Barista: A Distributed, Synchronously Replicated, Fault Tolerant, Relational Data Store

Anant Bhardwaj anantb@csail.mit.edu
Rebecca Taft rytaft@mit.edu
Manasi Vartak mvartak@mit.edu
David Goehring dggoeh1@mit.edu

MIT Computer Science & Artificial Intelligence Laboratory
6.824: Distributed Systems (Spring 2014)

Abstract

Barista is a *distributed, synchronously replicated, fault tolerant, relational data store*. It runs as a middleware service over database instances to provide an abstraction for a distributed relational data store. The data is replicated across a set of Paxos state machines to provide fault-tolerance and strong consistency. The real-time replication provides high availability; clients automatically failover between replicas. Barista supports SQL for data management. Client applications can use Barista with the same SQL code they used before.

1 Introduction

Barista is a middleware layer written over PostgreSQL [2] (Postgres). It replicates the data over multiple Postgres instances to provide fault tolerance. All writes are propagated synchronously using Paxos [7]. Barista provides strong consistency. Clients automatically failover between replicas. Barista exposes SQL for client applications.

The main contributions of this project are as follows.

A relational data store with the following guarantees:

1. *ACID:* Barista provides ACID guarantees. It can be used as a database backend for any JDBC-compliant application.

2. *Fault Tolerance and Recovery:* Barista can tolerate $\lfloor (b-1)/2 \rfloor$ faults and recover from server crashes by catching up the state of the crashing servers. It can also tolerate $\lfloor (b-1)/2 \rfloor$ disk failures and recover by bringing the new disks up-to-date.

Cross-Language support by providing Thrift bindings for the APIs: Barista APIs can be used in all the popular languages. We have provided sample client code for Go, C++, Java, Python, and JavaScript.

State Safety with ZooKeeper: All the running states of the system are stored in Apache ZooKeeper. ZooKeeper's atomic Read() and Write() APIs provide a safe mechanism for writing/reading system states. The states are necessary for recovering from various crashes/failures. Also, ZooKeeper provides an efficient way for log purging.

Evaluation with the TPC-C Benchmark: We implemented the industry standard TPC-C benchmark to evaluate the throughput and latency of client operations in a real-time scenario with our system. We describe this in detail in section 4.

Performance Optimizations: We made the following optimizations to our paxos-based protocol by implementing a version of Multi-Paxos [9]:

1. Avoid 2 round-trips per agreement by having a server issue Prepare messages ahead of time

2. Avoid dueling leaders under high client load by using a designated leader.

A Comprehensive Test Suite: To test Barista in a range of scenarios including concurrent client requests, network partitions, duplicate requests and machine failure, we wrote a set of automated test cases modeled after the tests from Labs 2, 3 and 4. The tests run on multiple Postgres instances on the same machine. The tests instantiate a number of Barista servers and clients and perform various combinations of operations including opening connections, executing arbitrary

SQL queries and closing connections. We test that the queries are applied in the same sequence across all replicas. Our test cases cover varying numbers of DB connections, different SQL queries and failure scenarios. We also test three recovery scenarios: (1) where a server restarts with disk intact, (2) server restarts without disk and (3) server restarts in the middle of a paxos agreement.

2 Implementation

Barista runs as a middleware service above each database instance. It intercepts client requests to the database. These requests include opening and closing a database connection, beginning and ending a transaction, and executing SQL queries.

2.1 Paxos Agreement

Once a client request arrives, the service initiates a paxos agreement between the replicas to get consensus on the slot-id in the paxos log for the client request. This ensures that all replicas agree on a single ordering for the client requests. The paxos state and logs are stored in Apache ZooKeeper so that in case of a failure, a recovering machine or a replacement machine can reconstruct its state.

As opposed to gets and puts, Barista supports the core operations required to interact with a database, namely: opening and closing connections to the database; starting, committing and rolling back transactions; and executing queries. Each of these operations is tracked in the paxos log. Since the presence of replication must be transparent to clients (and therefore they must not need to connect to different instances separately), we track the opening and closing of connections in the log in addition to query operations.

