CMPSCI 677 Operating Systems **Spring 2022**

Lecture 16: March 28

Lecturer: Prashant Shenoy *Scribe:* ***Shishir Verma(2017), Priyanka Mary Mammen(2022)***

16.1 Consistency and Replication

Replication in distributed systems involves making redundant copies of resources, such as data or processes, while ensuring that all the copies are identical in order to improve reliability, fault-tolerance, and performance of the system.

Types of Replication :

- **Data replication:** When the same data are stored on multiple storage devices.
- **Computation replication:** When the same computing task is available to be executed on multiple servers.

16.1.1 Why Replicate?

- **Reliability:**

  Data in distributed systems need to be replicated to improve the reliability of the system. If one of the replicas become unavailable or crashes, the data still remain available. For instance, in a distributed system, if one of the database servers crashes and we have a replicated copy of the same data on another database server, then the data is safe. We can point our system to the second replica of the database and continue to access the data without any problems. This is in general true with any storage system. If we have multiple copies of the data and the disk crashes on one machine or something else goes wrong, our data remain available because we have other copies.

  In many cloud-based storage systems, like Amazon S3, replication is done internally. It replicates the data in multiple locations. User can ask for a copy of the data and the system will get it from one of the replicas. The user doesn't have to know or specify from which replica the data should be accessed. There are many file-systems that support replication as well, e.g., hadoop file system (hdfs) or Google file system (GFS).

- **Performance:**

  Computation or data are also replicated to improve the performance of the system. Replicated servers can serve a larger number of users as compared to just one server. For example, if we have just one web-server, it would have a certain capacity, i.e., requests it can serve per second. After reaching the limit, it will get saturated. By replicating it on multiple servers, we can increase the capacity of our application so that it can serve more requests per second.

  Similarly, data can also be replicated to improve performance and capacity of the system. For instance, if we have a large number of web-servers and just one database server, eventually, the requests from web-servers will trigger more queries than what the database is capable of executing. If those are computationally expensive queries, the database might become the bottleneck in the system. In ideal

case, you would expect a linear increase in throughput, but in most cases you would get something less than ideal performance.

The replication can also be done in wide area networks, i.e, you can put copies of your applications in different geographical locations (which is called *geo-distributed replication*). Here, we are keeping copies closer to users, which aids better performance due to the decreased latency when accessing the application.

16.1.2 Replication Issues

Before we get into consistency, we will discuss replication issues that we have to consider:

- **When to replicate?**
  Similar to dynamic-or-static threadpool concept.
- **How many replicas to create?**
  If we need to sustain a certain request rate, we can find out how many replicas are required depending on the individual capacity of each replica.
- **Where should the replicas be located?**
  In a distributed application we can put the replicas in different locations. The general rule of thumb is to keep the servers geographically closer to the end-users. If the users are spread out in several locations, then it would be wise to keep replicas spread out in similar fashion. The users can connect to the replica that is geographically closest to them.

16.2 CAP Theorem

The CAP theorem was initially a conjecture by Eric Brewer at the PODC 2000 conference. It was later established as a theorem in 2002 (by Lynch and Gilbert). The CAP theorem states that it is impossible for a distributed system to simultaneously provide more than two out of the following three guarantees:

Consistency (C): A shared and replicated data item appears as a single, up-to-date copy

Availability (A): Updates will always be eventually executed

Partition-tolerance (P): The system is tolerant to the partitioning of a process group (e.g., because of a failing network)

16.2.1 CAP Theorem Examples

Consistency + Availability: Single database, cluster database, LDAP, xFS. If you want to have consistency and availability in your system, you have to assume that network cannot be partitioned to ensure that messages do not get lost.

Consistency + Partition-tolerance: Distributed database, distributed locking. They assume that the coordinator doesn't fail and there wont be any modifications in the system.

Availability + Partition-tolerance: Coda, web caching, DNS. DNS updates can take upto few days to propagate.

