Replicated and Consistent Distributed Data Storage

Roy Shadmon
CSE Department
University of California - Santa Cruz
Santa Cruz, California

Niharika Srivastav
CSE Department
University of California - Santa Cruz
Santa Cruz, California

***Abstract*—This project develops a distributed and replicated system to transmit, store, and query data for a simulated IoT infrastructure. The infrastructure contains one device which transmits real-time data to multiple servers. A client may then query a specific server to retrieve data. Clients may make read-only queries and the device may only make write-only queries. The decentralized system acts as a key-value store. In addition, the system supports fault-tolerance as long as at least one server is alive at any given moment in time.**

***Index Terms*—Distributed Database, P2P Network, Replicated storage, IoT, Fault Tolerance**

I. Introduction

In the world of big data, data is continually being recorded by some device and transmitted to some server to process and store the data efficiently. A client, also known as the data owner, is then able to make a request to a server to retrieve the data transmitted from the device. This project wants to prevent a single point of failure in the case a single entity unexpectedly fails.

In most cases today, data owners store their data using a single cloud provider such as Google Cloud, Microsoft Azure, or Amazon Web Services. If a client needs their data to be replicated, the data is replicated on multiple servers hosted by the single cloud provider. The problem with the common approach is that there is still no availability guarantee if data is hosted on one cloud provider [1]. What happens, for example, if there is an outage amongst Google Cloud servers [2]? Although this risk may be tolerable by some applications, there are other applications that cannot tolerate a single point of failure. Moreover, data is not truly replicated in an isolated manner if the same cloud provider hosts it, as the failure of one cloud provider causes a failure in the replication of the entire system. Yes, Google can host our data over multiple data regions. However, this relies upon requests to travel farther distances; hence, decreasing performance. If data were to be stored over multiple cloud providers, on the other hand, then there would be stronger availability guarantees. However, using multiple large cloud providers is rather expensive, and the setup of the system is not obvious [3], [4]. In addition, when a client sends a request to a server that is hosted within a data center, the request must travel through the data center to find the correct server, process the request, and then traverse through the data center again for the message to be sent back to the client. This arduous request processing hinders the performance of a client's request, which can be avoided by leveraging smaller compute nodes that are closer to the edge device [5].

Using direct nodes (possibly nodes in P2P networks), we can leverage nodes that offer data storage and processing services by providing the client and devices direct access to making requests; these node servers can process requests on the edge of the network, which mitigates the need of a request and response traversing through a data center. In addition, we can replicate the data over multiple independent nodes that offer similar service level agreements (SLAs). An SLA is an agreement made between a compute node and a data owner regarding the performance and storage capacity on data. For this project, we assume that SLAs have already been agreed to.

Assuming that the devices transmitting data is trusted meaning a device will always send the data to all nodes (since one cannot trust the data sent from an untrusted device), a requirement for this system is to ensure that when a client makes a request to the nodes, the nodes are all processing the request on consistent data. This is important because a client needs their data to be highly available in the case a node becomes benign (the node intends to act honestly, but fails for a short, finite period of time) [6]–[9]. In the first phase of this project, we assumed all nodes are honest, as well as; no node will crash. In the second phase, we assumed all but one node can become benign for some finite period of time. For the implementation, we have not considered byzantine nodes, however, future work will address a byzantine setting. In addition, we currently only support a single client, however, future work will also address this.

In this project we have developed a verifiable mechanism for independent nodes receiving transmitted data. The system ensures that when a client sends a request, all the nodes will process the request on the most recent processed state of the data [10]. One approach we considered is to develop an efficient batching mechanism where nodes agree to commit a predefined count of data. For example, nodes can agree to commit every 10 pieces of data received. This mechanism will also allow a client to determine which node(s) are not abiding by the SLA if one node is significantly behind in committing data compared to the other nodes. For this project, however, data was guaranteed to be received by the server, so servers committed data as it was received.

