CSCI-B649 Course Project Report: Achieving High Availability with Eventual Consistency on Edge Etcds

Kannan Mani Subramanian Maniarasusekar, Sathish Soundararajan, Max Roesler, Animesh Dhole, Divyank Sanjay Agarwal, Mrunal Yogesh Patil

Abstract—**This report covers our project's original idea, research performed, practical tasks implemented, basic practical simulation of a scenario within a distributed system, and our future work.**

I. Introduction

Blockchain [5] is a distributed ledger technology that facilitates tamper-proof, secure, and transparent transactions between multiple parties without the need for a centralized authority. The data is stored in blocks that are chronologically and immutably linked together in a chain. Each block in the blockchain contains a cryptographic hash of its previous blocks, making it difficult to tamper with the data.

Consensus algorithms play an important role in ensuring that all participants agree on the current state of the blockchain. They determine how Nodes in the network reach consensus on the order and content of transactions added to the blockchain.

There are various types of consensus algorithms used in blockchain, including Proof of Work (PoW) [7], Proof of Stake (PoS) [9, 8], Proof of Reputation (PoR) [6], Delegated Proof of Stake (DPoS), Practical Byzantine Fault Tolerance (PBFT) [1], and others. Each algorithm has its own strengths and weaknesses, and is suited to different use cases depending on factors such as network size, transaction volume, and energy consumption requirements.

II. Problem Statement

We wanted to achieve high availability of data and decisions to be made among vehicles on roads by sacrificing the consistency of data shared on a scalable network. And this required a tailored consensus algorithm working for different scenarios with a highly unreliable number of vehicles within the swarm network. With the limited timeline of a semester, our research, and practical implementations, we wanted to restrict our focus on a useful but yet a simpler research upgrade on this category. We understood that information storage also requires consensus methods in a distributed network. So, we decided to achieve a high available storage system in a scalable distributed network sacrificing consistency as shown in Figure 1 [4].

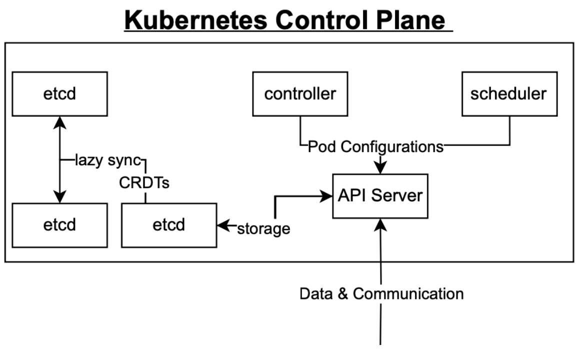


Fig. 1: Control Plane design to achieve eventual consistency with lazy sync

We pursued to understand Kubernetes object scaling with Pods, ReplicaSets, Deployments, Services, and more. LoadBalancers were prominent for scalability across the distributed system with Pods that failed. Research adapted to the swarm of vehicles for a real-world application, which requires consensus quickly on the road. This led to researching various methods for consistent, yet fast data transit across vehicles.

The Brewer Theorem (CAP Theorem) defines potential properties across a distributed system. The possibilities are broken into three outcomes, in which two can be focused on. In the event of a swarm of autonomous vehicles communicating on the road, the AP option (with Availability and Partition Tolerance), is the objective. Consistency can become lazy by extracting important and unimportant information from each vehicle with variable time for eventual consistency as seen in Figure 2 [4].

