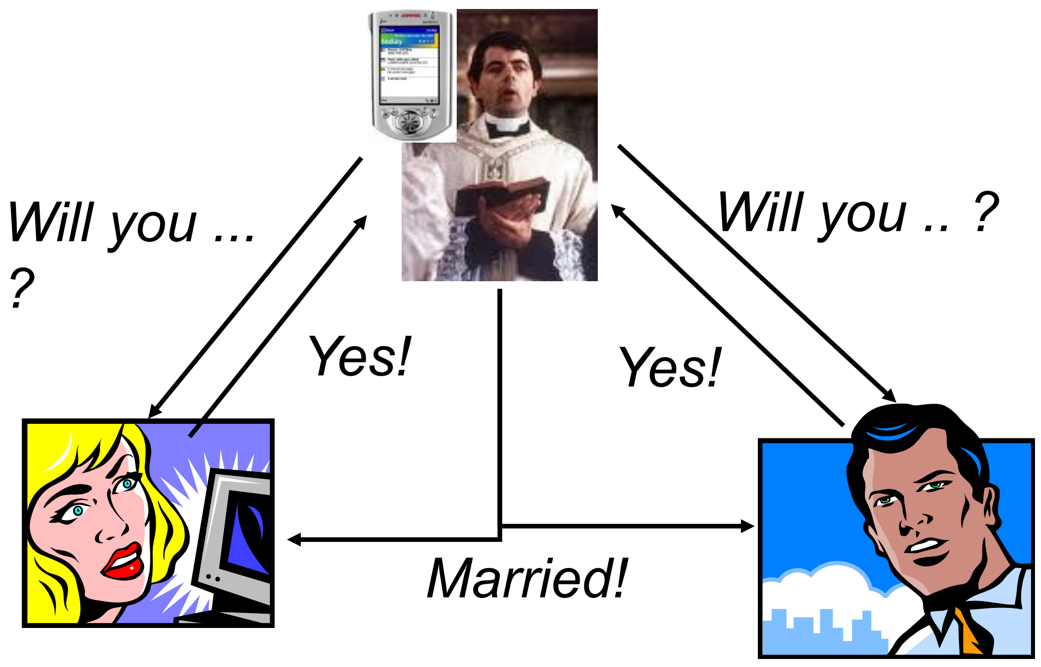
- All subtransactions of a distributed transaction release locks only at commit; and they must commit or all must abort
- An *atomic commit protocol*, initiated by a ***coordinator*** (e.g., the transaction manager), ensures this.
  - Coordinator polls *participants* to determine if they are all willing to commit

Why is this hard?

- Imagine sending to each node to say “commit this txn T now”
- Even though this message is on its way, any node might abort T spontaneously
  - e.g. due to a system crash
- What is then the final state of the system?

The Basic Idea: Getting Married over the Web

- Two round-trips of messages
  - Request to prepare/ prepared or aborted
  - Either Commit/committed or Abort/aborted (only if already prepared)



抄

2-Phase Commit Protocol

- Implemented as an exchange of messages between the coordinator and the participants
  - The participants are the individual subtransactions that participated in the transaction
  - The coordinator polls the participants to see if they want to commit
- When a distributed transaction wants to commit:

