# SUPRA, a distributed publish/subscribe protocol with blockchain as a conflict resolver

Jean-Philippe Abegg, Quentin Bramas, Timothée Brugière, Thomas Noël

# SUPRA, a distributed publish/subscribe protocol with blockchain as a conflict resolver

Jean-Philippe ABEGG *Transchain, ICUBE, University of Strasbourg, France*

Quentin BRAMAS *ICUBE, University of Strasbourg, France*

Timothée BRUGIÈRE *Transchain, Strasbourg, France*

Thomas NOEL *ICUBE, University of Strasbourg, France*

February 10, 2021

**Abstract**

Publish/subscribe is a communication paradigm used in distributed applications to easily exchange messages. This paradigm usually has a centralized architecture where a *broker* is responsible for transferring all the messages, hence can be a source of trust issues. In a threat model where the broker can be malicious, two honest entities cannot be sure of the origin of a message. There exist propositions for distributed publish/subscribe protocols replacing the broker by a blockchain [9, 13, 14]. Those propositions all have in common an extensive usage of the blockchain, which makes them expensive over time, due to blockchain fees, and not scalable.

In this paper, we introduce SUPRA, a distributed publish/subscribe protocol. This protocol has the same security guarantees than other solutions relying on blockchains, but where the vast majority of messages are off-chain. The message exchanges are done mostly directly between publishers and subscribers and the blockchain is only used in case of network issues, if a message is lost, or an entity is suspected to be malicious.

We have implemented a proof of concept of SUPRA to show experimentally that we outperform the best known previous solution.

Keywords: blockchain, publish/subscribe protocol, missing messages detection, MQTT

# 1 Introduction

Publish/Subscribe model is a paradigm for communication protocols. This communication model is more scalable and resource efficient than the request-reply model [6]. There are different kinds of publish/subscribe protocols and we will focus on topic-based publish/subscribe protocols [6]. In such protocols, subscribers declare their interest for data, associated with a given topic, from a publisher, by sending a subscription. Then, when the publisher generates a new data with this topic, it sends it to its subscribers. This paradigm allows a unidirectional message flow from the publisher to its subscribers.

These lightweight protocols are well suited for Internet of Things (IoT) applications but have a major drawback: the central entity in the system, called the broker. In most existing protocols, such as MQTT [3], the publisher never sends directly the messages to its subscribers, but to the broker, which forwards the data to the subscribers. The subscribers also send their requests to the broker and not directly to the publishers. This third-party between the publisher and the subscribers can create trust issues. This lack of trust makes most of the existing publish/subscribe protocols unable to be used for sensitive data.

### Contributions

The contribution of this paper it twofold. First, we present general concepts for distributed publish/subscribe architectures. The first concept, the Manager/Worker model, is an abstraction that makes it easier to represent the interactions between entities, especially when they involve connected devices behind gateways. The second concept is the definition of a unidirectional channel that uses off-chain messages by default, and on-chain message as a fail-over. The second contribution is a distributed publish/subscribe protocol that does not use a central broker. In our protocol, the publisher and the subscriber are always able to agree on a common state of communication, with the help of the blockchain technology. We named our protocol SUPRA, for Secured Update Protocol with Righteous Accusations.

# 2 Backgrounds and Related work

### Centralized Publish/subscribe protocols

The most famous topic-based Publish/subscribe protocol is MQTT [3] (Message Queuing Telemetry Transport). The purpose of this protocol is to be easily implementable on the client's side with the minimum resource consumption possible when used. To do so, the protocol has a short message format and the computation complexity is located on the broker. MQTT has a version dedicated to the IoT, MQTT-SN (Sensor network). This version adds compatibility for networks without TCP/IP support [4]. There exist implementations for distributed, or load-balanced, MQTT broker [8], but the aim is to improve

the scalability and allow more publishers and subscribers to be connected simultaneously. This is very important, especially for IoT applications, but it does not leverage the security issues with having a single entity responsible for the broker deployment.

A wide variety of publish/subscribe protocols exists, most of them can be considered distributed, see for instance the survey by A. V. Uzunov [17], but they do not solve the trust issues raised by having a single entity managing an important part of the protocol, such as the list of subscriptions.

### Blockchain

Blockchain is a distributed ledger technology that was first presented by Satoshi Nakamoto in 2008 [15]. The purpose of this technology is to make a network of nodes maintaining an immutable distributed ledger of transactions. These transactions are grouped in blocks. Each block is linked to the previous block using hash pointers, and then added to the ledger. It creates a chain of blocks, hence the name blockchain.

The author of a transaction is identified with public/private key cryptography. Each transaction is signed by its author using the private key and blockchain nodes verify the signature using the author's public key. The transaction is sent to a node of the blockchain and then broadcasted over the network. Each node saves incoming transactions in a pool used to build the next blocks. Once a new block is validated, transactions integrated in this block are removed from the pool. The way blocks are appended is a result of a consensus algorithm that depends on the blockchain technology (*e.g.*, in Bitcoin a single node is elected to append the next block).

Once a block is added on the chain, the transactions in it cannot be modified (or at least the probability that a modification can be made decreases exponentially fast over the time). That is why data on the blockchain is considered immutable. This property remains true as long as a certain amount of nodes follow the protocol honestly. The minimum amount of honest nodes depends on the consensus algorithm and the blockchain implementation.