16.2.2 NoSQL Systems and CAP

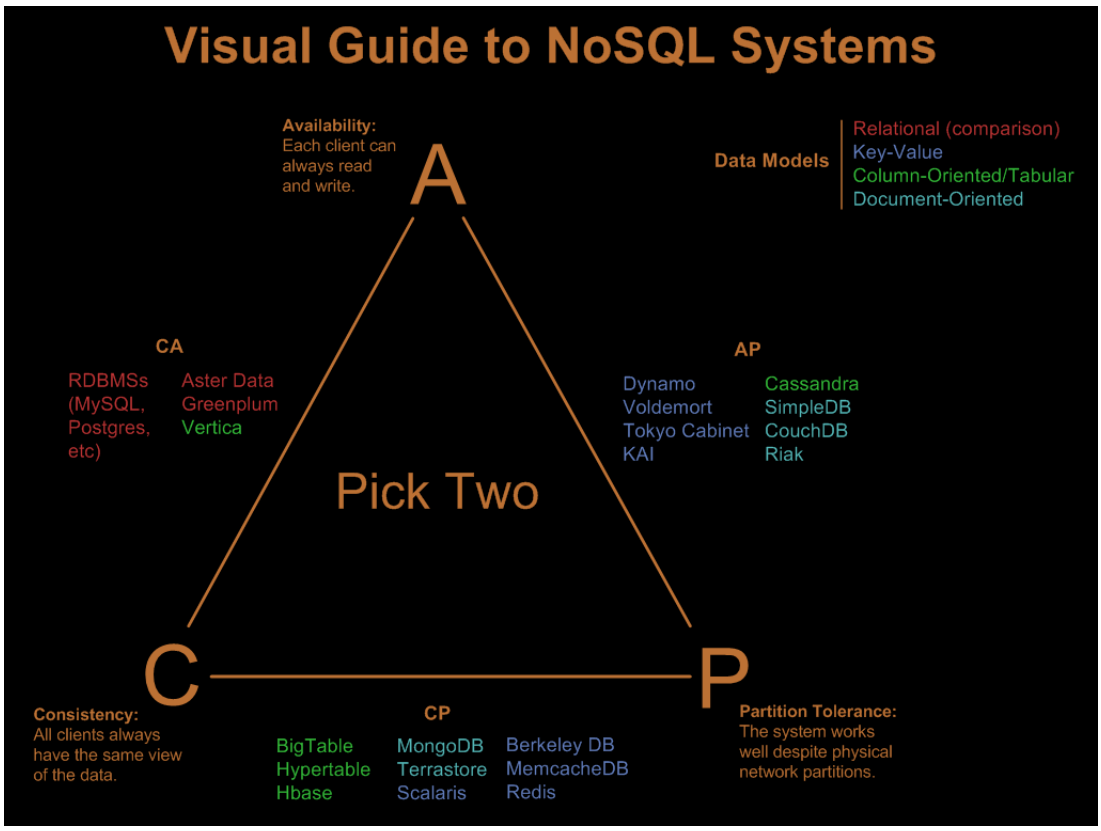

Figure 16.1: CAP in database systems.

NoSQL systems, as the name suggests, don't use a SQL database. Figure 16.1 shows some database systems and which properties they hold. Consistencies in the presence of network partitions are problematic.

Question: What is partition tolerance ?
Answer: Partition tolerance in CAP means tolerance to a network partition. Suppose there are some nodes in a distributed system and they are connected over the Internet. If any of the link goes down, the network essentially is partitioned into two halves. The nodes in first half can talk to one another, and the nodes in second half can talk to one another, but the nodes from first half cannot talk to the nodes in second and there are clients able to talk to either one or both of those nodes. In our case these are replicas. If there is an update on the node, that update can be propagate to other nodes, but since the network is partitioned it cannot communicate with the other nodes until the network is fixed. The system will be inconsistent if the messages are not flowing back and forth.

Question: Why is availability an issues?
Answer: Availability can be an issue if a node goes down and the system can't make any progress. For example, in the case of distributed locks, if a nodes go down, we cannot actually operate our system. Similarly in the case of 2-phase commits and and other situations where it is required for all the nodes to agree on something, if some nodes are unavailable, then they will not be able to agree.

Question: Is there any way we can relax one of the dimensions, e.g., consistency and get more of the other dimensions?
Answer: For specific systems we can make trade-offs. There is no general rule saying that if we relax

property A by 20%, we can get 30% more of property B because it all depends on the assumptions we make for that application.