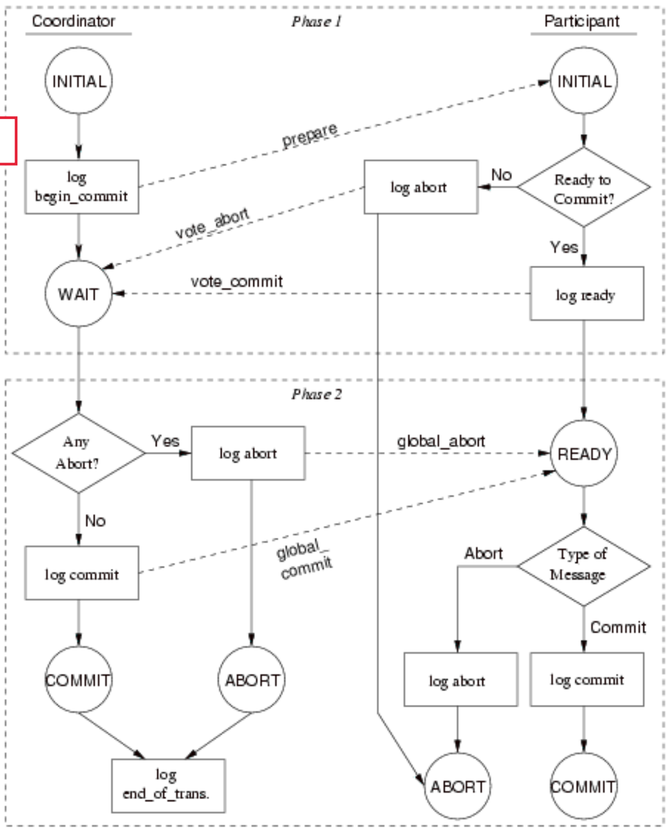
  Phase 1: Coordinator sends *prepare message* to each participant
  - Each participant ‘prepares’ (writes abort or prepare log entry) and then sends a *vote message* (*yes* or *no*) to the coordinator;

    => potential **blocking** danger, breach of local autonomy

  Phase 2: Coordinator decides on global outcome, logs it, and then sends a *commit/abort message* to each participant
  - Each participant writes a commit or abort log entry and then sends an *acknowledge message* back to the coordinator

The Two-Phase Commit Protocol

抄



Source: http://www.vermicelli.pasta.cs.uit.no/ipv6/students/andrer/doc/html/node18.html

2-Phase Commit Example

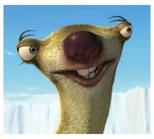
- Two Players: *P1* and *P2*
- Three replicas for play area: *r1*, *r2*, and *r3*
- *P1* wants to develop/buy field *F1* with Sid
  - Lets assume it tries to do so on *r1*

- *P2* wants to develop/buy the same field F1 with Scrat a bit later, but on *r2* and hence before he sees that *P1* occupied this field already

2-Phase Commit Example (cont'd)

- Assume we have an eager replication approach which propagates local updates to all other nodes at commit time.
- *P2* wants to commit
  - this means it will send the update of Scrat on F1 to the remaining replicas *r1 and r3*
  - Then it will try to commit all changes using 2PC

- Coordinator: *r2*, Cohorts: *r1* and *r3*
  - Coordinator *r2* sends two prepare messages to *r1* and *r3*
  - *r3* can place Scrat on *F1*, hence votes commit
  - *r1* sees the conflict with the previous update of *P1* and hence must vote abort
  - Coordinator r2 decides on abort and tell *r3*
- Sid wins



Global Serializability

- **Theorem**: If all sites use a two-phase locking protocol and a two-phase commit protocol is used, transactions are globally serializable
  - Transactions are serialized in the same order at every site – the order in which the transactions committed
- **Problems**:
  Global deadlock can be another result of implementing two-phase locking and two-phase commit protocols
  - At site A, T_{1A} is waiting for a lock held by T_{2A}
  - At site B, T_{2B} is waiting for a lock held by T_{1B}
- Systems use deadlock detection algorithms or timeout to deal with this
- 2PC protocol is blocking -> does not scale