Blockchain allowing arbitrary data in transactions could by used as publish/subscribe architecture as is, without any subscription management, because anyone connected to a blockchain has a read access to all the transactions. In public blockchain like Bitcoin, this means everyone. But we present next a solution that helps the management of publish/subscribe protocols using blockchain.

### Trinity

Trinity [13, 14] is to our knowledge the first proposition for a publish/subscribe protocol which relies on a blockchain to work. In this solution, brokers use the blockchain to replicate data and store them in an immutable way. When a client publishes data by sending it to its broker, the broker forwards it to a blockchain node. The data eventually appears in a block so that other brokers retrieve this

information by reading the blockchain and forward the published data to their local subscribers.

This interaction between a publish/subscribe protocol and a blockchain creates a secured system where we can trust the brokers, because each received data can be linked to the entity that published it. However, this approach does not scale for two reasons. First, writing on the blockchain is not free. Each new transaction/data has a cost (transaction fees). With this price, the network is able to stay alive by paying blockchain nodes for their work. Over time, if a lot of data are published, this kind of solution could become expensive for the brokers or the clients. Second, data in the blockchain is immutable, so each new published data increases the size of the blockchain. The more data is sent on the blockchain, the more storage is needed for nodes. It is based on this two reasons that we design a distributed publish/subscribe protocol that use as few blockchain transactions as possible.

After Trinity, other variants were presented [9], but keeping this extensive use of blockchain for each new published data.

# 3 General Concepts

In this section, we present two general concepts used by SUPRA. The first one is the Manager-Worker model. The second one is a communication model that allows secure and verified message transmissions between two entities. We use these new concepts in SUPRA but we believe it can be used in various contexts, hence is of independent interest.

## 3.1 Managers and workers

Several distributed application architectures can be translated in a Manager-Worker model architecture. In this model, workers are entities which are capable of creating or computing data and they can only send messages to their manager. Managers are in charge of the good behavior of a set of workers. They handle the security between them and their workers and provides a gateway service to them. Managers need to cooperate to make a common distributed application work. To do so, their workers need data from workers handled by other managers. This data dependencies can be translated in topics used by publish/subscribe protocols. Managers can be publishers and subscribers.

In this model, managers are the only entities able to use directly the blockchain, because workers can only send messages to their managers. Managers can be blockchain users *i.e.*, they send transactions to a blockchain full node, or they can be blockchain full nodes themselves.

This abstraction layer allows us to map our protocol to several use cases, as presented below. Workers can be IoT devices and managers are their gateways. Workers can also be servers and a manager can be a cloud provider. The same manager can have IoT devices and servers as workers.

For instance, the supply chain use case presented in [13] can be translated to a Manager-Worker model. Each food producer is a manager and sensors that compute humidity and temperature are workers.

Access control architecture can be mapped to the Manager-Worker model. See for instance the architecture presented in [10]. In this architecture, entities called management hubs, are linked with IoT devices. These hubs, connected to a blockchain node, use the blockchain to grant or deny access to theirs devices. In this architecture, management hubs are managers and IoT devices are workers.

LoRaWAN architecture can also be translated with this model. As an illustration, Syed Muhammad et.al. [5] presented a blockchain-based two factors authentication system in LoRaWAN. In this system, the gateways and the network server are managers, and the application server, the join server, and the end-devices are workers.

### 3.2 The use of Unidirectional Channels in Distributed Publish/Subscribe Architectures

We saw in the previous section, that the Manager-Worker model is an abstract representation of Publish/Subscribe architecture. In this abstraction, only the managers can be publishers and/or subscribers, and the communication between workers and its manager are left at the discretion of the manager.

In a distributed Publish/Subscribe architecture, without a central server/broker, each publisher manager sends its data directly to the subscriber managers. The communication between one publisher and one subscriber is unidirectional, and is totally independent with the other communications between the other managers. Indeed, if there is a problem between a publisher $A$ and a subscriber $B$ that breaks up their relationship, the communication between $A$ and another subscriber should not be impacted. Of course, one manager can be at the same time publisher and subscriber. Since we are interested in having secure, reliable and auditable communications, we have to make sure that each unidirectional channel has those properties, independently from the other channels.

This observation implies that the design of a distributed Publish/Subscribe architecture, and so of SUPRA, can be decomposed in two main parts. The first part is to define a unidirectional channel protocol that allows one manager to send messages to another manager, with strong guarantees. This protocol is defined independently from our global architecture, but requires several assumptions. The second part is to define a global protocol, that allows a set of managers to setup unidirectional channels on-the-fly, allowing subscriber managers to request data from publisher managers. Setting up a channel requires, among other things, to verifies the condition of the unidirectional channel.

SUPRA is the combination of those two parts. We define in the following the first part of our architecture, which is a unidirectional channel protocol. Then, in Section 4, we define SUPRA, which dictates how managers should interact to set up secure subscriptions and use our unidirectional channel for all data communications.

