2022 Prelim – SOLUTION SET

Q1 problem setting: One very famous use of a key-value store (DHT) is to maintain the Amazon shopping cart. The system is called Dynamo. When you click to add something to your cart, a DHT put occurs, and the purchase is saved into a shard determined by your customer id (for example, "Purchase 1" by "Ken Birman" is saved in a shard obtained by hashing "Ken Birman"). Each different purchase has a separate id, so items won't overwrite one-another unless you revise a selection, such as by changing the size or the number of units: Amazon allows you to edit a purchase or delete it, before checking out. Checking out scans your shopping cart and then charges you for everything.

In their paper about this system, Amazon describes how a system failure sometimes makes it hard to reach the members of the correct shard. For this, Amazon does something unusual: they just increment the shard number until they find a shard where they *can* store the key-value item. Eventually, when the DHT recovers from whatever the issue might have been, items will automatically migrate back to the shard where they really should reside. For example, if "Ken Birman" normally maps to shard 6, maybe "Purchase 1" temporarily ended up on shard 7. But then the nodes in shard 6 recover, they notify the nodes in shard 7, and the key-value pairs get shifted from shard 7 to shard 6. There is no locking used when this kind of transfer occurs.

Amazon is a big believer in the CAP principle and in BASE.

a) State the CAP principle in your own words, ***briefly***. We already know that C stands for consistency, A for availability and P for network-partition tolerance. We want you to explain what those words mean, and what CAP is saying we should "do" when designing scalable, rapidly responsive systems.

CAP is a conjecture originated by Eric Brewer in which he asserts that there is a fundamental tradeoff between consistency, high availability and partition-tolerance in cloud settings, namely that we can only have two of the three properties at the same time. Eric argues that we should relax consistency to ensure quick responsiveness.

b) BASE is short for basically available, soft state with eventual consistency. "Basically" means "to the degree possible." Explain what the words "available", "soft state" and "eventual consistency" mean, and then with one additional sentence, what BASE is recommending.

"Available" means that the system is able to respond to requests immediately, even if this involves using potentially stale cached data.

"Soft State" means that the system responds to requests using some form of cached data, and that the real state is held elsewhere.

"Eventual consistency" means we should clean up any inconsistencies later, if necessary.

c) In words, tell us what Jim Gray's study tells us about transactions on a cloud-hosted database. Don't show us the formula – tell us what Jim's conclusions were and what they are based on.

Jim Gray says that if you do want consistency (he focused on transactional ACID properties in a database), there will be a dramatic growth in overhead and hence a slowdown if we increase the number of servers running the database and then try to send an increased number of transactions to the database. He attributes this to lock contention and delays, aborts due to deadlock, and rollback/restart when deadlocks or very low delays arise. Jim concludes that the only way to get true scalability is to break our database into shards: small self-complete mini-databases with separate data, and then map the transactions to the shards so that each shard sees just a portion of the total set of transactions. Here we should get linear speedup in the number of shards.

d) Now, show us the formula Jim derived, and define the terms it uses. Tell us what it "means".

*Gray's formula is that if N is the number of servers and T the number of active transactions, we should expect overhead to rise as N^3T^5. An example is that this tells us that if we double the number of servers, N, and also send twice as many transactions, T, then we should expect a slowdown of 8*32 = 256x.*

e) Suppose that we have a transaction that touches objects that live in different shards. For example, it reads A which is in shard X, then uses what it read to update B, in shard Y. What are we supposed to do when we follow Jim's advice? Will this impact the normal transaction guarantees? How?

Basically we have to break our transaction into a set of parallel (or in some cases, sequential) but separate transactions, one per shard. For example one transaction could read shard X and learn A, and then the second could compute B and write it to shard Y. We would lose standard ACID guarantees.

Q2: Paxos Protocol. 5 parts.

Initially, for parts (a), (b) and (c), focus on Paxos with a fixed set of servers: N=3, Q_W=2 and Q_R=2.

a) Paxos was created as a solution to the state machine replication model. Define the model. Why can't we run Paxos with Q_W = N?

With Q_W=N we wouldn't be able to tolerate even a single slow or failed server.

b) Why can't we just use Q_R=1?

In Paxos an update might run on just some of the logs, leaving gaps (missing updates) in others that were inaccessible at the time of the update. As a result we have to merge logs into a single complete log, and for that purpose need Q_R large enough to overlap with all possible prior updates. Q_R=2 addresses this issue: with two logs, we can see every committed update in at least one of them.

c) Are there other possible values of Q_W and Q_R for this case of fixed N, with N=3? Explain.

Not if we want tolerance of 1 failures. We need $Q_W+Q_W>N$ and $Q_W+Q_R>N$, and only having them both set to 2 will work for N=3.

Now for (d) and (e) suppose that Paxos has been deployed in a virtual synchrony membership context:

d) Virtual synchrony reports a new view if a process crashes or freezes up. Why does this enable Paxos to run with $Q_W = N$? Explain.

