

BARISTA

A distributed, synchronously replicated,
fault tolerant, relational data store

Anant Bhardwaj

Rebecca Taft

Manasi Vartak

David Goehring

Barista

- What is Barista?
 - a middleware layer written over Postgres
 - enables real-time replication of data among multiple Postgres instances in a distributed infrastructure
 - client applications can use Barista with the same SQL code they used before
- Why one should care?
 - an application using database as the backend would like to have:
 - fault-tolerance (automatic failover among replicas) with strong consistency
 - automatic recovery

The Main Contributions

- A distributed, synchronously replicated, relational data store
 - fault-tolerance, automatic recovery, ACID, strong consistency, SQL
- Cross-language support through Thrift
 - sample client code in Go, C++, Java, Python, and JavaScript
- State Safety with ZooKeeper
 - efficient purging and recovery
- Evaluation with the TPC-C Benchmark
 - validation against the industry-standard database benchmark
- Performance Optimizations
 - multi-paxos
- A comprehensive test-suite

Design Choice: Paxos Agreement

- Track the opening and closing of connections in the log in addition to query operations.
 - the presence of replication must be transparent to clients (and therefore they must not need to connect to different instances separately)

```
// open connection to machine 1
con, err = clerk.OpenConnection(machine_1)

// create the table on machine 2
_, err = clerk.ExecuteSQL(machine_2, con,
    "CREATE TABLE IF NOT EXISTS courses (
    id text, name text)", nil)

// insert a record on machine 3
_, err = clerk.ExecuteSQL(machine_3, con,
    "INSERT INTO courses values('6.831', 'UID')", nil)
```

- all operations must affect all the machines in the same order

Design Choice: Enforcing Ordering

- Postgres is multi-threaded
 - transactions can run as different threads.
 - the threads might get scheduled in any order
 - the commit order can be different from the order in which the transactions were submitted
- The current approach
 - only one outstanding transaction at a time
- Future Work:
 - analyze queries waiting to get executed for conflict
 - order the conflicting ones
 - run the non-conflicting ones in parallel

Design Choice: State Safety

- For each paxos instance
 - there is a node in the ZooKeeper with the path
`/barista/paxos/{machine_name}/store/{seq_num} = Paxo {
N_P, N_A, V_A, Decided}`
- ZooKeeper's `Write()` and `Read()` APIs are atomic
 - we don't have to worry about consistency

- Paxos code update the state in ZooKeeper

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

Design Choice: Log Purging

- 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}/store/{seq_num}` nodes if the `{seq_num} < paxos.Min()`
- The purging allows us to keep the ZooKeeper logs small
- The choice of ZooKeeper allows us to do efficient purging
 - if we used file, it'd require us to implement efficient purging mechanism
 - we also considered using sqlite

Design Choice: Recovery

- `AP` : the last applied `seq_num` to the database
 - this is not stored in ZooKeeper?
 - no, we need this to be atomic with the client query execution
 - store this in `sqlpaxoslog(lastseqnum int)` table
 - intercept the client txn and make AP update as part of the client transaction to ensure atomicity

Design Choice: Recovery

- Recovery from crash & restart (no disk failure)
 - reconstruct the paxos state

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

- the [AP](#) can be recovered by reading the sqlpaxoslog table.
- paxos fills holes in its log to ensure that everything after the [AP](#) can be retrieved as part of the paxos protocol.

Design Choice: Recovery

- 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
 - if it serves a new request it will change its state during the recovery and would lead to inconsistent data/state transfer.
 - once the data & the state `{AP}` is copied, the normal recovery protocol kicks in

