Simulating Secure Data Exchange and Storage for Urban Air Mobility Environments

Aidan Jones*
NASA Ames Research Center, Moffet Field, CA 94035, USA.

Norbert Gillem†
NASA Ames Research Center, Moffet Field, CA 94035, USA.

Abdullah Abdul‡
Morgan State University, Baltimore, MD 21251.

Nishant Sharma§
Intrinsyx Technologies Corporation, Moffet Field, CA 94035, USA.

Chaz Chang¶
San Jose State University, San Jose, CA 95152, USA.

Kenneth Freeman‖
NASA Ames Research Center, Moffet Field, CA 94035, USA.

Urban Air Mobility (UAM) defines an environment for managing operations of vertical takeoff and landing (VTOL) and short takeoff and landing (STOL) vehicles in an urban environment. Within a UAM environment, UAM operators manage fleets of vehicles, relying on Providers of Services for UAM (PSUs) for managing flights in a region of airspace. Flight plan deconfliction is primarily performed by the Discovery and Synchronization Service (DSS), and the Federal Aviation Administration (FAA) maintains control over the UAM space via the FAA-Industry Exchange Protocol (FIDXP). UAM is a federated environment with many different entities owning and operating vehicles, PSUs, and other services. These entities often need to interoperate or access data generated by other organizations. This paper demonstrates the feasibility of using blockchain to facilitate a secure data exchange and storage for this flight information in a UAM environment. In particular, this paper is focused on flight plans and telemetry data. A blockchain network was developed with a set of smart contracts for managing relevant flight data. Hyperledger Fabric was chosen as it is performent, scalable, and allows organizations to reuse existing public key infrastructure (PKI) for identity management. A set of simulated UAM services were also developed. These services propose flight plans and negotiate with other UAM services for airspace access. All interactions between UAM services, as well as vehicle telemetry data, is recorded onto the blockchain. Vehicle telemetry data is generated by a vehicle flight simulation service. This paper successfully demonstrates the feasibility of using blockchain as a secure data exchange and storage mechanism in a UAM environment.

I. Background

Urban Air Mobility (UAM) defines an environment for managing the operations of vertical takeoff and landing (VTOL) and short takeoff and landing (STOL) vehicles in an urban environment [1]. Within a UAM environment, UAM operators manage vehicle operations, relying on Providers of Services for UAM (PSUs) and Supplemental Data

*Software Engineer, NASA Ames Research Center, Moffet Field, CA 94035, USA.
†Aerospace Engineer, NASA Ames Research Center, Moffet Field, CA 94035, USA.
‡Intern, Morgan State University, Baltimore, MD 21251.
§Systems Engineer, Intrinsyx Technologies Corporation, Moffet Field, CA 94035, USA.
¶Intern, San Jose State University, San Jose, CA 95152, USA.
‖Aerospace Engineer, NASA Ames Research Center, Moffet Field, CA 94035, USA.

Service Providers (SDSPs) for support. PSUs are of particular importance, as they are responsible for managing flights for multiple UAM operators within a region. This is comparable to the role of air traffic control in the National Airspace System (NAS). In addition, the Federal Aviation Administration (FAA) must maintain certain controls over UAM environments in order to coordinate UAM and NAS operations. The FAA accomplishes this through the Flight Information Management System (FIMS), for which the primary interface for UAM participants is the FAA-Industry Exchange Protocol (FIDXP). As UAM environments are cooperative in nature, with multiple organizations participating, there is a need for synchronization and conflict management among the different entities. The primary mechanism in place for this is the Discovery and Synchronization Service (DSS), which tracks operations within different airspace regions. The notional architecture for UAM can be seen in figure 1.

Ensuring a strong cybersecurity posture is paramount to successful UAM operations. In order to achieve public acceptance of aerial transportation in densely populated areas, strong security and safety guarantees must be made. Cybersecurity in UAM is required to secure both physical infrastructure and data. In this case, data can belong to individual customers, operating organizations, or be public.

