A Delay and Disruption Tolerant (DDT) Transaction Key-Value Store

Yash Patil

1 Introduction

A Delay and Disruption Tolerant Transaction (DDT) key-value store uses asynchronous replication to provide a highly available key-value store in the presence of network partitions and intermittent network connectivity. This key-value store combines transactions implemented with RIFL along with Bayou. Entities in this system are users (those who interact with DDT clients), DDT Clients (the application instance itself), and replicas. This DDT key-value store meets transactional/strong consistency for globally committed transactions, but transactions that are tentative have weaker consistency guarantees (discussed in section 3). In order to resolve conflicts between transactions, the system uses optimistic concurrency control (OCC) along with version vectors for stronger guarantees.

2 User Requests

By the property of high availability, a user that is able to contact a DDT client will receive a response, regardless if that client is partitioned from the rest of the replicas in the network. A key assumption we make is that once a user begins a transaction with a specific DDT client, it will only conduct operations that are part of the transaction with that client, until the transaction has been completed or the user has crashed. This creates a sticky session between a user and a DDT client. If a user were to lose connection with the DDT client before a transaction has been marked as tentatively completed, it must re-transmit all operations that were part of the transaction. Just as RIFL specifies, each user will send a request with a clientID and a sequenceNum. The sequenceNum must be monotonically increasing. If a user were to crash and come back to the exact same DDT client, the user may request previous tentative transactions and the list of operations that were part of the transaction that was ongoing. In addition, to ensure that the sequenceNum remains monotonically increasing, the client can save the previous sequenceNum to disk. However, if the user were not to come back on line to the same DDT client within a specific time interval (ie. timeout of the session), then the DDT client will abort the transaction.

3 DDT Clients

DDT Clients are replicated (can be done through paxos); as such, if a single instantiation of a client were to fail, the DDT client would be alive. If the entire replica set were to crash, all transactions that were being processed will be lost. Trivially, all committed state (tentative and global) is written to disk, so that data would not be lost. DDT clients will also be involved in anti-entropy exchanges, so that they can see tentative transactions and not create conflicts with them.

3.a User Requests

A transaction starts with a *begin* operation and is completed (not globally committed) with the *commit* operation. A *begin* operation is written as a *begin* record in the log. Each operation that the user sends to the DDT client will include a clientID that is provisioned by a coordinator, sequenceNum, and operation. When an operation is received by a DDT client, the client first checks to see if it has been received before. If so, the DDT client will find the completion record associated with that sequenceNum and will

return the RPC result and if the transaction associated with that result is ongoing, temporary, or completed.

Multiple users can be communicating with a single DDT Client. As such, DDT Clients must ensure that ongoing transactions from users that are communicating with them do not result in inconsistent states. This can be done by the DDT client holding locks on objects that are being updated (*Note: this lock is being held by a single DDT client and does not block other DDT Clients from acquiring the same lock for another transaction*). In addition, DDT clients must also ensure that transactions and their operations do not conflict with other tentative and globally committed transactions. In the context of globally committed transactions, the DDT client must only ensure that the new transaction does not enter a state that is inconsistent with regards to the invariants of the system. In addition to maintaining a consistent system, if the operation in the current transaction depends on an operation of a tentative transaction, then it is noted in the operation completion record. For comparing operations to both tentative and confirmed transactions, if the version number is greater than the version number a DDT client expects, then the transaction is aborted. This can happen if there is a read that gave one version number and then when the object is written the version number has incremented due to a temporary transaction (*Note: we assume no blind writes in this system to maintain conflict serializability*). With this system, given a read on a key, it is possible to return multiple values with different timestamps and versions for reads on keys that are part of tentative transactions. Thus, we can return multiple values for a read (similar to DynamoDB); however, it most likely that the user will find the most recent write to that key relevant.

Once the operation is known to not produce any conflicts, the DDT client will then create a completion record with the client ID, sequenceNum, RPC result, object, timestamp, status (ongoing, tentative, confirmed), and a list of temporary transactions that the operation has observed a write from, which allows the system to resolve conflicting transactions on entropy-exchanges. This completion record is stored and the operation update is written to log. The DDT Client can identify potentially conflicting transactions through maintaining a multi-map of the key that is being operated on to a list of completion records.

3.b Commits

Transactions are tuples of completion records. When a commit operation is done, it is necessarily the case that the transaction will not create an inconsistent state with any known tentative or committed transactions to the DDT client. When a transaction is committed, the DDT client will create a commit record in the log as tentatively committed. It will also execute all tentative writes that were in the write buffer (note them as being tentative), and change the status of all operations in the commit tuple to being tentative. When the transaction is tentatively committed, the DDT client will let the user know of this tentative commitment. In addition, it will send the tentative commitment to all replicas that it can make contact with. If the DDT Client cannot reach any replicas, then the transaction will remain tentatively committed until it can make contact with a replica and receive a reply about whether it is committed or aborted. The DDT Client can also continue to serve replicas. In addition to a tuple of completion records, transactions will have an ID of the DDT Client it came from, a timestamp, and a union of the list of transactions that the operations are dependent on (taken from the commit record of each operation). Once a transaction x is tentatively committed, if it is dependent on another tentative transaction y, then the final commit sequence order must observe this causal order between the two transactions if both transactions are not aborted.