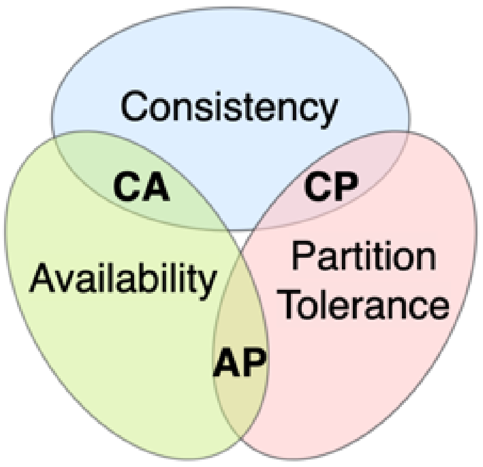


Fig. 2: CAP (Brewer) Theorem for distributed systems

An article shows common day technology development for autonomous vehicle research. Autonomous vehicles are facing a problem today which involves real-time communication on the road. Although vehicle-to-vehicle communication with 5.9-GHz frequency is becoming capable over various distances, an adaptive hybrid network concept is also important reflecting European standard IEEE 802.11ad which focuses on providing a framework for vehicle-to-infrastructure communication with radar technologies at 60-GHz frequency (shorter distances, faster speed). This includes utilizing wireless technologies like 5G and LTE mobile communications for distributed systems to transmit important data promptly while leaving non-essential data to sync eventually across a swarm of vehicles in a specific radius. Criteria for selecting wireless communication includes distance availability and signal quality [10].

III. Our Research

A. *Consensus Algorithms*

Our research began with improving any of blockchain's technologies and adapted to explore consensus algorithms. We sought to improve the consensus algorithms established, combine, or test out different methodologies and possibilities of consensus algorithms to tailor them for the vehicle and road scenarios. We analyzed Paxos [2, 3] and Raft algorithms, which are open-sourced and available on GitHub, by going through their source codes, we concluded that they are already working in optimal states.

We tried combining different ideas such as integrating Avalanche's leaderless-state algorithm with Proof of Reputation, and dividing reputation based on vehicle manufacturer's brands. We also thought of vehicles delivering information quickly among themselves within a range using a consensus algorithm known as Directed Cyclic Graphic. Although, since we wanted to achieve high availability, despite sacrificing the consistency on a scalable network, we focused on improving the Kubernetes etcd key-value store to become highly available across various Nodes while behaving with eventual consistency.

B. *Conflict-Free Replicated Data types (CRDTs)*

Within a kubernetes environment, Minikube's centralized database subscribes to edge Nodes' etcd to look for periodic changes for new Services (not Service location within containers of Pods). If all system clocks are synchronized across the network, CRDTs are used to induce lazy syncing between etcd Nodes and a leader etcd Node. This allows the etcd leader Node within the etcd cluster to read and write for quick responses from the API Server while all other etcd Nodes are prevented from consistent communication to the leader etcd Node and API server. This inconsistent communication can create conflicts of concurrent data write requests among Nodes. CRDTs resolve conflicting data synchronization by decentralizing the system [13].

State based conflict-free replicated data types (known as CvRDTs) transfer the entire combined local state of an etcd Node to the leader Node. The edge-etcd Node does not pass on the operation(s) it applied to change the data because it sends the entire new state of the data. This method requires all CRDT replicas to transmit the entire state to other replicas at one point to merge by function. The merge yields all previous state updates to the replica to monotonically increase the internal state and keep the same order of states. Despite being costly with large data transfer, the merges are commutative, associative, and idempotent where the operations can be scaled without affecting the state. A specific variant, called delta state CRDTs, provides recently applied state changes to replicas without sharing the entire state [13].

Operation based conflict-free replicated data types (known as CmRDTs) provides limited bandwidth which allows pushing to the remote state. This method only transmits update operations which contain small changes, yet accumulate to a large quantity of transactions. An example applied to a swarm of vehicles would be an update operation to an integer such as the velocity of the car changing 2 km/h from +60 km/h to +62 km/h. CmRDTs are commutative operations, not idempotent, and require guarantees from communication protocol between replicas to transmit all updates. This guarantee ensures no dropped or duplicated operations are prevalent along with ensuring transmitted operations are in causal order. The replicas receive updates and apply them locally to their own datastores to update their key-value pairs. The data is transmitted with protobuf schema files between etcd Nodes where a single round trip syncing process is necessary for edge Nodes [13].

The impact CRDTs leave on Kubernetes key-value stores includes potential stale data (which is not updated frequently) updates in other etcd Nodes while operating on data in a specific Node. An additional scenario which occurs is when two separate Nodes increase count in a ReplicaSet which causes two new Pods to be scheduled. The synchronized controller decides if it should keep both replicas (usually takes maximum value in key-value pair). These CRDT edge-cases show the potential that CRDTs bring when decentralizing Kubernetes with a control-plane on each worker Node to promote autoscaling, reduce cloud workload, lower latency, and permit 5G networks across edge devices [13].