II. Motivation

The objective of this work is to leverage a permissioned blockchain for secure data exchange and storage in a simulated UAM environment. UAM environments are federated in nature, meaning there are multiple organizations managing their own services, identities, and infrastructure. However, these entities often need to interoperate, necessitating a secure way to exchange data. Blockchain is ideal for such use cases, allowing multiple organizations fine-tuned, granular, and consistent control over shared data. Examples of such data are identities, flight plans, customer and employee data, logs, and vehicle telemetry. The simulation discussed in this paper focuses on flight plans and telemetry data.

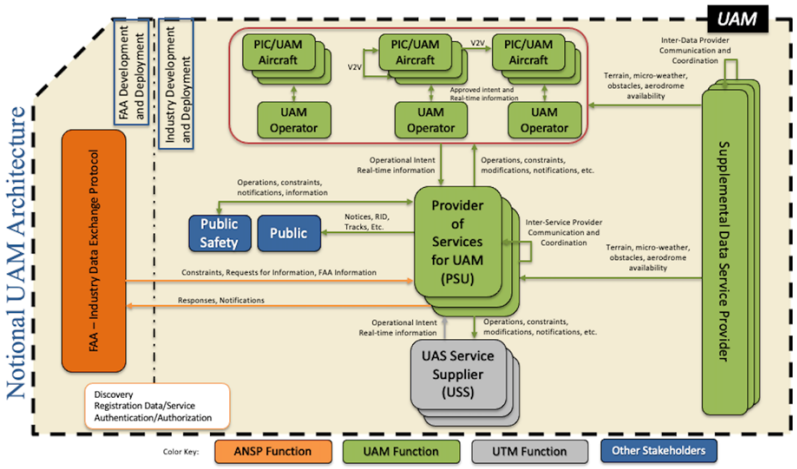

Fig. 1 Notional UAM Architecture

Previous research [2] into using blockchain for UAM data exchanges focused on the general feasibility of such an approach, as well as addressing various cybersecurity concerns. In contrast, this paper aims to better simulate an actual UAM environment, mocking many of the core services. Specifically, this paper focuses on information exchanged between UAM operators, PSUs, and a DSS during the process of a UAM operator claiming a specific airspace for a flight. The flight plan, also referred to as operational intent in this paper, is saved on the blockchain, as are any constraints imposed by other organizations who have their own claims on said airspace. This paper also explores the process of generating and recording realistic telemetry data after the flight plan is accepted.

III. Blockchain

A blockchain is a distributed and immutable ledger [3]. That is, it is a data structure that records transactions between various entities in a distributed environment, where each transaction is appended. It should be noted that the term transaction here refers to an arbitrary addition of data recorded on the blockchain. Each transaction is signed by

the private key of the entity submitting the transaction, so that other users can verify with the corresponding public key. Each node in the blockchain network typically has a full copy of the ledger.

A block in the blockchain consists of a set of transactions grouped together, a hash of the previous block, as well as some additional metadata. Including the hash of the previous block creates a chain of blocks, where the state of data at some block i can be verified by computing the hash of the $i-1$ block, and so on up the chain. The first block in a blockchain is the only one without a hash, and is called the genesis block. This chaining together of blocks with cryptographic hashes is how blockchains can guarantee immutability, as changing the value of a previous transaction will change the value of the hash. In addition, all nodes in a blockchain network must reach an agreement in terms of which transactions are included in a block. This process is called consensus. There are many different mechanisms for consensus, though the most common are proof of work and proof of stake.

Permissioned blockchains, also called private blockchains, restrict access to a specific group of verified users. This is in opposition to public blockchains, such as Bitcoin, where anyone can participate by running nodes or submitting transactions. Hyperledger Fabric (HLF) [4], the blockchain used in this research, is an example of a permissioned blockchain.

A. Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain framework created by the Hyperledger Foundation, a nonprofit organization created to host open-source enterprise blockchain frameworks and other distributed ledger technology (DLT) [5]. HLF is a modular framework that grants blockchain developers a lot of control over how they build their blockchain network and applications. Some of its key characteristics include channels, the way HLF implements smart contracts and transactions, its modular consensus mechanism, and how it handles identity management.