Example: Locking and 2PC

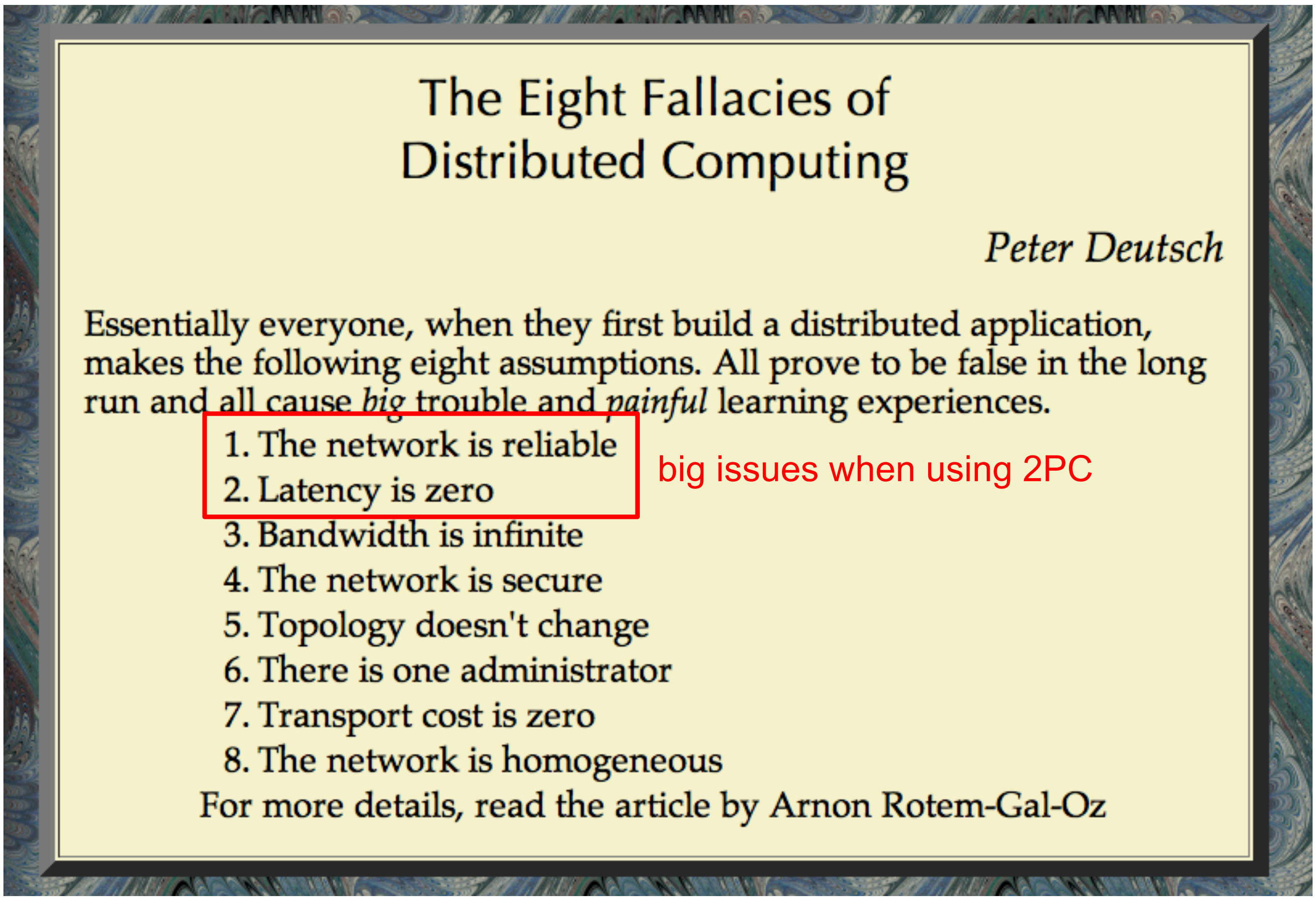
- Consider again our scenario from before:
  both players would gain an exclusive lock on their copy of field F1
  - P1 locks (exclusively) *r1* to place Sid
  - P2 locks (exclusively) *r2* to place Scrat
- With eager replication, the updates have to propagate to all replicas at commit time
  - if *P2* wants to commit first
    it will send the update of Scrat on F1 to the replicas *r1 and r3*
    - at *r3*, it also can get an exclusive lock now and update it to Scrat
    - at *r1*, it will block, because it is already exclusively locked by P1
      => P2 has to wait
- in the meanwhile P1 might want to commit too
  - it will send its update of Sid on field F1 to *r2* and *r3*
  - on *r3*, it can't update yet because this is still locked by P2
    => **deadlock**

Is this enough for large-scale data sharing?

- Problem of 2-Phase Commit is its blocking nature
  - Timeout helps some way, but it still means waiting
  - In general, a participant cannot complete the protocol until some failure is repaired, it is said to be blocked
  - Blocking can impact performance at the cohort site since locks cannot be released
- This is not good enough for large-scale web applications

Goals for a Shared Data System?

- Strong Consistency
  - We would like to have ‘all-or-nothing’ semantics
    and ideally have all copies of the same data always consistent as if there is only one copy
- High Availability
  - A data system should always be up.
    E.g. Werner Vogels keynote: Amazon will always take your order
  - The challenge: the larger the system, the higher the prob. of failures
- Partition Tolerance
  - If there is a network failure that splits the processing nodes into two groups that cannot talk to each other, then the goal would be to allow processing to continue in both subgroups.


