Trinity: A Byzantine Fault-Tolerant Distributed Publish-Subscribe System with Immutable Blockchain-based Persistence

Gowri Sankar Ramachandran, Kwame-Lante Wright, Licheng Zheng, Pavas Navaney, Muhammad Naveed, and Bhaskar Krishnamachari
USC Viterbi School of Engineering
University of Southern California
Los Angeles, USA
{gsramach, kwamelaw, lichengz, navaney, mnaveed, bkrishna}@usc.edu

Jagjit Dhaliwal
Deputy CIO, Office of CIO
County of Los Angeles
Los Angeles, USA
jdhaliwal@cio.lacounty.gov

***Abstract*—Internet of Things (IoT), Supply Chain monitoring, and other distributed applications rely on messaging protocols for data exchange. Contemporary IoT and enterprise deployments widely use the publish-subscribe messaging model because of its resource-efficiency. However, the systems with publish-subscribe messaging model employ a centralized architecture, wherein the data from all the publishers in the application network flows via a central broker to the subscribers. Such a centralized architecture makes the publish-subscribe messaging model susceptible to Byzantine failures. For example, it provides an opportunity for the organization that owns the broker to tamper with the data. In this work, we contribute Trinity, a novel distributed publish-subscribe broker with Byzantine fault-tolerance and blockchain-based immutability. Trinity distributes the data published to one of the brokers in the network to all the brokers in the network, and stores the data in an immutable ledger through the use of blockchain technology. Through the use of consensus protocols and distributed ledger technology, Trinity can guarantee ordering, fault-tolerance, persistence and immutability across trust boundaries.**

Our evaluation results show that Trinity consumes minimal resources. To the best of our knowledge, Trinity is the first framework that combines the components of the blockchain technology with the publish-subscribe messaging model. Furthermore, we plan to use Trinity in a real-world use case for increasing the transparency of racial profiling.

***Index Terms*—IoT, Broker, Blockchain, Supply Chain Monitoring, Multi-stakeholder, Ledger, Smart Contract.**

I. Introduction

Enterprises and industrial applications, e.g., Microsoft Azure Service Bus [1] and IBM WebSphere [2] use the publish-subscribe communication pattern to exchange data between various data producers and consumers. This messaging pattern isolates the publishers and subscribers in time, space, and synchrony [3]. Publish-subscribe communication pattern is an alternative to synchronous and point-to-point request-reply communication models. The key advantages of the publish-subscribe protocol are its ability to support loosely-coupled message exchanges between producers and consumers [3].

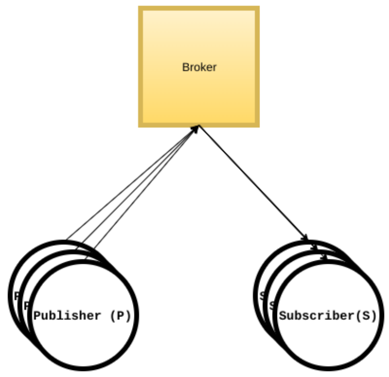


Fig. 1: Publish-Subscribe Communication Model.

Figure 1 shows the interaction model of the publish-subscribe messaging model. In the publish-subscribe system, publishers and subscribers interact via a central broker. A broker is a centralized software that orchestrates the communication between the publishers and subscribers [3]. Enterprise systems and IoT applications employ this messaging model because of its resource-efficiency and scalability.

Although the publish-subscribe messaging model is lightweight, scalable, and resource-efficient, it relies on a central broker for data communication between publishers and subscribers as shown in Figure 1. Such a centralized architecture makes the publish-subscribe messaging model vulnerable to central points of failure. Besides, publishers and subscribers interact through a central server owned by a single organization. Failure of a broker may impact the publishers and subscribers. Furthermore, the traditional publish-subscribe brokers do not provide strict guarantees regarding the ordered delivery of messages to the subscribers, although some implementation supports provide ordered delivery in a single broker setting [4].

In this work, we contribute Trinity, a novel distributed publish-subscribe broker with blockchain-based immutability by integrating the broker system with a Byzantine Fault-Tolerant (BFT) consensus protocol and distributed ledger technology. Contemporary blockchain platforms consist of a consensus layer for state replication and ordering and a distributed tamper-proof ledger for persistent storage. Trinity, therefore, uses a blockchain platform to provide fault-tolerance, message ordering, and immutable storage. Trinity is implemented using MQTT broker [5] and a modular choice of "*pluggable*" blockchain platforms including Ethereum [6], IOTA [7], HyperLedger Fabric [8], and the Tendermint [9] blockchain framework. Our formal specification and analysis assume the Blockchain provides BFT consensus; however, the platforms such as IOTA and Ethereum or the consensus models used under Hyperledger Fabric don't provide a provable guarantee in this regard today. So this represents a gap between our theory and the system implementation for all the evaluation candidates except Tendermint. The evaluation results show that the Trinity framework introduces reasonable timing and performance overhead in exchange for providing assurances when transacting in a multi-stakeholder environment.

Section II introduces the publish-subscribe broker and discusses the related work. Section III presents the formal definitions and the architecture of Trinity. System description of Trinity is presented in Section V. The implementation and evaluation of Trinity is presented in Section VI. Section VIII concludes the paper with the future work.