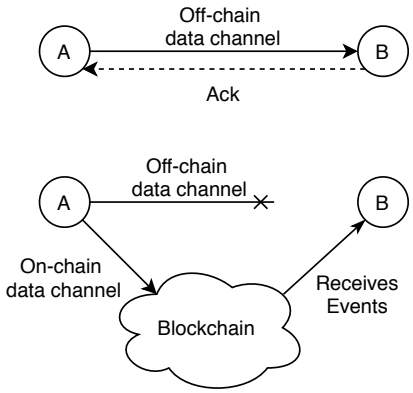


Figure 1: The two modes of communication of our unidirectional on/off chain channel protocol

## 3.3 Unidirectional Channel with On-Off Chain Proof of Delivery

We present now a new communication protocol, used by SUPRA, but can be of independent interest. The purpose of this protocol is to allow one manager to send messages to another, with delivery guarantees, and such that each manager obtains proofs for each of those guarantees. The communication protocol is a unidirectional channel, but uses similar principles and motivations as the bidirectional channels defined in the Lightning network [11].

Informally, the publisher manager sends its messages to the subscriber manager using two methods, as illustrated by Figure 1. With the first method, the messages are sent like any other messages on the internet, directly to the subscriber manager, using its IP addresses, such messages are said to be sent *off-chain* because the blockchain is not used by this method. With the second method, the messages are sent on the blockchain, included in a block, and the subscriber manager indirectly receives the message by reading the blockchain. In this case the messages are said to be sent *on-chain*.

Choosing which method to use depends on several parameters. By default, messages are sent off-chain, and if a manager tries to deviate from the protocol, or if there is a connectivity issue, messages will go on-chain. We now present what are the initial assumptions for our protocol to be used, how it works in more details, and what guarantees it provides.

### 3.3.1 Assumptions

A Unidirectional Channel with On-Off Chain Proof of Delivery between a publisher manager $A$ and a subscriber manager $B$ requires the following assumptions. Those assumptions should be verified to consider the channel *open*, and the way the two entities agree on those assumptions is not part of the protocol

(in our architecture, this is done by SUPRA).

- Each manager has a pair of private-public cryptographic keys. Each manager is aware of the public key of the other manager. A manager can update its pair of keys, but if it does, the other manager has to be aware of it.

- Both entities have to be connected to the same blockchain, either by being part of it (*i.e.*, being a full node) or by being connected to another trusted full node (or set of full nodes) in a reliable manner. In other words, we assume that each manager can receive events from the blockchain, for instance, every time a transaction associated with a specific ID or address is included into a block.

- The maximum number of transactions the publisher manager can send to the blockchain is known. So we assumed that the publisher manager defined what it considers to be the maximum number of on-chain transactions. Once all the available transactions are sent, the channel is considered closed. To reopen the channel, the two entities should agree again on this maximum number of transactions.

- Each manager can receive messages, of any type, on the blockchain. Here, we assume that each manager has a unique ID on the blockchain and there is a specific kind of transaction that can contain data of any type and is associated to a specific ID. So that when sending on-chain messages, a sender can create a transaction with the data and the corresponding subscriber manager ID. This message will be received by the subscriber manager eventually as it is supposed to be connected to the blockchain. In practice, this can also be implemented using a smart-contract.

- The size of the messages are smaller than the maximum transaction size of the blockchain. This assumption can easily be removed by considering that only a fingerprint of the message is included in the transactions on the blockchain, and that a publicly available distributed cloud storage is used to store the message. Doing so the messages are public and auditable, exactly like they are if they were included in the blockchain.

- Messages are weakly timestamped. This means that messages includes the timestamp of the sender but managers will ignore a received message if the timestamp is in the future compared to its local clock or if the timestamps of consecutive messages are not increasing.

- The managers agree on a value $T_{acknowledged}$, which represents the maximum acceptable duration between the first transmission of a message and its acknowledgment.

### 3.3.2 The Two Modes of Communication

Let $m$ be the message manager $A$ wants to send to manager $B$. The first mode of communication used by $A$ is **the off-chain transmission**. $A$ signs the concatenation of $m$ with the signature of the previous message $s_{previous}$ to obtain $s = \mathbf{sign}_A(m||s_{previous})$. Then, $A$ sends the message and the signatures $m||s_{previous}||s$ directly to $B$. When $B$ receives the message $m||s_{previous}||s$, it signs an acknowledgment $Ack_m$ and sends it to $A$. The acknowledgment is basically, the signature from $B$ of the signature of the received message $\mathbf{sign}_B(s)$. Manager $A$ is allowed to send a new message before receiving the acknowledgment, but it has to store all the non-acknowledged messages in order to re-transmit them if needed. Each new message from $A$ follows the same procedure.

From the way messages are linked by signature, an acknowledgment implicitly acknowledges all the previous messages. So if $B$ receives several messages from $A$ that are correctly linked together (*i.e.*, no message is missing), $B$ can send an acknowledgment only for the last received message.

If $A$ does not receive the acknowledgment from $B$ before a certain amount of time has passed (due to a connectivity issue or because $B$ does not send it), $A$ can send again the same message to ask $B$ to send an acknowledgment again. $A$ is free to decide when it re-transmits the message, however, it needs a proof of delivery before $T_{acknowledged}$ after the first transmission. To ensure $B$ receives the message before $T_{acknowledged}$, $A$ can use the second transmission mode: the on-chain method, described later. The second method may take a time $\Delta_{on-chain}$ before a message is confirmed. Hence, after a delay $T_{off-ack} = T_{acknowledged} - \Delta_{on-chain}$ from the first transmission of a message, if no acknowledgment is received, $A$ uses the second method of transmission to ensure the correct delivery before $T_{acknowledged}$.

**For an on-chain transmission**, $A$ sends the message and its signature $m||s_{previous}||s$ to the blockchain associated with the ID of destination $B$. Doing so, as soon as the message is included into a block in the blockchain, manager $A$ knows that $B$ is aware of the message, by assumption.