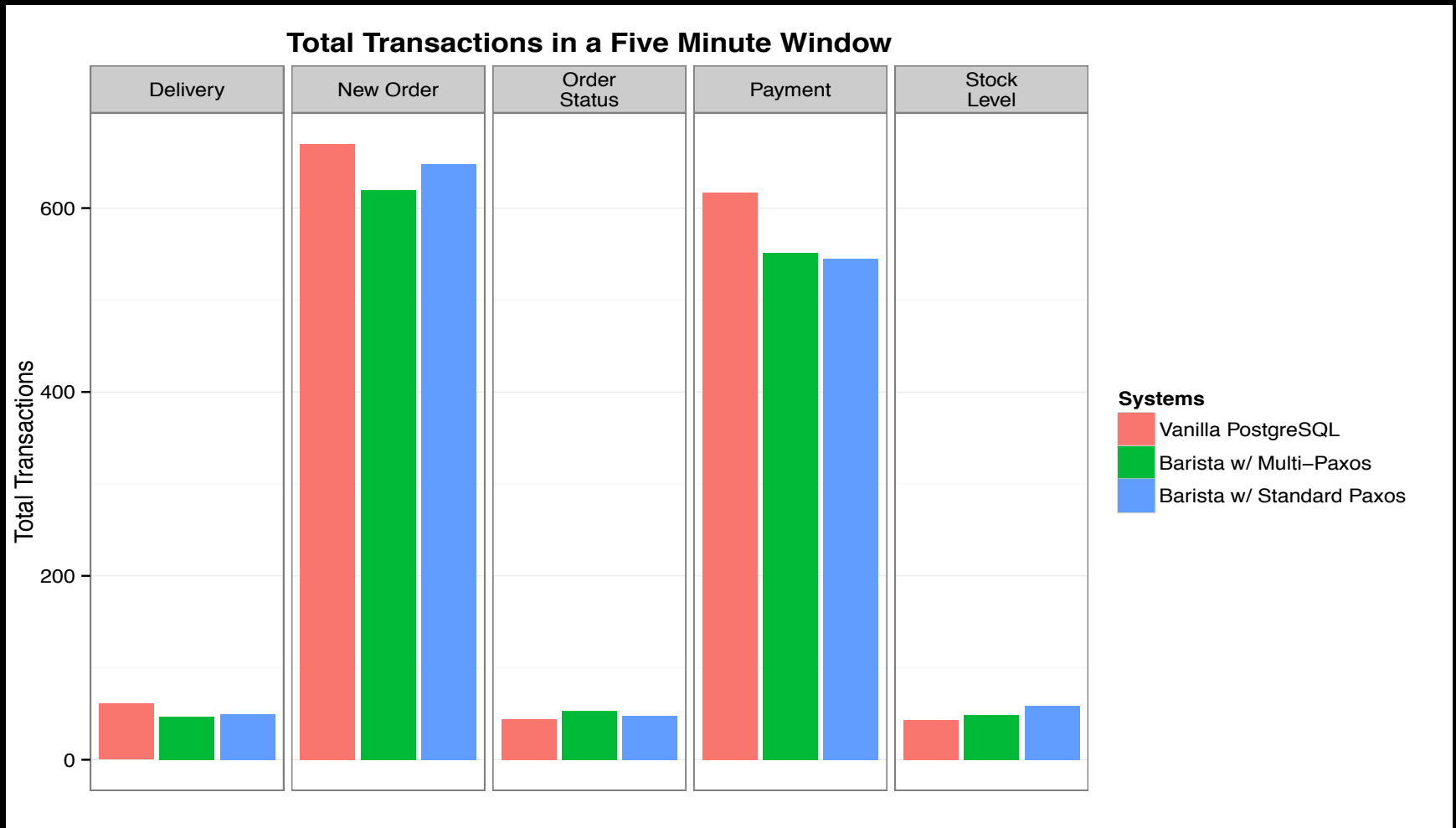
Design Choice: Multi-Paxos

- We made the following optimizations to our paxos-based protocol by implementing a version of Multi-Paxos:
 - avoid 2 round-trips per agreement by having a server issue Prepare messages ahead of time
 - avoid dueling leaders under high client load by using a designated leader
- We present the effect of this optimization in the evaluation section