II. State of the Art and Related Work

A. *publish-subscribe Messaging Model*

Over the past several years, the publish-subscribe messaging has proven itself as a dominant messaging paradigm for IoT and enterprise systems. By decoupling publishers (data sources) from subscribers (data sinks), clients (i.e., publishers and subscribers) can operate more freely with less prior knowledge of the network they run in while having isolation in time, space, and synchrony [3].

Distributed applications rely on a messaging model for coordinating and collaborating with other systems in the network. A wide-array of messaging models were proposed for distributed interactions including RPC [10], CoAP [11], and MQTT [12]. Remote Procedure Calls (RPC) [10] allows a program or a function in one machine to interact with a program or a function in other remote machines in the network without knowing the details of the network protocols. Although RPC offered a powerful approach for distributed communication, it is not suitable for resource-constrained IoT applications because of its high resource and communication overhead. Protocols such as CoAP and MQTT were created for resource-constrained IoT devices.

On the one hand, CoAP follows the request-reply communication model, wherein the interested party has to request the data source for the new data periodically [11]. MQTT, on the other hand, follows the publish-subscribe messaging model, in which a broker is used to orchestrate the communication between the data producers and consumers. Even though both MQTT and CoAP are suitable for IoT and enterprise applications, CoAP is not widely used because of its request and reply communication model, whereas MQTT requires one request to be submitted to a broker, after which the data from the producers are forwarded to the subscribers by the broker.

The publish-subscribe communication model [4], [12] typically consists of three components; broker, publisher, and subscriber. Publishers in the system send data to a broker following the concept of the *topic*. Topic typically refers to the metadata, which describes information about the data in a string format. A topic can have multiple levels. For example, the data generated by a temperature sensor deployed at room 123 of building A can have its topic defined as $\backslash buildingA \backslash room123 \backslash Temperature$. Consumers of the data can receive data from the temperature sensor by subscribing to the $\backslash buildingA \backslash room123 \backslash Temperature$ topic. In the next section, we discuss the related work addressing distributed publish-subscribe broker.

B. *Related Work*

The idea of Byzantine fault-tolerant publish-subscribe broker is already discussed in the literature. Chang *et al.* [13] motivate the need for a distributed publish-subscribe broker with Byzantine fault-tolerance and examines the Byzantine behaviors of publishers, subscribers, and brokers. Bhola *et al.* [14] present a loss tolerant publish-subscribe with support for exactly-once delivery and message ordering using a concept knowledge graph, which relies on a network formation protocol to establish links between brokers. Meling *et al.* [15] propose Byzantine Proposer Fast Paxos, a novel consensus algorithm based on the Paxos for Byzantine faulty clients. P2S [16] is a fault-tolerant distributed publish-subscribe broker based on Paxos consensus algorithm. P2S is developed to tolerate faults using the Paxos algorithm, and it is complementary to Trinity, except Trinity is a Byzantine fault-tolerant publish-subscribe broker capable of handling both Byzantine and crash faults while providing immutable persistence storage.

Broker distribution using overlay networks is extensively addressed in the literature. Kazemzadeh *et al.* [17] present a partition tolerant publish-subscribe broker based on network routing algorithm, which reacts to broker and link failures by discovering new brokers and updating the routing table on all the broker nodes. A few works [18]–[20] contribute approaches to managing the publish-subscribe broker on overlay networks. Unlike these routing approaches, Trinity depends on the underlying blockchain platform, which includes a consensus algorithm and the distributed ledger, to distribute brokers.

Kafka [21] is a more powerful publish-subscribe broker developed for use in data centers, and it has a rich set of features. Kafka uses a proprietary protocol for communication. It is designed to run in a distributed fashion with built-in support for partitioning and replication. Partitioning is a method for load-balancing across different instances, while replication involves copying the same data across multiple instances. Although Kafka provides ordering guarantees and configurable persistence of messages, it assumes that the software is managed by a single organization, meaning that no trust boundaries are traversed during the operation of the system. There is nothing in place to prevent data tampering as every instance of a Kafka deployment is expected to be owned and operated by a single entity. In our design of Trinity, we do not make this assumption regarding trust. Through the use of a blockchain network, Trinity guarantees persistence, ordering, and immutability across trust boundaries.

Implementations of publish-subscribe protocol can be found in brokers such as Mosquitto [4], which is primarily designed for use as a single instance, but it has support for bridging, providing support for multiple connected brokers to operate together through the bridging feature, which effectively duplicates all transmitted messages at every broker, allowing publishers and subscribers to connect to any instance. However, this method of distribution does not guarantee the same ordering of messages at every broker. There is also an implicit assumption of the system being run by a single entity, so there are no trust concerns, and the immutability of data is not provided. We address the issues of ordering and immutability with our blockchain-based broker system.

III. Formal Description of Trinity: architecture, preliminaries, definition, assumptions

A. System Model and Architecture

We consider a system with domains. Each domain has a consensus node. Consensus nodes maintain a distributed ledger that results from an atomic broadcast [22] (ordered consensus) process. Each domain also has a publish-subscribe broker and a collection of clients that can host publishers and subscribers. Publish-subscribe clients can interact with a local domain broker.