Another thing that can happen, is that $B$ does not receive a message from $A$, and $A$ does not send it on-chain. Then $B$ has no way to know that it missed a message until $A$ sends another message. When $B$ finally receives a message $m||s_{previous}||s$ from $A$, it can verifies whether the signature $s_{previous}$ equals the signature of the last received message. If it is not equal, $B$ knows that a message is missing. $B$ knows that $A$ is supposed to send the message again off-chain or on-chain, so it just waits. If $B$ does not receive the message after $T_{acknowledged}$, it knows that $A$ did not, or could not, follow the protocol.

### 3.3.3 Delivery Guarantees

Our protocol offers several delivery guarantees, and generates proofs that can be shown publicly to prove to anyone that a manager did followed the protocol.

**Proof of integrity and origin of data:** In the case where an off-chain

message is correctly received, the message
$m||s_{previous}||s$ is a proof for $B$ that $A$ sent the message. If there is a connectivity issue and $B$ does not receive the message directly from $A$, then $A$ sent the message on-chain, so the proof is the same. Hence, our protocol verifies the non-repudiation property.

**Proof of delivery:** In the case where an off-chain message and its acknowledgment are correctly received, the acknowledgment from $B$ is a proof, for $A$, that $B$ correctly received the message. If there is a connectivity issue and $A$ does not receive an acknowledgment from $B$, then the on-chain message of $A$ containing the data is a proof that $B$ received correctly the message, by assumption.

**Proof of non-delivery:** If $B$ detects that a message is missing, and does not receive it (off-chain or on-chain) before a delay $T_{acknowledged}$, then the signature of the missing message can be used to prove that $A$ did not delivered correctly the message. Indeed, the signature proves that $A$ sent a message, and if $B$ did not received this message, $A$ cannot provide a proof of delivery.

Our protocol is used in our SUPRA architecture for almost all the communications, and the proofs are used in our trial section where conflicts are handled, making sure honest entities can prove they followed the protocol to avoid penalties.

# 4 SUPRA

SUPRA is a distributed publish/subscribe protocol using the manager/worker model and unidirectional channels with on-off chain proof of delivery. Managers are publishers or subscribers and use topics as a filter for data, just like MQTT. This protocol is focused on inter-manager communications as all communications are between managers. There is no restriction on which protocol managers use between them and theirs workers.

We are using Universal Unique Identifiers (UUID) to retrieve transactions from the blockchain. Each transaction is associated with a source and destination UUID, and each manager (being either connected to a blockchain node or a blockchain node itself) can listen to given UUIDs and receive the associated transactions when they are included in a block. UUIDs are used in the Katena [16] blockchain and could be replaced by a Smart-contract in other blockchains where this is possible.

SUPRA is divided into fives modules. The first one is the public key module, it manages the public identities of managers. The second one is the subscription module that handles the subscriptions. The third one is the publishing module, used to publish data with our previously defined communication channel. The fourth one is the fail-over module, which retrieves communication state after a crash from a manager. The last module is the trial module, it detects malicious managers in the system.

We now explain SUPRA in more details. We assume that messages are transferred in a partially synchronous model on unreliable links.

## 4.1 Communication example

In Figure 2, we present a communication example between two managers. In this example, A is a subscriber and B is a publisher. After declaring theirs public keys on the blockchain, the manager A retrieves the public key of B and sends a subscription demand to B. At the reception of the demand, B retrieves A's key on-chain and checks the signature of the message. Thereafter, signature checking is performed by A and B at each message reception. In this example, the first data published by B is received and acknowledged directly by A using the off-chain channel. The message containing the second data publication is lost but the message disappearance is detected only at the reception of the third data publication, because A detects a missing signature in the message. This missing message detection was explained in Section 3.3. Hence, A warns B of a missing message, the message is resent, and A acknowledges it. Notice that A acknowledges the third data and not the second one, because acknowledging a message implicitly acknowledges all the previous messages. Another way to retrieve a missing message is shown with the fourth data publication. B resent the fourth publication on-chain because it did not receive in time the acknowledgement. $A$ retrieves the message from the blockchain directly (either it receives event from the blockchain node, or $A$ requests periodically the blockchain for new messages). Observe that $A$ does not acknowledge the last message. Indeed, an on-chain message is assumed to be always acknowledge (see Section 3.3). We will now provide in the next sections further details for this exchange between A and B.

## 4.2 Generic message format

All the messages of SUPRA are represented on Figure 3. They all have in common the following specifications. They start with a two hexadecimal digits code on one byte. The first digit is for the module, and the second digit for the type of message inside the module. We will write the message codes with this format: $X - Y$. Each code is associated with a specific action and format. Next to it, there is a timestamp. We use the UNIX timestamp format which is a 8 bytes long unsigned integer with milliseconds precision. This timestamp adds unpredictability in messages. This property is important in a security environment based on signature. Indeed, if you have every possible outputs signed by someone, you can say everything you want on his behalf without his consent (*e.g.*, replay attacks). The timestamp prevents this type of behavior by creating a chronology between messages. This property is explained in further detail in Section 4.7. We will assume that two messages can not have the same timestamp. Next to the timestamp, the message has the UUID of the destination. The last part of the message is a signature to prove the identity of the author. We use ED25519 as a signature protocol, where signatures are 64 bytes long and public keys are 32 bytes long.