A DDT Client will mark a transaction as globally committed after it receives transactions to be committed from a replica (discussed further in section 4). Once a DDT knows that a transaction is globally committed, it will notify the user that the transaction has been committed. DDT clients will garbage collect transactions that are known to be globally committed and acked by the client. DDT Clients must wait for the ack because if the user does not update to the globally committed state, then it could have many transactions aborted because it doesn't know about a certain state.

DDT Clients will transfer transactions to replicas that it can make contact with. When sending these transactions to replicas, replicas will also reply with transactions that have been globally committed and aborted. DDT Clients must observe the commit sequence order that the replicas send. If a transaction is

aborted, the DDT Client will then go and rollback that transaction in addition to any other tentative transactions that it has knowledge of that depended on it (ie. observed a write from it). In this case, the DDT Client will then delete the completion record along with rolling back the database state by looking at the log.

The DDT Clients observe a consistent state with regards to the globally committed operations at all times. This is based on the assumption that the DDT Clients are transitively connected to the Primary. As such, the DDT Clients are eventually consistent in that they will eventually converge to the same state, and the transactions that are globally committed observe transactional consistency. As mentioned before, DDT Clients are replicated, so if they fail there should be another instance that can communicate with users and replicas. If the DDT Client is disconnected, tentative transactions can still commit, but the user that initiated the transaction with that DDT Client will have to communicate with another DDT Client to get an update on the transaction.

4 Replicas

Replicas are the lifeblood of this system. Replicas serve to propagate transactions from the DDT Clients to other replicas and eventually reach the primary. Using Bayou gives us high availability; however, it does not give us an exact protocol of resolving conflicts with transactions (it is able to do so with general operations using vector clocks).

4.a Anti-Entropy Exchange

The vector clock/version array in this system will consistent of a list of DDT Clients and their most recently sent transaction. When two replicas make contact with one another, we can look at the vector clocks and see what transactions have timestamps that are greater than what the replica knows about for any other DDT Client. Once this anti-entropy exchange is completed, the replicas must first handle all known-committed updates based on the exchanged commit sequence number. The replica will commit transactions in the known committed order, applying these updates to its log and database. If a replica can communicate with a client that has a committed transaction, then it will notify the DDT Client as such.

The replica will then detect which transactions have conflicts between them. If a transaction depends on another transaction (eg. the DDT Client knew of another tentative transaction), then the replica must obey that causal order between two transactions. There are multiple ways that conflicts can arise in this system, and replicas will follow a deterministic protocol for choosing which transactions to abort. Replicas can detect transaction conflicts by looking at the version numbers of operations in transactions that modify the same key and seeing if they conflict with one another. If they do conflict, the replica can then check if a tentative transaction knew about another (ie. they are causally ordered) and they will not abort. Otherwise, the replica will abort the transaction with the higher timestamp. This is done under the assumption that the transaction with the lower timestamp may have traveled further and potentially to the primary. If the timestamps have the same transaction time , then the transaction is aborted based on the ID of Client DDT.

The replica can now process transactions. If the replica sees a new transaction that is causally ordered with another transaction, then the replica will update its log and completion record as such, including a potential roll back and roll forward of state to meet this causal order. The replica must also abort transactions that observe writes from the aborted transaction. These are identified by the list of transactions that the transaction is dependent upon (as explained in section 3). If the replica can communicate with the corresponding DDT Client, then it will notify it of a transaction abort, only if it is finalized (described in section 4.b). The replica will order the transactions that it has according to first the causal order specified and then can order them based on the timestamp and DDT Client ID. The transaction order on the replica is now conflict-serializable with respect to the transactions that it knows of. Other replicas may have different orderings given the transactions they have. As replicas learn about commitments, they may have to reorder updates in their log that they set in the tentative commit in order to observe the commit sequence order that is set by the primary. For example, the replica may learn about tentatively committed transactions 1 3, but not tentatively committed transaction 2. If commit sequence order denotes a total order of 1, 2, and

then 3, then the replica must roll back its state and reapply 2. Replicas with two different orderings will eventually converge to the same state because the primary will create a total order of transactions. Replicas that are transitively connected to the primary will make progress and be eventually consistent.

Replicas will maintain a list of aborted transactions. These aborted transactions are shared on anti-entropy transfers to update other replicas as to aborted transactions (discussed further in section 4.b). With this in mind, when anti-entropy exchange is done, the replica will first commit, then abort all non-aborted transactions including cascading aborts, and then follow the protocol outlined in the previous paragraph.

If a replica is disconnected from the primary, then it will only be able to update its commit history to replicas that it is partitioned with. These replicas can communicate with DDT Clients and propagate tentative transactions among themselves. One instance of a replica should be replicated through paxos. If an entire replica instance were to completely fail, the DDT Client may have to restart that transaction depending on a timeout (this must be set after conducting an analysis on how long it takes to commit a transaction on average, network latencies, etc., and may vary between DDT Clients).