The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

big issues when using 2PC

Source: Blog by James Gosling (https://blogs.oracle.com/jag/resource/Fallacies.html)

"The Network is reliable"

- “Network partitions should be rare but net gear continues to cause more issues than it should.”

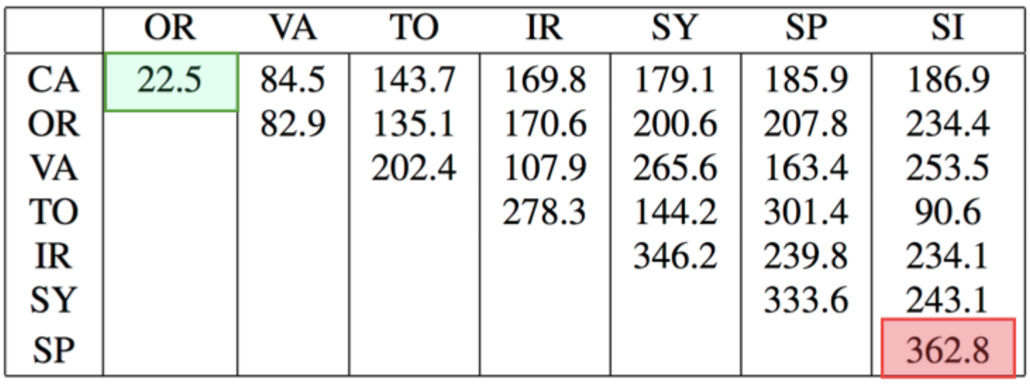
  -- James Hamilton, Amazon Web Services [perspectives.mvdirona.com, 2010]

- Some published numbers:
  - MSFT LAN: avg. 40.8 failures/day (95th %ile: 136)
    5 min median time to repair (up to 1 week) [SIGCOMM 2011]
  - HP LAN: 67.1% of support tickets are due to network;
    median incident duration 114-188 min [HP Labs 2012]
  - UC WAN: avg. 16.2–302.0 failures/link/year
    avg. downtime of 24–497 minutes/link/year [SIGCOMM 2011]

Source: Talk Slides by Peter Bailis

"Latency is Zero"

Latencies between Amazon EC2 Regions (2013)



	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

Source: Talk Slides by Peter Bailis

All times in ms, average of 1 week of pings, 1/sec

"Latency is Zero"

Rule of thumb:

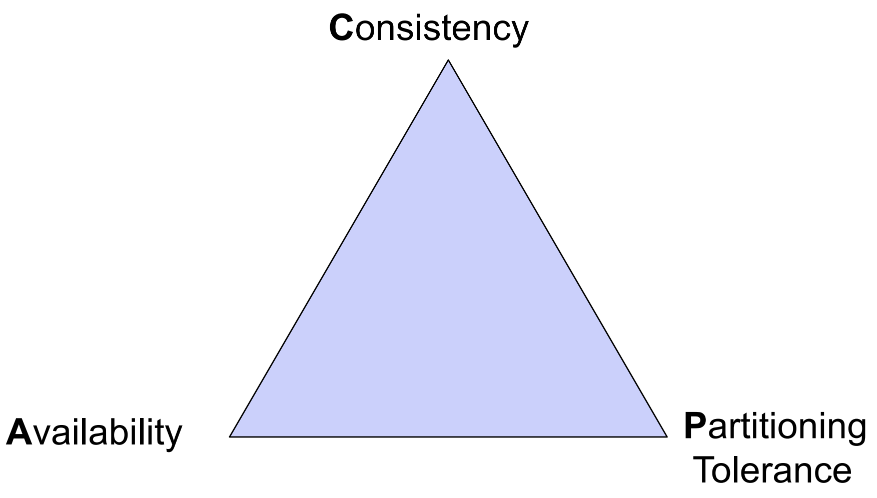
LAN	0.5 ms	1 x
Co-located WAN	1 – 3.5 ms	2 – 7 x
WAN	22 – 360 ms	44 – 720 x