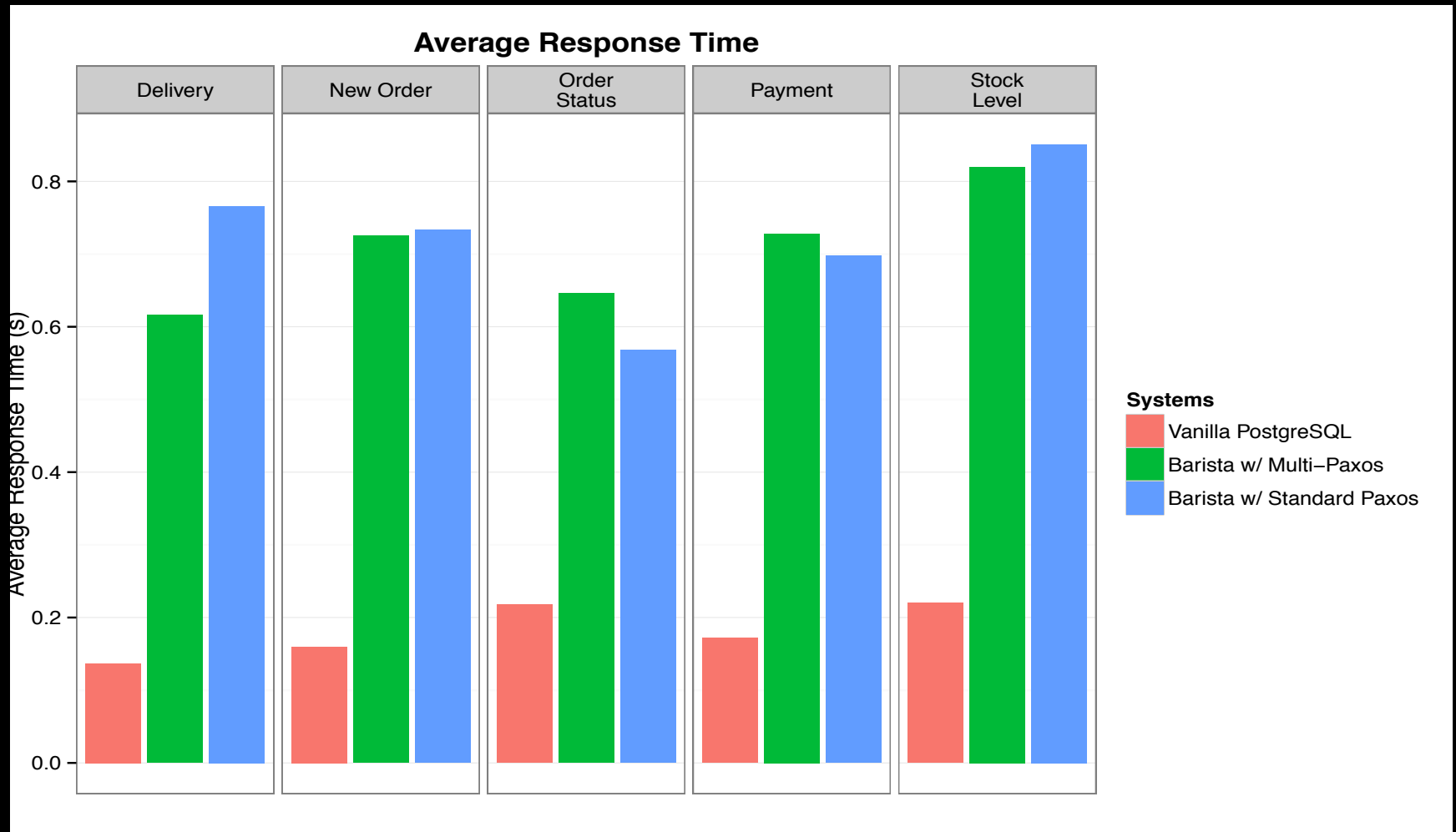
Evaluation

- We implemented the TPC-C benchmark
 - 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.
- The intent of this benchmark is to simulate a realistic real-time OLTP system.