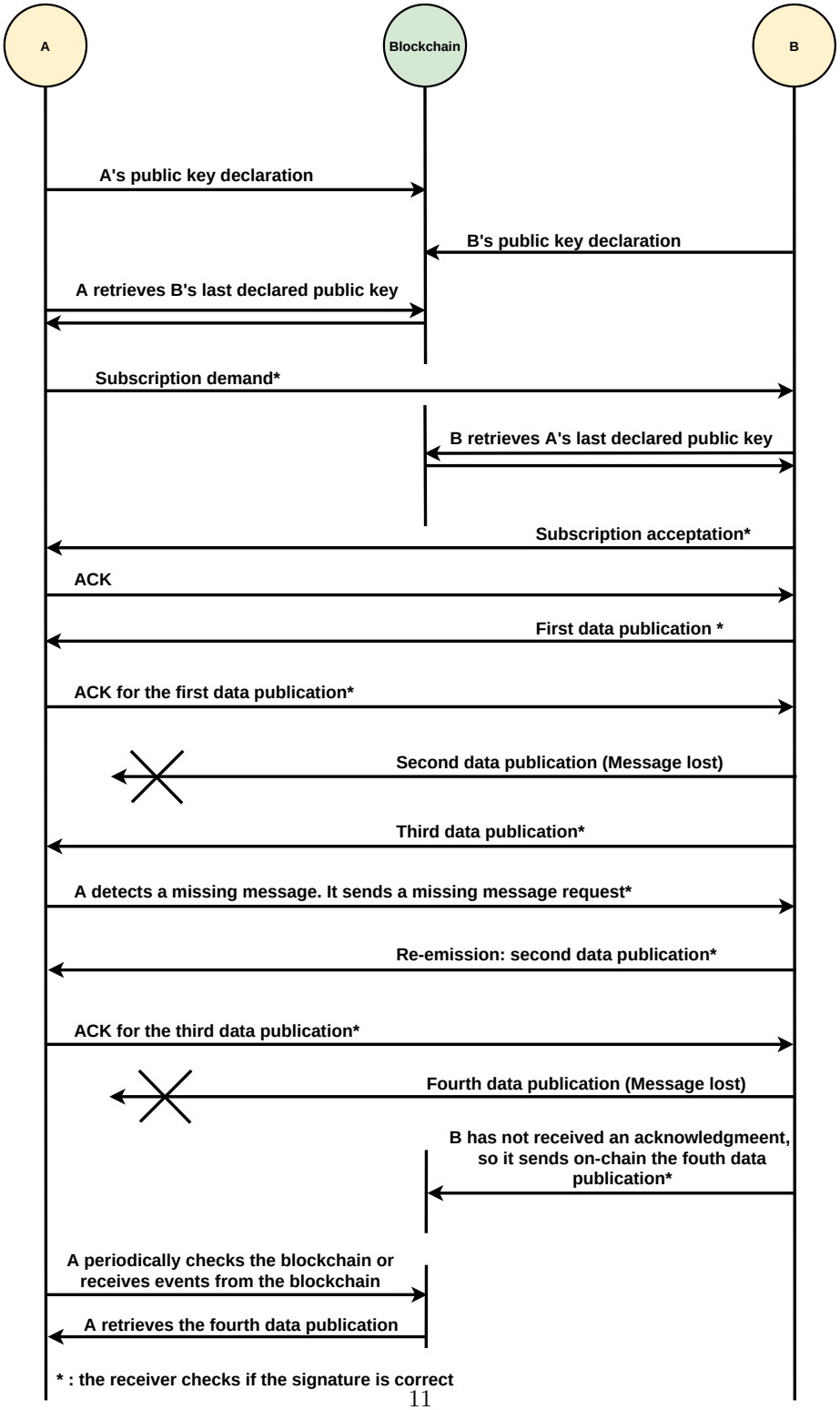




Figure 2: Message exchanges between two managers

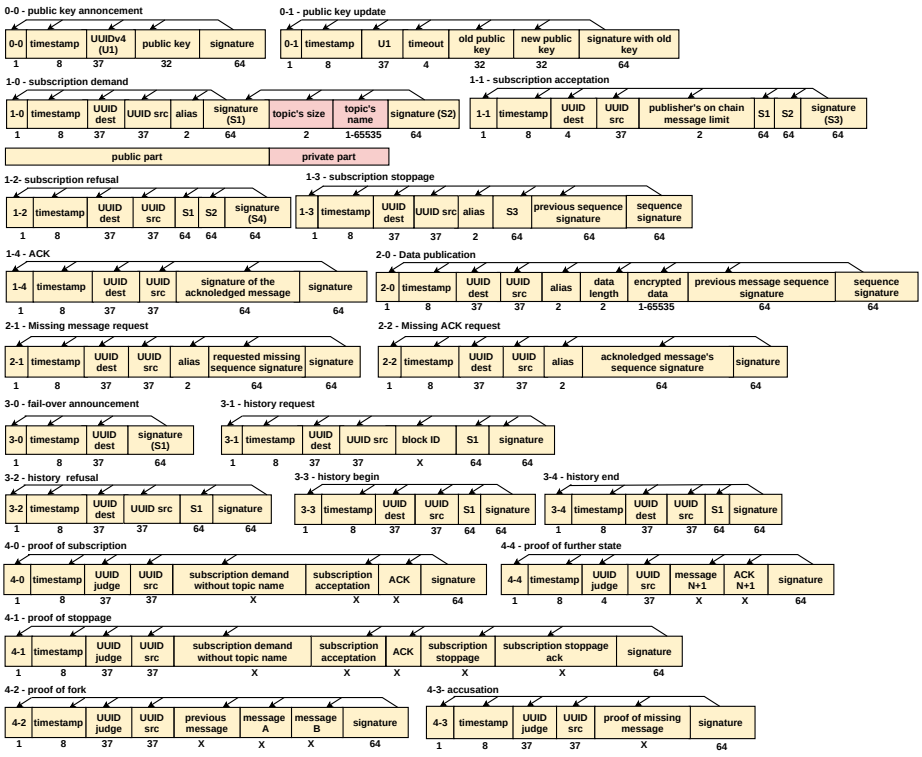


Figure 3: SUPRA message format

## 4.3 Public key module

In this part we explain how the public key module works and the message format it uses. The purpose of this module is to begin the setup of a channel described in Section 3.3 by sharing public keys between managers.

Messages in the public key modules are sent directly on the blockchain. The first message has the code 0-0 and is used to declare the first public key of a manager. This message is signed with the private key associated with the public key inside the message. The manager can then broadcast the corresponding UUID to the other managers, to make them aware of its identity. It is important to notice that the identity of a manager corresponds to its UUID and not directly its public key.

The first manager who declares a key is the owner of the UUID, and the only manager allowed to update keys associated with this UUID.

The purpose of the second message of the module, with code 0-1, is to allow a manager to update its private key, for instance when the integrity of its private key is at risk. The message contains the new public key but is signed with the old private key and must use the same UUID. Other managers listening to this UUID are then notified. For a given UUID, anyone can have a history of every public keys used by a manager at each point in time. It creates traceability on the author's identity and it is used by the trial module in Section 4.7. The timeout attribute of message 0-1 allows managers to prepare the key switch. The key switch takes effect immediately after reaching the timeout, which is a timestamp on 4 bytes.

Messages exchanged in other modules are signed with the private key associated with the last declared public key of the sender. For security purposes, sensitive part of message, such as the data, is encrypted with the public key of the receiver.

## 4.4 Subscription module

We now explain how the subscriptions are done by explaining the two parts of a subscriptions, the subscription demand and the subscription stoppage.

### 4.4.1 Subscription demand

The subscription demand is the first part of a subscription. During this part, a manager/subscriber is asking for the manager/publisher's permission to get messages associated with a topic. This is done by sending a message with code 1-0. This message contains a public and a private part. The public part is used by the subscriber to declare an alias.

The private part of the message 1-0 is the topic name. A second signature is used to link the topic name with the chosen alias. Using an alias has two advantages. The first one is to identified each subscription by using less space than a topic name. If a subscriber has several subscriptions to the same publisher at the same time, it needs an information to know from which subscription a