Most blockchains have a single ledger where all of the transactions are written to. However, HLF introduces the concept of channels. A channel is a distinct ledger, but due to HLF's architecture, the identity management and consensus components can be reused across all channels. This makes it easy to partition data into different ledgers based on use case without any additional overhead. In the UAM use case, telemetry and operational intent can be kept on separate ledgers. This is useful as the ledgers and associated smart contracts can be tuned differently. For example, high throughput is important in the telemetry use case due to the large volumes of data. On the other hand, a channel for operational intent might be more concerned with decreasing latency, as it could be blocking flight operations.

Smart contracts and transactions work differently in HLF than in other blockchains. Ethereum was the first blockchain to introduce smart contracts [6], and as such will be the comparison for this section. In Ethereum, contracts are written in a deterministic language called Solidity and deployed onto the chain itself. This means that transactions are first ordered and then the smart contract is executed on each peer. Due to the nature of Solidity, deterministic output from the contract can be guaranteed. HLF fabric changes this process in a few ways. First of all, in HLF every transaction is the result of invoking a smart contract [4]. There are no tokens or currency in HLF, so the concept of transferring ownership of a token does not inherently exist. Second of all, smart contracts can be written in a variety of languages such as Go, Java, and NodeJS. These languages are naturally non-deterministic, thus smart contracts written in them must be deployed off chain. In practice, smart contracts typically run in Docker containers, and it is up to the peers who run the contract to verify the output is correct. After peers execute the contract, the result can then be ordered into blocks and validated against some policy before finally being committed to the blockchain. Using general purpose programming languages to execute smart contracts off chain allows a subset of peers in a network to execute the contract which can improve network scalability and performance. Furthermore, as it is only the result of the contract that is used to check if the execution was successful, the non-deterministic nature of general purpose programming languages does not make the output of the contract non-deterministic. In addition, general purpose programming languages offer significantly more flexibility in terms of the logic that can be implemented in smart contracts. A more generalized example of the transaction flow in HLF can be seen in figure 2.

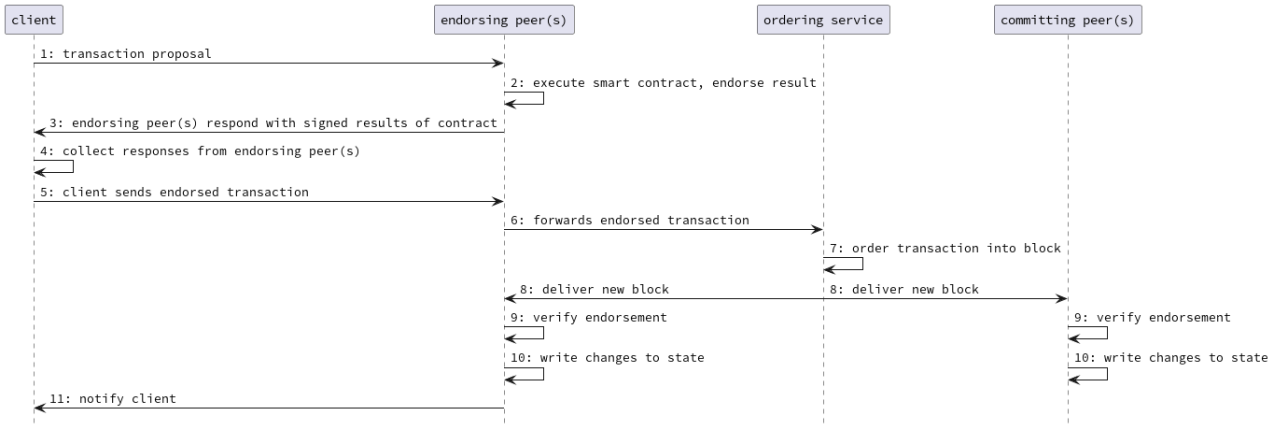

Fig. 2 Hyperledger Fabric Transaction Flow

Consensus is the process in which each node in the blockchain network reaches an agreement on what the state of the ledger is. The most common protocols are proof of work and proof of stake, though many others exist as well [7] [8]. In public blockchains any node can participate in this process, whereas in HLF there is a centralized service for ordering transactions into blocks. Benefits to this method include allowing deterministic algorithms to be used for consensus as well as separating the logic of ordering transactions with that of executing smart contracts [4]. This generally improves performance and scalability of the network, while also preventing forking of the ledger. While logically centralized, the ordering service typically consists of multiple nodes. HLF's modular architecture makes it simple to support a variety of algorithms, namely Raft, Kafka, and practical Byzantine Fault Tolerance [9]. Raft is the most common due to its simplicity and performance.

