VGuardDB: An Extension of VGuard for Efficiently Storing and Accessing Data from V2X Networks

Abstract

VGuard proposes a new permissioned blockchain that achieves consensus for vehicular data under changing memberships. However, VGuard only produces chained consensus results of data entries but does not store them in a database but only on a log file, making it challenging to retrieve data from the blockchain for analysis and decision-making. Additionally, the VGuard paper has not specified the structure of the supported data entries or the use cases that generate them. To address these issuses, we introdudce VGuardDB, which adds a database layer to VGuard, where each vehicle has a distributed database to store data, provide access to the agreed data through the read capabilities of the database layer and define specific data structures. As such, the proposed enhancements to the VGuard blockchain system will enable an efficient and reliable data storage and retrieval, improving the usability of the system and enhancing its practical application.

1 Introduction

With the quickly growing popularity of smart vehicles as well as smart transportation systems, the need for efficient and reliable communication between vehicles and the surrounding infrastructure is becoming more and more important. Vehicle-to-everything (V2X) communication is a promising technology that enables vehicles to communicate with each other and the surrounding infrastructure, such as traffic lights, road signs, and other vehicles. V2X communication can be used to improve road safety, reduce traffic congestion, and improve the efficiency of transportation systems.

However, the vast amount of data generated by V2X networks are often unaccessible to users and monopolized by auto manufacturers. In response, there has been an increasing demand for a distributed data storage option for V2X data. However, the unpredictable dynamic nature of road and vehicle conditions renders applying traditional distributed data storage solutions such as blockchians difficult in a V2X environment. To address these issues, VGuard[] has been proposed as a permissioned blockchain that achieves consensus for vehicular data under changing memberships. VGuard is designed to be used in V2X networks, where the consensus is achieved by the vehicles in the network. However, VGuard only produces chained consensus results of data entries but does not store them in a database but only on a log file, making it challenging to retrieve data from the blockchain for analysis and decision-making. Additionally, the VGuard paper has not specified the structure of the supported data entries or the use cases that generate them.

To fill in this missing component, we introduce VGuardDB, which creates and additional database layer on top of VGuard, where each vehicle has a distributed database to store data, provide access to the agreed data through the read capabilities of the database layer and define specific data structures. VGuardDB ensures data avaialbility as long as a the number the number of unavaible vehicles, including the proposer, is smaller than a configurable value. Additionally, we disign VGuardDB such that only the configured number of randomly selected vehicles need to store data on the chain at any given point in time, in order to save storage space on vehicles. As such, the proposed enhancements to the VGuard blockchain system will enable an efficient and reliable data storage and retrieval, improving the usability of the system and enhancing its practical application.

The rest of the paper is organized as follows. Section 2 provides an overview of related work in the field. Section 3 describes the architecture and design of VGuardDB, including the extension of VGuard, the storage layer, and the use of MySQL. Section 4 presents the experimental results and evaluation of VGuardDB's performance. Finally, Section 5 concludes the paper and outlines future research directions.

2 Related Work

There have been some existing works that aim to create a fusion between blockchain and distributed databases in order to combine the advantages of both worlds. BlockchainDB[]