After supporting non-benign servers, we attempted to support node recovery. For example, if a node crashes, it can receive the data it failed to receive when it became unavailable–we do this using the gossip protocol. A gossip protocol [11] is an algorithm to implement peer-to-peer communication that is based on the way epidemics spread. Distributed systems use P2P gossip to ensure that data is disseminated to all members of a group. In addition, each node keeps a log containing all of its commits, which allow for a quick way to catch up with a benign node.

II. Description of the Prototype

The system consists of a device, a client and multiple servers. The device continuously sends data to the servers, where each server saves the data to a PostgreSQL database[1]. We presume that the client knows the number of servers in the system as the system is up to the client's specification.

To retrieve data, the client sends a request for a value on a specified date (similar to a key-value store); the client must specify which server it wants to query.

A. Threat Model

The threat model of this system assumes that the data source from the device is trusted. The reason for this is that if the system cannot trust the data source, then why even query the data. There must be implicit trust that the creation of data is correct. Moreover, we assume that in the worst case, servers are benign. Benign servers may fail or go offline, but they will never maliciously compute incorrect results or transmit incorrect data to a node in recovery. Moreover, we assume that the server logs every data item committed to the database. The servers cannot send messages to the device, as well. The client is assumed to be trusted. The client can only send read-only requests and cannot cause the server to fail or insert malicious data. The client owns the data transmitted from the device to the servers, so there is no risk that the client would commit a Denial of Service (DoS) attack against the servers. The client also has no means to contact the device, so there is no threat of a DoS attack to the device from the client.

B. Architecture

The architecture of our system is a single device that transmits data to multiple servers. A client can then send requests to servers regarding the transmitted data. Figure 1 illustrates an architecture with one client, three servers, and one IoT device. Forward arrows signify messages are sent to entity, which also implies entity facing forward arrow receives sent message.

C. Messages

The servers, the client, and the device use PubNub [12], a global data stream network that acts as our node-to-node communication tool. PubNub is a real time Infrastructure as a Service (IaaS) tool that allows for decentralized entities to send and receive messages through channels [13].

[1]Other SQL-based databases may be used, however, consider that the syntax some queries may need to be modified.

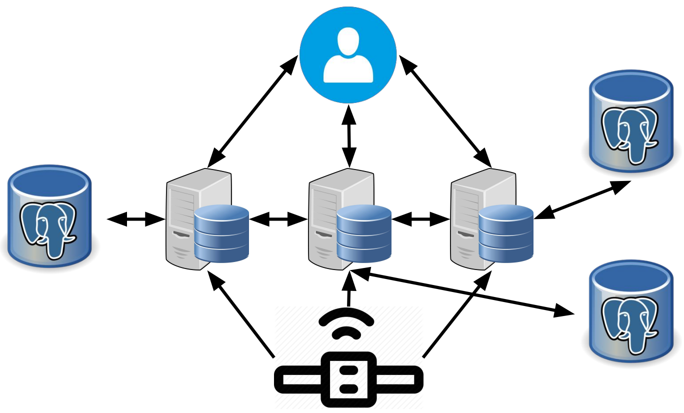
Fig. 1. This is an example architecture of our system that has one device, three servers, and one client. Forward arrows indicate messages are sent to the certain entity, as well as all servers may communicate with each other. There are also three Postgres instances as each server maintains their own database instance.

1) Channels: A channel can be thought of as a host and port entry point where nodes can communicate with each other in real-time–similarly to socket programming. Specifically, channels in PubNub are defined by a unique name, such as 'client-channel' or 'server1-to-server2'. To receive messages on a channel, the entity must *subscribe* to the channel. To send a message, the entity must specify on which channel(s) to send the message, the message itself, and the metadata associated with the message. The reason to specify the channel and message is obvious, however, the sender must also specify the metadata so each entity can filter messages based on the metadata. For example, a message sent on the 'client-channel' is sent to everyone subscribing to the 'client-channel' including the sender if they are also subscribing to the channel. To prevent the sender from receiving their own message, the sender specifies their user id (uuid) in the metadata. Thus, if a receiver's uuid matches the uuid in the metadata of the message, the receiver will filter out the sent message[2].