For ease of exposition, we consider just one topic and assume that all publishers and subscribers are communicating

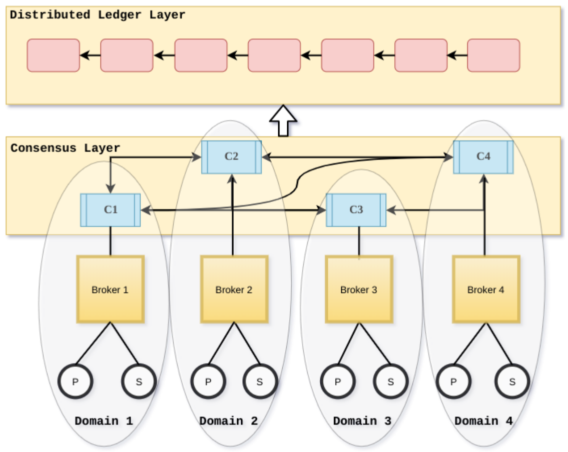


Fig. 2: Architecture of Trinity: A Byzantine Fault-Tolerant Publish-Subscribe Broker.

on that topic. We assume that all domains under consideration are capable of hosting clients that can publish to or subscribe from the given topic. The claims made hold true for multiple topics, by considering only the relevant publishers/subscribers and domains for each topic, one at a time.

B. Definitions

We define the key concepts of Trinity in this section.

Definition 1. *A domain consists of one consensus node, one broker node that communicates with the consensus node, and a collection of publish and subscribe clients that communicate with the broker within that domain.*

Definition 2. *A domain is said to be faulty if either the Consensus node or the Broker node of that domain shows Byzantine faults; else it is said to be correct. In particular correct consensus and broker nodes must be always online.*

Definition 3. *A publish-subscribe client is said to be correct if it is not faulty and if it is in a non-faulty domain.*

C. Assumptions

The key assumptions and the specifications of Trinity are described in this section.

Assumption 1. *We assume that strictly fewer than one-third of all the domains under consideration are faulty.*

Assumption 2. *Under the above assumption, the consensus layer implements a Byzantine fault-tolerant atomic broadcast (here received/delivered means in a confirmed manner):*

- ***2.1** All messages sent by correct consensus nodes are delivered identically to all correct consensus nodes within a bounded delay period.*
- ***2.2** If one correct consensus node receives a message, all correct consensus nodes will receive that message within a bounded delay period.*
- ***2.3** If one correct consensus node receives message m before m', so too will all correct consensus nodes.*

Assumption 3. *Other nodes in the system can not forge all messages that are digitally signed by any publisher unless the publisher loses or gives away its private key.*

Assumption 4. *We assume that brokers in correct domains deliver all messages from its publishers to its consensus node in order and deliver all messages from its consensus node to all its subscribers in order.*

Assumption 5. *We assume that brokers in correct domains deliver any message received from a publisher to the consensus node or from the consensus node to a subscriber within a bounded delay.*

Assumption 6. *Consensus nodes forward any message from the broker to the consensus layer and vice versa also within a bounded delay.*

Assumption 7. *We don't assume any buffering, so subscribers that go offline are not guaranteed delivery of messages. Hence,* *for simplicity, we assume that all correct subscribers are always online.*

Assumption 8. *Correct publishers are online whenever they send messages.*

In the next section, we prove the safety and liveness properties of Trinity.

IV. Safety and liveness properties: formal statement and proof sketches

The safety property proves that ”something bad will never happen” [23], whereas the liveness property informally guarantees that the system will make progress and ”something good will always happen” [23]. In this section, we prove the safety and liveness properties of Trinity.

A. Safety Properties

Safety Property 1. ***Publisher Safety:*** *All correct publishers can guarantee that all their messages are delivered to all correct subscribers.*

Proof of Publisher Safety: A correct publisher by definition is in a correct domain (Definition 3). Hence its messages are delivered to the consensus layer correctly without omission. The consensus layer can guarantee delivery to all correct-domain consensus nodes (by Assumption 1, Assumption 2.1). The correct subscriber will hear this delivered message.

Safety Property 2. ***Subscriber Consensus:*** *All correct subscribers receive same messages.*

Proof of Subscriber Consensus: All correct-domain consensus nodes receive the same message (Assumption 2.2), and in these correct-domains, all correct subscribers will receive that message (Definition 3). Hence all correct subscribers receive the same message.

Safety Property 3. ***In-order Delivery:*** *If one correct subscriber sees message m before m', all correct subscribers will see it in the same order.*

Proof of In-order Delivery: This follows from Assumption 2.3 and the fact that all brokers in correct nodes order published and subscribed messages correctly.

Safety Property 4. ***Non-Forgery:*** *If publishers use digital signatures and A3 holds, correct subscribers will not receive any forged messages from any publisher.*

Proof of Non-Forgery: Follows from Assumption 3.

