Lab2: Enforcing Causal Consistency in Distributed System

CSE 513 - Distributed Systems
Due Date: November 11

1 Introduction

In this lab, you will design and implement a simple distributed system that follows causal consistency. The system has multiple geographically replicated data centers. Each data center has a data store, and the data center supports many clients to read/write the data store. The data store is replicated at multiple data centers to improve performance; however, it may introduce data consistency issues and the goal of this lab is to enforce causal consistency.

2 Causal Consistency

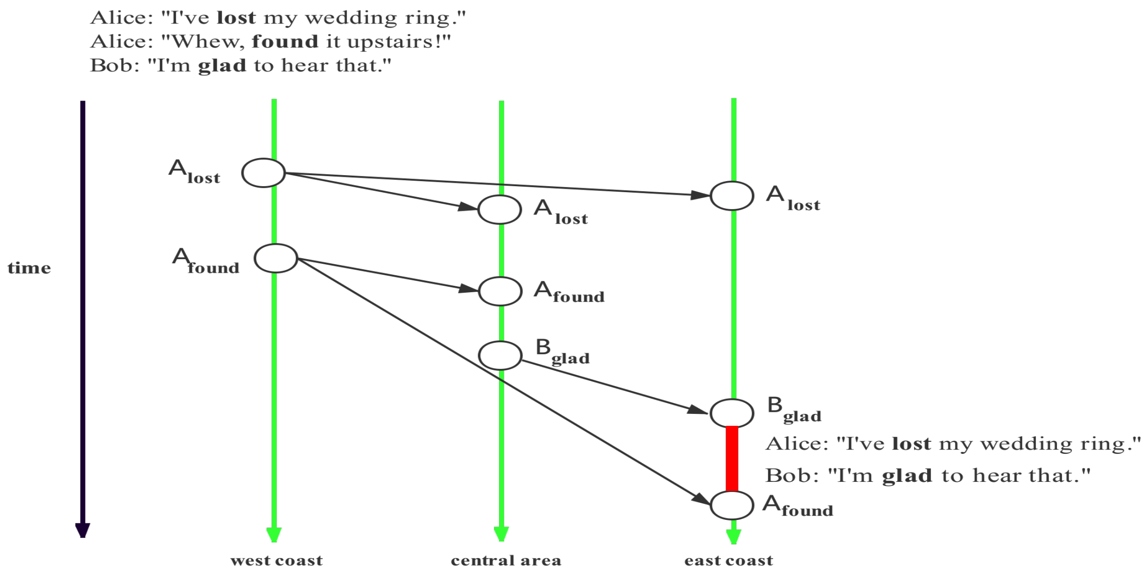
Alice: "I've **lost** my wedding ring."
Alice: "Whew, **found** it upstairs!"
Bob: "I'm **glad** to hear that."
time
A_{lost}
A_{found}
A_{lost}
A_{found}
B_{glad}
A_{lost}
B_{glad}
Alice: "I've **lost** my wedding ring."
Bob: "I'm **glad** to hear that."
A_{found}
west coast
central area
east coast

Figure 1: An example of real life causal consistency problem

We use an example to show the importance of causal consistency. In Figure 1, there are three data centers, west coast, central area, and east coast. Alice uses the data center in the west coast, and Bob uses the data center in the central area. Alice posts "I've lost my wedding ring", and then comments "Whew, found it upstairs". Then Bob reads comments via another client process at the central area, and comments "I'm glad to hear that". In the east-coast data center; however, due to the communication delay, Alice's comment A_{found} arrives later than Bob's comment B_{glad}, which makes Bob's comment look weird. The problem is because the operations in the east-coast data center does not follow causal consistency which states that Alice's comment A_{found} should happen before Bob's comment B_{glad}.

3 Achieving Causal Consistency

There are different solutions to achieve causal consistency, and we provide one solution based on dependency check. You are not limited to use this solution as long as your solution works.

The data center maintains a $dependency$ data structure for each connected client. It is a list of <key, version> pairs, where $version$ consists of two integers: $timestamp$ is the logical time for writing this key following Lamport's clock, and the $datacenter\_id$ is the data center that executes this write operation. When a client connects to the data center, the data center creates an empty $dependency$ for this client. Further read and write operations will update this $dependency$. Replicated write request will attach this $dependency$ to the remote data centers so that they can use $dependency - check$ to determine if they can process that write request or delay it due to potential violation of causal consistency.