D. Device

The device broadcasts a tuple message containing a date and time of type *datetime* and a value of type *int* onto the 'device-channel' channel, which each server is subscribed to. In addition, messages are broadcasted every three seconds to simulate an IoT device transmitting data.

E. Server

Every server maintains its own Postgres database instance and a server log to keep track of every key and value it receives from the device. Upon receiving a *datetime* date and an *int* value, the server first tries to commit that value into the database. If successful, the server logs the recorded date to a newline in its log file. The only reason a data item should fail is if the sent *datetime* date is identical to a date already

[2]In future work, we plan on building our own socket messaging ecosystem, however, due to time limitations we decided to use PubNub so we can focus our time on developing a decentralized key-value store.

committed to the database or the server runs out of memory. Since the device is trusted and we currently assume the server has infinite memory, we don't expect the need to implement an insert failure feature. Future work will develop this feature.

The server is also responsible to respond to client requests. As explained earlier, the client sends a *datetime* date to the server, and upon receiving this request, the server processes the request and sends the result back to the client. If the date does not exist, then the server also notifies the client that the date does not exist in the database. Lastly, the server is responsible to help other servers recover data that a server missed due to some failure. The fault-tolerance process will be discussed in Section III.

1) Server Database: Servers store data in a PostgreSQL database. The *datetime* date is the *primary key* of the database, that prevents the insert of any duplicate data item. A *primary key* of a table in a database requires that only a unique value of the *primary key* type may be inserted into the table. For example, a data item containing a *datetime* date and *int* value, such as ('2019-11-30 18:00:55-08', 5) can only be inserted if and only if '2019-11-30 18:00:55-08' is unique in the *datetime* date column of the table in the database. If at some point another data item with an identical *datetime* date, such as ('2019-11-30 18:00:55-08', 10), the database will immediately reject this insert request. This is because we presume that only one value will be recorded at any given point in time.

Inserts to the database are also processed with the following PostgreSQL syntax:

INSERT INTO [table_name] (created_time, input) VALUES([datetime], [int]) ON CONFLICT(created_time) DO NOTHING;

where the values in between the brackets are actual values. The query is relatively simple to understand, however, the '*ON CONFLICT*' states that if there is a pre-existing *created_time* value, rather than returning an error, the Postgres database management system will do nothing. This error handling prevents the node from manually needing to resolve database errors caused from having errors–similarly to catching exceptions.

F. Client

The client can only issue a *GET* command to retrieve the value at a certain point in time. It is assumed that the client is always aware of which server node is available to retrieve requests–although if a client makes a request to an actively benign server node, the client would never receive a response. After a timeout period, the client can assume that the server node is experiencing a failure since the threat model only assumes possibly benign nodes.

III. Fault Tolerance

If a server were to fail, the server upon coming back alive will broadcast a message to all active servers subscribed to the recovery channel notifying them that they are ready to process

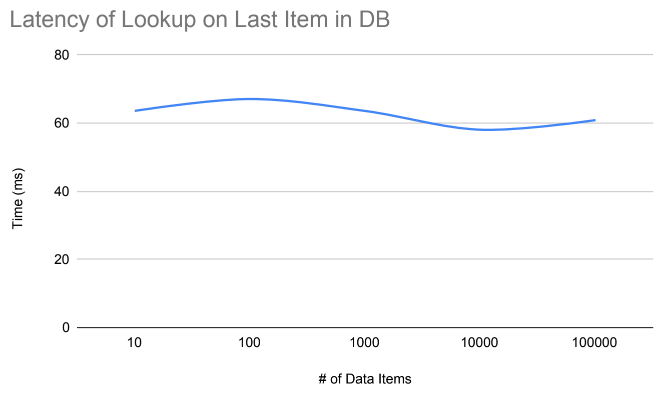

Fig. 2. This figure shows the lookup time for the last data item inserted into the database in ms.

data again. However, due to the down server missing data (since data is continuously being streamed from the device), the node who needs to recover missed data will use its log to determine the point at which it failed[3].

