CS380D Distributed Systems Review Questions

Zeyuan Hu, iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2018

1 Jan 19 Review Questions

1. **How do the 8 fallacies of distributed systems differ between designing a system over the world wide web (say between India and the US) and between servers inside a single datacenter?**

   (a) The network is reliable. The network for a system that spans globablly may be highly likely unreliable. However, for a system resides only in a single datacenter. We can think the network is reliable. However, there are still chances for network outage even in a single datacenter.

   (b) Latency is zero. The latency for a global system is necessarily non-zero and large. However, for servers in a single datacenter, we can assume that the latency is very small but we still cannot assume that the latency is zero.

   (c) Bandwidth is infinite. No matter which scenarios, the bandwidth is limited.

   (d) The network is secure. For servers within a single datacenter, if we have built a good firewall to isolate the network inside the datacenter from outside, we are likely to have a secure network. However, for a system that relies on a global network, we need to put much more effort to ensure the security of the network.

   (e) Topology doesn't change. We can assume this one for servers in a single network. However, for a system that utilizes multiple datacenters, the topology may change over the time because we may switch to other datacenters during the time.

   (f) There is one administrator. In either scenarios, we cannot assume we have one administrator. This is especially true for a global sytem that needs to provide a 24x7 service.

   (g) Transport cost is zero. Similar to the latency, we cannot assume the transport cost is zero in either cases.

   (h) The network is homogeneous. For servers in a single datacenter, we can assume this one holds. However, for a global system, this assumption can never be true because the network infrastructure is necessarily different from country to country.

2. **What would you expect to be different between Tandem's breakdown of outage reasons and the breakdown of outage reasons for a modern service, say Amazon EC2?**

   I think the administration outages still are the major source of the outage for both Tandem and a modern service given that the notable outages from last year (GitLab[1] and AWS[2]) are due to the human errors. In addition, the software outage may also take the second place for the outage reasons given the complexity of the system has increased. However, I would expect that the outage reasons due to the hardware and environment may no longer hold for a modern service nowadays. The stability of hardware has increased since Tandem's paper and for a large scare sevice like Amazon EC2, people use multiple datacenters to provide redundancy instead of multiple servers reside in the same datacenter. Providing service backup using multiple datacenters across different locations make the outage due to power, communications, and facilities very unlikely.

2 Jan 26 Review Questions

1. **Jeff Dean's talk mentions several patterns in distributed systems (Single Master, Many Workers; Canary Requests; ...). Describe how some of these could be applied in a current system not described in the slides.**

   Amazon EC2 system adopts "Elastic Systems" design pattern described in the talk. A user can rent their own server(s) on the Amazon EC2. However, it is hard for user to forsee the exact peak load for his usage. It is expensive to rent multiple servers in the anticipation of high workload. On the other hand, the current server instances may suffer from the sudden jump in the workload, which can only be handled by renting additional servers. To handle this type of user's needs, Amazon EC2 system is designed to have the elastic capability to handle the workload: Amazon EC2 can automatically shrink capacity when there is not much workload and grow capacity as the workload grows. In addition, Amazon EC2 as a cloud platform can have huge amount of users' service requests. It is unfeasible to have one single machine to handle the requests from the beginning to the end. Thus, "Tree distribution of Requests" and "Multiple Smaller Units per Machine" design patterns can be helpful here. We restrict the amount of requests can be handled by each machine (i.e., 10-100 requests/machine) and we distribute the requests (e.g., create 100 instances) to

[1] `https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/`
[2] `https://aws.amazon.com/message/41926/`

50 leaf machines and with each leaf machine only needs to perform small amount of tasks (e.g., create two instances) . Those 50 leaf machines can connect to another tree sturcture parents and root machines to collect the status of tasks progress requested by the user.

2. **Provide an example of modern day events that are related by the happens-before relationship and an example of events that are concurrent. What are the possible relationships of the logical clock times for the events?**

   The preference ranking on goods by a consumer. A consumer prefers A over B is denoted as $A \succ B$. This is similar to the "happen-before" relationship in the sense that consumer's more favorable object (i.e., A) always comes before the less favorable object (i.e., B). "Concurrent" concept is also perserved in the preference ranking in the sense that if a consumer is indifferent between two goods A and B, we can denote them as $A \sim B$. In other words, if a consumer cannot tell if he perfers A over B or perfers B over A, we say he is indifferent between two goods. Mathematically, $A \sim B$ when $A \not\succ B$ and $A \not\prec B$, which mimics concurrent definition exactly. The "logical clock times" for preference ranking is utility function where we assign numbers to each good based on the consumer's preference ranking. We have $U(A) > U(B)$ if and only if $A \succ B$. In words, if the utility function value of A is greater than B, we know that the consumer prefers A over B. Reverse statement also holds. If two objects' utility function values are equal to each other (i.e., $U(A) = U(B)$), we know that the consumer is indifferent between two goods (i.e., $A \sim B$).

2.1 Feedback

While $C(A) = C(B)$ would imply that events are concurrent, the events being concurrent doesn't actually imply $C(A) = C(B)$. For concurrent events A and B, any relationship between $C(A)$ and $C(B)$ is valid. The similarity to a utility function would hold better for vector clocks, where the happens-before relationship can be inferred from the clock times.

3 Feb 02 Review Questions

1. **The state machine approach paper describes implementing stability using several different clock methods. What properties does a clock need to provide so that it can be used to implement stability?**

   The clock should be able to assign the unique identifier to each request whose issuance corresponds to an an event. In addition, the clock should ensure the total ordering

on the unique identifiers. In other words, the clock has to satisfy: 1) clock value $\hat{T}_p$ incremented after each event at process p 2) Upon receipt of a message with timestamp τ, process resets the clock value $\hat{T}_p$ to $\max(\hat{T}_p, \tau) + 1$.