Identity management in HLF is different than most blockchains as well. Public blockchains are completely decentralized, meaning an identity simply consists of a private and public keypair. There is no concept of organizations, and anyone can create any number of identities. HLF, however, leverages traditional public key infrastructure (PKI) to manage identities [4]. In a HLF network, each identity belongs to one of the organizations that is allowed to participate. These organizations issue identities internally using certificate authorities, and access control is performed by verifying each identity belongs to the correct organization using its membership service provider (MSP). An MSP is a data store of certificates that is shared with each participating organization so that external identities can be verified and authorized [4]. This has the benefit of not introducing a new identity management paradigm, instead allowing organizations to leverage existing infrastructure for blockchain operations.

IV. Secure Data Exchange Blockchain Network

The focus of this work was to demonstrate the feasibility of using blockchain for a secure method of data exchange and storage in a UAM environment. The focus was not to test the fault tolerance, resiliency, or performance of a permissioned blockchain. As such, the developed blockchain was built with a focus on simplicity, keeping the number of organizations, peers, and ordering nodes low. The network consists of four organizations, three of which represent private companies that operate PSUs, vehicle fleets, and other services. The fourth organization represents the FAA. Each organization has a single peer node that operates as both an endorsing peer and a committing peer, as well as a certificate authority (CA). The FAA organization also owns the ordering service, which consists of a single node. The architecture of the blockchain network can be seen in figure 3.

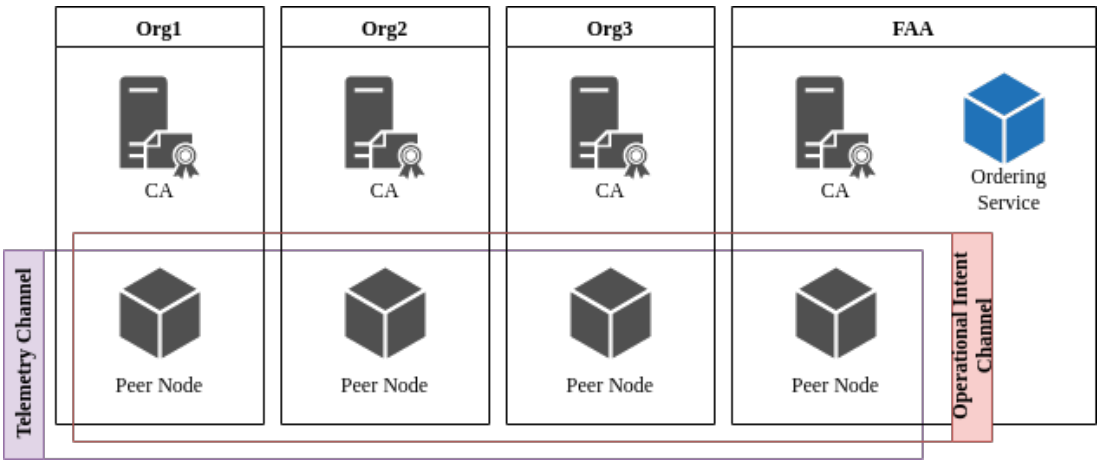

Fig. 3 Secure Data Exchange Blockchain Network

There are two channels, one for telemetry and one for operational intent. Each channel has a corresponding smart contract with a set of functions for managing data in that space, as well as a set of read-only functions. For example, there is a function for proposing an operational intent; a function for recording the response of the DSS, FIDXP, and alternate PSU; as well as a read-only function for accessing recorded intents. Read-only functions do not generate new transactions, though to be invoked the endorsement policy must still be met.

Smart contracts were developed for the different use cases of the simulation. Of note, there is a contract with a set of functions for managing flight plans, as well as one for managing telemetry data. Other contracts were developed in previous work [2], though these contracts were not used in this simulation.

