LAST PRECEPT:

1) Amazon (probable) system architecture:
- Web tier with servers running the 'applications' on top of them. We specifically look at the 'shopping cart' application because it has some intereting properties.

- Storage tier: (DYNAMO) objects are stored as <key, value> pairs. Keys (we believe) describe a userID (his/her shopping cart) while the value is just a 'blob' of information (structured/unstructured).

- We can imagine three types of requests from the client to the application server:lookup (look at what's in the cart), add or delete item. These client requests are mapped to two different API calls to the data store system (DYNAMO itself) get() and put().

- the get() operation is called on a key. The get returns one single object (the shopping cart) or a list of objects if there was some conflicts within DYNAMO.

- the put() operation is called on a (key,context). The context is in reality the vector clock use by the DYNAMO system.

Question (still don't understand): do we have to call a get() before calling a put()? In the paper it says that the put() operation creates a new 'context' (updates the vector clock) but needs the old context (this is also needed because DYNAMO resolves conflicts at read time and not at write time).

- There are two types of conficts: semantic and syntactic conflicts.

Semantic conflicts are resolved by the application. These conflicts arise when DYNAMO is unable to resolve the syntactic conflicts (due to concurrent writes / network partition? / failures). Syntactic conflicts are discrepancies between vector clocks. The only conflicts DYNAMO itself can resolve are by looking at each component in the vector clock and if all elements in one vector clock are 'older' than another then the old copy of the object can be discarded.

- DYNAMO vector clocks: how are they built? (this ties in with how the partitioning is done, not sure if I should talk about the consistent hashing + virtual nodes and replication before talking about vector clocks)
Vector clocks are used for data versioning... let's talk about data versioning

Data versioning:

- Dynamo provides eventual consistency, in fact it allows for updates to be propagated to all replicas asynchronously.
A put() call may return to its caller before the update has been applied to all the replicas (but it has to return after the W parameter specified in the quorum right?)

- Problems are caused by network partitions, concurrent actions or failures. Due to this what can happen is version branching: example? look at figure 3

To avoid this Dynamo uses vector clocks to capture causality between different versions of the same object. If the vector clocks can be causally ordered then the conflict is syntactically resolved, otherwise it must be semantically resolved.

Causal consistency (causal ordering) review:

a stronger notion than partial ordering. what is partial ordering?
In partial ordering we only had two rules:
- on same process if a --> b then a comes before b
- send(m) precedes a receive(m)

- implemented through Lamport logical clock: one timestamp per process. Timestamp is updated locally in the following method:
max (receiver-counter, message-timestamp) + 1

Casual consistency instead?
- concurrent writes maybe seen in different order
- Writes that are poten!ally causally related must be seen by all processes in the same order.

Partitioning: (usual picture)

- Partitioning is achieved through consistent hashing. We imagine to have a 'ring' of values and each node owns a portion of the ring (in this specific case it handles all the items preceding its hash(nodeID) value up to its predecessor. Each key is inserted into the ring by computing hash(key) and then assigned by the closest node clockwise

What are two problems associated with this:
- not uniform data distribution
- oblivious to performance of nodes (heterogeneous nodes - some have less capacity)

SOLUTION:

- virtual nodes: each node in the ring is a 'virtual node'. One node owns multiple nodes. Effectively when a new node joins the system he takes up different 'virtual nodes' and thus takes multiple positions in the ring.

advantages:
- when a real node becomes available the load is split across multiple nodes (not just 1 gets overloaded)
- when a new node joins the system it accepts a roughly equivalent amount of load from each of the other nodes (takes load off of many nodes)

Replication:

How is data replicated across nodes?
- each data item is replicated N times (N is tunable by the system). The node who 'owns' the key is the coordinator and it is in charge of replicating the data items that fall within its range.
The N successors of the coordinator node are the nodes where the replication is made. Thus each node owns items for his own range and the previous N nodes.
The list of nodes responsible for a certain range of values is called the preference list

Execution of get() and put() operations:

get() and put() should be directed towards the coordinator but what can happen is that we have a load balancer in front of the DYNAMO and the request will be forwarded to a random node. if the node is in the top N preference list it will handle the request, otherwise it will forward the request to the coordinator.

How does DYNAMO maintain consistency?

consistency protocol used in quorum systems. We have two configurable values R and W. Setting a R+W > N we get a quorum system. But this system does not provide great performance as it goes as fast as the slowest between the R & W servers.

Upon receiving a put() request for a key, the coordinator generates the vector clock and then sends the new vector clock (and the new object) to the highest N ranked. returns when receives W-1 responses

for a get() request the coordinator requests all existing versions of data for that key from the N-highest ranked nodes (waits for R responses)

sloppy quorum: all read and write operations are performed on the first N healthy nodes which may not always be the first N nodes encountered while walking the consistent hashing ring.

so when a node fails (say A) then another node (D... but why D?) will hold the item in place for A until node A comes back online. When this happens then D will attempt to deliver the replica to A. D is a called a hinted replica and this procedure is called a hinted handoff

what happens if the hinted replica goes offline? DYNAMO implements an anti-entropy replica synchronization protocol to keep the replicas synchronized.

A merkle tree is a tree of hashes. The leaves are the hashes of each key, the parent is the hash of all its children. In Dynamo each node keeps a merkle tree for each key range it has to hold (each virtual node).
comparing the set of items two nodes have is rather easy (just compare hashes and then explore the tree if the hashes are different).

the problem is when a node leaves the system, the ranges for each other node change and the merkle trees have to be recalculated.