IV. Design Architecture

A. Project Plan

While keeping eventual consistency as an objective, we are working to improve autonomous vehicles on roads with eventual consistency capable of exchanging individual metadata among others through cell tower infrastructure for distributing data. The car swarm could include vehicles traveling in the same direction when driving at approximately the same speed.

Eventual consistency would be used for absolute speed, distance, fuel consumption, and other variables which fluctuate continuously when driving (acts as operation-based changes). These values would be published from each car when the subscribed cell tower in range receives the updates about every minute.

Full consistency is necessary for state-based features of the system; this includes the number of vehicles within the swarm, important vehicle updates (breaking from single vehicle or proximity sensor alerts), cluster ID number for the swarm, and individual vehicle ID numbers active. Load utilization would also be fully consistent to ensure computation cycles within the edge cloud are not exploited from specific vehicle sources. These state-based changes would be written and received between each vehicle along with the cell tower every 10 seconds.

The cell tower subscribe to this data, partition the information, and republish this data back to the supplementary vehicles within the swarm to update their state's key-value stores accordingly (such as slow down, speed up, etc.). Figure 3 below denotes a scenario of three vehicles approaching a human hazard while transmitting data across 5G mobile communication to a cell tower to distribute the data back to the other vehicles for safety protocol [10].

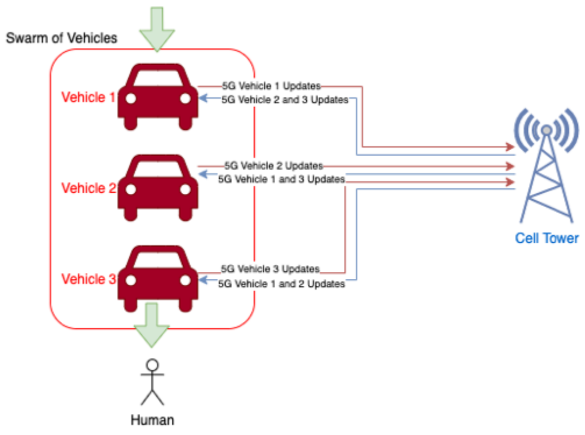


Fig. 3: Vehicle swarm approaching hazard and communicating metadata between vehicles

B. Emulated Ideation

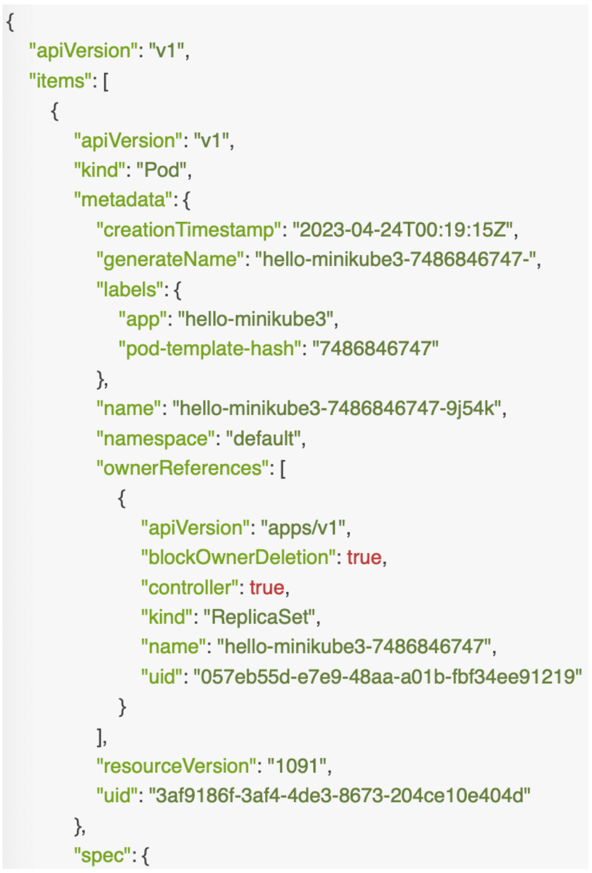
We came up with a variety of simulated scenarios to map the technologies required. We decided to work on two practical design implementations on Kubernetes.

