ResVision: Final Report

Abstract

This report encapsulates the culmination of our efforts in visualizing the Practical Byzantine Fault Tolerance (PBFT) protocol, a prominent consensus algorithm widely employed in distributed systems like Resilient DB. PBFT addresses the critical challenge of achieving consensus among distributed nodes even in the presence of faulty or malicious components. The protocol's significance lies in its ability to maintain functionality and data integrity, making it an indispensable tool for resilient and fault-tolerant distributed systems.

In the face of increasing reliance on distributed systems for critical applications, ranging from financial transactions to healthcare records, ensuring the integrity and availability of data becomes paramount. PBFT emerges as a robust solution, offering a practical approach to Byzantine fault tolerance that is crucial for maintaining the trustworthiness and reliability of distributed systems.

As we delve into the visualization of PBFT, our project provides a comprehensive understanding of the protocol's real-time dynamics. The visual representation not only serves as an educational tool but also offers a practical lens into the inner workings of the protocol. Through meticulous front-end development using React.js and D3.js, and the integration with Resilient DB on the back end, our visualization strives to bridge the gap between theoretical understanding and practical application. Not only this, but we have deliberately designed the visualization in such a way that it is versatile enough to visualize various replica communications other than consensus and expandable to visualize other BFT protocols.

Motivation

Our motivation stems from the ubiquity of Byzantine Fault Tolerance (BFT) mechanisms in contemporary decentralized networks. With the proliferation of blockchain technologies and distributed systems, safeguarding data integrity against malicious nodes becomes non-negotiable. The Practical Byzantine Fault Tolerance (PBFT) protocol, chosen for its practicality and prevalence, exemplifies the resilience required in these dynamic ecosystems. Our decision to visualize PBFT in real-time is fueled by the recognition that these intricate consensus protocols are best understood through visualization, offering a tangible representation of their operations. This project becomes a pivotal educational tool, providing developers and students with a user-friendly interface to comprehend the nuanced dynamics of PBFT. By bridging the gap between theory and application, our venture aims to empower individuals navigating the complexities of fault-tolerant consensus in distributed systems. In doing so, we contribute to the broader discourse on the significance of BFT algorithms in fortifying the reliability and trustworthiness of decentralized networks.

Our motivation is further enriched by the multifaceted benefits our project brings to the educational landscape. By enhancing the learning experience, we provide students and practitioners with a visually intuitive tool to comprehend the Practical Byzantine Fault Tolerance (PBFT) consensus protocol. The incorporation of real-time visualization not only allows users to observe but also to analyze the protocol's intricate behavior, offering a hands-on understanding of the dynamics at play in distributed systems. This project serves as a valuable teaching aid, empowering instructors to convey complex concepts effectively. By fostering a clearer understanding of distributed systems and consensus algorithms, our visualization tool becomes instrumental in the educational journey, ensuring that learners can grasp the nuances of PBFT and its applications in real-world scenarios. This educational value underscores our commitment to not only advancing the understanding of

Byzantine Fault Tolerance but also to cultivating a rich and accessible resource for both educators and learners in the field of distributed systems.

Introduction

The technological innovation that we propose is with respect to the visualization of the consensus protocol known as PBFT. PBFT, also known as Practical Byzantine Fault Tolerance, is a popular consensus protocol used in many distributed systems, such as Resilient DB. Visualization of this protocol in real-time can not only serve as a teaching tool, it can also provide greater insight into the machinations of the protocol itself. Showing real-time transmission of data amongst the replicas and primary can enable one to observe/ascertain if any relevant patterns emerge via the elucidation of this protocol. Furthermore, if patterns emerge with respect to the flow of data within this protocol, one can begin to use this information so as to generate novel protocols that compensate for any potential weaknesses observed.

Implementation

Front-End:

In the landscape of Practical Byzantine Fault Tolerance (PBFT) protocol visualization, our front-end development serves as the user-centric gateway, seamlessly blending intuitive design with real-time data representation. The collaborative effort embraced modern technologies, employing React.js for user interface construction and D3.js for dynamic data visualization.

User Interface Design:

Commencing with conceptualization and design, we utilized Figma to craft a landing page and visual elements. The subsequent translation of these designs into a modern web application was executed using React.js, emphasizing a user-friendly interface.