B. Liveness

Liveness Property 1. *Messages from any correct publishers are delivered to correct subscribers within a bounded delay.*

Proof: Correct publishers are online to send their messages (Assumption 8). The broker forwards these messages to its consensus node within a finite delay (Assumption 5). The consensus layer will receive it within a finite delay (Assumption 6), and confirmed delivery at all correct consensus nodes will occur within a further finite delay (Assumption 2.1). Finally, these correct consensus nodes will forward to their correct broker within a finite delay (Assumption 6), and those, in turn, will deliver to the correct subscriber that are assumed to be online (Assumption 7) within a finite delay (Assumption 5).

The safety and liveness properties of Trinity and their proofs show that Trinity is capable of handling Byzantine faults. We describe the system details of Trinity in Section V.

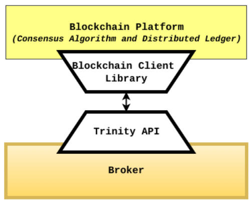

Blockchain Platform
(Consensus Algorithm and Distributed Ledger)
Blockchain Client Library
Trinity API
Broker

Fig. 3: Interface overview of Trinity.

V. System Description of Trinity

Figure 2 shows the three main components of a domain: the blockchain network, broker, and publish-subscribe clients. A single organization or a stakeholder own each domain. The Trinity deployment enables each stakeholder to publish information not only to their brokers but also to other brokers via the underlying blockchain platform. The rest of the section discusses the building blocks of Trinity.

A. Publish-Subscribe Broker

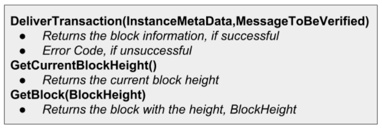
The broker coordinates the communication between clients (publishers and subscribers) and the blockchain platform. To realize Trinity, a publish-subscribe broker with support for topic-based communication model is desired. As shown in Figure 3, each broker instance connects to a consensus node maintained by the domain via the blockchain client library provided by the underlying blockchain platform.

B. Trinity Blockchain Integration

The blockchain network is responsible for the consensus and persistent storage. Trinity exposes a set of APIs to interact with a blockchain platform using the APIs listed in Figure 4 following the interface architecture shown in Figure 3.

The *DeliverTransaction* API initiates the consensus and the block creation process by delivering the message to the underlying platform. Trinity instance sends the published messages using the *DeliverTransaction* API along with the necessary metadata such as node identifiers to the underlying blockchain framework. The blockchain network replicates the data among the Trinity consensus nodes following the BFT consensus protocol. We will discuss the role of the blockchain frameworks below.

The blockchain frameworks typically consist of a consensus protocol, block creation logic, distributed ledger, and public-key cryptography. Trinity does not depend on a particular blockchain framework since the broker interfaces with a



DeliverTransaction(InstanceMetaData,MessageToBeVerified)
- *Returns the block information, if successful*
- *Error Code, if unsuccessful*

GetCurrentBlockHeight()
- *Returns the current block height*

GetBlock(BlockHeight)
- *Returns the block with the height, BlockHeight*

Fig. 4: Trinity APIs to Interact with a Blockchain Network.

Blockchain Platform	Public or Permissioned	Consensus Algorithm	Transaction Fee
Tendermint	Permissioned	BFT	No
HyperLedger Fabric	Permissioned	Ordering (Solo, Kafka)	No
IOTA	Public	PoW	No
Ethereum	Public	PoW	Yes

TABLE I: Blockchain Platforms Used in the Evaluation

blockchain framework via a set of APIs for managing the blockchain-related functionalities. Each Trinity instance has APIs for querying the blockchain network. The *GetCurrent-BlockHeight()* API allows the Trinity instance to get the current block height from the blockchain network. Similarly, the *GetBlock(BlockHeight)* API returns the entire block at the height denoted by the *BlockHeight* argument. The implementation of these functionalities should be carried out using the APIs and client libraries provided by the underlying platform. Note that the interface model for Hyperledger Fabric has to follow its ledger model, which is different from contemporary blockchain platforms such as Ethereum and Tendermint. Although the Trinity implementation is platform agnostic, the underlying blockchain framework must have the following components to guarantee immutability and fault-tolerance:

Consensus Algorithm: The Trinity framework operates in a distributed network owned by multiple domains or organizations. All the authorized consensus nodes must verify the messages received by Trinity. This verification process relies on a consensus protocol. A system state perceived by one broker in the network must be replicated to other consensus nodes owned by other domains, and the transaction (message) should be approved by the majority (two-thirds) of the nodes in the network. Our system can work with Byzantine fault-tolerance (BFT) protocols such as Tendermint [9], but it is not strictly limited to BFT (though then the formal properties shown earlier do not hold strictly). Note that the resource-consumption and the ability to tolerate Byzantine failures depend on the consensus protocol.

Public Key Cryptography: Blockchain framework use the public key cryptography to secure the transactions. Each consensus node in the network create a pair of keys and share their public key to the network to participate in the consensus and block creation process.

Validators: Validators are a special type of nodes in the network that are authorized to participate in the consensus process and create blocks. In case of the permissioned blockchain, only a designated set of nodes can act as the validators, while any capable node can perform validation (and mining) in public blockchains such as BitCoin and Ethereum. We believe that Trinity framework is better-suited for permissioned blockchains as the parties involved in the transactions are known to the system, and each subscriber will have to register their interest before receiving the data.