1) MQTT Broker and Client: First, we implemented a HiveMQ MQTT broker with a publisher and subscriber client to emulate the transfer of information from cars to edge towers as shown in Figure 6. But we restricted our use-case to pass information from etcd to subscriber machines to make updates on edge infrastructure in-use. We planned to create a web-app using this data, so that an admin can keep track and monitor edge infrastructure.

To emulate this car swarm example, we determined that a cell tower server was needed to host a publishing and subscribing interface for edge devices to the edge cloud. We utilized Minikube to host a single Kubernetes cluster for architecting the system design. We started with multiple 'Hello-World' image containers across different Nodes to understand the JSON etcd output from all Kubernetes objects within the cluster through the API Server. The Kubernetes command-line tool 'kubectl' was used to analyze Minikube cluster state changes to extract etcd information from Deployments, Services, Nodes, Pods, and more. This led to our project of emulating a CvRDT between etcd Nodes within the cluster to establish unnecessary state data to be suitable for eventual consistency.

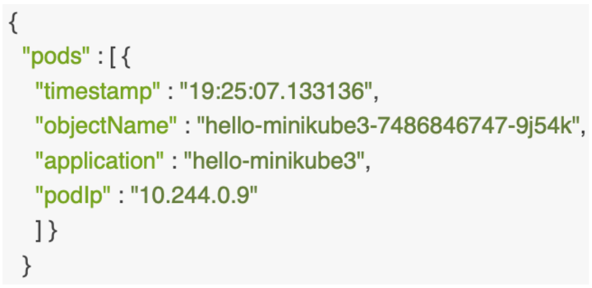
In the real-world application, the cluster (i.e. the swarm) would contain Deployments of Pods to imitate vehicles (edge-state etcd) which would communicate with the server or HiveMQ MQTT broker (cloud-state etcd). The broker Service would host a client capable of publishing and subscribing (a radio within a car). Another client (emulating the edge-cloud in the cell tower) would subscribe to important metadata as JSON output from the Deployment of a Pod (to mimic radios publishing personal car data) and partition the state details such as the object address (without knowing the internal mechanics of containers), number of Pods active within cluster, Pod IPs, and publish back to the other Pods. The client Service would analyze cluster state along with ReplicaSets to understand if 0 replicas are present, take action, and if 3-5 replicas are present, do not do anything. The Pods would subscribe back to the publisher to receive partitioned state updates from other Pods and run computations on the edge for merging states.

Figure 4 resembles actual unfiltered etcd data and Figure 5 shows the important metadata extracted to republish.



```
{
  "apiVersion": "v1",
  "items": [
    {
      "apiVersion": "v1",
      "kind": "Pod",
      "metadata": {
        "creationTimestamp": "2023-04-24T00:19:15Z",
        "generateName": "hello-minikube3-7486846747-",
        "labels": {
          "app": "hello-minikube3",
          "pod-template-hash": "7486846747"
        },
        "name": "hello-minikube3-7486846747-9j54k",
        "namespace": "default",
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "blockOwnerDeletion": true,
            "controller": true,
            "kind": "ReplicaSet",
            "name": "hello-minikube3-7486846747",
            "uid": "057eb55d-e7e9-48aa-a01b-fbf34ee91219"
          }
        ],
        "resourceVersion": "1091",
        "uid": "3af9186f-3af4-4de3-8673-204ce10e404d"
      },
      "spec": {
```

Fig. 4: Pod metadata example provided from etcd key-value pairs



```
{
 "pods" : [ {
  "timestamp" : "19:25:07.133136",
  "objectName" : "hello-minikube3-7486846747-9j54k",
  "application" : "hello-minikube3",
  "podIp" : "10.244.0.9"
  ] }
}
```

Fig. 5: Publisher partitioning the important data from above before republishing to subscribers

For initial emulation, a HiveMQ-CE (community edition) Docker image was deployed and exposed as a Service for hosting a small environment consisting of a separated local Python Paho publisher and subscriber. Using the command line terminal, the 'minikube tunnel' command exposed the HiveMQ Service to an external websocket Service target port (1883) from the internal NodePort 30005 defined within the Service YAML specifications. This tunneling permits Localhost accessibility through targetPort 1883 with web-host link: http://127.0.0.1:1883 [11].