See also:
http://www.bailis.org/blog/communication-costs-in-real-world-networks/

Source: Talk Slides by Peter Bailis

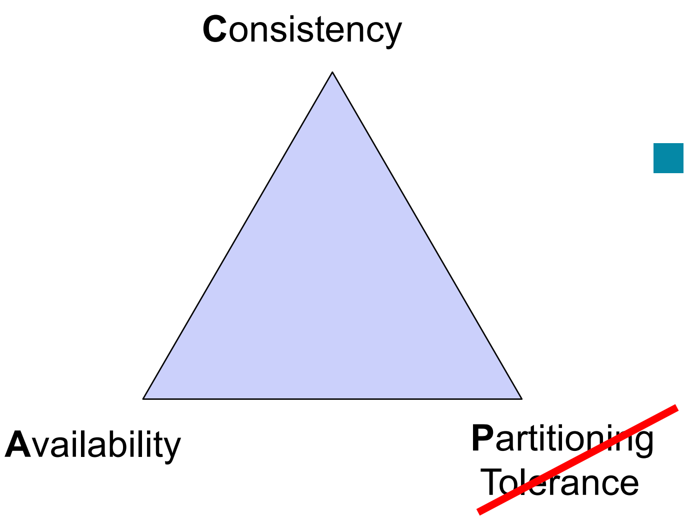
The CAP Theorem

[Brewer, PODC2000]



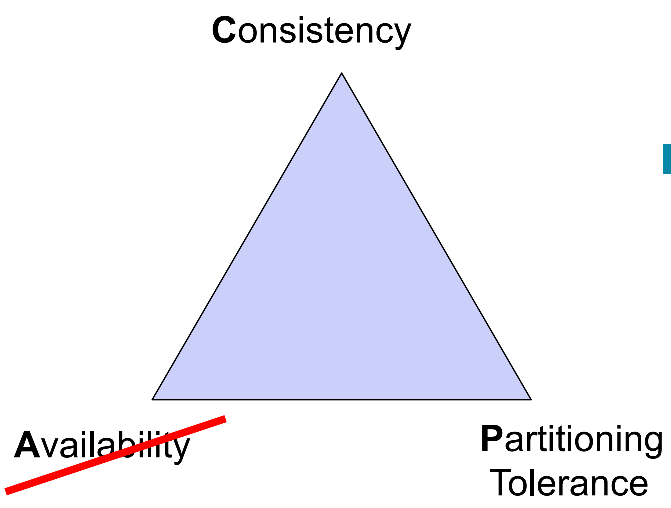
- Theorem:
  You can have **at most two** of these properties for any shared-data system.

Forfeit Partitioning Tolerance



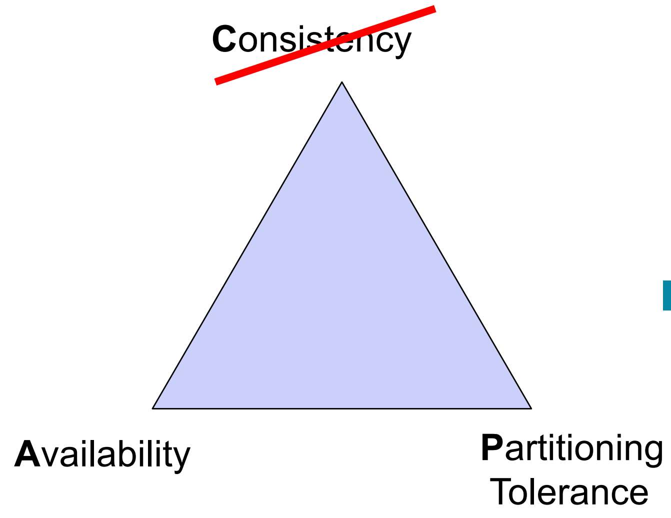
- Examples:
  - Single-site DBMS
  - Cluster of databases
- Techniques:
  - 2-Phase Commit
  - Cache Validation Protocol
  - 1cp Replication Protocols

Forfeit Availability



- Examples:
  - Distributed databases
  - Majority Protocols
- Techniques:
  - Pessimistic Locking
  - Make minority partitions available