The steps to recover missed data is as follows:

1) The failed server retrieves the first and last row in its log file, which are of type *datetime*.
2) The failed server sends the first and last *datetime* to all servers on the recovery channel.
3) All current active nodes compute and send all the rows recorded in time before the first date in the recovery node's log and all the rows recorded in time after the last date in the recovery node's log.
4) The recovery node attempts to insert all data sent to it by the other active nodes.
5) The recovery node must also sequentially order its log file to keep the commit history ordered.

While the node is processing all the data it missed, it is also asynchronously processing the streamed data from the device. During the recovery time for the node in recovery, a client may receive an incorrect response. Future work of this project will support consistent execution at all times.

IV. Performance

The performance of our system was measured on a MacBook Pro containing a 2.4 GHz dual-core Intel i7 processor with 16 GB of memory. The model used to measure performance contained one client, three servers, and one device. The rest of this section explains why the overall performance of our system is tolerable.

Figure 2 illustrates the lookup time for the last data item in the stored in the database. As shown (in log-scale), the lookup time for our system hovers around 60 ms regardless of the number of data items stored in the database. Running the same lookup query directly on a Postgres database instance processes in 0.861 ms–a reduction of 100x. However, it is

[3]The point at which a node stops committing data to the database is when a node is considered to have failed.

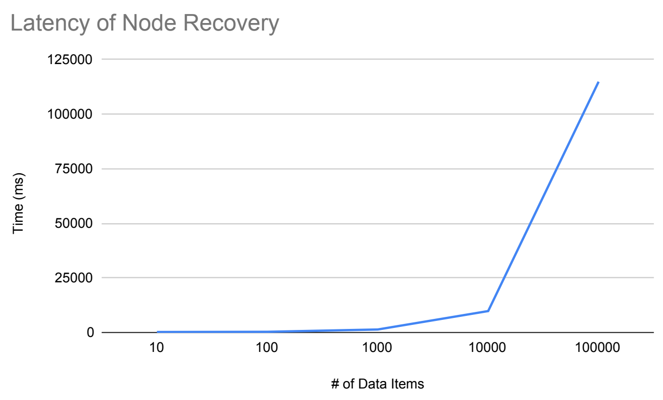

Fig. 3. This figure shows the time it takes for a benign node to recover data.

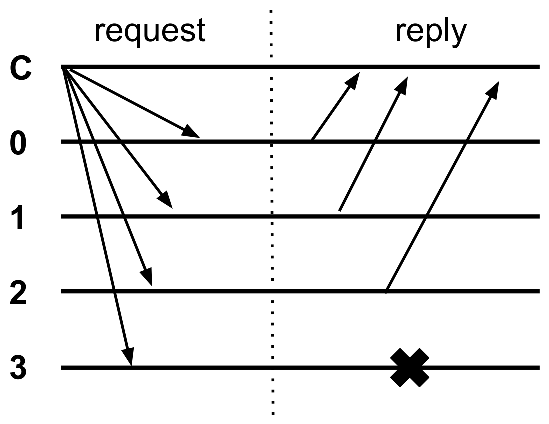

Fig. 4. A successful 2-Phase lookup protocol in a byzantine setting.

important to note that a lookup requires two PubNub messages that take 25 ms each. Therefore, the performance of our system is justified.

Figure 3 illustrates (in log-scale) the time it takes for a benign node to recover data when it comes back alive after a period of downtime. Unfortunately, performance significantly decreases the longer a node remains offline, however, the exponential decrease is expected as an exponential amount of data items needs to be recovered. There is also an exponential increase in time for a benign node to recover data due to the limitations of message size. The maximum message payload PubNub supports is 32Kb. Therefore, as more data items need to be recovered, more messages need to be sent. In our simple (*datetime*, input) data item, we measured that the most data items that can fit in a single message is 100 data items. Thus, messages are sent in batches of 100 data items upon recovery.