2.2 Enforcing the ordering on Postgres

PostgreSQL is a non-deterministic database. This can cause transactions to appear in a different order than the order agreed upon by the replicas during paxos-agreement. This is because Postgres is multi-threaded and transactions can run as different threads. The threads might get scheduled in any order and thus the commit order can be different from the order in which the transactions were submitted. To work around this, we do not allow more than one pending request on any replica. Before submitting a new query to the db, Barista makes sure the last one has returned. This is done by synchronizing database calls with a lock. This affects the performance in terms of number of client requests per second, but makes consistency and recovery easier.

2.3 State Safety

We use Apache ZooKeeper for Paxos state safety. Apache ZooKeeper provides a distributed configuration service for maintaining configuration information for large distributed systems.

For each paxos instance there is a node in the ZooKeeper with the path `/barista/paxos/machine_name/{seq_num}` = `Paxo {N_P, N_A, V_A, Decided}`. The paxos instances update the state by calling `Set(path, value)` API. ZooKeeper's `Write()` and `Read()` APIs are atomic which guarantees consistency. The following is a bit of sample code that updates the state of a paxos instance:

```
px.path = "/paxos/" + px.Format(px.peers[px.me])
...
...

if args.N_A >= paxo.N_P {
  paxo.N_P = args.N_A
  paxo.N_A = args.N_A
  paxo.V_A = args.Value
  reply.Status = OK
} else {
  reply.Status = REJECT
}

if px.use_zookeeper {
  px.Write(
    px.path + "/store/" + strconv.Itoa(args.Seq), paxo)
} else {
  px.store[args.Seq] = paxo
}
...
...
```

We also do write-ahead logging of database operations. The log is stored as a path (`/barista/sqlpaxos/machine_name/{seq_num}`) in Apache ZooKeeper. We call this log `sqlpaxos` log. The write-ahead logging allows us to run paxos agreement for multiple client queries in parallel.

We maintain an application pointer (AP) to track the last transaction that has been applied to the database for each replica. *The update to AP and the actual transaction must be atomic* – thus this state variable can't be in a log file outside the database. We maintain a table called `sqlpaxoslog (lastseqnum int)` that tracks the Application Pointer (AP) for that Postgres instance. We change each client transaction by adding the AP update as part of the transaction itself to achieve atomicity. The AP is used to recover the database state as discussed in the recovery section.

2.4 Log Purging/Garbage Collection:

When `paxos.Done()` from other peers updates `paxos.Min()` – all paxos instances in ZooKeeper with `seq_num < paxos.Min()` are purged. This is done by removing all `/barista/paxos/machine_name/{seq_num}` nodes if the `{seq_num} < paxos.MIn()`.

We garbage-collect `sqlpaxos` logs by deleting `/barista/sqlpaxos/machine_name/{seq_num}` nodes if the `{seq_num} < Global AP`. The `Global AP` is the sequence number below which all the operations have been applied on all the replicas.

The purging allows us to keep the ZooKeeper logs small.

2.5 Multi-Paxos Implementation

We present an implementation of Multi-Paxos that builds on the Paxos algorithm/system. The system automatically selects one of the paxos replicas as the leader (via paxos) who serves as the distinguished proposer for future paxos instances until a new leader is elected (i.e. the old one was detected as failed). Leader reigns are broken into epochs specified by an epoch number, a strictly increasing number that demarcates the successive leaders. Once a leader is elected, only that replica can make proposals which allows for two optimizations: 1) reduced contention when attempting to reach agreement on a particular paxos instance and 2) the leader only needs to send Accept and Learn messages instead of the going through the traditional Prepare/Accept phase of the normal paxos algorithm. Thus, in the common case where the leader remains relatively stable and reachable by a majority of the replicas for a long period of time, these optimizations allow for increased performance and shorter agreement latencies for state machines built on top of this Multi-Paxos system.

2.5.1 Normal Case Design (Stable Leader)

The core algorithm of the Multi-Paxos (MP) system is quite simple. The system starts off at instance -1 and elects a leader in this instance using traditional Paxos before processing any later instances (i.e. initiating Paxos agreement for any later instance). Once a leader is elected, the leader becomes the distinguished proposer for the Multi-Paxos system in that it is the only replica with authority to make new proposals on behalf of clients and mutate the Paxos log during its reign. Since only the leader can issue new proposals, this allows the following optimizations:

- Elimination of the dueling proposers problem encountered by normal Paxos.
- The leader only needs to send Accept messages to a majority of the Paxos replicas and can completely skip the Prepare phase (this is called FastPropose).

