DDT Transactions

The goal of this exercise is to design a transactional key-value store that is “delay and disruption tolerant” (DDT). DDT transactions use asynchronous replication to provide service even when network connectivity is intermittent. In this model, a locally committed transaction must be confirmed with other replicas, and might revert if a conflict is detected. Thus DDT might also mean “delayed or disrupted transactions”.

DDT transactions execute in the usual way for client-server transactions, e.g., using a subset of the RIFL API discussed in class (begin, put, get, commit). Each DDT client (an application instance) has a co-located full replica of the database. The DDT commit operates on the local replica only, and global commit is asynchronous. Replicas learn of remote transactions and make progress as connectivity permits, and eventually converge to the same database state.

Specifically, after a local commit on transaction T returns success, T is tentative until the system confirms T. The system might also revert (abort) T unless/until T is confirmed. The system confirms global commit for transaction T after it determines that T is stable: there can be no transaction S that commits earlier than T in the final serial order and where S conflicts with T.

For availability, any writes of a locally committed T are visible after commit in the usual way, on any node that knows of T, even while T is tentative. Each transaction observes a consistent database state that results from a conflict-serializable schedule of all committed transactions known to that client, including both the tentative and confirmed transactions. This choice is a tradeoff in that it makes DDT vulnerable to cascading aborts: if a tentative transaction S aborts, then any later transaction T that observed a write from S must also abort (transitively).

DDT transactions preserve conflict-serializability of any transaction ordering observed by a client. If the system discovers that a transaction S conflicts with another transaction T, then at least one of S and T aborts, and the system notifies the client. You may find the transactional concurrency control scheme used in RIFL to be useful to detect conflicts.

You may assume that the configuration of the replica group (members, addresses, roles) is fixed and known to all participants.

As with the midterm exercise, this exercise is open-ended. Your mission is to identify the essential issues and propose a workable solution, applying concepts and techniques from the course as needed. Your submitted proposal is a PDF document of at most 6 pages. You may organize the document and present your design in your own way. Here are some questions to consider:

- How do transaction records propagate among the replicas? What information is included with each transaction record?
- Can transactions commit globally if some node is failed or disconnected?
• How to ensure that clients observe consistent states at all times, and that replicas eventually converge to the same final state? (Presuming that they are reachable from one another.)
• How is the transaction ordering determined? How are transactions ordered on a replica that knows of only a subset of transactions?
• How do replicas learn of global commit (confirmation) and abort?
• What properties does the transaction ordering require, and why? For example, is the transaction ordering always a causal ordering?
• Under what conditions or applications is such a system useful (safe, live)? Are additional assumptions or restrictions required?
• Are there techniques that seemed promising but that you chose to avoid for reasons of cost and/or lack of necessity?
• Does DDT violate CAP? It assures transactional consistency, and yet is highly available despite the possibility of network partitions.