new data is. Adding the alias with the published data, allows the subscriber and the publisher to identify a subscription by using a 2 bytes long value, instead of a topic name, which can be 65 535 bytes long. This optimisation reduces the size of the exchanged messages. Notice that the same optimization is done in MQTT-SN [7].

The second advantage is to prevent information leaks. Topic names can reveal private information on someone (Names, addresses) and to resolve issues between managers we need to publicly reveal on-chain some messages. The public part of the message, with only the alias, is enough to prove the existence of a subscription, and it is better to reveal only a small random alias, instead of the whole topic name. This revealing process is explained in Section 4.7.

At the reception of a subscription demand, a manager/publisher answers to it favorably or not. The unfavorable answer begins with a code 1-2 and is composed of the signature from the publisher of the public part of the demand, and the signature of the private part. The favorable answer has a code 1-1 and is used to declare the maximum amount of on-chain messages that the publisher is ready to send (which is needed to use our unidirectional channel described in Section 3.3). These two messages need to be acknowledged with a code 1-4 by the subscriber.

When the subscription has a favorable answer from the publisher and is acknowledged by the subscriber, then the unidirectional channel described in Section 3.3 is ready to be used to transfer data from the publisher to the subscriber.

### 4.4.2 Subscription stoppage

At any point in time, the subscriber or the publisher can end the subscription by sending a message with code 1-3. The reason could be that the subscriber is not interested anymore by the topic or, for the publisher, that the topic is no longer available. The message includes the signature of the last data publication message of the channel (message with code 2-0). The manager that initiates the stoppage needs an ACK (message with code 1-4) from the other manager, or it needs to send the subscription stoppage on-chain. The subscription is considered ended at the reception of this acknowledgement or when the subscription stoppage is added in a block, and the data-publication messages sent after the stoppage can be ignored by both managers.

Another way to stop a subscription is to reach the limit of published messages on the blockchain. It implicitly stops the subscription without the need to send a stoppage message. Managers need to make a subscription again, if they want to keep sharing data.

## 4.5 Publishing module

This module is in charge of the data transmission from the publisher to the subscriber. This module uses the unidirectional channel with on-off chain proof of delivery presented in Section 3.3.