More specifically, for a client's $read(key)$, the data center will return the $value$ of this key, and attach this new dependency <key, version> to the current $dependency$ list. For a client's $write(key, value)$ request, the current $dependency$ is attached to the replicated write request and propagated to other data centers. After local updating, the key will have a $new\_version$ number. After the write is propagated to other data centers, its dependency list will be cleared, and updated with <key, new_version>. When a remote data center receives a replicated write request, it will check the attached $dependency$. If any <key, version> in the dependency list hasn't arrived yet, it will delay this replicated write; otherwise, it will commit this update. There are two ways to create dependency:

- If a and b are two operations in a single thread of execution, then $a \rightarrow b$ if operation a happens before operation b.
- If a is a *write* operation and b is a *read* operation that returns the value written by a, then $a \rightarrow b$.

Let's consider the example in Fig 1. Assume the id of the three data centers west coast, central area, east cost are: 0, 1, 2. Assume A_{lost} is $write(x, \text{“}lost\text{”})$, A_{found} is $write(y, \text{“}found\text{”})$, B_{glad} is $read(y, \text{“}found\text{”})$ and then $write(z, \text{“}glad\text{”})$. There may be other ways to generate these read/write operations for this example (e.g., using the same key instead of multiple keys). How to generate these read/write operations is not required for this lab; i.e., you will be given these read/write operations as input. In this lab, our focus is the dependency tracking and checking. Similarly, the distributed data store can be read/write of large video files, instead of simple messages.

At west cost, after $write(x, \text{“}lost\text{”})$, the dependency list becomes <x, (1,0)>. After $write(y, \text{“}found\text{”})$, the dependency list becomes <y, (2, 0)>. Then, after the central data center receives the replicated writes A_{lost} and A_{found}, Bob performs $read(y, \text{“}found\text{”})$. Then, the dependency list will be <y, (2, 0)>. When Bob propagates $write(z, \text{“}glad\text{”})$ to the east cost, it attaches this dependency list <y, (2, 0)>. When $write(z, \text{“}glad\text{”})$ arrives at the east cost, it checks this dependency. Since <y, (2, 0)> has not arrived yet, it will delay this write operation. Later when A_{found} arrives, the dependency condition satisfies, and then we can see A_{glad}.

4 Requirements and Test Cases

You can use this example of Figure 1 as the initial setup, but your code should be general. You can use one machine to simulate the whole process; i.e., data centers and clients have the same IP address, but different port addresses. For a client, after connected to its data center, it can read and write. The data center should track the dependencies of all clients connected to itself. It will also send replicated updates to other data centers. When a data center receives a replicated write, it should check the dependency before committing the write. It may delay the update until the dependency is satisfied.

The data center needs to intentionally delay the sending of the replicated write request for a random/preset interval to simulate the communication delay. To achieve this, you may consider to let data center create a thread, which first sleep() for a random/preset time before sending the updates.

We still use the above example. At real life time 0, Alice and Bob's clients connect to the data center 0 and 1. Alice sends a write(x,"lost"). At time 5, Alice sends a write(y, "found"). At time 6, Bob performs a read(y), and then write(z,"glad"). You can intentionally delay the propagation of write(y, "found") to east cost, so that it is received after $write(z, \text{``}glad\text{''})$ has been received. Your code should print out how the data center checks the dependency relationship, and how it delays the update. Your code should be general and can handle more complex test cases, for example, a write may depend on multiple operations. You will be given these read/write operations as test input, and how to generate them is not required.

There are some details that this lab does not consider, for example write conflicts and garbage collection. When two write operations on the same key happen concurrently (namely, there is no causal relation between them), there is a write conflict, which means your data centers may have inconsistency copies of this key eventually. To solve this, you may consider a simple "last-write-wins" policy. In this policy, the data center always keeps the newest $version$ of this key. With garbage collection, the $dependency$ structure can be reduced in some case. For example, when a key's $version$ listed in the $dependency$ has already arrived at all data centers, we can remove this <key, version> pair from the $dependency$ list. You don't have to consider write conflict and garbage collection in this lab.

You can form a group of at most two people for this lab. For example, your partner may be good at socket programming and you are good at implementing the causal dependency part.

5 What and How to submit

Your submission should include:

1. A detailed description of the system you built, along with protocol specifications.
2. A description of the structure of your program.
3. A description of which part works and which part doesn't.
4. A sample output of your program. Please notate the output so that it can be understood easily.
5. Well-commented source code.
6. A system description with makefile or compile command. Note, we only accept documents in pdf or txt format.

Then,

1. Compress all above materials in a tar file or zip file named lastname_firstname_2.zip(or .tar).

2. Submit it through Canvas.
3. Please prepare a demo for your project.