Forfeit Consistency



- Examples:
  - Web Caching
  - Amazon's Dynamo
  - Google …
- Techniques:
  - Eventual consistency protocols
  - Access to stale data
  - Conflict resolution in Apps

Open Research Area

- The jury is out: CAP or ACID?
- E.g. Blog by Stonebraker:
  http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext
- Or latest work by UC Berkeley and Alan Fekete (U Sydney) on HAT: Highly Available Transactions
  - It depends on which form of ACID consistency we are talking about
  - Serialisabilty? Repeatable Read? Read Committed?
  - Some are actual possible together with A and P!
- Sparked new interest in replication techniques
  - Eventually consistency protocols
  - Better scalable SI-based replication protocols
  - Still problems with WAN setting though

Paxos Protocol

- Proposed by Leslie Lamport in 1998
  and in 2001 refined (or: explained in plain English ;)
- Central Question:
  "How to reach consensus / data consistency in distributed systems where each node can propose a data value that can tolerate (non-malicious) failures?"
- Recall: 2-Phase Commit can block, especially with node failures.
- How to avoid this but still result with a single consistent value?

Paxos – Preliminaries

- Distributed System with multiple nodes
- Shared, global state
  - In database speak: data replication
- Processes are concurrent, asynchronous, and error-prone
  - Each node can 'propose' a new value for a data item
    - In database speak: update-everywhere
  - Nodes communicate via asynchronous messages
    - Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.
  - Nodes can fail
    - But after failure, will eventually restart
    - Can remember *some* information if restarted after failure

Requirements for Correct Consensus

Safety Requirements:

- Only a *value that has been proposed* may be chosen
- Only a *single value* is chosen, and
- A process never *learns* that a value has been chosen unless *it actually has been*.
  - Note: no restriction on *what* value is actually chosen as long as it satisfies these three safety requirements

Liveness Requirements:

- Some proposed value is eventually chosen.
- If a value has been chosen, a node can eventually learn the value.

Paxos: Terminology

- Paxos is a distributed consensus algorithm
- Three roles for the participants (‘agents’) of the algorithm:
  - **Proposers** (2PC: coordinator)
  - **Acceptors** (2PC: participant)
  - **Learners** (no real equivalent in 2PC)

  - A specific node might play any number (one, two or three) of these roles simultaneously for different data items; the role names are just a tool for better understanding the algorithm.
  - By merging roles, the protocol "collapses" into a client-master-replica style deployment
- **Proposal**: A message consisting of two parts
  - request number *n*
  - data value *v*

Paxos Phase 1: Prepare & Promise

- A **proposer** creates a proposal identified with a number *n* and sends a *prepare request* with this proposal to some quorum of *A* acceptors
  - e.g. via a broadcast to some majority of the acceptors
  - *n* must be greater than any previous proposal number seen by proposer
- If an **acceptor** receives a prepare request with number *n* greater than that of any *prepare request* it saw,
  - it responds YES to that request with a promise not to accept any more proposals numbered less than *n*, and
  - includes the highest-numbered proposal less than *n* (if any) that it has accepted before (typically from a concurrent proposal); *nil* otherwise
- Acceptors can ignore any prepare request with a number *n* smaller or equal to another request that it already saw

Paxos Phase 2: Accept

- If the **proposer** receives a YES response to its prepare requests from a quorum of acceptors, then it sends an *accept request* to each of those acceptors for a proposal numbered *n* with the value *v* which is the value of the highest-numbered proposal among the responses.
  - Majority *m* where $m > A/2$
  - if acceptors reported no proposals, proposer can select any value *v*

- If an **acceptor** receives an accept request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a prepare request having a number greater than *n*.
  - Only higher-numbered proposals overwrite lower-numbered ones
  - In case of acceptance, answers OK to proposer

Paxos: Definition of Chosen

- A value is chosen at proposal number *n* **iff** the majority of acceptor nodes accept that value in phase 2 of the algorithm

- Why does this work?
  - With $m > A/2$, any two majorities of acceptors must have at least one acceptor in common.
  - An acceptor can accept only one value at a time. Therefore, any two majorities that choose a value must choose the same value.
  - Just need to make sure acceptors don't accept something else once a value is chosen.

