ACID Transactions in Crux

Kojiro Murase
Yale University
Team: Christine Hong, Sage Price
Advisors: Bryan Ford, Jose Faleiro

April 30, 2015

Abstract
Distributed systems are in widespread use due to their reliability, performance, and modularity. Most distributed systems, however, suffer from latency issues whereby the worst-case latencies across the network are transferred to even local interactions. Crux is a novel general framework to create locality-preserving deployments of distributed systems by deploying multiple instances of the distributed algorithm. We test the applicability of Crux to the widely used distributed database MongoDB. We also use MongoDB as a platform on which to show that a trading application requiring ACID capabilities can be deployed in Crux.

1 Introduction
Most distributed systems suffer slowed communication speeds due to a loss of locality. Because interactions can be routed through arbitrary nodes in large networks, interactions between nodes that are close in
the network topology can be slowed to the worst-case communication speed across the network. For instance, communication between two users both based in New York might be routed by the distributed system through a third user in London, incurring significantly higher latency or bandwidth costs.

Though such communication delays can be problematic, most high-level scalable distributed protocols fail to provide a solution. Instead, some systems provide ad-hoc solutions without strong guarantees on performance.

1.1 The Crux Scheme

A paper by Nowlan, Faleiro, and Ford describes Crux, a framework for preserving locality in distributed systems. Crux is a general framework for transforming scalable distributed algorithms into scalable and locality-preserving algorithms, at the cost of overhead.

Given an underlying algorithm $A$, Crux build a collection of instances of $A$ of varying sizes, ensuring that nodes within smaller instances of $A$ can communicate with greater locality preservation. Crux treats users as nodes in a connected graph, probabilistically assigns a level to each node, and then builds subgraphs of shortest path trees around nodes to guarantee that two given nodes share at least one subgraph.

Distant nodes are therefore able to communicate through higher-level nodes acting as landmarks, while nearby nodes should be able to communicate through lower-level local landmarks. As a result, latency between two nodes is decreased to be proportional to the distance between the nodes, while per-node overhead is increased.
2 Deploying Crux on MongoDB

Previous demonstrations of Crux have deployed the scheme on three distributed systems: the memcached caching service, the bamboo distributed hash table (DHT), and a redis publish/subscribe service. These relatively simple and transparent systems demonstrate the flexibility of Crux, but not its practical applicability to widespread, consistent, and scalable distributed applications with secondary indexes. Crux can theoretically be implemented on any given distributed database; in this paper, we demonstrate that it can be used on the NoSQL database MongoDB.

2.1 Choosing MongoDB

We explored several different NoSQL options, including Cassandra, HyperDex, and MongoDB. An initial attempt to implement Crux in Cassandra was halted by practical difficulties, as Cassandra did not allow for multiple instances to communicate through different ports on a single machine. We therefore refined our understanding of the limitations of Crux: Crux can be implemented on any given distributed system, provided that system has no architectural barriers preventing the replication of multiple overlapping databases on a single machine.

Given these limitations, we decided to implement Crux on MongoDB. Not only does MongoDB allow for multiple overlapping instances to be created, but it is also one of the most widely used available NoSQL database systems.

2.2 MongoDB Architecture
MongoDB is a document-oriented database storing JSON-like BSON documents. Though MongoDB is not by default a distributed system, it can be called as such. To support a single distributed MongoDB cluster, multiple instances of MongoDB must be called. A central configuration server manages a database storing the cluster’s metadata, one or more routing services (which is for our purposes a single service located on the same node) processes queries the cluster and routes them appropriately, and multiple shards distributed across different nodes store portions of the database.

3 ACID Transactions in Crux

ACID transactions present a significant challenge to the reliability of database systems, particularly to distributed databases. Crux complicates consistency issues by the nature of its replication of databases across rings. We demonstrate in this paper that the architecture of Crux can in fact support ACID transactions by implementing a simple Crux-transformed trading application.

3.1 Consistency Issues

Consider a simple trading application that allows users to interact with one another to buy and sell atomic commodities. Such a system could be deployed on a given distributed database quite simply; users could make their items “available” by placing entries in the database, and other users could remove those entries to “purchase” or claim these items. These transactions do not present a significant problem as long as the underlying distributed database ensures consistency.
In a Crux-transformed setting, however, a database that guarantees consistency may lose this property. Suppose one followed an identical protocol, while following the Crux scheme. Because databases—and therefore items—are replicated across the network, users may purchase items from the application even when they should not be able to do so.

For instance, suppose user $u$ initiates a request to sell one item of stone. This user will post this request to $u$’s own ring and all larger rings. Now, two different buyers, one located in $u$’s smallest ring and one in only its largest ring, may try to purchase an item of stone at the same time. Because the two buyers issue requests to two different rings at the same time, they will both be able to remove entries from their respective rings and complete their transactions. One of these transactions, however, must be invalid.

### 3.2 Solutions

To address this issue we worked with Jose Faleiro to discuss several different solutions.

One potential solution would involve resolving inconsistencies by using several databases and allowing bid posters to arbitrate. Suppose user $u$ initiates a request to sell one item of stone, and posts this request to all of its rings. Users $v$ and $w$ both want this item. Instead of taking the item, however, the users signal their interest in purchasing the item by annotating user $u$’s request in the database. User $u$ then serves as the arbiter of the sale; as $u$ iterates through its rings, it finds the annotation made by the closer of $v$ and $w$ and decides to sell to this buyer. To do so, $u$ posts a final sale entry to a second database, addressed
specifically to the desired buyer. This buyer is then authorized to remove the item.

A less unwieldy solution that we decided to implement removed the intermediary steps of contacting the seller through the database. Suppose user $u$ initiates a sale request on all of its rings. $u$ includes in its request the information needed to directly contact it over the network, such as a hostname and port. User $v$ is the first user to notice this request and want the item—likely because it is the closest such user—and uses this contact information to directly open a transaction with $u$. Because all requests are overseen by the users that post them, atomicity is guaranteed.

4 Implementation

We built a trading application built on a Crux-transformed MongoDB. To do so, we built configuration file generation from the ground up in Python, as well as a method for configuring and calling MongoDB from Python using Javascript fragments. Finally, the logic of the trading application and its API was built in Python.

4.1 Personal Contribution

My own role on the team involved overseeing the database interactions and trading algorithm of our application. To do so, I familiarized myself extensively with MongoDB, particularly with its Javascript scripting; with the Crux algorithm, particularly with its handling of requests; and with the novel trading application which we developed for this paper.

My trader.py file served as the interface for our trading application, building upon the text outputs of the configuration programs (handled
by my team). Specifically, for each node, trader.py takes different configuration files describing how many instances of our algorithm to replicate, and which other instances to connect to.

Trader.py also takes an instruction file specifying tests to be run on the application. Given these tests, the application outputs buy/sell results and timing data of interactions.

5 Discussion

Our initial tests demonstrate that Crux can in fact be applied to a noSQL database like MongoDB. Moreover, we describe here a general approach to address ACID-type transactions within a Crux framework.

6 Acknowledgments

Christine Hong and Sage Price for their invaluable work, Bryan Ford and Jose Faleiro for their expertise and insight.