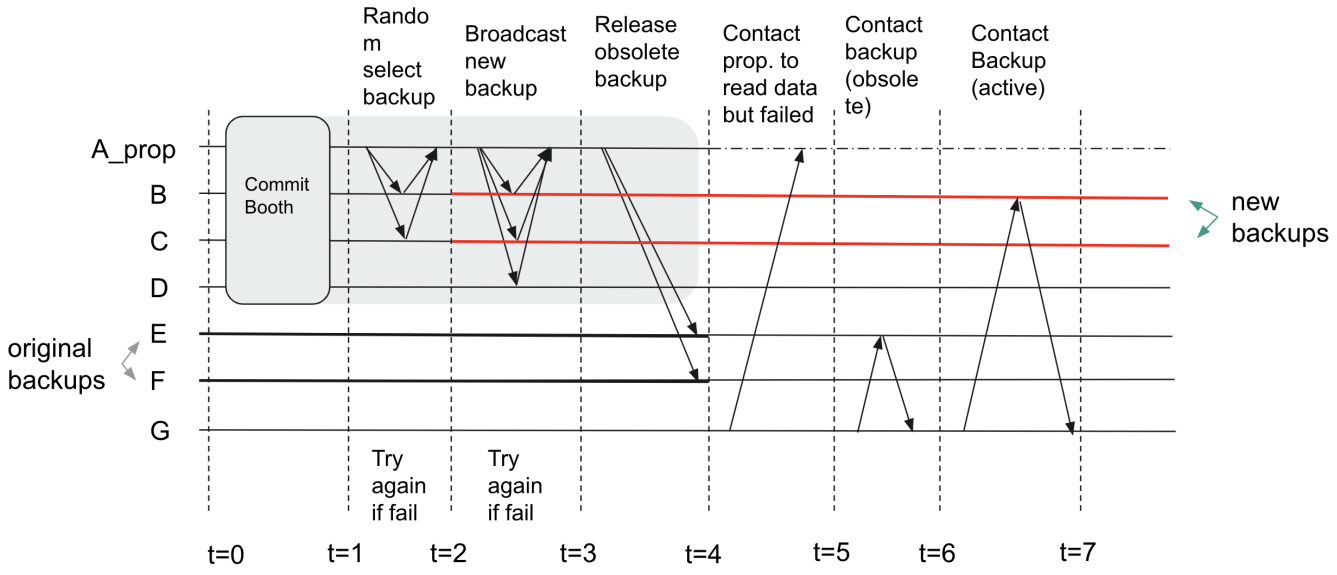

Figure 1: The overall workflow of VGuardDB, illustrated with an example of one proposer (A) and six validators (B to G). At t=0, A hosts a commit booth with B, C, and D. E, and F are vehicles that are chosen as backups from a previous booth. The back-up replication phase is triggered at t=1, when the commit booth ends. A first randomly chooses B and C to be backups, and upon success, broadcast the information to vehicles in the booth. A then startes the data-release phase by informing E and F that they are no longer the backups, and are now free to delete their local data. At t=4, Vehicle G initializes the read phase by sending a read request to A. By default, all vehicles are configured to read from the proposer first, but in this case, A has gone offline (as illustrated in dashed line), so G attemps to read from of the backups according to its local backup list. Since G is not informed that B and C are the current active backups, it makes the request to E, one of the now obsolete backups. E then informs G that B is the active backup, and G makes the request to B. B then informs G that it has the data, and G receives the data.

proposes a novel architecture that uses a scalable and efficient database as a storage layer and a blockchain as an append-only log that records the history of data modifications. Another example is Blockchain Relational Database (BRD)[], a novel architecture for a blockchain relational database that leverages the rich features of a replicated relational database and extends them with blockchain properties such as immutability, provenance, and consensus. Regarding the applicability of V2X blockchain technology, some research works have analyzed and categorized some real use-case scenarios. For example, [] presents a taxonomy of design use cases and system architectures for blockchain applications in V2X communication and discusses how blockchains can be used to record and distribute data such as software updates, driver behaviors, and accident reports among different stakeholders.

3 Assumptions

Due to unique nature of V2X networks, VGuardDB makes the following assumptions:

- All vehicles are (eventually) reachable through cellular (or other) network via their unique identification (e.g. public key). (Note this is different from the direct vehicle-to-vehicle connection in the booth).
- Vehicle to vehicle connections is available for vehicles in the same booth.
- At most N replica unavailable simultaneously at any given time, where for k backup databases, $N \leq k$.
- All vehicles are willing host a backup databse as long as there is reasonable fairness in the selection process.

4 Design

The core logic of VGuardDB can be divided to 3 independent phases, which are the backup-replication phase, the data-release phase, and the data-read phase. The backup-replication phase is responsible for making backups of the data on the chain. The data-release phase is responsible for releasing data from obsolete replicates. The data-read phase is responsible for reading the data from the database. The overall workflow is illustrated in figure 1. The following subsections describe the design of each phase in detail.