Although, this expanded to utilizing a HiveMQ-4 Docker image with nginx Deployment (granting bash shell terminal access for debugging) as a cluster-based broker for high availability, scalability, and fault tolerance in the Minikube cluster [12].

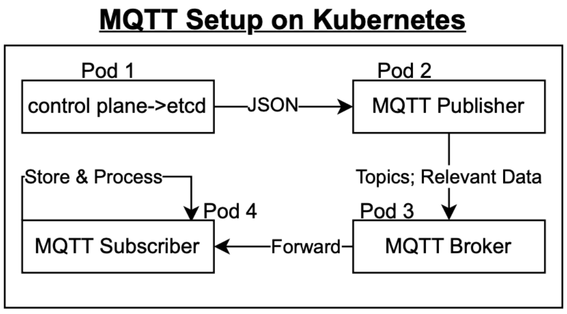


Fig. 6: MQTT implementation in Kubernetes to update infrastructure information

2) Kafka: Secondly, for a more robust approach for managing many brokers, we initially used a Kafka ZooKeeper setup to create a broker for hosting the cluster IP to communicate through port 9092. The broker permits messages through topics between a separated Kafka producer and consumer client along with access to Spark Streaming for pipelining data from cars to respective cell towers within regions and process the data by the consumer Services as shown in Figure 7. In later additions to this architecture, Kafka's robustness was used to partition JSON data for the ability to store large and complex data from edge Nodes in a structured database (acting as an edge cloud in a cell tower). The current example includes a vehicle driving from Bloomington to Indianapolis with several cell towers in between. These tower servers would partition the subscribed vehicle state data to identify a location and republish the data to a swarm of various other vehicles on the road in the same region. These updates would allow the vehicles to update their car speed accordingly.

To emulate this design, we utilized Kafka for separating the cell towers into regions (which is used to identify vehicles near the distinct towers). As the idea evolved, we researched various adaptations to use Kafka to partition data through our current HiveMQ broker Service. Due to complex data mapping with the Minikube Node interface, we speculated the Kafka extension of HiveMQ within the enterprise version trial for actual vehicle data simulations (refer to GitHub source code).

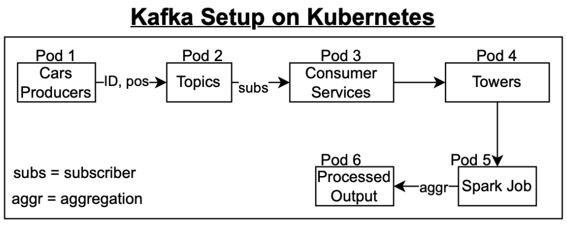
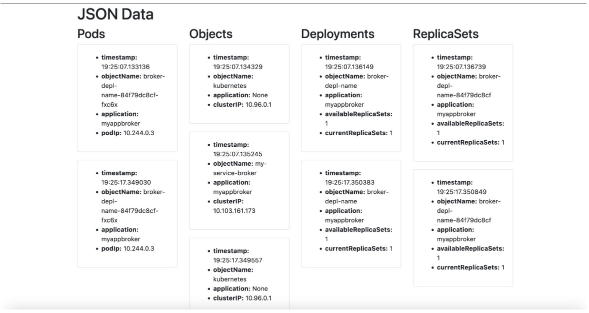


Fig. 7: Kafka implementation in Kubernetes to automate vehicle decisions

C. Web Application UI

The web application shows the overall infrastructure cluster details which can be used to see the locations and details of the cars (Pods) that are currently present on the edge. As of now, it is a static page where we are manually taking the output JSON file (which is replicated from the Paho pubisher client topic outputs) from the Kubernetes cluster and providing important metadata details on the user interface visualization utility (which includes the time of the sample, kind of Kubernetes object, cluster IP, Pod IP, image application name, name of the object, available replicas, and current replicas within the specific objects partitioned which is seen in Figure 8). We are yet to deploy this as a separate Pod on Kubernetes to dynamically fetch the changing JSON file from the publisher and update the user interface in real time. Additionally, this utility could be expanded and utilized for an admin to analyze Pod load utilization, swarm updates, IP address changes, and more.