Question: In Figure 16.1, there are a lots of databases mentioned. Some of them offer availability and partition tolerance, but not consistency. Why would a database not want consistency?
Answer: In these cases it means that we are not getting good consistency guarantees. A very lose form of consistency is called "eventual consistency." The best way to understand it is by taking DNS as an example. We can think of DNS as a very large database that stores hostname to IP address mappings. There are no consistency assumptions made. If we make an update, it may take up to 24 hours for it to propagate. Until then, things may be inconsistent with respect to one another. We do this because we want availability and partition-tolerance. If our application needs a better guarantee than that, we should not choose these databases.

16.3 Object Replication

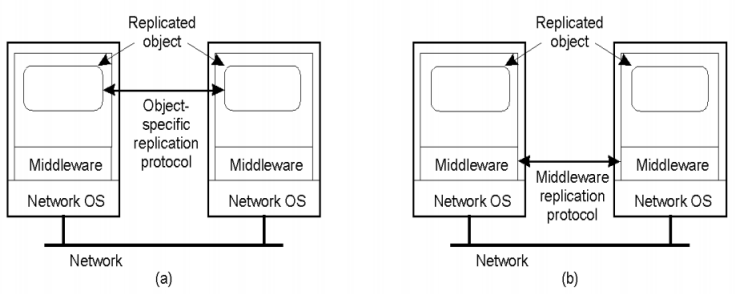

Figure 16.2: Two types of replication. (a) The application does the replication and handles consistency. (b) The middleware does the replication and handles consistency.

Question: Is it beneficial to implement replication in the middleware than in the application level?
Answer: It depends on the type of application you use.

16.3.1 Replication and scaling

Replication and caching are often used to make systems scalable. Suppose an object is replicated N times, the read frequency is R, and the write frequency is W. Stricter consistency guarantees are worthwhile if $R >> W$, otherwise they are just wasted overheads (tight consistency requires globally synchronized clocks). The overheads increase as we make the consistency guarantees stricter. Thus, we try to implement the loosest consistency technique that is suitable for our application.

16.4 Data-Centric Consistency Models

We can analyze from the perspective of data items. Tthere are consistency models from the perspective of clients too. All of the consistency models have the goal to retrieve the most recently modified version. There is a contract between the data-store and processes, i.e, if processes obey certain rules, the data store will work correctly. All models attempt to return the results of the last write for a read operation.

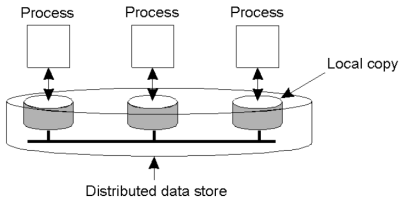

Figure 16.3: Data-centric consistency models.

16.4.1 Strict Consistency

Strict consistency is when the system always returns the results of the most recent write operation. There is no inconsistency. It is hard to implement as it assumes a global clock and compares the read and write timings. There is some delay in propagation of messages as well. Suppose a copy at location A gets modified and A sends a notification to B about its write which takes 1ms to travel. If B gets a read request before the message from A has arrived but after A has been modified, B will not know that there has been an update. This is one of the reasons why it is so hard to implement.

16.4.2 Sequential Consistency

Sequential consistency is weaker than strict consistency. All operations are executed in some sequential order which is agreed upon by the processes. Within a process, the program order is preserved. We can pick up any ordering for operations across different machines.

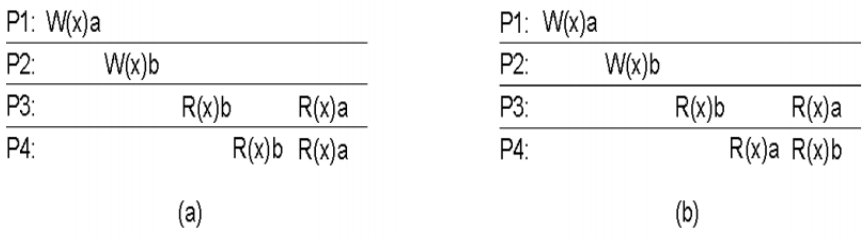

Figure 16.4: Sequential consistency.