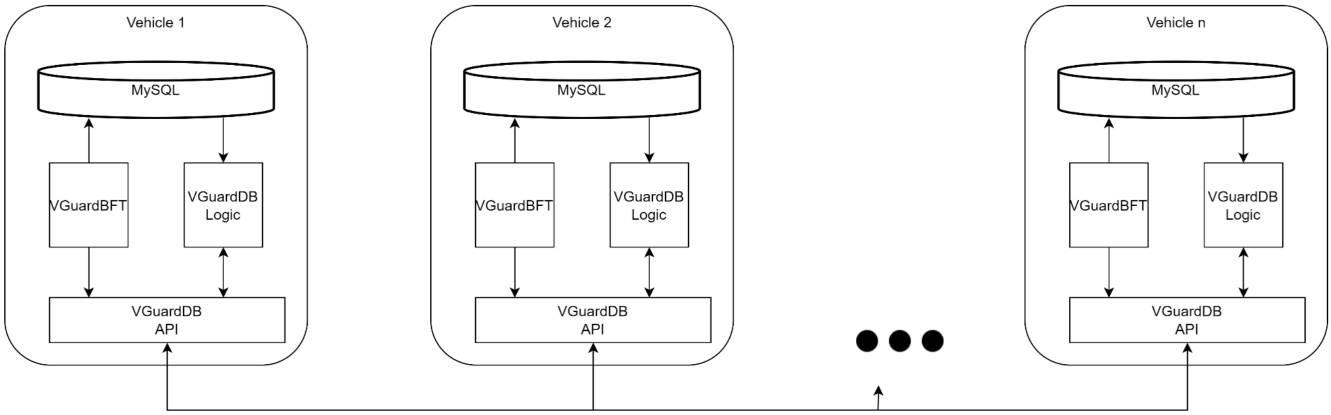

Figure 2: The overall architecture of VGuardDB implementation.

Backup-Replication Phase The backup-replication phase is responsible for making backup database on the chain. The backup-replication phase is triggered when at the end of the commit consensus phase in VGuard. Upon which, VGuard sends a request to VGuardDB to make backups of the data on the chain. VGuardDB then randomly selects a k vehicles from the current participants in the commit consensus phase, and sends a request to each of them to make a backup of the data on the chain. If the request is successful, VGuardDB then sends a request to the proposer to broadcast the information of the new backups to the vehicles that participated the commit consensus booth. The vehicles then update their local backup list with the new backups.

The following pseudocode illustrates the logic of the `backup_replication` function:

Algorithm 1 backup_replication(payload, conn)

```
status ← False
participants ← payload['participants']
backup_list ← current_backup_list
while not status do
    status, chosen ← make_random_backups(participants, conn)
    if not status then
        continue
    end if
    status ← broadcast_backups_to_booth(chosen, participants)
    backup_list ← update_backup_list(chosen, backup_list)
    request_delete_obsolete(backup_list)
end while
```

Data-Release Phase. In order to save storage space on vehicles, VGuardDB only requires a configurable number of k vehicles to store data on the chain at any given point in time. The data-release phase is responsible for releasing data from obsolete replicates. The data-release phase is triggered when a vehicle receives a request to delete obsolete backups. Upon which, the vehicle deletes the data from its local databse and sends and updates its own backup list with the news backup inofrmation.

Data-Read Phase The data-read phase is responsible for reading the data from the database. The data-read phase is triggered when a vehicle in the chain receives a requests for triggering the data-read process. Upon which, the vehicle act as a client and sends a read GET request to the VGuard proposer instance. This is because the proposer will always have most recent version of data. Under no failures, the proposer would simply receive the request and returns the requested data to the client vehicle.

However, in where case that the proposer fails, the client vehicle will recieve the failure response and sends the data request to one of the backup vehicles according to its local backup list. It is worth noting that the client vehicle would only have a copy of the most rencent backuplist if it has participated in the latest booth. However, this may not be the case, hence its local backup list may be outdated. In this case, the client vehicle would send the data request to one of the *obsolete backup* vehicles. There are two scenarios when this happens: 1) the *obsolete backup* has been informed that it is obsolete through the data-release phase, or; 2) It is oblivious that is is now obsolete. In the first case, the *obsolete backup* would return a failure response and forward a copy of its own backup list to the client vehicle. The client vehicle would then update its own backup list with the new backup list and send the data request to one of the new backup vehicles. In the second case, the *obsolete backup* would return the requested data to the client vehicle. Note that this means that the data may not necessarily contain the newest update. This can be seen as a trade-off between data freshness and storage space.

5 Implementation

We design VGuardDB in a way that it can be easily integrated into existing VGuard implementations in a modular fashion. The VGuardDB architecture consists of three main components: A Python Flask-based API, A main logic layer, and a MySQL-based data storage layer. The API layer is responsible for receiving requests from the vehicles and forwarding them to the main logic layer. The main logic layer is responsible for the main logic of the system, including making database replicas as well as handling data-read functionality. The storage layer is responsible for storing the data and is shared with VGuard.

The architecture of VGuardDB is shown in Figure 2. As illustrated, each vehicle runs an instance of a SQL databse, a VGuard instance, a VGardDB logic instance, and a VGuardDB API instance. On the end of commit phase of VGuard, VGuard sends a GET request to VGuardDB API along with the list of vehicles that participated in the commit phase. VGuardDB API then forwards the request to VGuardDB logic. VGuardDB logic then initializes the processes described in §4, utilizing VGuardDB API to faciliate communication with other vehicles, while exchanging data with VGuard directly through the shared SQL databse for tasks such as data-replication.

6 Evaluation