The correctness of the first property is obvious, but correctness for the second property is proved by the following invariants that ensure correctness even in failover and unreliable communication scenarios:

1. Leader reigns are broken into epochs, a strictly increasing number that demarcate the successive leaders.
2. The replica with the highest epoch number associated with it is currently the true leader in the system
3. All Accept messages from previous leaders (who may not know they are no longer the leader) with lower epochs will be rejected
4. Leader transitions occur one at a time (i.e. cannot jump from leader 1 to leader 10)
5. The operations that a leader can contribute to the log are the operations between its leader election operation with epoch i and the first appearance of a leader election op with epoch $i+1$ (election of the next leader)
6. After a leader with epoch i has been processed any operations that may have been agreed upon by an earlier leader (with epoch $0..i-1$) after that slot in the paxos log are invalidated and the new leader owns them.
7. The result of agreement for instance j is not exposed to the Multi-Paxos API user (via Status) until the results of instances for $0..j-1$ are all known because the result may be invalidated if a leader election was agreed upon in that range (the old leader approved it because instance agreements can be pipelined, but in that time the leader become unresponsive and so the other replicas may have initiated leader failover at an earlier slot to avoid the state where there is no leader)

These invariants ensure that leaders can only mutate parts of the log corresponding to their reign and that operations are not exposed as agreed upon until they have actually been solidified by the current leader (no leader change op occurred before a propose op that was accepted by the majority from the current leader, the leader change op would have ended its reign and made that agreed-upon op invalid). Thus, the Multi-Paxos API only exposes operations proposed by the current leader and not contended by a leader change (for correctness during leader change see the failover section).

The way these invariants are enforced is actually done by running a state machine on top of the original Paxos library (with a few modifications to support FastPropose for the leader). This state machine processes the paxos log and leader change events to ensure that no operation is exposed as agreed upon before it actually is.

Finally, since only the leader can propose new values, client requests made to non-leader replicas are forwarded to the leader for agreement using one additional RPC (in the unreliable case the system retries until it gets through to the current leader at least once and as it is retrying it can also process leader change ops so that the forwarded requests are sent to the correct leader). The reason for this is that we wanted to encapsulate the leader in Multi-Paxos and not expose it to users of the API so that they can treat it like normal Paxos. Furthermore, the additional overhead incurred by forwarding is minimal since these non-leader replicas are mostly sitting idle (just waiting for leader Accept messages and not proposing anything themselves). However, there is one difference between the Multi-Paxos and Paxos API. Since the Multi-Paxos system uses the Paxos log for its leader agreement operations, some instances cannot be used by the client state machine. In these cases, Status() will return return true,Nil in which case the user will know that the instance has been taken by a leader election op (without exposing it) and can treat this like a NOP (ignore it and move on). Generally, a major design goal of our Multi-Paxos algorithm/implementation was to keep the API the same as normal Paxos used in the labs and encapsulate leader election so that API users don't have to think about it.

The Multi-Paxos code is located here: `https://github.com/abhardwaj/barista/tree/multipaxos/src/server/src/multipaxos` (on a branch on barista).

2.5.2 Failover and Catch Up Protocol

All Multi-Paxos replicas ping the replica that they think is the leader every PINGINTERVAL milliseconds (note that some replicas may be behind and not know who the current leader is or may be pinging an old leader). Pings are also used to bring lagging replicas up to speed. In the ping request the replica sends the maximum instance that it knows about, and in the reply the leader returns the instances between that number and the maximum instance its knows about. The pinging replica then updates its instance log using the data retrieved from the leader. This is used in case the replica missed notification messages and increases the speed at which a failed replica can be brought back up to speed. Furthermore, since only the leader is sending Accept/Learn messages, maximum done values of each replica are also piggy-backed on the ping RPC. This allows the leader to calculate the earlier instance that cannot be discarded (Min()) and relay that to the replicas using the reply to the ping RPC. Finally, if a replica detects that the leader has missed NPINGs then that replica does two things:

- Sends a quick round of find leader RPCs to all the paxos replicas that return a (leader address, epoch) pair. The current replica takes the reply with the max epoch number and if that number is greater than the epoch of its current leader it pings that leader and requests all paxos instances between the last instance it processed and the max the new leader has processed and then processes all these instances.