In Figure 16.4, lets say x is a web page. Process $P1$ writes a to x and process $P2$ writes b to x. Process $P3$ reads x's value as b and then later reads it as a. If we had a global lock, we would know that $P1$ wrote it first and then $P2$. So, once $P3$ sees a it shouldn't see b. But since we do not have synchronized clock, we don't really know if that is what has happened, because $P1$ and $P2$ did not communicate with each other and hence we don't know if they are concurrent events. So processes just agreed that $P1$'s write happened before $P2$'s write. In 16.4 (a) The processes agree on the order that $P2$ wrote before $P1$ and both $P3$ and $P4$ read in that order. Figure 16.4 (b), $P3$ and $P4$ see in different orders and that is not allowed.

Question: Process $P1$ has written to the web page, so why does it not see a before b?
Answer : It will process a before b but the question is when the update b arrives, $P1$ has to decide if that update occur before $P2$. j]Just like in totally ordered multicasting, we will have to wait for all the writes to figure a global ordering and then commit them in that order.

16.4.3 Linearizability

Along with all the properties of sequential consistency, we also have the requirement that if there are two operations x and y across different machines such that time-stamp of x, $TS(x)$ < time-stamp of y, $TS(y)$, then x must precede y in the interleaving. There is an implicit message passing. The reads and writes are done on shared memory buffers and if we read some value from a variable, the write must have happened before. If there are concurrent writes, then their order can not be determined. Linearlizability is stricter than sequential consistency but weaker than strict consistency. Consider the difference between serializability and linearizability: serializability is a property at transaction level, whereas linearizability handles reads and writes on replicated data.

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Figure 16.5: Linearizability example.

Figure 16.5 shows three processes. Each process writes one variable and reads variables written by the others. Thus, there is an implicit communication here about the ordering. The valid interleaving is shown in Figure 16.6.

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011 (a)	Signature: 101011 (b)	Signature: 110101 (c)	Signature: 111111 (d)

Figure 16.6: Valid interleaving for Figure 16.5 satisfying the property of linearizability.

An invalid ordering would be when after assigning a value to a variable we still print a 0. Another scenario will be if we do not agree to a program order.

16.4.4 Causal Consistency

Causally related writes must be seen by all the processes in the same order. In Figure 16.7 (a), $P2$ read a from x and then wrote b which means that $P1$ wrote before $P2$ and thus, a will be read before b. Process $P3$ does not agree to it and thus is not consistent. For concurrent writes, the processes do not need to

agree upon an interleaving and can read in any order (Figure 16.7 (b)). Causal consistency is weaker than linearizability as the latter fixes an order.

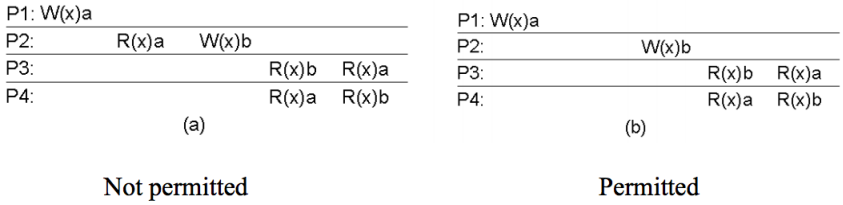

Figure 16.7: Causal consistency.

Question: In Figure 16.7 (b), if $P3$ reads it twice, is it possible for it to read b and b again?
Answer : That is valid because in that case $P1$ writes a on x first. Then $P2$ writes b on x and it overwrote the content written by $P1$. $P3$ reads that subsequently and keeps seeing b as many times as it reads.

Question: Could you give an exmaple for what is allowed in sequential consistency but not in linearizability (Figure 16.7 (b)?
Answer : Processes can agree on some order, say b, a, then $R(x)b$ $R(x)a$ and $R(x)b$ $R(x)a$ is allowed in sequential consistency, which is not allowed in linearizability.

Question: In lineraizability is $R(x)a$ $R(x)b$ and $R(x)a$ $R(x)b$ allowed (Figure 16.7 (b)?
Answer : Yes, in fact that is required. You have to see a followed by b if there is a happen-before relation.