The CAs are responsible for issuing identities to users within an organization, as well as verifying identities both internally and externally. Identities outside of an organization can be verified by using the Membership Service Provider (MSP), which defines the preconfigured accepted organizations that can participate in a HLF network. The root certificate from each organization's CA is shared through the MSP. Identities within an organization are verified using traditional PKI mechanisms.

V. Secure Data Exchange UAM Simulation

The Secure Data Exchange UAM simulation defines multiple services, many of which mock services found in actual UAM environments. There are three primary organizations in the simulation, Org1, Org2, and the FAA. Org1 and Org2 represent private organizations that own and operate vehicles, a single UAM operator, and a single PSU. There is also DSS used for discovery and synchronization between the two organizations, a FIDXP service to mock FAA authorization of flight plans, a flight simulation service that generates telemetry data once a flight plan is approved, and an API wrapper around the blockchain. The DSS, FIDXP, flight simulation service, and blockchain API all belong to the FAA organization. The architecture of the UAM simulation can be seen in figure 4.

Each UAM operator has the capability to submit flight plans to its corresponding PSU. The PSU then ensures the airspace is available, modifying the flight plan and communicating with the DSS and other PSUs if necessary. Once the PSU has a valid flight plan, it returns it back to the UAM operator, which then can notify the flight simulation service to begin the flight. All exchanges, updates to the flight plan, as well as final telemetry data is recorded on the blockchain. This process is transparent to the UAM operator.

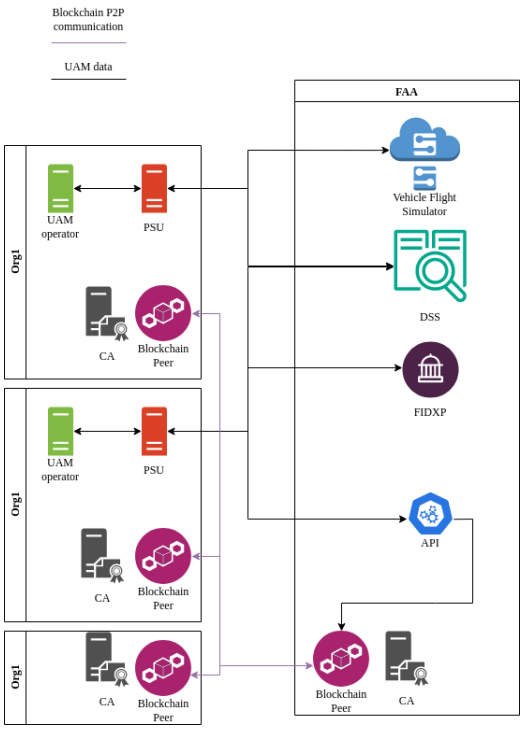

Fig. 4 UAM Simulation Architecture

Both the UAM environment and the actual functionality of the services was simplified compared to other work in the UAM space. The objective of this simulation was not to develop a functional PSU, DSS, or other UAM services, but to demonstrate that the information exchanged when these services operate can be captured on a blockchain. In particular, the process of operational intent deconfliction has been simplified.

A. Data Flow

This section outlines first the possible exchanges between services when a UAM operator submits an operational intent. Then it discusses how the telemetry data flows from the flight simulation service to the blockchain. When this simulation was designed, a decision was made to make PSUs the only services that interacts with the blockchain. This interaction is done exclusively via the blockchain API service.

1. Operational Intent

From a UAM operator's perspective, the process for submitting a flight plan is simple. It makes an API call to the PSU with its operational intent included in the request, and then receives a response. The response will either be the same intent back if it is clear to fly, a slightly modified intent if there are resolvable conflicts, and a rejection if there are conflicts that cannot be resolved.

When the PSU receives an intent, it first records the proposed intent onto the blockchain. Then, if there is no conflict with other flights managed by this PSU, it will make an API call to the DSS to check if other PSUs have flights that conflict. The DSS will either return a constraint or an all clear message. This response is recorded onto the blockchain. If there is a constraint, the constraint is recorded as well. In the case of a constraint, the PSU will try to adjust the intent if able, making an additional API call to the DSS after each adjustment and recording the additional responses. The number of retries the PSU makes is called the RETRY_COUNT, and is a configurable parameter.