- If the first start returned no leader then the replica initiates leader failover using paxos with an epoch number that is 1 greater than its current leader.

The first phase is to prevent wasting time on paxos agreement if a leader was selected but the replica is just behind. In which case the ping to the new leader brings that replica up to speed in one RPC. The second phase is to start leader transition and in essence lock the old leader out, because when a replica receives a Prepare with a higher epoch number, it updates its internal epoch number and stops accepting Accept messages with a lower epoch number. This is to guarantee that the old leader, if it is still alive, realizes that it's dead, and if it still has in-progress proposals, they cannot overwrite a leader change proposal that's been agreed upon. Furthermore, if the current leader already got a majority to Accept on an instance, the normal Paxos algorithm ensures that the failover does not overwrite this already accepted value. The leader failover process (both phases) stops when a leader change op with an epoch number 1 greater than the current leader's epoch is processed by the Multi-Paxos replica (i.e. the correct leader change was agreed upon). One additional benefit of this system is that if the leader gets overloaded with requests and crashes, the entire system can transfer leadership to an idle replica with a single round of paxos, thus giving time for the original leader to recover and rejoin the paxos group (in a way, distributing the load across all servers).

2.6 Recovery

Recovery from crash & restart (no disk failure): The paxos state is recovered by reading the saved states from the ZooKeeper. The ZooKeeper Get/Set APIs are atomic which ensures that the entire state can be recovered consistently. Below is a snippet that shows how we recover Paxos state for a particular slot:

```
px.path = "/paxos/" + px.Format(px.peers[px.me])
...
...
if px.use_zookeeper {
  paxo, ok = px.Read(
    px.path + "/store/" + strconv.Itoa(args.Seq))
} else {
  paxo, ok = px.store[args.Seq]
}
...
...
```

The `sqlpaxos` write-ahead logs don't have any state other than the application pointer (AP). The AP is updated as the part of the transaction itself in the database and thus can be recovered by reading the `sqlpaxoslog (lastseqnum int)` table in the database. Paxos fills holes in its log to ensure that everything after the AP can be retrieved as part of the paxos protocol.

Recovery from disk failure: This recovery is slow and requires some manual steps. To recover from a complete disk wipe out, we provide a script that copies the database data files from a `{healthy_machine}`. The recovery requires that the `{healthy_machine}` is not serving any request during the recovery because if it serves a new request it will change its state during the recovery and would lead to inconsistent data transfer.

3 Client APIs

Barista APIs are exposed as a Thrift IDL file (*barista.thrift*). Thrift [5] is a framework for scalable cross-language services development. It combines a software stack with a code generation engine to build RPC services that work efficiently and seamlessly between C++, Java, Go, Python, Ruby, JavaScript, and various other languages. A Thrift IDL file is processed by the Thrift code generator to produce code for the various target languages to support the defined data types and services in the IDL file. Although we are implementing Barista in Go on the server side, clients can be implemented in any language. We have provided sample client code in Go, C++, Java, Python, and JavaScript.