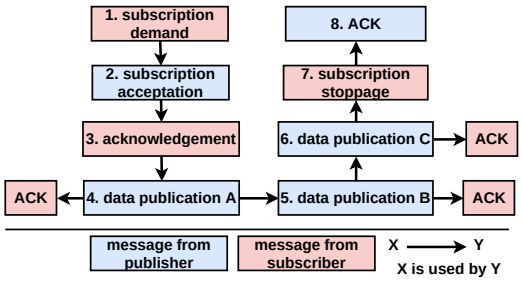


Figure 4: Example of communication between managers

The main message of this module is the data publication, with the code 2-0. This message can only be sent by the publisher. Figure 4 represents a message exchange between two managers with the dependencies between messages. With these dependencies, called sequence signature, a manager can detect missing messages, like explained in Section 3.3, because each message includes the signature of the previous message.

This module provides warnings for missing messages, with a code 2-1, and missing acknowledgements, with a code 2-2. These two additional messages can be used to quickly solve missing messages. Warnings don't have sequence signatures and are only sent off-chain, because the protocol can work without them.

Since some messages are publicly revealed, either because they are on-chain or because we are using the trial module defined in Section 4.7, to prevent information leak, the published data inside the message 2-0 is encrypted with the subscriber's public key. Hence, the subscriber is the only manager capable of decoding this data.

Messages with code 2-0 are acknowledged explicitly, by the receiver, with an ACK (code 1-4) or, by assumption, by sending the message on-chain.

## 4.6 Network issue and fail-over

Network issues can be the result of a network congestion or of a malicious entity who purposefully drops messages. These events have the same result: a message is dropped and does not reach the destination. SUPRA uses the mechanism describe in Section 3.3 to handle this events.

When off-chain warnings are not enough to retrieve missing messages, managers need to send messages on the blockchain. With this method, managers are sure that the messages will reach the destination, because the managers are supposed to remain reliably connected to the blockchain.

SUPRA has a fail-over module to allow managers to retrieve the aliases and the last sequence signatures for theirs current subscriptions when they crash. After a manager has crashed (when it is back online), it needs first to declare the event on-chain. This is done by sending to itself a message with a code 3-0

on-chain. This message is called a fail-over announcement. Then, it can join the other managers with a history request, code 3-1. This request contains the block where the fail-over announcement is on the blockchain. Managers can accept or refuse the request. If the request is refused, all the active subscriptions between the two managers are stopped, because one manager is not able to verify the signatures anymore, since it has lost the current states of the subscriptions. If the request is accepted, the manager resends off-chain all the stored previous messages intended to the crashed manager. The crashed manager has to acknowledge each message. When all the messages are received, then the other manager sends a message with a code 3-4 to warn the crashed manager that it has all the messages. We explain in Section 4.7.2 how managers need to store messages.

The same fail-over announcement can be used to join several managers. So a manager can contact all its known managers (subscribed to or from) and retrieve all its subscription states. On the other hand, the same fail-over announcement cannot be used for two different crashes. If a manager tries to do so, it will be easily detected, because the other manager already have received a history request with this fail-over announcement or because the timestamps do not match. For instance, subscriptions created after the on-chain fail-over announcement message cannot be used to recover the sate of this subscription.

Declaring errors on-chain makes manager behaviors publicly auditable. Those behaviors can be used by other managers to measure manager's reliability. Managers want to spend as less money as possible in on-chain transactions, so they prefer to cooperate with managers who force the least number of on-chain messages but also who are reliable and do not crash.

## 4.7 Trial module

In this section, we describe how the trial module works by explaining the concept of proof in SUPRA and the different proofs available.

### 4.7.1 Proofs, accusations and the judge

The purpose of this module is to detect malicious managers *i.e.*, managers that do not follow the protocol, by using the judge. The judge is a program with a trusted execution. To do so, the judge can be a smart-contract inside the blockchain, because the execution of a smart-contract can be verified by every users. Detecting malicious managers allows to reduce the risk of paying fees, avoiding honest manager to send on-chain messages.

The trial works as follow, the accuser manager sends his accusation to the judge and the defendant manager has an amount of time to present counter-arguments. The judge then decides which manager is malicious. An interesting property for the trial is that an honest manager will always be able to defend himself from a false accusation and will always win when accusing a malicious manager. Also, if every managers are honest, this module should never be used.

The messages sent by managers to the judge are called proofs. They are generated by managers while using SUPRA. These proofs are composed of previously exchanged messages between managers. This generation is due to the dependencies in SUPRA's messages and the properties of the unidirectional channel with on-off chain proof of delivery. The proofs have two important properties. First, they are verifiable by everyone outside the system. Second, they do not reveal sensitive information from managers. We assume that every managers know the judge's UUID. To prevent a certain amount of false accusations, at the detection of a missing message, a manager needs to wait $T_{acknowledged}$ before accusing a manager of an error. This delay allows managers to resend missing messages. There is also a limitation period for the accusation called $T_{limit}$. It is used to prevent the storage of every exchanged messages by the managers. $T_{acknowledged}$ and $T_{limit}$ need to be known by every manager, and they could be declared in the judge, as variables in the smart-contract.

### 4.7.2 Message conservation

The trial is based on comparisons between exchanged messages, so managers need to retain messages for a certain amount of time. Not storing messages removes the chance for a manager to defend itself in case of a false accusation.

Messages from the subscription and fail-over modules should be kept during the whole lifetime of the subscription because a manager needs to show them if there is an error.

Messages from the publishing module that required an acknowledgement need to be stored until the acknowledgment is received. Then, managers only need to store, for each subscription, the last off-chain acknowledged message, and its ack. Keeping these messages allows managers to reveal the sequence signature reached before the last network issue.