Graph Visualization:

The creation of a PBFT visualization prototype, showcasing the real-time transmission of data among replicas and the primary node was achieved. D3.js played a central role in dynamically rendering the intricate graph representation.

Web Application Integration:

The seamless integration of the entire front-end interface into a web application using React.js in Visual Studio Code established a proof of concept and laid the foundation for a Minimum Viable Product (MVP).

Educational Impact:

Beyond aesthetic appeal, the front-end development significantly enhances the learning experience. The visually intuitive web application offers a gateway for students and practitioners to dynamically comprehend the PBFT consensus protocol.

We have completed the task of creating a front end with simulated data. This is a critical part of the project as it not only creates a proof of concept but also sets the stage for a minimum viable product. Furthermore, by creating the front end/UI elements and inserting simulated data, we get a sense of how the data should flow through the visualization, what data we need, and what we expect in a fully functional visualization process.

The front end was created using the React library for Java Script, and the graph visualization has been done using D3.js which is a visualization library for javascript.

A landing page has also been developed using React.js into which we will integrate our Visualization. The initial design was first created in FIGMA which helped us design the application efficiently and effectively. Then we used our FIGMA design to create the Web UI using react.js in VSCode.

Fig 1: Web Application Front End

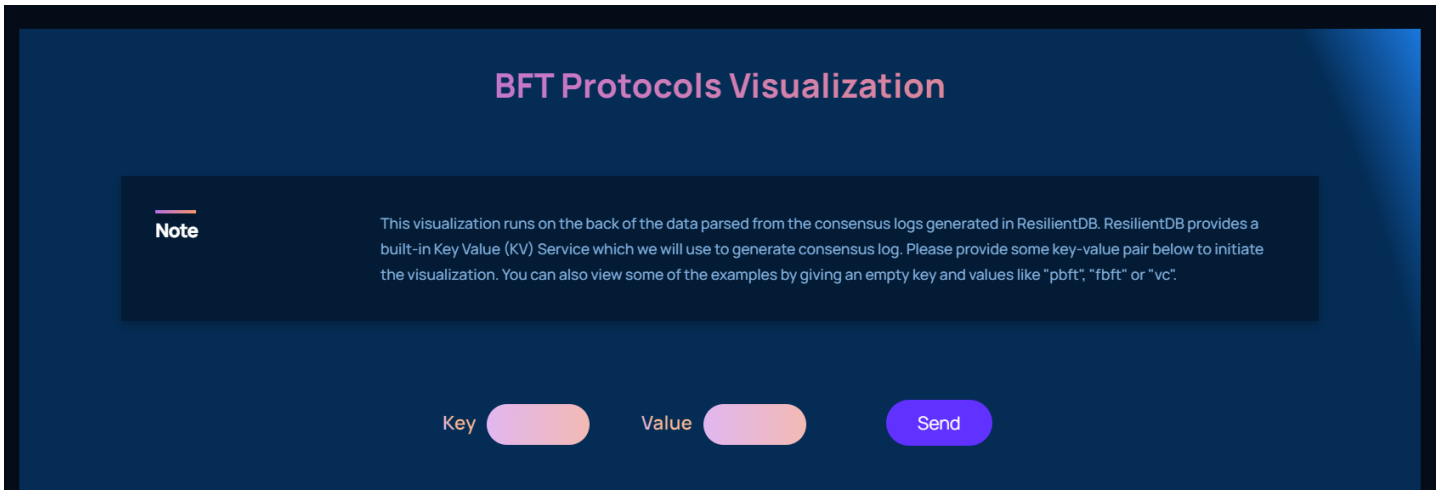

Fig 2: Key Value Input for Initiating Visualization

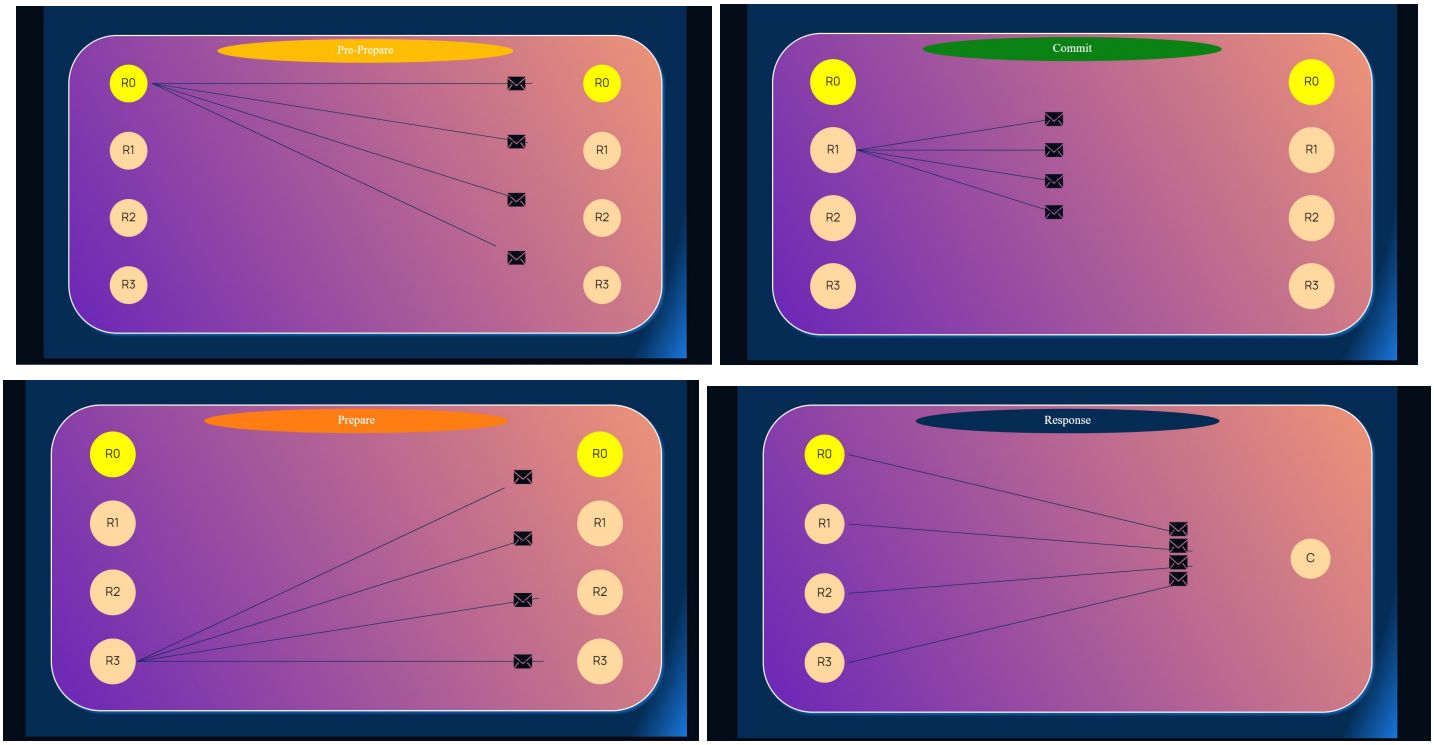

Fig 3: Visualization of various phases of PBFT

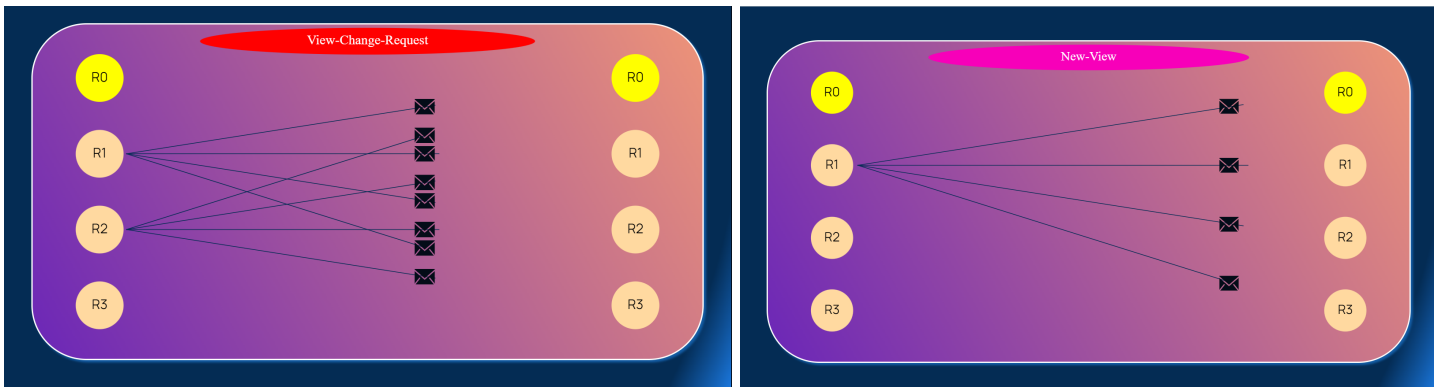

Fig 4: Visualization of View Change of PBFT

Previous Work and Enhancement:

In addition to our current visualization, we have also taken inspiration from the previous class's project "Pretty PBFT". We noticed that the project used dummy data that was automatically generated at run time and felt that it would be a great addition to our project if we leveraged our classmates' prior code and modified it such that it would accept code from Resilient DB. This required combing through the code and significant amounts of debugging. One of the obstacles that we ran into was the fact that the visualization was overwriting the data file that we appended to and inserting dummy data instead. Hence, after debugging we were able to pinpoint the location of this code and change the file specified accordingly such that we were able to insert our custom data. We also had to create a separate parser for the "Pretty PBFT" visualization as the format it accepted was slightly different from what our current parser was outputting.

Back-End:

In the backend of our platform, we have set up ResilientDB on Google Cloud Platform (GCP) to leverage its scalable and secure infrastructure for handling our distributed ledger transactions. Our crafted 'execute command' function is pivotal in this orchestration, as it receives key-value pairs from the frontend interface and diligently executes transactions on the server. Parallel to this, a Python parser tirelessly operates in a continuous loop, meticulously iterating every second to generate a JSON file within the server's filesystem. This file is not just created and forgotten; it is under the vigilant surveillance of a file watcher. Any modifications trigger an immediate action—the change is captured and, through the magic of WebSockets, this updated data is streamed in real-time to the connected clients. This seamless backend mechanism ensures that our clients are always interacting with the most current state of the ledger, facilitating a dynamic and responsive user experience.

Initializing the transaction from frontend:

On our platform, we've harnessed the power of Node.js to directly interact with our system's shell, enabling us to execute key-value transaction commands seamlessly. When a request hits our server, the executeCommand function springs into action, checking for the required parameters—a key and its corresponding value. If these parameters are in place, it meticulously constructs a shell command to