Paxos Acceptors

- Allow acceptors the *option* of responding to any requests they receive.
  - Lost messages act just like an acceptor that chooses not to respond
  - In practice they should usually respond if they can, but proposers can't depend on that behavior
- Acceptors' behavior can be defined by saying what messages they are allowed to respond to.

- An acceptor needs to remember the following even if it fails and gets restarted:
  - The highest-numbered proposal it ever accepted.
  - The highest-numbered *prepare* request it ever responded to.

Paxos Example

- Proposers are *p1* and *p2*.
- Acceptors are *a1*, *a2*, and *a3*.

- *p1* sends *prepare* for proposal 1 to *a1* and *a2*.
  - *a1* and *a2* reply to *p1*.

- *p2* sends *prepare* for proposal 2 to *a2* and *a3*.
  - *a2* and *a3* reply to *p2*.

Paxos Example (cont'd)

- *p1* sends *accept request* to *a1* and *a2* for proposal 1 with value "Sid"
  - *a1* and *a2* had not accepted any proposal before
  - *p1* hence got to select which value to propose; it chose "Sid".

- *a1* accepts proposal 1.
- *a2* does not accept proposal 1.
  - *a2* promised *p2* it wouldn't accept proposals < 2.

Paxos Example (cont'd)

- *p2* sends *accept* request to *a2* and *a3* for proposal 2 with value "Scrat"
  - *p2* also got to select which value to propose,
    because at the time of its prepare request, no proposal was accepted yet.

- *a2* accepts proposal 2.
- *a3* accepts proposal 2.
- {*a2*, *a3*} is a majority of acceptors, so proposal 2 is chosen.
  - The chosen value is "Scrat"

Paxos Example (cont'd)

- *p1* sends *prepare* for proposal 3 to *a1* and *a2*.

- *a1* replies; it last accepted proposal 1 for “Sid”
- *a2* replies; it last accepted proposal 2 for “Scrat”

- *p1* sends *accept* request to *a1* and *a2* for proposal 3 with value “Scrat”
  - Value must match the one from proposal 2; this guarantees that the all nodes now agree on the same state.
  - Note that the intuition is from law makers in Paxos who do not change existing laws ;)
- *a1* and *a2* accept proposal 3.

Example as Message-Exchange Diagram

- tbd. in class

Differences to 2-Phase Commit

- Recall: A *proposer* is like the transaction coordinator
- Not all nodes have to respond to a prepare, but a *majority*
  - Typically more than half the acceptors
  - Participants can fail without blocking the prepare phase; actually, no difference between *fail* and *choose not to respond*
- Not all nodes have to accept a value
  - The message number *n* is like a version value which is guaranteed to monotonically grow
  - Acceptors can fail between Phase 1 and 2 without blocking Paxos
- Both proposers and acceptors include in their messages the latest data value they have accepted
  - Allows nodes to *learn* the latest value, even if they were down before

2PC+Locking vs Paxos

- Paxos does not block – 2PC with locking would block
- 2PC+Locking supports in-place updates (hence the locks)
  - Paxos is a consensus protocol about the outcome of one global state
  - For updating a variable, need additional sequence or version number
    - achieve consensus of the 'ith value' of a variable
    - similar to get consensus on the value of a log entry in the commit log, with each update creating a new log entry

Multi-Paxos

- If the leader (proposer) is stable, an optimisation to efficiently process multiple consensus decisions in a row is:
  - Do phase 1 (Propose / Promise) only once
  - Thereafter, the same proposer can conduct multiple Accept-Request/Accept cycles
  - Include 'instance' number with each accept-request to keep track of in which round a value was chosen
  - Instance number is increased by proposer for each new decision

- Avoids phase 1 for the majority of rounds
- Reduces failure-free message delay per proposal from 4 to 2 messages (accept-request and answers by majority of acceptors)

Any Practical Usage of Paxos?