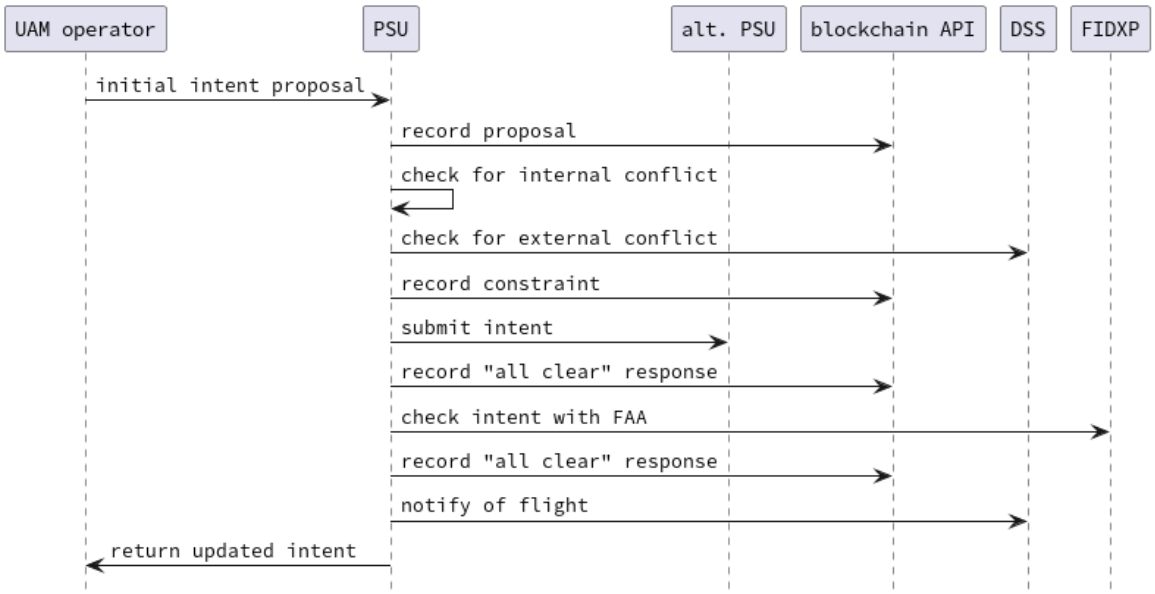

Fig. 5 Operational Intent Submission Sequence

If the DSS does not return an all clear message, then the PSU must communicate with the other PSU that has a conflicting flight. The PSU responsible for the conflicting flight is returned by the DSS when it returns a constraint. The process for communicating with the alternate PSU is similar to the DSS. The intent is sent and a response is received and recorded onto the blockchain. This response is either an all clear message or an additional constraint. If a constraint is returned, the PSU may try to adjust its intent before retrying. If the intent cannot be deconflicted after RETRY_COUNT + 1 attempts, the PSU returns an error to the UAM operator that their flight plan cannot be accepted.

If an operational intent has been successfully deconflicted, then one last check must be made with the FIDXP. The FIDXP will never return a constraint, but rather either an all clear or a deny message. In this simulation, the FIDXP will accept all flight plans. This response is recorded onto the blockchain as well. Now that the operational intent has been accepted, the PSU makes one last API call to the DSS to notify it that a vehicle will be flying on that intent, records the intent in their own internal database, and returns the final approved intent to the UAM operator. This information flow can be seen in figure 5 in the case where a conflict is found in the DSS and must be resolved with an alternate PSU. In that example, RETRY_COUNT is set to 0.

2. Telemetry

Once a UAM operator receives the approved intent back from its PSU, it sends the intent to the flight simulation service. The flight simulation service simulates a vehicle flying along the given operational intent, generating a telemetry data point every second that gets recorded into the blockchain. The telemetry information goes from the flight simulation service, to the UAM operator, to the PSU, and then it gets saved into the blockchain via the blockchain API. To prevent additional overhead from message handshakes and access control logic, a persistent connection is made between the four services to allow a stream of data to be forwarded between the services for the duration of the simulated flight.