invoke a Bazel-built tool that alters the configuration of our service. This process isn't just about execution; it's about precision. If any errors arise or if the standard error stream captures anomalies, we log these events internally while keeping our users informed with appropriate feedback. Finally, upon a successful operation, we parse the output to confirm the transaction's success, sharing this confirmation as a JSON response. This blend of Node.js with shell commands exemplifies our commitment to delivering responsive and efficient server management solutions.

The parser:

There is a python parser to parse log files generated by ResilientDB, which utilizes the Practical Byzantine Fault Tolerance (PBFT) protocol. These logs are detailed and contain a lot of information, but for the purpose of visualizing the PBFT process, only specific data points are needed. To begin with the logs are generated on every transaction with sender and receiver ID. There are 5 logs files, client.log which proposes transactions, then 4 server node logs, where server0.log is primary and rest of the logs are from 3 replicas of resilient db.

With regards to what the parser contains, the first component is a function to parse_pbft_log(file_path) to parse a log file and extract information about the PBFT stages (e.g., New-Txns, Pre-Prepare, Prepare, Commit, Execute, Response). This function uses regular expressions to identify lines in the log file that correspond to different PBFT stages.The script also handles each stage by creating a dictionary with the phase, sender, and receiver information and then appending it to a list of stages. With respect to saving and processing Data, the parsed data is saved into a JSON file using the save_to_json(pbft_stages, output_file) function. The script then reads the JSON file, filters out any stages without senders using the remove_empty_senders(data) function, and saves the filtered data back to the file. The script reorders the phases to ensure they follow a specific order (New-Txns, Pre-Prepare, Prepare, Commit) using the reorder_phases(data) function. It also adds missing New-Txns phases where necessary with the add_new_txns(data) function.