In summary, the blockchain-specific tasks of Trinity are loosely coupled with the broker-specific activities, wherein the interaction between the broker and the blockchain network happens via a set of APIs. We believe that this architecture would inspire application developers to replace the publish-subscribe communication model with other messaging protocols to create novel blockchain-based frameworks.

VI. Implementation and Evaluation

Section VI-A discusses the implementation details and the evaluation setup of Trinity. The timing overhead of Trinity is presented in Section VI-D. Section VI-E discusses the network overhead of Trinity. CPU and Memory overhead are presented in Section VI-F.

A. Implementation of Trinity

We implemented Trinity using Mosquitto (MQTT) Broker [5]. The broker-specific functionalities are implemented on top of MQTT using the APIs presented in Figure 4 For the blockchain part, we used Tendermint [9], HyperLedger Fabric [24], IOTA [7], and Ethereum [6]. Table I and Section VI-B provides an overview of the platforms used in the evaluation. Except Tendermint BFT, all the other blockchain platforms uses a different form of consensus mechanism. However, we implemented and evaluated Trinity on other blockchain platforms to show the blockchain-agnostic nature of Trinity and make the code available as open source software (https://github.com/ANRGUSC/Trinity).

B. Overview and the Setup of the Evaluation Platforms

Tendermint [9] blockchain platform consists of a set of tools for achieving consensus on a distributed network and the creation of blocks. The platform isolates the blockchain-related functionalities from the application-specific features, which means any application can be developed atop Tendermint framework, ranging from cryptocurrencies to a distributed chat server. The consensus engine of Tendermint allows the application developers to replicate the state of an application across all the Tendermint consensus nodes in the network using a variant of BFT consensus [9]. The state information is fed into the consensus engine using Application Blockchain Interface (ABCI). For the implementation of Trinity, we integrated MQTT [5] with Tendermint blockchain platform using the ABCI. Tendermint platform uses the Byzantine Fault-tolerant (BFT) consensus protocol, which means the two-thirds of the consensus nodes in the network must approve the transactions. When the majority of the consensus nodes (two-thirds) in the network approve the transaction, Tendermint platform adds the transaction into a block.

HyperLedger Fabric: HyperLedger Fabric is a permissioned blockchain maintained by the HyperLedger Foundation [8]. In permissioned blockchain platforms, the network is formed by authorized members with a known identity, who is responsible for the validation of transactions and ledger maintenance. HyperLedger Fabric uses an ordering service to order and validate the transactions before writing the verified information to the distributed ledger. Although HyperLedger Fabric does not provide Byzantine fault-tolerance, we implemented Trinity using it to show the blockchain-agnostic architecture of Trinity.

Ethereum: Ethereum [6] is one of the widely used public blockchain platforms after BitCoin. The PoW consensus algorithm and the ledger technology provide persistence, ordering, and fault-tolerance guarantees. We developed a Web3 NodeJS client application along with a smart contract in Solidity to evaluate Trinity on Ethereum. Unlike Tendermint and HyperLedger, the implementation of Trinity on public blockchain platforms such as Ethereum and IOTA does not require dedicated consensus nodes.

IOTA: IOTA [7] is a public distributed ledger technology that uses a directed acyclic graph (DAG) storage structure, Tangle, for storing transactions. IOTA uses a special type of PoW consensus, wherein each node that posts a transaction to the IOTA ledger has to verify two other transactions. Such a verification model enables IOTA to provide free transactions unlike other PoW-based platforms such as BitCoin and Ethereum.

C. Evaluation Setup

Tendermint: Trinity was evaluated using 20-node Raspberry Pi3 (RPi) test network. Raspberry Pi (Version 3) has ARM Cortex-A53 Quad Core CPU with 1 GB of RAM. All the RPi devices were connected through a LAN. Each data point in the evaluation results is collected by publishing 1000 messages to an MQTT broker in the network. Upon receiving the message, the broker may choose to relay the message to the subscribers as in contemporary publish-subscribe systems (referred as loopback) or deliver the message to the blockchain platform (validated), which runs the BFT consensus for ordering and a distributed ledger for persistence. The former is denoted as *loopback* in Figure 5a and Figure 5b, while the later is referred to as *committed* (validated). Our evaluation was carried out on 5, 10, 15, and 20 nodes to compare the network performance and end-to-end delay with the scale. We used the default configuration of the Tendermint platform for the evaluation [9].

HyperLedger Fabric Setup: The HyperLedger Fabric version of Trinity was evaluated using 20 Docker containers comprised of full nodes including an orderer.

Ethereum: Ethereum version of Trinity was evaluated using Ropsten test network due to the transaction fees in the main network. The smart contract was realized using the Solidity language, and it was deployed using Remix IDE.

IOTA: The IOTA version of Trinity was evaluated on the test net. We considered IOTA for the evaluation as it was targeted for resource-constrained IoT devices [7].