JSON Data

Pods

- **timestamp:** 19:25:07.133136
- **objectName:** broker-depl-name-84f79dc8cf-fxc6x
- **application:** myappbroker
- **podIp:** 10.244.0.3

- **timestamp:** 19:25:17.349030
- **objectName:** broker-depl-name-84f79dc8cf-fxc6x
- **application:** myappbroker
- **podIp:** 10.244.0.3

Objects

- **timestamp:** 19:25:07.134329
- **objectName:** kubernetes
- **application:** None
- **clusterIP:** 10.96.0.1

- **timestamp:** 19:25:07.135245
- **objectName:** my-service-broker
- **application:** myappbroker
- **clusterIP:** 10.103.161.173

- **timestamp:** 19:25:17.349557
- **objectName:** kubernetes
- **application:** None
- **clusterIP:** 10.96.0.1

Deployments

- **timestamp:** 19:25:07.136149
- **objectName:** broker-depl-name
- **application:** myappbroker
- **availableReplicaSets:** 1
- **currentReplicaSets:** 1

- **timestamp:** 19:25:17.350383
- **objectName:** broker-depl-name
- **application:** myappbroker
- **availableReplicaSets:** 1
- **currentReplicaSets:** 1

ReplicaSets

- **timestamp:** 19:25:07.136739
- **objectName:** broker-depl-name-84f79dc8cf
- **application:** myappbroker
- **availableReplicaSets:** 1
- **currentReplicaSets:** 1

- **timestamp:** 19:25:17.350849
- **objectName:** broker-depl-name-84f79dc8cf
- **application:** myappbroker
- **availableReplicaSets:** 1
- **currentReplicaSets:** 1

Fig. 8: Web UI

D. Results

An automation script was created to easily deploy a Minikube cluster containing the two implementations mentioned combined in one virtual environment. First, the script deletes previous Minikube objects (Deployments, Services, ReplicaSets, and Pods) then creates and exposes a new HiveMQ broker in a Pod within a ReplicaSet. Next, the script builds, tags, and pushes a Paho publisher in a Pod within a ReplicaSet, and Paho subscriber in a Pod within a separate ReplicaSet utilizing Dockerfiles (containing a version of Python and Paho as installation requirements) to the user's Docker Hub account where the images can be deployed. Similarly, Kafka ZooKeeper extensions were modified to build, tag, and push the broker, consumer, and producer to the user's Docker Hub. Partitioning tools are used to extract the cluster IP from the HiveMQ-4 or Kafka ZooKeeper broker Pod image. The IP addresses that the publisher/producer and subscriber/consumer clients connect to through port 1883 or 9092 is updated to utilize this cluster IP address. This updated IP hosts the clients such as the publisher partitioning etcd JSON input and the subscriber receiving condensed JSON cluster state details through MQTT topics and subtopics. These clients within Pods mimic vehicles that act as half-duplex communicators (for only containing a publisher or subscriber as of now).

The first implementation plan succeeds in utilizing a single Kubernetes cluster through Minikube for accessing etcd key-value stores of distributed data across edge objects. The Docker container Deployment with the HiveMQ MQTT broker image exposes and hosts a LoadBalancer Service for publishers and subscribers through the cluster IP for first steps to developing a Service that can handle large amounts of data partitioning [12]. The Paho Python publisher client successfully partitions dumped cluster JSON metadata for specific attributes (seen through Kubernetes web-dashboard or bash shell) which can be utilized further for overall Kubernetes object state updates in a distributed system [14]. The Paho subscriber can view these key value-pairs in topics with latency induced between partition iterations to emulate real-time data transit with mobile communication protocols [14]. The second implementation successfully defines a real-world traveling scenario where a vehicle's data would be processed between various cell towers in-route to distribute among other vehicles on the road. The Kafka implementation was necessary to provide a robust method for vehicle data pre-processing on vehicles whereas the MQTT broker emulated a Kubernetes environment.