3.1 Barista Client APIs

Below is the list of data types and methods available to client applications through *barista.thrift*:

```
/* Barista constants */

// version info
const double VERSION = 0.1


/* Database Connection */

// connection parameters
struct ConnectionParams {
  1: optional string client_id,
  2: optional string seq_id,
  3: optional string user,
  4: optional string password,
  5: optional string database
}

// connection info -- must be passed in every
execute_sql call
struct Connection {
  1: optional string client_id,
  2: optional string seq_id,
  3: optional string user,
  4: optional string database
}

/* ResultSet */

// A tuple
struct Tuple {
  1: optional list <binary> cells
}

// A result set (list of tuples)
struct ResultSet {
  1: required bool status,
  2: Connection con,
  3: optional i32 row_count,
  4: optional list <Tuple> tuples,
  5: optional list <string> field_names,
  6: optional list <string> field_types
}


/* Barista Exceptions */

// Database Exception
exception DBException {
  1: optional i32 errorCode,
  2: optional string message,
  3: optional string details
}


/* Barista RPC APIs */

service Barista {
  double get_version()

  Connection open_connection (1: ConnectionParams
con_params)
      throws (1: DBException ex)

  ResultSet execute_sql (1: Connection con, 2: string
query,
      3: list <binary> query_params) throws (1:
DBException ex)

  ResultSet execute_sql_txn (1: Connection con, 2:
string query,
      3: list <binary> query_params) throws (1:
DBException ex)

  void begin_txn (1: Connection con)
      throws (1: DBException ex)

  void commit_txn (1: Connection con)
      throws (1: DBException ex)

  void rollback_txn (1: Connection con)
      throws (1: DBException ex)

  void close_connection (1: Connection con)
      throws (1: DBException ex)
}
```

3.2 A Sample Python Client

Barista RPC stubs for Python can be generated as follows:

```
thrift --gen py barista.thrift
```

Once RPC stubs are generated, a python program can call Barista APIs. Below is a sample client code snippet in Python:

```
transport = TSocket.TSocket('localhost', 9000)
transport = TTransport.TBufferedTransport(transport)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Barista.Client(protocol)

transport.open()

con_params = ConnectionParams(
    user="postgres", password="postgres",
database="postgres",
    client_id="1234567890", seq_id="1")

con = client.open_connection(con_params)
res = client.execute_sql(con=con,
    query="SELECT 6.824 as id, 'Distributed Systems' as
name",
    query_params=None)

print "\t".join(res.field_names)
for tuple in res.tuples:
  print "\t".join(tuple.cells)

client.close_connection(con)
transport.close()
```

4 Evaluation

In order to test the performance of our system, we implemented the TPC-C benchmark [3], an industry standard for comparing the performance of OLTP database systems. TPC-C simulates the operation of a wholesale parts supplier, in which a population of terminal operators executes a set of transactions against a database. These transactions include monitoring the stock level of a warehouse, creating a new order for a customer, accepting payment from a customer, making a delivery to a set of customers, and checking the status of an order. Clearly, the intent of this benchmark is to simulate a realistic real-time OLTP system.

The TPC-C specification is quite complex, so rather than implementing it ourselves, we modified an existing open-source implementation for PostgreSQL [1]. The open-source implementation uses C code to communicate directly with the database, so in order to test our replicated, fault-tolerant framework, we needed to modify this code to communicate via the Barista client instead. Fortunately, Thrift allows us to create clients in several different popular programming languages, so we created a C++ Barista client and linked it with the modified C benchmark code. In this way, we were able to compare the performance of our fault-tolerant, replicated version of PostgreSQL with vanilla PostgreSQL. The results of running TPC-C on Barista (with standard Paxos and Multi-Paxos variants) and vanilla PostgreSQL are shown in tables 1 to 3.

As expected, vanilla PostgreSQL performs better than Barista since it does not require paxos agreement for each transaction. However, the performance of Barista is still impressive given that each transaction can take 5-8 paxos agreement instances, and each paxos agreement instance can require up to 20 round-trip messages between replicas. Despite this overhead, Barista only increases latency by about 4-5X and barely decreases throughput at all. Furthermore, the average round-trip time between our virtual machines is 0.9 ms, so given the number of messages passing between machines, an added latency of a few hundred ms is reasonable. In addition, Barista is extremely fault tolerant while vanilla PostgreSQL relies on a single point of failure.

Theoretically, Multi-Paxos should perform faster than traditional Paxos due to the enhancements described (primarily the reduced contention and number of messages needed for agreement) which produce smaller latencies. Generally, for an N replica system the number of messages required for agreement is 2(N-1), N-1 Accept and Learn messages to other servers from the leader. This is much smaller than 1) the 3(N-1) messages required in the case of a single normal Paxos proposer (N-1 Prepare, Accept, Learn messages) and 2) a possibly unbounded number messages in the case of dueling proposers. In our 5 replica system this translates to 8 RTTs instead of 12 for the best case. Thus, since the total number of round trips for agreement is minimized, the total end to end latency for a particular operation is also minimized. However, the results indicate that performance is comparable to normal paxos despite the performance improvements. This could be a result of the increased overhead of pinging the leader and updating replicas over a ping. However, we are still investigating why the performance of Multi-Paxos is not as high as it should be.