D. End-to-End Delay

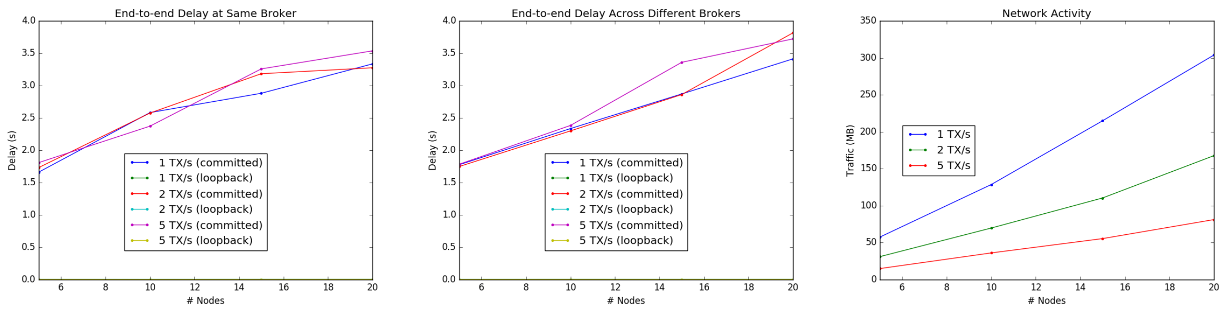
Trinity verifies and orders the published messages using a blockchain platform and records the transactions in a distributed immutable ledger, which allows the subscribers of Trinity to receive the recorded and ordered messages while providing Byzantine fault-tolerant guarantee. The consensus and ledger storage processes add a delay since the Trinity consensus nodes must execute the consensus algorithm and the block creation protocol, which means the subscriber receives the published data after a slight delay.

1) Tendermint: Figure 5a and Figure 5b shows the timing overhead of Trinity when publishing to and subscribing from a broker within a domain and publishing to one broker and subscribing for a same verified and ordered topic from a different broker (from a different domain) respectively. We measured the end-to-end delay with a publisher sending data every 0.2s, 0.5s, and 1s, which translates to 5 transactions per seconds (TPS), 2 TPS, and 1 TPS respectively. The end-to-end delay increases with the number of consensus nodes in the network due to the increase in the number of validators participating in the consensus process.

Figure 5a shows the timing overhead when publishing to and subscribing from the same broker within a domain. The maximum delay is roughly 3.5s for 20 nodes with 5 TPS, and the delay for *loopback* transaction is negligible (approximately 90 milliseconds). When subscribing to the verified data from a different broker from another domain, the maximum delay increases from 3.5s to 3.7s. From Figure 5a and Figure 5b, it is clear that the blockchain-based immutability and ordering through BFT consensus increases the end-to-end delay, but we believe that this cost outweighs the Fault-tolerance and persistence guarantees of Trinity.

2) HyperLedger Fabric: The HyperLedger Fabric [24] version of Trinity uses the Solo consensus, which is a centralized orderer for ordering the transaction before writing it into a distributed ledger. We evaluated the end-to-end delay for various buffer sizes since the transactions get dropped when the buffer is full. Our evaluation for HyperLedger Fabric presents only the results for the broker in a single domain since the difference is negligible for brokers in different domains as indicated by Section VI-D1. As shown in Figure 6a, the maximum end-to-end delay for HyperLedger Fabric is 142 milliseconds when the buffer size is 10 and the number of transactions per second is 30. The lower end-to-end delay of HyperLedger Fabric makes it ideally suitable for a permissioned setup, which is typically encountered in applications such as the racial profiling application presented in Section VII.

3) Ethereum: The Ethereum version of Trinity was evaluated using Ropsten test network, which is functionally similar to the Ethereum main net. The average end-to-end delay for Ethereum is *14 Seconds* on Ropsten testnet, whereas the average block creation time on Ethereum mainnet is *15.8*



(a) Delay with Same Broker. (b) Delay with Different Broker. (c) Network Overhead

Fig. 5: Tendermint-version of Trinity: End-to-end delay when publishing and subscribing from same broker in Figure 5a and Figure 5b presents the delay when publishing from one broker and subscribing from a different broker. Network usage results are presented in Figure 5c.

Seconds according to https://ethstats.net/. Besides, the user has to pay a transaction fee in Ethereum, which at the time of writing, fell between *0.0021 USD (1 gwei)* and *51.71 USD (12500 gwei)* according to https://ethgasstation.info/.

4) IOTA: The IOTA version of Trinity was evaluated using the test net. The transactions gets added in the mempool in *10 Seconds* on average, but it takes approximately *36 seconds* to be confirmed. On the mainnet, the average confirmation time is *5.2 minutes*https://tanglemonitor.com/. Note that there is no transaction fees in IOTA although it uses a lightweight PoW consensus since each full node in IOTA is expected to verify two other transactions whenever it submits a transaction [7]. Note that the network uses a centralized coordinator at the time of writing.

E. Network Overhead

The Trinity framework uses a blockchain framework to perform consensus and block creation. All these functionalities are achieved through the coordination and collaboration of all the consensus nodes in the network. This process generates network traffic. Figure 5c shows the network overhead of Trinity framework for the Tendermint version. The networking overhead of the Trinity framework increase as the number of consensus nodes in the network increases for the Tendermint BFT.

Interestingly, the lower the amount of transaction per second, the higher the networking overhead, which is due to the creation of a large number of blocks. The higher TPS typically result in multiple transactions being recorded in a single block, whereas the lower TPS would lead to a single block per transaction. We believe that the average network overhead of approximately 300 KiloBytes/ Second for 20-device with one TPS is insignificant compared to the benefits offered by Trinity.