B. Technology Stack

1. UAM Services

The UAM services were developed as HTTP APIs using Python and Flask. Each service runs independently, and each message exchanged between services during negotiation of an operational intent is a discreet HTTP request and response. Since telemetry data flows consistently once the simulated flight begins, websockets were used to decrease overhead. An HTTP call is made to initiate each websocket connection, and then the telemetry points, formatted as JSON objects, are passed over the chain of websockets. Websocket connections exist between the flight simulation service and the UAM operator, the UAM operator and the PSU, and the PSU and the blockchain API.

The DSS and PSUs must maintain an internal state in regards to what flights they are managing. These services use MongoDB instances to record this data. In addition, the vehicle flight simulation service has knowledge of different routes simulated vehicles can take. This information is saved in a PostgreSQL database, and is retrieved when the service receives an intent. This database, as well as the MongoDB databases, were all deployed in Docker containers, and all databases and services are deployed on the AWS cloud.

2. Blockchain & Smart Contracts

The blockchain was also deployed on the AWS cloud using Docker containers. Each peer, CA, ordering node, and smart contract runs in its own container. The network is orchestrated using docker-compose to easily manage nodes and CAs. When a smart contract is deployed, a new Docker container is created for that contract. In this network, each peer has two smart contracts deployed on it, so two containers get created for each peer in addition to the container where the peer itself runs. For CAs, the HLF-provided fabric-ca was used. The fabric-ca certificate authority integrates well with other HLF software, though its use is not required.

The smart contracts are written in Go and consist of, at minimum, three files: go.mod, main.go, and contract.go, where contract.go is replaced with the name of the contract. For example, for the operational intent smart contract, the file is called operational_intent.go. The go.mod file is a necessary artifact of all Go modules, listing dependencies and where to find them. The main.go file defines a smart contract data structure that inherits the contract fields and functions from HLF's contract data type, as well as a main function that runs the contract. This is all boilerplate code, and will be more or less the same regardless of what the contract does. The logic of the contract occurs in contract.go.

Listing 1 Definition of IdLocation Data Structure

```
type IdLocation struct {
    Timestamp       string
    VehicleId       string
    Latitude        string
    Longitude       string
    ...
}
```

In contract.go, a data structure is defined for the type of data that is being managed. An example of this can be seen in listing 1, where a struct called IdLocation is defined. This object describes where a specific vehicle was at a specific time. There are additional fields in the struct that have been omitted in the example. Also in the file are a set of functions for managing the struct. These functions belong to the custom smart contract object defined in main.go, and always have the first parameter be the transaction context. The transaction context is how smart contract developers can access blockchain data. All other parameters to the functions can be custom set by developers depending on what the function does. An example of a smart contract function can be seen in listing 2, where the function GetLatestIdLocation is defined. In this function, the most recent position of a vehicle is retrieved and returned. Smart contracts can be as simple or as complex as needed. The logic is simple in the telemetry use case, as the main functionality is simply periodically recording where a vehicle is.

Listing 2 Definition of GetLatestIdLocation

```
func (s *SmartContract) GetLatestIdLocation(ctx contractapi.TransactionContextInterface,
       vehicleId string) (*IdLocation, error) {
    idLocationAsBytes, err := ctx.GetStub().GetState(vehicleId)
    // error handling
    idLocation := new(IdLocation)
    err = json.Unmarshal(idLocationAsBytes, idLocation)
    // error handling
    return idLocation, nil
}
```

3. Access Control

Access control was achieved at both the simulation and blockchain level. In terms of blockchain access control, the only requirements are that the entity invoking smart contracts belong to a member organization and that the endorsement policy is met. The endorsement policy chosen simply requires a valid signature from a member peer. Allowing other private organizations to prevent information being written to the blockchain is undesirable as it could lead to competitor companies sabatoging each other, As such, a simple and permissive endorsement policy was chosen.