With virtual synchrony, we reconfigure to drop failed servers, then repair them later using state transfer. As a result we can update every log because we know that every process is healthy during the epoch.

e) Virtual synchrony introduces the idea of state transfer. What state would we transfer to a joining process if we combine virtual synchrony with Paxos? Explain.

We would use Paxos learning to merge the remaining logs and then transfer this complete log to the joining process. If we ran Paxos with Q_W=N, we can just transfer some single log without merging.

Q3: Some true/false questions. Put "T" or "F" in the box on the left to show your answer.

T/F	Assertion
F	a) A stateless program needs to be coded in a functional language, like O'Caml because any variables to which values can be assigned while the code is running are a form of state.
T	b) When we use state machine programming in the cloud, we have to implement logic to store and update the state, a mechanism to trigger state transitions, and the action that each state transition will initiate.
T	c) Serialization of an object containing child objects requires design decisions such as whether each we should have one key-value pair per child object, versus recursively serializing child objects as part of the serialization of the parent object.
T	d) If a sender machine and a destination machine use different CPU types (like Intel and AMD), data sent in messages will need to use a standard representation for objects like ints or doubles, resulting in overhead to translate in and out of the message format.
F	e) The Byzantine failure model is used in Paxos libraries and systems like Derecho/Cascade
F	f) To ensure availability even with 1 failure we must run Paxos on at least 5 servers
T	g) A consistent cut is a set of instants in the timelines of the processes in a system, which could have occurred simultaneously in real-time.
F	h) RDMA runs Paxos in the network routers.
F	i) With Lamport's logical clocks, we can conclude $A \rightarrow B$ iff $LT(A) < LT(B)$
T	j) With vector clocks, we can conclude $A \rightarrow B$ iff $VT(A) < VT(B)$

Q4: Consistent snapshots. Suppose that a banking application supports deposit, withdraw and transfer operations. Just like in homework 2, the bank is physically implemented by a sharded DHT for scalability, and different accounts could be on two different shards.

A transfer is done this way: using reliable message passing, the transaction is "sent" to the shard holding the source account, and it starts there by verifying that there is enough money in the account, then deducting the transfer amount. Next, again using reliable messaging, the transaction is forwarded to the destination account. Once it arrives, it is executed and will add the transfer amount to this account.

a) Lamport defined the "happens before" relationship, $\rightarrow$, in such a way that if event a might have caused event b, $a \rightarrow b$. In our bank transfer operation, would it be correct to say that if w is the operation that withdrew money and d is the operation that deposited the transfer, $w \rightarrow d$? Explain.

The transfer request must first ensure that the source account has enough money, and only then can run the deposit. Thus the transfer request r happens before the withdraw w, which happens before the deposit, d.

b) Suppose that b is included in a consistent snapshot, and a $\rightarrow$ b. Would we expect a to be included too, or might a be omitted from the snapshot? Explain, giving an example from the bank to help us visualize exactly what this issue is about.

Yes: We can express this by saying that a consistent snapshot is closed under potential causality. For example, if a snapshot includes a deposit for a transfer, it must also include the withrdraw operation.

c) We run a consistent snapshot algorithm. After it terminates, we have a set of files that represent the output – they constitute the actual snapshot. How do the numbers of files relate to the number of bank servers and network connections amongst them? What will the files contain?

There will be files to hold a checkpoint for each server, and also a checkpoint for each connection.

d) Given a snapshot, we want to compute the total amount of money the bank is managing. Describe how we would calculate this total amount from the data in the snapshot.

We can scan the checkpoints to find the current status of each account, then need to scan the channel data to look for deposits that are still in flight. Adding these amounts will give us the total.

Q5: Stateless first tier. In the cloud we use a *stateless* layer to handle client requests.

a) If we have N clients currently making requests, how many stateless functions will be running?

Normally, N: 1 per client.

b) Define *elasticity* and explain why the first-tier is considered to be an elastic cloud layer.

It means that we vary the number of instances to handle varying load. The first tier has one instance per client, so it is highly elastic.

c) Suppose that we want to keep a count of how many requests of a particular kind each customer makes, in order to charge for them (for example, in a game we might count purchases of extra strength points, or special weapons). Since the first tier is stateless, where would this state be kept?

These counters would probably be in key-value tuples held in a key-value store, but they could also be maintained in a scalable file system or a database.

d) Do we need locking to implement counters of the kind arising in (c)? Justify your answer.

We need some form of protection against concurrent state updates that might interfere with one another, but in class we learned that KVS systems can sometimes provide this with versioned put (a form of compare-and-swap), in which the stateless function reads the current counter, adds to it, but then does a put that also specifies which version it expects to find. The put will be ignored in the version is different, and the request can then loop.

e) Explain what a *container* is, and why container virtualization is popular for first-tier servers.

A form of virtualization that virtualizes individual processes and any local data they require. Containers are very cheap to launch – much cheaper than full VMs – so we often use them in the cloud.