Yes, for example:

- Google BigTable (OSDI2006)
  - Via its use of Chubby for meta-data management and master selection (see below)
- Google Spanner (OSDI2012)
  - Distributed relational datastore, successor of BigTable
- Spinnaker datastore (VLDB 2011)
  - Distributed datastore by IBM Research + LinkedIn/Google

- Google Chubby (OSDI2006)
  - Distributed lock system and meta-data repository
- Hadoop Zookeeper (DISC2009)
  - Open-source implementation of Chubby; uses own ZAP protocol that is inspired by/based on Paxos

Paxos Alternatives

- **Raft**: like Paxos but better understandable
- **Blockchain**: similar to Paxos, but includes provisions to proof the validity of an update (assumes malicious players)

The Costs of Consistency

- 2-Phase Commit and Paxos still strive to achieve a fully consistent distributed state
- The cost for this is
  - Blocking nature with 2PC
  - a high number of message exchanges with both 2PC and Paxos as in both cases we need to get an agreement between a (majority) of nodes with two phases of message exchanges
- Does this scale?
  - Number of messages? Throughput?
  - Latency?
- If not, what can we do instead?

Notions of Consistency

- Strong Consistency (aka 1-copy-Serializability)
  - behavior as if there is only one copy of each data item, and
  - only serializable accesses permitted
- Weak Consistency
  - No guarantee that a subsequent (read) access from this or any other client will return a just updated value
  - Might mean several versions of data (i.e. not 1-copy), might mean not serializable
  - Updates will be propagated 'eventually' to all replicas after some delay ('inconsistency interval')

    => **Eventual Consistency**

Eventual Consistency

- A model originally proposed for disconnected operation (e.g., mobile computing)
- Different nodes keep replicas and each update is "eventually" propagated to each replica
  - And eventually, there is agreement on which update is the latest
  - DNS is the most well-known system implementing eventual consistency
- Usual definition is *counterfactual:*
  "once updating ceases, and the system stabilizes, then after a long enough period, all replicas will have the same value"

Extra Properties for Eventual Consistency

- Programming an application is much harder if storage supports only eventual consistency
  - What to do until everything settles down??
  - Handling inconsistency in the sequence of reads
  - Cf Hellerstein et al PODS'10/CIDR'11: eventual consistency data model supports monotonic programs (a very limited class)

- A range of extra properties, which (if the storage provides this) can make programming not quite so hard
  - e.g., read-your-own-writes, monotonic reads, …

Properties of Eventual Consistency

- Causal Consistency
  - If client A has communicated with client B that an item has been updated by A, B will see the updated value of A
- Read-your-writes Consistency
  - Special form of Causal Consistency for A=B
- Session Consistency
  - Practical version of Read-your-writes Consistency that guarantees this property just within one 'session', but not between separate sessions
- Monotonic Read Consistency
  - A subsequent read will never return an older version of a data item than a previous read
- Monotonic Write Consistency
  - Serializability just for writes

Summary

- **Strong Consistency**
  - is most desirable for consumers, but seen as not achievable with network partitioning tolerance
  - Cf. CAP Theorem
- **Eventual Consistency:**
  - Allows nodes to disagree on current data value
  - But current algorithms/systems differ widely in how this is achieved
- **Current Cloud Storage Implementations**
  - Provide different variants of 'eventual consistency' without disclosing the implementation details or clear SLAs
  - Currently missing SLAs (observable, but no guarantees):
    - Rate of inconsistent reads, time to convergence, performance under variety of workloads, availability, costs

References

- Werner Vogels: *Eventually Consistent*. Communications of the ACM, Volume 52, No. 1, Jan 2009.
- Eric Brewer: *CAP Theorem*, talk at PODC 2000.
- Peter Bailis, Alan Fekete, Ali Ghodsi, Joe Hellerstein and Ion Stoica: *HAT, not CAP: Towards Highly Available Transactions*. HotOS 2013.
- 2-Phase Commit
  - Kifer/Bernstein/Lewis (2nd edition)
- Paxos Protocol
  - Leslie Lamport: The Part-Time Parliament. *ACM Transactions on Computer Systems 16, 2, pp. 133-169,* May 1998.
  - Leslie Lamport: Paxos Made Simple. 2001.
  - Jun Rao, Eugene J. Shekita, Sandeep Tata: *Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore*, VLDB 2011. (arxiv.org/pdf/1103.2408.pdf)
  - Corbett et al: *Spanner: Google's Globally-Distributed Database*, OSDI 2012.
  - Good Blog post explaining the functioning of Paxos with some neat animations: http://harry.me/blog/2014/12/27/neat-algorithms-paxos/