16.4.5 Other Models

FIFO consistency does not care about ordering across processes. Only the program ordering within a process is considered. It may also be sometimes hard. It is also possible to enforce consistency at critical sections, i.e., upon entering or leaving a critical section but not within a critical section. This can be a weak consistency or entry and release consistency. All transactional systems like databases use this kind of consistency. Consistency is done at commit boundaries only.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Figure 16.8: Consistencies (weaker as you go down).

16.5 Client-centric Consistency Models

Consider reads and writes performed by different clients (processes). There are following types:

Monotonic Reads: All reads after a read will return the same or more recent versions. It does not necessarily have to be the most recent.

Monotonic Writes: The writes must be propagated to all replicas in the same order.

Read your writes: A process must be able to see its own changes. For example, if you update your password and log back in after sometime while the changes have not been replicated. But still, the system should not say incorrect password.

Writes follow reads: The writes after read will occur on the same or more recent version of the data.

Question: What is the concept of "you" in the above description?
Answer: "You" means a machine or a process or a user who uses the machine.

16.6 Eventual Consistency

Because of their high costs, many systems do not implement the consistency models described previously. According to eventual consistency, an update will eventually reach all of the replicas; there are no guarantees regarding how long it will take. DNS uses eventual consistency. The only guarantee is that in the absence of any new writes, all the replicas will converge to the most recent version. Write-write conflicts occur in this model because there can be conflicting writes across machines and eventually there will be a conflict when the updates propagate. Source code control systems are also eventually consistent. Some exmaples of systems that use eventual consistency include:

- DNS: Single naming authority per domain. Only naming authority allowed updates (no write-write conflicts).
- NIS: User information database in Unix systems. Only sys-admins update database, users only read data. Only user updates are changes to password
- Cloud storage services such as Dropbox, OneDrive, and iCloud all use eventual consistency.

Question: If DNS uses eventual consistency, can it lead to network problem?
Answer: The problem occurs because many of the DNS servers cache entries. If you want to avoid it, you have to reduce the size of the cache value. In this case, when you make the same request again, that server needs to look upto the origin server again, which might lead to an increase in load. Thus, expiring the cache values will generate more requests at the origin server.

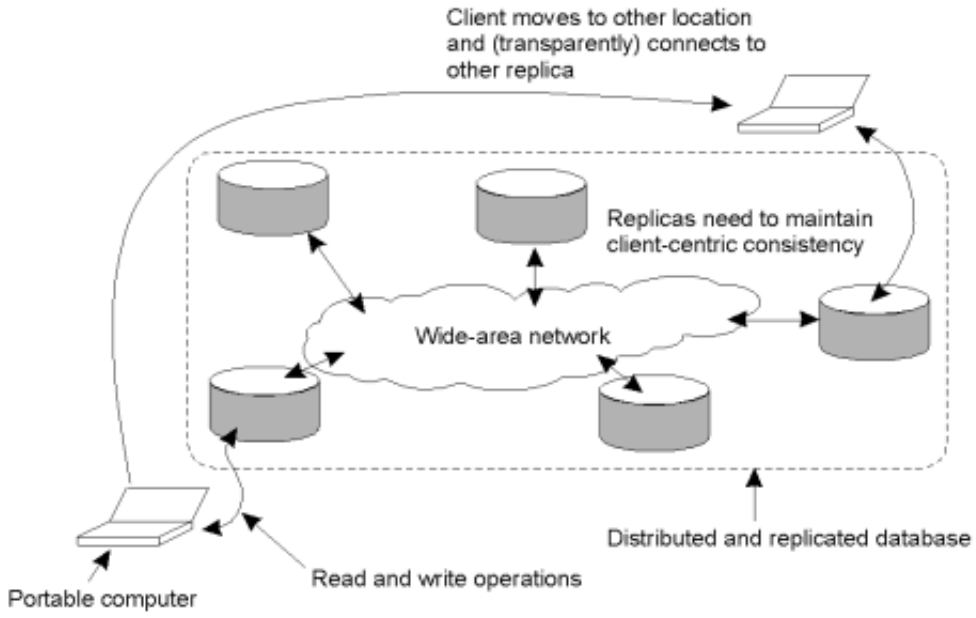

Figure 16.8: Eventual consistency.