V. Future Work

We would like to conduct further research and discussions to tune partitioning within the current data from the Kubernetes cluster JSON dump for important and unimportant information. Currently, we extract Kubernetes object types, Cluster IPs, Pod IPs, object/image names, image applications/containers, current state replicas, and available replicas within a cluster, but we would like to gain a firm understanding of truly important state data along with emulating real vehicle data in the best-suited Kubernetes object type (e.g. Pod with ReplicaSet, Node, or even a Deployment exposed as a Service). Additionally, we would like to multi-process important and unimportant data across threads for inducing suitable latency (e.g. 10 seconds, 1 minute, 2 minutes, etc.) for eventual consistency within vehicles to remain safe on the road. This method would allow parallelism to synchronize unimportant metadata with longer delays but require additional consensus analysis and CRDTs.

For the nginx Paho client applications, full-duplex bi-directional network data transmissions would permit simultaneous publishing and subscribing within currents Pods to simulate a vehicle transmitting its own real data from the Kafka simulation (i.e. Pod metadata) and receiving data from other vehicles (i.e. partitioned Pod metadata) through the cell tower which partitions the data (which is the MQTT broker acting as publisher and subscriber for releasing important metadata). This process would involves conflicts and require eventual consistency for merging states (i.e. CvRDTs) or updating values (e.g. velocity) solely from operation changes (i.e. CmRDTs) with other vehicles (e.g. second vehicle slows down 5 km/h) [13].

References

[1] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[2] Roberto De Prisco, Butler Lampson, and Nancy Lynch. “Revisiting the Paxos algorithm”. In: *Theoretical Computer Science* 243.1-2 (2000), pp. 35–91.

[3] Leslie Lamport. “Paxos made simple”. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[4] Nancy Lynch and Seth Gilbert. “Brewers Conjecture and a characterization of the limits, and relationships between Consistency, Availability and Partition Tolerance in a distributed service”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59.

[5] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized business review* (2008), p. 21260.

[6] Fangyu Gai et al. “Proof of reputation: A reputation-based consensus protocol for peer-to-peer network”. In: *Database Systems for Advanced Applications: 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part II 23*. Springer. 2018, pp. 666–681.

[7] Nada Lachtar et al. “A cross-stack approach towards defending against cryptojacking”. In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 126–129.

[8] Fahad Saleh. “Blockchain without waste: Proof-of-stake”. In: *The Review of financial studies* 34.3 (2021), pp. 1156–1190.

[9] Wenbing Zhao et al. “On peercoin proof of stake for blockchain consensus”. In: *2021 The 3rd International Conference on Blockchain Technology*. 2021, pp. 129–134.

[10] EBV Elektronik. *Communication in autonomous vehicles*. URL: https://future-markets-magazine.com/en/markets-technology-en/communication-in-autonomous-vehicles/#:~:text=Communication\%20in\%20autonomous\%20vehicles\%20will,their\%20surrounding\%20area\%20via\%20WLAN. (accessed: 05.01.2023).

[11] *hivemq/hivemq-ce*. URL: https://hub.docker.com/r/hivemq/hivemq-ce. (accessed: 05.01.2023).

[12] *hivemq/hivemq4*. URL: https://hub.docker.com/r/hivemq/hivemq4/. (accessed: 05.01.2023).

[13] Roshan Kumar. *When to use a CRDT-based database*. URL: https://www.infoworld.com/article/3305321/when-to-use-a-crdt-based-database.html. (accessed: 05.01.2023).

[14] Roger Light. *Paho Python - MQTT Client Library Encyclopedia*. URL: https://www.hivemq.com/blog/mqtt-client-library-paho-python/. (accessed: 05.01.2023).