5 Related Work

Synchronous replication schemes for PostgreSQL have been explored in the literature. The main challenge is ensuring that transactions are committed in the same order on all replicas. In our work we assume that there are no distributed transactions. One method to accomplish consistent ordering is 2 phase commit [4] across all replicas – however, this requires all replicas to be alive and communicating at all times, thus nullifying any benefit from Paxos. Moreover, 2 phase commit significantly increases latency. Systems like H-store [6] adopt a concurrency protocol that is defined by having one thread of execution per data partition and thus eliminating locking in single-partition transactions.

Transaction	%	Response Time (s) Average	Response Time (s) 90th %	Total	Rollbacks	%
Delivery	3.99	0.137	0.218	61	0	0.00
New Order	43.82	0.160	0.508	670	6	0.90
Order Status	2.88	0.218	0.903	44	0	0.00
Payment	40.35	0.172	0.606	617	0	0.00
Stock Level	2.81	0.220	0.901	43	0	0.00

Table 1: Results from running the benchmark for five minutes on a single, non-replicated instance of PostgreSQL

Transaction	%	Response Time (s) Average	Response Time (s) 90th %	Total	Rollbacks	%
Delivery	3.48	0.766	2.407	50	0	0.00
New Order	45.06	0.733	2.325	648	5	0.77
Order Status	3.34	0.568	2.191	48	0	0.00
Payment	37.90	0.698	2.332	545	0	0.00
Stock Level	4.10	0.851	2.528	59	0	0.00

Table 2: Results from running the benchmark for five minutes on Barista with standard Paxos

Transaction	%	Response Time (s) Average	Response Time (s) 90th %	Total	Rollbacks	%
Delivery	3.96	0.806	2.482	57	0	0.00
New Order	44.69	0.948	2.508	644	4	0.62
Order Status	2.78	0.756	2.069	40	0	0.00
Payment	39.69	0.719	1.997	572	0	0.00
Stock Level	2.91	0.762	2.240	42	0	0.00

Table 3: Results from running the benchmark for five minutes on Barista with Multi-Paxos

In a synchronous replication scheme, the equivalent would be to only execute one transaction at a time on each replica and start the next one only after the current transaction finishes. Another technique that has been explored on unreplicated systems is speculative execution [8] where the system runs subsequent transactions even before the current one has finished. It is likely that some of these transactions may have to be redone based on the results of the current transaction. In our scenario, we want to avoid having to undo transactions. We can get the benefit of speculative execution instead by batching non-conflicting transactions. Some systems also adopt the technique of using a pre-processor that determines the serial order of all transactions and transactions must request locks in the same order. The Paxos log can help with creating a universal order of transactions.

6 Supplementary Materials

- **Source Code:** `https://github.com/abhardwaj/barista`
- **Benchmark Code:** `http://people.csail.mit.edu/anantb/files/barista/6.824/`

References

[1] Database Test Suite: DBT-2 OLTP Benchmark. http://sourceforge.net/apps/mediawiki/osdldbt/.

[2] Postgresql. http://www.postgresql.org.

[3] TPC Transaction Processing Performance Council: TPC-C OLTP Benchmark. www.tpc.org/tpcc/.

[4] Two-phase commit protocol. http://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[5] Agarwal, A., Slee, M., and Kwiatkowski, M. Thrift: Scalable Cross-Language Services Implementation. Tech. rep., Facebook, 2007.

[6] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1496–1499.

[7] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[8] Lampson, B. W. Lazy and Speculative Execution in Computer Systems. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2008), ICFP '08, ACM, pp. 1–2.

[9] Prisco, R. D., Lampson, B. W., and Lynch, N. A. Revisiting the Paxos Algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (London, UK, UK, 1997), WDAG '97, Springer-Verlag, pp. 111–125.