HyperLedger Fabric's network overhead depends on the buffer size and the number of transactions per second as shown in Figure 6b. The networking overhead cannot be measured for Ethereum and IOTA since the full nodes run by the community members in a public network.

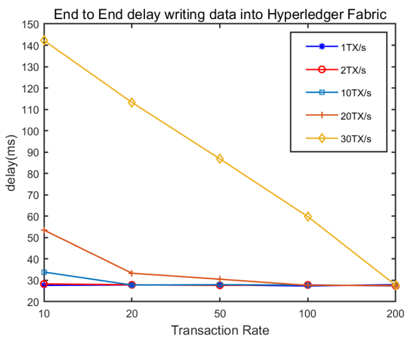
F. CPU and RAM Usage

Figure 7b shows the CPU usage of Trinity for the Tendermint version. On RPi platform, Trinity uses approximately 85% of the CPU when publishing 1000 transactions at a rate of 1 TPS in 20-node network, since the framework executes a consensus algorithm, and block creation protocol every second. Similarly, the maximum RAM usage (see Figure 7c) is measured in a 20-node network when publishing 1000 transactions at a rate of 1 TPS over 15 minutes time interval. The resource overhead of Trinity increases with the number of blocks. By adding multiple transactions in a single block, the CPU and memory overhead can be minimized.

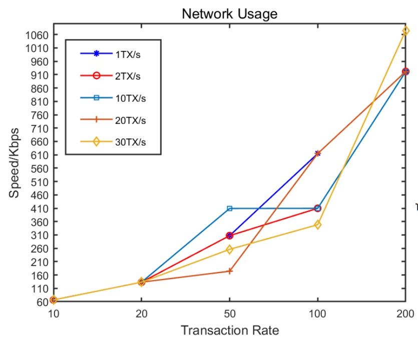
The CPU overhead of HyperLedger Fabric on a Docker container is approximately 51% as depicted in Figure 7a. HyperLedger's memory overhead is approximately 932 MB, and it increases with the buffer size as shown in Figure 6c. The CPU and memory overhead cannot be measured for Ethereum and IOTA since we don't maintain a full node to run the PoW consensus algorithm.

VII. Real-world Application of Trinity

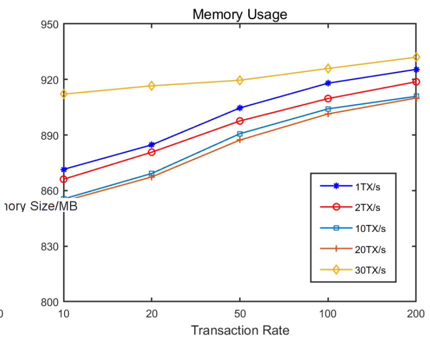
The Racial and Identity Profiling Act of 2015 (AB 953) [25] aims to curb the harmful and unjust practice of racial and identity profiling and increase transparency and accountability with law enforcement agencies. Racial and identity profiling occurs when law enforcement officers stop, search, property seize, or interrogate a person without evidence of criminal activity. One of the requirements of AB 953 focuses on creating a system for collecting and reporting basic information on police-community interactions. The Department of Justice (DOJ) has developed a web application for collecting and reporting stop data. As an alternate option, some agencies have also started developing local applications which provide easier integration with their local systems including their RMS. These agencies are planning to collect data locally and then submit it to DOJ as per defined timelines. There was a concern raised by RIPA Board in a June 2018 board meeting regarding the integrity of the data collection. The Board asked DOJ to assure that the data is not manipulated by any intermediaries



(a) Delay for HyperLedger.

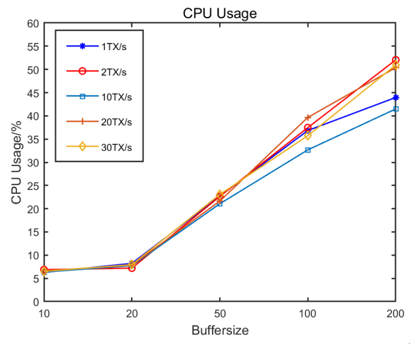


(b) Network Overhead.

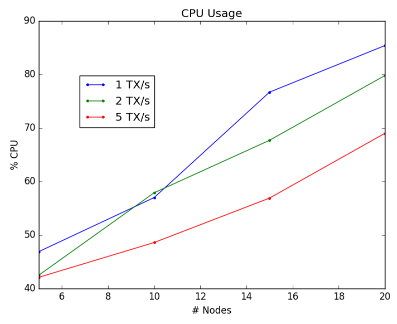


(c) Memory Overhead

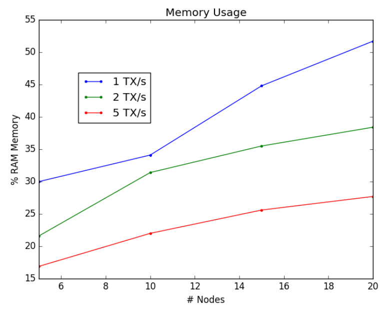
Fig. 6: HyperLedger Fabric-version of Trinity: End-to-end delay is presented in Figure 6a. Network and memory usage results are presented in Figure 6b and Figure 6c respectively. HyperLedger Fabric consists of a buffer which influences the performance. X-axis shows the different sizes used in the evaluation.