V. Byzantine Fault-tolerance Enhancement

A. Not Specialized BFT

To support byzantine fault-tolerance (BFT) for lookup requests, we can simply use Practical Byzantine Fault Tolerance (PBFT) [14]. For PBFT in our system to work, a client would make a lookup request to the current leader of the protocol. Once the leader receives a client's request, the leader would broadcast the signed request to all other servers. If the leader and the other servers receive a majority of accept messages ($2f+1$ including their own total matching messages of $3f+1$ total nodes), then the servers process the request and broadcast to each other the computed result. If a server receives $2f+1$ identical computed results (including its own), then the server sends the client the computed result. Once the client receives $2f+1$ identical messages containing the same result, then the client knows that they received the correct result.

B. Specialized BFT

A more novel approach to support BFT in our client read-only system is analyzing the threat model of our system. As shown in Figure 4, if we assume that the client will always make legitimate read-only requests, then we can implement an efficient 2-phase lookup protocol. First, the client will send its request to all servers. Each server will process the lookup request and send the client the response. If the client receives $2f+1$ identical responses from a total of $3f+1$ servers, then the client knows they received the correct lookup. If the client does not receive $2f+1$ identical responses in time δ, then the client must retry its request and consider their original message as failed.

VI. Future Work

Future work for this project is to support all types of Postgres queries rather than just a single data item lookup. We believe this is simple to accomplish, as we just need to support this message type and prepare a message batching algorithm in case the SQL lookup request results in the return of multiple data items. Moreover, we would need to prevent any type of client request that would update, insert, or delete data items from the database. We think this is also simple to accomplish by comparing all requests against a regex.

More exciting work would be to support complete SQL queries that parallelizes the servers in the BFT environment. If data is anyways being replicated, taking advantage of the replication to increase performance would be a significant improvement.

The most challenging future work for this project is to develop more efficient mechanisms to verify correctness of the computation done by the byzantine servers. Databases such as IntegriDB [15] support verification of SQL queries, however, the set up time is multiple hours and isn't tolerable in IoT infrastructures.

VII. Conclusion

We have successfully developed a database system to transmit, store, and query data for a simulated IoT infrastructure. The infrastructure consists of one device which transmits real-time data to multiple servers. A client may then query a specific server to retrieve data. Clients may make read-only queries and the device may only make write-only queries. The decentralized system acts as a key-value store. The system can support node failure and guarantees node and data recovery as long as at least one server is alive at any given moment in time.

References

[1] N. Majadi, "Cloud computing-research issues and challenges," in *Proceedings of the Global Engineering, Science and Technology Conference 2012*, 2012.

[2] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 1–16, ACM, 2016.

[3] A. Bala and I. Chana, "Fault tolerance-challenges, techniques and implementation in cloud computing," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 288, 2012.

[4] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[5] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *2012 Second Symposium on Network Cloud Computing and Applications*, pp. 137–142, IEEE, 2012.

[6] O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[7] A. Gaur, B. Scotney, G. Parr, and S. McClean, "Smart city architecture and its applications based on iot," *Procedia computer science*, vol. 52, pp. 1089–1094, 2015.

[8] L. Catarinucci, D. De Donno, L. Mainetti, L. Palano, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An iot-aware architecture for smart healthcare systems," *IEEE Internet of Things Journal*, vol. 2, no. 6, pp. 515–526, 2015.

[9] T. Neudecker, P. Andelfinger, and H. Hartenstein, "Timing analysis for inferring the topology of the bitcoin peer-to-peer network," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pp. 358–367, IEEE, 2016.

[10] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *International conference on reliable software technologies*, pp. 38–57, Springer, 1996.

[11] K. Jenkins, K. Hopkinson, and K. Birman, "A gossip protocol for subgroup multicast," in *Proceedings 21st International Conference on Distributed Computing Systems Workshops*, pp. 25–30, IEEE, 2001.

[12] PubNub, "Pubnub: Apis for connected experiences." https://pubnub.com/.

[13] K.-L. Wright, B. Krishnamachari, and F. Bai, "Noctua: A publish-process-subscribe system for iot,"

[14] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[15] Y. Zhang, J. Katz, and C. Papamanthou, "Integridb: Verifiable sql for outsourced databases," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1480–1491, ACM, 2015.