16.7 Epidemic Protocols

These protocols help implement eventual consistency. In Bayou, a weakly connected environment is assumed, i.e., clients may disconnect. Offline machines are made consistent when they re-connect (e.g., pulls in git). The updates propagate using pair-wise exchanges similar to diseases. Machines push/pull updates when they connect to another machines and eventually all the machines will have the updates.

Many systems that you encounter in practice, use this form of consistency. For example, DropBox essentially uses this type of model, except that DropBox has a centralized server. You might have many Dropbox clients. When you make an update on one device, your Dropbox client at some point going to contact the centralized server and tell it "Here are some changes," and push it. It might also pull for new updates. Once you pushed your changes to the server, other clients can pull the changes from the centralized server after some time. So you have a pairwise exchange of information between two machines which happens at random intervals.

Question: Will you waste a lot of messages trying to spread an infection? When do you stop?
Answer: There are two algorithms based on epidemic protocols discussed in the following sections and will answer this question there.

16.7.1 Spreading an Epidemic

Algorithms:

- Anti-entropy:
  - Server P picks a server Q at random and exchanges updates.
  - Three possibilities: only push, only pull or both push and pull.

  - **Claim**: "A pure push-based approach does not help spread updates quickly."
    Explanation:
    Suppose there is a system with N nodes and we make a change at one of the nodes. This node will randomly pick another node and push that update. Next, these two nodes will pick two other nodes randomly and push the update. The number of nodes which have the update increase exponentially. In the end there will be a very small set of servers which haven't received the update. The probability of picking a server in a large system is $1/N$ i.e. for a large value of N, it is a small probability. We may end up picking up the same servers which have already seen the update. So, the remaining small number of nodes may not get the update quickly. We will have to wait until one of these infected nodes end up picking them and push the update.

  It works much better if we combine push and pull because nodes are pro-actively pulling and pushing.

- Rumor Mongering (also known as "gossiping"):

  This works similar to how rumors are spread. Inspired by class of protocols called *gossip protocol*, which are same as epidemic protocols with one small difference: in Rumor mongering there is some probability that you will stop. Just as initially if we have news item, we try to spread it, but after a while we feel like everybody knows it, so we stop calling friends. Rumor mongering is a push-based protocol.

  - Upon receiving an update, P tries to push to Q.
  - If Q already received the update, stop spreading with prob 1/k.
  - Analogous to "hot" gossip items => stop spreading if "cold."
  - Does not guarantee that all replicas receive updates.
    * Chances of staying susceptible: $s = e^{-(k+1)(1-s)}$

Question: Can you push faster and at a higher rate in anti-entropy?
Answer: The rate at which you push or how frequently you push is a parameter you can set in both anti-entropy and rumor mongering, so both of them can control the rate at which spread is happening.

Question: There are many ways to do this, are there any reasons why choose this?
Answer: That's right, this is an entire area of research and there are hundreds of papers published on approaches similar to these.

Question: If a file is changed at location 1 and some other client changes the same file at another location at around the same time, what happens?
Answer: This is called a write-write conflict. This will often occur in systems like Dropbox. If you login on two machines, open the same file on both the machines, make two different changes and save the file more or less at the same time, you will see both of those clients will try to contact the server and server will see that the files are changing more or less at the same time, it will declare a write-write conflict and create two copies, saying that the file changed at the same time. This can often happen because the consistency guarantee is weak.

16.7.2 Removing Data

Deletion of data is hard in epidemic protocols. Lets say we delete a file from Dropbox. Our Dropbox client contacts the server asking for updates. It will compare the two directories and find a file on the server which is not available on the client. If we simply do pairwise exchange blindly, we will recreate the same file on the client which was deleted. There has to be a way to distinguish between an "update" and a "delete." A "delete" that leaves no sign of it will not allow you to figure out whether it is a deleted file or a new file

that got added. This problem is solved using *death certificates*, which means when a file is deleted, an entry is kept for the file that has been deleted. So, "delete" is now an "update," which has to be propagated and cause other nodes to delete the file as well.