(a) CPU Overhead



(b) CPU Usage for Tendermint version of Trinity.



(c) Memory Overhead

Fig. 7: CPU usage of HyperLedger Fabric-version of Trinity is presented in Figure 7a. CPU and memory usage of Tendermint-version of Trinity is presented in Figure 7b and Figure 7c respectively.

when agencies are using locally developed applications to collect data.

The HyperLedger Fabric-version of Trinity is considered for this application due to the involvement of multiple stakeholders. We plan to deploy a permissioned Trinity set up with five peers including an orderer at field offices and the office of DOJ. Whenever a field officer enters the profiled information, it will get replicated to all the Trinity instances via the blockchain network. When DOJ wants to track the profiled information, they can subscribe to the verified data stream of Trinity. Although local agencies all run Trinity instances, they cannot manipulate once the data is entered onto the blockchain, as the data is signed, ordered and stored in the immutable ledger by the consensus node deployed on the DoJs office.

VIII. Conclusion

Blockchain technology has made a significant impact in the world of cryptocurrencies. However, the building blocks of blockchain technology such as consensus protocols and distributed ledgers are promising for applications beyond cryptocurrencies. In this work, we have presented Trinity, which is a distributed fault-tolerant publish-subscribe broker with blockchain-based immutability. Trinity's fault-tolerance capabilities come from the consensus nodes that run a Byzantine Fault-tolerance consensus algorithm. The implementation of Trinity showed how the blockchain-specific tasks are decoupled from the broker-specific functionalities. Our implementation and evaluation showed that Trinity can be practically implemented using a contemporary publish-subscribe broker and blockchain platforms as demonstrated by its implementation and assessment using MQTT and multiple blockchain platforms including Tendermint, HyperLedger, Ethereum, and IOTA. We believe that the fault-tolerance and trust benefits of Trinity outweigh the computational and communication overhead introduced by Trinity and the underlying blockchain platform. Nevertheless, our future work will focus on optimizing computation and networking resources.

Acknowledgment

This work is supported by the USC Viterbi Center for Cyber-Physical Systems and the Internet of Things (CCI).

References

[1] "Service Bus queues, topics, and subscriptions," Microsoft, September 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions

[2] F. Budinsky, G. DeCandio, R. Earle, T. Francis, J. Jones, J. Li, M. Nally, C. Nelin, V. Popescu, S. Rich, A. Ryman, and T. Wilson, "Websphere studio overview," *IBM Syst. J.*, vol. 43, no. 2, pp. 384–419, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1147/sj.432.0384

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[4] "Eclipse Mosquitto," Eclipse Foundation, May 2018. [Online]. Available: https://mosquitto.org/

[5] A. Banks and R. Gupta, "MQTT version 3.1.1," *OASIS Standard*, vol. 29, 2014.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[7] S. Popov, "The tangle, iota whitepaper," 2018.

[8] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," *CoRR*, vol. abs/1805.11390, 2018. [Online]. Available: http://arxiv.org/abs/1805.11390

[9] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, 2016.

[10] B. J. Nelson, "Remote procedure call," 1981.

[11] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Tech. Rep., 2014.

[12] A. Banks and R. Gupta, "Mqtt version 3.1. 1," *OASIS standard*, vol. 29, 2014.

[13] T. Chang and H. Meling, "Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure," in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, Oct 2012, pp. 454–456.

[14] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *Proceedings International Conference on Dependable Systems and Networks*, June 2002, pp. 7–16.

[15] H. Meling, K. Marzullo, and A. Mei, "When you don't trust clients: Byzantine proposer fast paxos," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, June 2012, pp. 193–202.

[16] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang, "P2s: a fault-tolerant publish/subscribe infrastructure," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 189–197.

[17] R. S. Kazemzadeh and H. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, Oct 2011, pp. 101–110.

[18] C. Chen, H. Jacobsen, and R. Vitenberg, "Divide and conquer algorithms for publish/subscribe overlay design," in *2010 IEEE 30th International Conference on Distributed Computing Systems*, June 2010, pp. 622–633.

[19] M. Onus and A. W. Richa, "Minimum maximum-degree publish-subscribe overlay network design," *IEEE/ACM Trans. Netw.*, vol. 19, no. 5, pp. 1331–1343, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1109/TNET.2011.2144999

[20] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito, "Efficient publish/subscribe through a self-organizing broker overlay and its application to siena," *The Computer Journal*, vol. 50, no. 4, pp. 444–459, 2007. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxm002

[21] "Apache Kafka," The Apache Software Foundation, April 2018. [Online]. Available: https://kafka.apache.org/

[22] F. Cristian, H. Aghili, R. Strong, and D. Dolev, *Atomic broadcast: From simple message diffusion to Byzantine agreement*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[23] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed computing*, vol. 2, no. 3, pp. 117–126, 1987.

[24] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. 310, 2016.

[25] "AB 953: The Racial and Identity Profiling Act of 2015," State of California Department of Justice, 2015. [Online]. Available: https://oag.ca.gov/ab953