In addition to blockchain level access control, there are controls on the services. Each HTTP message includes a JSON web token (JWT) which is used for authentication and authorization. The token includes the requester's organization, their certificate, as well as some other metadata such as when the token was issued and how long it is valid for. When a service receives a request, it first extracts the token from the request, and then the certificate from the token. Then the service validates that the token was signed by the private key that corresponds to the certificate. This proves that the entity making the request is in fact the the same as is identified by the included certificate. Then the service fetches the root certificate from its MSP that corresponds to the organization identified in the token, and verifies that the included certificate has been signed by the root CA of that organization. This proves that they belong to the organization that they claim to. If either of these verifications fail, the endpoint will return an error saying the user is unauthorized.

VI. Conclusions and Future Work

This work was able to successfully demonstrate that a blockchain can be used as a secure mechanism to exchange and store data in a UAM environment. After running the simulation and submitting some flight plans to the UAM operator, it can be shown that each interaction during the negotiation of airspace, as well as all constraints and approvals that are generated in this process, is recorded on the blockchain. Due to the immutable nature of blockchain, there is a clear picture of all proposed flights, whether they were approved, where all constraints originated from, as well as when the communication occurred. Furthermore, there is a clear picture of all telemetry points indicating where the vehicle has traveled. This information could prove useful in audits, either routine or in case of an accident. In addition, any information saved is immediately available to all member organizations, as they each operate a blockchain peer.

The work in this space is ongoing. While this paper demonstrated the general feasibility of using blockchain in UAM using a simulated UAM environment, future work is focusing on integrating actual flight tests into the simulation. In that work, there will both be simulated and actual vehicles generating telemetry information and flight plans, which will all be recorded onto the blockchain. In addition, the UAM and flight simulation services are being expanded to support a wider range of configurations and tests.

References

[1] Fontaine, P., "Concept of Operations v2.0 Urban Air Mobility (UAM)," , 2023. URL `https://www.faa.gov/sites/faa.gov/files/Urban%20Air%20Mobility%20%28UAM%29%20Concept%20of%20Operations%202.0_0.pdf`.

[2] Freeman, K., Gillem, N., Jones, A. R., Sridhar, B., and Sharma, N., *Immutable Secure Data Exchange and Storage for Urban Air Mobility Environments*, ???? https://doi.org/10.2514/6.2022-1092, URL https://arc.aiaa.org/doi/abs/10.2514/6.2022-1092.

[3] Kolb, J., AbdelBaky, M., Katz, R. H., and Culler, D. E., "Core Concepts, Challenges, and Future Directions in Blockchain: A Centralized Tutorial," *ACM Comput. Surv.*, Vol. 53, No. 1, 2020. https://doi.org/10.1145/3366370, URL https://doi.org/10.1145/3366370.

[4] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A. D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., and Yellick, J., "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *CoRR*, Vol. abs/1801.10228, 2018. URL http://arxiv.org/abs/1801.10228.

[5] Hyperledger Foundation, "An Overview of Hyperledger Foundation," Tech. Rep. msu-cse-06-2, Hyperledger Foundation, October 2021. URL https://8112310.fs1.hubspotusercontent-na1.net/hubfs/8112310/Hyperledger-Foundation-White-Paper.pdf.

[6] Wood, G., et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, Vol. 151, No. 2014, 2014, pp. 1–32.

[7] Xiao, Y., Zhang, N., Lou, W., and Hou, Y. T., "A Survey of Distributed Consensus Protocols for Blockchain Networks," *IEEE Communications Surveys & Tutorials*, Vol. 22, No. 2, 2020, pp. 1432–1465. https://doi.org/10.1109/COMST.2020.2969706.

[8] Fu, X., Wang, H., and Shi, P., "A survey of Blockchain consensus algorithms: mechanism, design and applications," *Science China Information Sciences*, Vol. 64, No. 121101, 2020. https://doi.org/https://doi-org.libaccess.sjlibrary.org/10.1007/s11432-019-2790-1.

[9] Yang, G., Lee, K., Lee, K., Yoo, Y., Lee, H., and Yoo, C., "Resource Analysis of Blockchain Consensus Algorithms in Hyperledger Fabric," *IEEE Access*, Vol. 10, 2022, pp. 74902–74920. https://doi.org/10.1109/ACCESS.2022.3190979.