Further processing is done to ensure that the 'Response' phase follows after every three 'Commit' phases.The script then compiles all the data into a final JSON format, which includes metadata (primary ID, number of replicas) and the detailed PBFT phase. The entire process is wrapped in an infinite loop (while True:), indicating that this script is intended to run continuously, possibly to monitor and parse new log files as they are generated. At the end of each loop iteration, the script pauses for 1 second (time.sleep(1)). This is likely to prevent the script from overwhelming the CPU or reading the same file multiple times in rapid succession.


```
{
  "primary_id": 1,
  "numberOfReplicas": 4,
  "phases": [
    {
      "phase": "New-Txns",
      "senders": [
        5
      ],
      "receivers": [
        1
      ]
    },
    {
      "phase": "Pre-Prepare",
      "senders": [
        1
      ],
      "receivers": [
        1,
        2,
        3,
        4
      ]
    },
    {
      "phase": "Prepare",
      "senders": [
        2
      ],
      "receivers": [
        1,
        2,
        3,
        4
      ]
    },
    {
      "phase": "Prepare",
```


Fig 5: Parsed Data from ResilientDB log files

Websockets & File Watcher:

In our latest infrastructure update, we have introduced an innovative file-watching service paired with WebSocket technology to ensure real-time data synchronization between our server and clients. The file

watcher is meticulously engineered to monitor changes to a specific JSON file, 'newer_final.json'. When a modification is detected, the service leaps into action, reading and parsing the updated content before broadcasting it to all connected clients through a WebSocket connection.

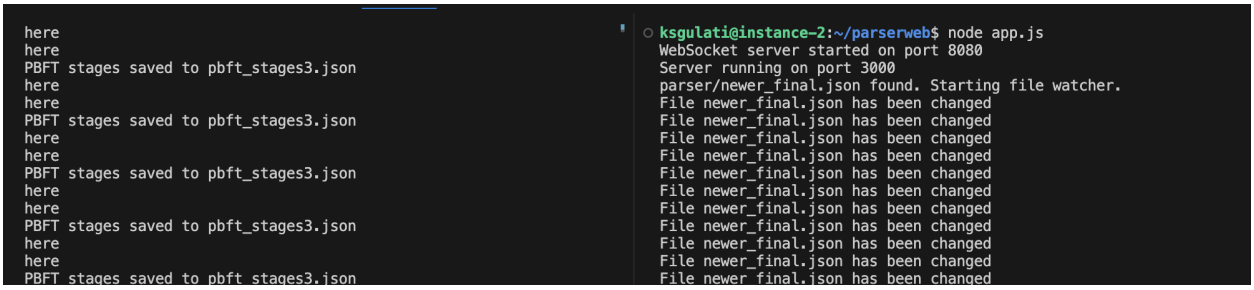

Fig 6: Web-socket updating Parsed data

This setup guarantees that every client receives the latest data almost instantaneously. The WebSocket server operates on port 8080, establishing a persistent connection with each client.
Upon any new client connection, the server immediately pushes the current state of the monitored file, ensuring that the client's initial view is as up-to-date as possible. With this robust system in place, we promise an interactive user experience where data flows seamlessly in full-duplex, keeping everyone on the same page without the need for manual refreshes as shown below.

Limitation/Associated Risks

Having multiple bash scripts may make the process of setting up/configuring Resilient DB more arduous than doing it manually. Hence, we have the next step of abstracting lots of this functionality with simple prompts to the user by means of a Python script. Another risk is the changing container IDs, and if Resilient DB is restarted there may be some configuration changes required to the python scripts. Hence, to mitigate this risk, we need to generalize the Python script and think of any potential user actions that may confound the automation. During the setup of the GCP, the first issue was more of a logistical one in terms of impending workflow. This was very slow connection times, thereby making the entire process take longer. This was followed by the issue caused by the overdrawing on the GCP credits we were allotted since compute power is rather high priced on GCP. Finally, moving on to the risks associated with the integration of backend data. They are mainly based on the fact that the visualization latency on the front-end side will be exponentially increase. This is due to the fact that we are using real-time data from resilient DB. Hence, network speed, consensus time, etc. can all factor into making the visualization appear a lot slower. However, we don't believe that this latency penalty will be too severe since the number of nodes we are handling is minimal.