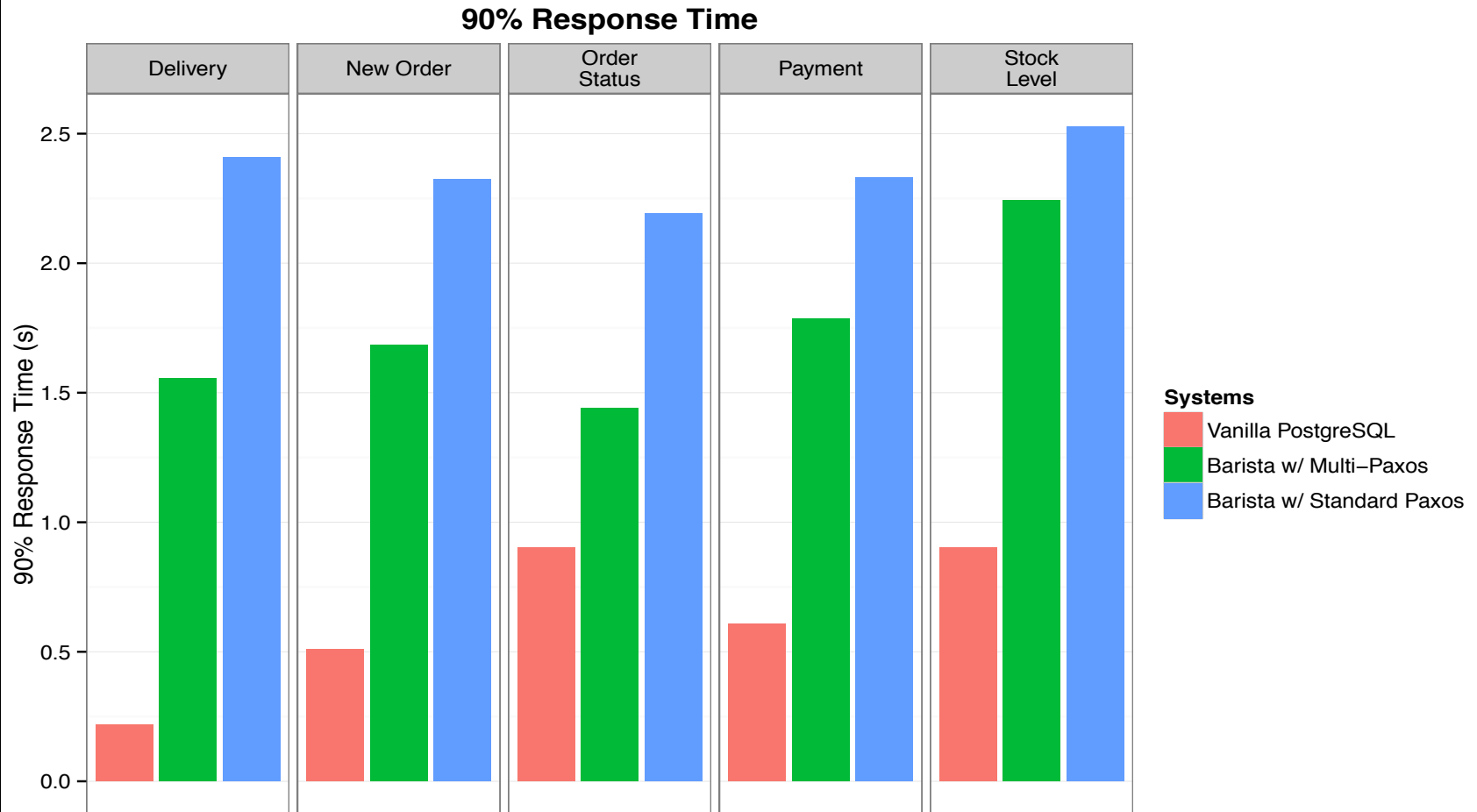
Evaluation



Evaluation



Evaluation



Demo

- We will do a live demo of:
 1. synchronous replication with strong consistency
 2. fault-tolerance (auto failover)
 3. Recovery
 - a. a crashed machine should catch up with other peer by executing all the missing queries after the restart
 - b. paxos safety (paxos should tolerate server restarts)
 - c. disk wipeout (reconstructing state by copying a healthy machine)

Barista Project: Summary

- A distributed, synchronously replicated, relational data store
 - fault-tolerance, recovery, ACID, strong consistency, SQL
- Cross-language support through Thrift
 - sample client code in Go, C++, Java, Python, and JavaScript
- State Safety with ZooKeeper
 - efficient purging and recovery
- Evaluation with the TPC-C Benchmark
 - validation against the industry-standard database benchmark
- Performance Optimizations
 - multi-paxos
- A comprehensive test-suite