2. **For linearizability, sequential consistency, and eventual consistency, describe an application (real or imagined) that could reasonably use that consistency model.**

   - linearizability example: transactions on RDBMS enforces linearizability in the sense that the read/write operations on a table in RDBMS have to be "atomic". Once a tuple is modified and commited, the changes to the tuple become visible to the following read immediately. A real life example is that when I deposit the money into an account, I want to see the the money reflect the latest deposit immediately in my bank account.
   - sequential consistency example: isolation requirement for RDBMS. When two transactions that are modifying the same table, each operation within the transactions are executed in the order they are issued. However, the operations from two transactions may be interleaved (i.e., under "Read uncommitted" isolation level). Another example is one person A issues "unfriend", "post" operations and the other person B issues "scroll the facebook page". The operation order from these two people are kept: "post" never goes before "unfriend" but operations may interleave: "unfriend", "scroll the facebook page", "post" instead of doing A's operation first ("unfriend", "post") and then B's.
   - eventual consistency example: high available key-value store like Dynamo. The order status display service may not see the user's update on the shopping cart but eventually those updates can be seen by every services. Another example is when you like a post, the other people who can see the post may not see your "like" immediately in their facebook page but eventually, they will see your "like" on the post.

3. **What's one benefit of using invalidations instead of leases? What's one benefit of using leases over invalidations?**

   One benefit of using leases over invalidations can be seen from the following example: suppose we use the eventual consistency model and a write is waiting on invalidations. Invalidation has negative impact to the system performance because the user cannot use the cache due to the invalidation but it is ok for client reads the old value because

of eventual consistency. However, for the lease, if the cache is hold by the lease holder and still within the lease term, user can still read the data from the cache. This scenario also indicates the benefit of the invalidations over leases in the sense that the user may not get the latest updated value since they can read the old value from the lease-holder-protected cache directly. Thus, for the linearizability consistency model, we cannot use the leases because the write may not be approved by the leaseholder and there might be a read that happens immediately after reading from the out-dated cache. This violates the linearizability, which guarantees that reads always reflect the latest write. Thus, there will be a delay in write, which hinders the system performance. However, for eventual consistency model, leases is more favorable than invalidations.

4 Feb 09 Review Questions

1. **What are differences between strong leases and weak leases and the guarantees they provide?**

   One difference between strong leases and weak leases is that strong leases do not allow the write while the lease held by the cache hasn't expired or the leaseholder hasn't approved the write. If one object is covered by multiple strong leases, the write can happen if all the leaseholders approve the write or all the leases covered that object has expired. This mechanism guarantees that there will be no conflict of write to the object in cache. However, that's not ture for the weak lease. Weak leases allow the writes and reads simultaneously. Same object can be owned by several weak leases with different expiration time. This implies that some leases that are expired allow the writes while other non-expired leases allow the read only. Thus, we may get the conflict on the object in cache. In the write-back cache, the conflict is resolved on the server side. In addition, for the weak leases, the server doesn't need to keep track of the leases granted but that's not true for strong leases. Lastly, strong leases guarantess a strict consistency while the weak leases have a weak consistency. As the consequence, weak lease has better performance than the strong lease as we don't need to queue the writes say in the write-through for the weak lease.

2. **Use the types of anomalies allowed to per-object sequential consistency and read-after-write consistency to compare the two consistency models. Is one stronger? If so, which one?**

   For anomalies in vetrices, both per-object sequential consistency and read-after-write

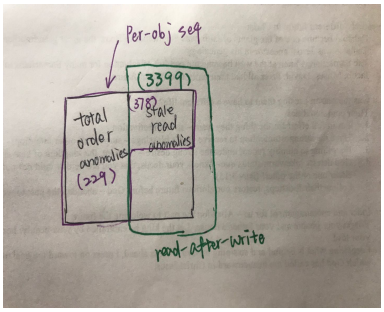

Figure 1: Illustration of statistic relationship between two consistency models

consistency have stale read anomalies. In addition, read-after-write consistency also has total order anomalies whereas the per-object sequential consistency doesn't. If we take a look at the actual statistics, per-object sequential consistency contains 607 total anomalise including 378 stale read anomalies and 229 total order anomalies. Read-after-write consistency contains only total order anomalies. Since there is overlap between read-after-write consistency anomalies and per-object sequential consistency anomalies but one of them is not the subset of the other. Thus, we cannot determine which consistency model is stronger. The above text is illustrated in the Figure 1 as well.

3. What performance vs consistency tradeoffs are made by Yahoo's different read types (read-any, read-critical, and read-latest)?

`Read-any` emphasizes performance over consistency.Since the latest read doesn't need to reflect the latest write (i.e., we can still read the stale version of the record even after a sucessful write), we can maximize our performance by return a valid version of record from the records' history. However, for `Read-critical` and `Read-latest`, we emphasize the consistency over the performance. `Read-critical` requires the returned version has to be strictly newer or the same as the specified version and `Read-latest` requires the return has to be the latest copy of the record that reflects all the previous successful writes. These two types of reads require the squential consistency for the record and at the same time, if the local cahced copy becomes stale, we need to fetch a new copy from remote replica, which implies that the performance of these two APIs will be lower than `Read-any`'s. In addition, since `Read-critical` can return any version of the record as long as the requirement is met, there are a fraction of record history that satisfy the constraint and any one of them can work. However, `Read-latest` can only return the latest one. Thus, `Read-latest` has the strongest consistency among the tree with the worst performance.