4.b Primary

A primary is a special type of replica in this system. If a DDT Client is transitively connected to a replica, that transaction will commit globally if all of the other transactions that are in a causal order before it have also globally globally committed. In general, transactions will eventually commit as long as there exists some transitive relationship to a primary for a given amount of time it takes some n anti-entropy exchanges between replicas to to a primary to occur.

The primary first reconciles conflicting transactions that it has received from other replicas with the protocol described above, with minor changes. If there is an abort with a transaction that is already committed, the primary will not abort that transaction. If there is a transaction that is causally ordered after another transaction, but the earlier transaction has not reached the primary, then the primary will wait until it receives either an abort message or commit for the earlier transaction. Building off of this situation, a DDT Client could be making many transactions that are dependent on an initial transaction in a causal order that has not reached the primary (a replica with the transaction could be disconnected). This could cause a very large cascading abort, but is a conflict we must be mindful of in order to have high availability.

The primary will abort transactions that conflict with one another including cascading aborts, and then create an order just as replicas do. This order, however, is a total order that must be observed by all other replicas. Sets of transactions can observe a causal order in this total order, and must do so if they are doing reads on earlier writes. Once a primary broadcasts commit orders and its abort lists to the other replicas, those replicas must commit the transactions in the commit order and abort the corresponding transactions.

The primary must handle the case where a replica aborts transaction A because it conflicts with transaction B, but another doesn't see transaction A. Here, the primary will just choose whichever transaction reaches it first. Once the primary has decided on a transaction to abort, it too will update its list of aborted transactions. However, transactions that aborted by the primary are considered finalized aborts, meaning the Replica can notify the client of the abort. In a system where there are not finalized aborts, a replica could inform a DDT Client of an abort, only for that transaction to end up later being confirmed by the primary.

If a primary were to be disconnected from the system, then only transactions that have previously been globally committed and sent while the primary was connected will be able to be committed by all other replicas. Replicas can still make progress on transactions that are previously globally committed that they have not seen and for tentative transactions they have not seen. If the primary were to fail, then the system could try to provision a new primary. This would involve a Paxos-like leader election and catch up procedure, with replicas that have the most up to date transaction state. However, there are many issues that can arise from this (committing transactions that have been aborted, not seeing certain transactions, etc).

4.c Garbage Collection

We must increase the size of messages shared in an anti-entropy exchange in order to garbage collect. Lists of aborted transactions and logs can be garbage collected. Replicas will maintain a list of which transactions the other replicas have aborted. Once a DDT Client has acked an abort, that ack will be tagged onto every anti-entropy exchange. Replicas can garbage collect the transaction once it knows that all other replicas it is connected to have also aborted the transaction. Logs can also be truncated by having an omitCSN stamp that is sent between replicas.

4.d DDT Clients acting as a Passive Replica

One optimization in this system could be the DDT Client acting as a passive replica in that it does anti-entropy exchanges. This could allow the DDT Client to create more transactions that have a lower chance of being aborted; however, the cascading abort would involve much more work to handle. The DDT Client is passive in this case; it is not proactively going and making contact with other replicas unless it has to send a tentatively confirmed transaction from a user. Rather, replicas will communicate with a DDT Client just as if it is another replica. At the very least, it is advantageous for DDT Clients to see a list of aborted transactions from replicas, which will enable faster updates to the user about the status of a transaction. In addition, it will allow the DDT Client to roll back its state periodically and not create more transactions that will eventually be discarded due to a cascading abort.

5 Optimizations

In order to reconcile servers that are lagging, or those that have just come into the system, a replica can send the new replica instance of all of the globally committed transactions. A key design decision in this system is how to provision replicas and DDT Clients. If there are many DDT Clients and a few replicas, it may be difficult to. Another optimization that a replica could make is not to abort transactions based off of the DDT Client ID (if the timestamp check fails), but to do so based on the transaction that will result in fewer cascading aborts.

6 Application-Level Discussion

This DDT key-value store is highly available and provides different consistency guarantees depending on the connectivity of the network. All replicas will eventually be consistent if they have some transitive relationship with the primary. The system is available whenever a user can connect to a DDT Client and "fully" available when DDT Client can connect to a replica.

A technique that is synonymous with transactions is 2 phase commit (2PC). This method, however, does not work well in this setting given that 2PC is optimized for strong consistency. We could have DDT Clients have a 2PC protocol with its replicated instances, but this would also add unnecessary overhead. Another technique we considered using was causal multicast for locks. However, this results in deadlock for DDT Clients to get if a DDT Client with the requested lock was not transitively connected to others. Finally, we did consider using sharding with regards to how the users communicate with DDT Clients, but shards would have to constantly be reassigned to balance load if users had intermittent connections with the DDT Clients.

DDT does not violate CAP. First, it is important to note that CAP does not strictly mean choosing 2 of consistency, availability, and partition tolerance. Rather, these properties are continuous. This system is highly available with and without partitions. What our system tunes is consistency. More specifically, the system does not have to provide strong consistency; rather, it is eventually consistent. When there is a partition, especially with a primary, it will take longer for all the transactions to be globally committed or aborted. When the partition heals, we are able to recover because of roll backs in our system.