Future Steps

Our future steps for this project would be to incorporate other consensus protocols such as Hot Stuff, Narwhal and Tusk, etc. Furthermore, we would like to further refine the parser and add a level of flexibility/interactivity to our visualization. For example, if a user wants to simulate a view change by purposefully making the network faulty to see the view change protocol in action, the user should be able to do so. Also, the user should be able to modify the configuration such that we are able to increase the number of nodes and see how consensus plays out rather than just being limited to just four nodes.

Conclusion

Given the ubiquity of blockchain and cryptocurrencies, we feel that our project would be an excellent teaching tool in order to better illustrate the machinations of various different consensus protocols.

The integration of React.js and D3.js has empowered us to seamlessly represent the real-time transmission of data among replicas and the primary node, providing users with a tangible insight into the protocol's complexities. Beyond the aesthetic appeal, our front-end development carries an inherent educational impact, offering students and practitioners a dynamic platform to comprehend the nuances of PBFT.

As we navigate the complexities of Byzantine Fault Tolerance, our project, motivated by the imperative need for resilient consensus mechanisms and their comprehension, contributes to the broader discourse on distributed systems. In conclusion, our journey into PBFT visualization reflects a harmonious blend of technological innovation and educational empowerment, unlocking a deeper understanding of consensus protocols in the realm of distributed systems.

Contributions

Divyanshu Malik: Learned React and developed a modern web app using JSX, Virtual DOM, and D3.js for real-time BFT visualizations, collaborating closely with the team on backend integration. The design process involved Figma, and implementation was done in VSCode & React.js.

Aditya Sharoff: Established a local Resilient DB using WSL and Ubuntu, automating Docker image handling and parsing server logs into CSV. Initially considering a GCP HTTP Server, opted for Apache due to issues, successfully accessing GCP server data locally and modifying CPP files for new visualization data. Extended pretty PBFT to use data parsed from ResilientDB logs requiring a modified parser for compatibility.

Devashree Kataria: Collaborated with Akshit to create the PBFT front-end visualization, explored the Proto file and enhanced CPP files for expanded data logging, and implemented React code to fetch data from the API for seamless integration. Deployed Resilient DB locally and attempted to set up the HTTP server.

Akshit Parmar: Created the initial PBFT visualization prototype with animations for message transport between nodes, aimed at proof of concept and pending back-end integration. Contributed to defining front-end data requirements, proposed an agnostic visualization, and played a crucial role in editing and modifying cpp and proto files for logging.

Karamjeet Singh Gulati: Implemented ResilientDB on GCP with an HTTP server capable of parsing POST request bodies containing key-value pairs from the frontend. Established SSH-based command execution, resulting in new data in the logs. Set up a file watcher to monitor the JSON file generated by Parser, along with a WebSocket server to relay file changes to the client. Also, implemented a WebSocket client to receive and store data in a file, continuously updating it with real-time data. Deployed frontend and backend, in GCP.

References

1. https://2019.wattenberger.com/blog/react-and-d3
2. https://observablehq.com/@nyuvis/d3-introduction
3. https://d3js.org/what-is-d3
4. https://www.smashingmagazine.com/2015/12/generating-svg-with-react/
5. https://www.figma.com/best-practices/guide-to-developer-handoff/
6. https://blog.resilientdb.com/2023/11/22/Deploying-ResilientDB.html
7. https://blog.resilientdb.com/2021/08/11/GettingStarted.html
8. https://www.tutorialsteacher.com/d3js/animation-with-d3js
9. https://legacy.reactjs.org/docs/getting-started.html
10. https://rapidapi.com/guides/fetch-api-react
11. https://blog.resilientdb.com/2021/08/21/PBFT_Commit.html
12. https://expressjs.com/en/5x/api.html
13. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
14. https://nodejs.org/api/child_process.html
15. https://docs.python.org/3/library/re.html