After a subscription stoppage, every messages can be deleted after reaching $T_{limit}$ from the last sent message of the subscription.

### 4.7.3 Accuser's proofs

The accuser needs to show three mandatory messages to generate a valid accusation. These messages are: the proof of identity, the proof of subscription, and an accusation or a proof of fork.

The proof of identity is the managers' UUID. This UUID is in every exchanged messages and managers can use this UUID to retrieve the key history of a specific manager. With the timestamp inside the messages and the key history, the judge can check which key should have been used at each moment in time and verify the signatures.

The proof of subscription is composed of every exchanged messages during the subscription process. In this messages, we removed a sensitive part, the topic's name, because it can leak information on a manager. These messages show that both managers agreed on a subscription with a certain alias, but it do not reveal the topic name.

The last mandatory message is the accusation with the code 4-3. This message presents a proof of missing message *i.e.*, a message with a sequence signature that, according to the accuser, do not match the previously received message. This *accusation message* can be a data publication or a missing acknowledgment request, and cannot be forged because it is signed with the suspected manager's private key. The meaning of this accusation is to declare that a manager has not followed the protocol and a message was not delivered before $T_{acknowledged}$. To be valid, the accusation message must not be older than $T_{limit}$.

Another possible issue with messages is a fork on the sequence signature. As presented in Section 4.5, dependencies between messages create a chain of signatures. If a manager detects a fork in sequence signature with two messages from the same manager, it means that this manager sent two different messages with the same previous one, which is an error that the manager can only do on purpose.

By revealing these messages, the accuser proves the identity of the suspected manager, proves a subscription, and proves the existence of an incorrect signature sequence.

### 4.7.4 Defendant's proofs

In this part, we assume that the judge has checked the accusation's proofs and has considered them as valid. It means that every messages are signed with the right keys and that no on-chain message denies a proof. For instance, the judge could easily check if the missing messages are included in the blockchain, in a block older than the block containing the accusation, if it does find them, the accusation is invalid.

A defendant that is victim of a false accusation, and that followed the protocol, is always able to present one of the proof below, because it has at least one message that denies a proof from the accuser.

If the accusation is false, but no on-chain message proves it, there are two possibilities. First, the accusation message is from an older subscription with the same alias between the same managers. Second, the accusation message is an old message, correctly acknowledged, from the current subscription. Both possibilities can be denied with off-chain messages stored by the defendant.

If the accusation message is from an old subscription, then it was stopped recently (because the accusation message is not older than $T_{limit}$ and was generated by the publisher) so the defendant should not have deleted the subscription stoppage. Showing the subscription stoppage to the judge proves that the accusation is false.

If the accusation message was correctly acknowledged, then the defendant should have an acknowledgement for this message, or a more recent message of the same subscription. The defendant can reveal the last sent message and its acknowledgment to the judge. This message's timestamp and the accuser's acknowledgment prove that the accusation is invalid.

# 5 Comparison with existing solutions

In this section, we compare SUPRA with two solutions: MQTT and Trinity, which is the first combination of MQTT and blockchain to our knowledge.

### MQTT

We will compare MQTT and SUPRA to verify that SUPRA removes the issues linked with central broker.

The two common ways to add security in MQTT is to use SSL/TSL encryption and to use an authentication system on the broker. It is enough to remove overhearing by a third-party but not enough to remove the star topology with the central broker. This broker can be the source of trust issues, if it is malicious. In SUPRA, managers communicate directly between each other, to create subscription and send data. It removes the central authority from the system.

MQTT runs over the L4 protocol TCP [1]. TCP gives guaranties on message delivery: messages are ordered and they reach the destination, if it is available. The cost of these guaranties are a longer message header and a lower throughput than other L4 protocol, like UDP [12]. SUPRA also guaranty messages order, with the sequence signature, and messages delivery. So SUPRA can work on a L4 protocol that does not provide this guarantees, such as UDP. The on-chain and off-chain communications allow SUPRA to have the level 2 in MQTT's QoS classes *i.e.*, messages are delivered exactly once, because managers will ignore re-transmissions of an previously received message.

These differences between MQTT and SUPRA make SUPRA more secured and also increase its number of use-cases. SUPRA can be used when traceability is needed in data exchanges or if the central broker is not trusted.

### Trinity

To compare Trinity [14] and SUPRA, we use the 19 distributed publish/subscribe protocol threats presented in [17]. These threats were used in [17] to compare several security solutions in publish/subscribe protocols. Each threat is presented by its ID in the survey, $TX$ where $X \in \{1, 2, ..., 19\}$.

Trinity and SUPRA are alike against several threats, because these protocols share similar properties. Messages are signed, which prevent attacks based on data alterations or spoofing ($T4, T5, T6, T7, T10$) and there is a mechanisms to prevent malformed messages ($T9$). In Trinity, it is the API used by the brokers, and, in SUPRA, it is the judge. Malformed proofs are ignored by the judge. The blockchain also allows these protocols to handle crashed manager/broker, because messages lost during a crash are available on-chain after the crash recovery ($T18$).

In Trinity, all the data published on a blockchain is only accessible by the brokers. In SUPRA, we use a new communication paradigm to send data off-chain as much as possible. The blockchain is used to create trust in managers'

identities and as a backup communication link for data. Despite this difference between Trinity and SUPRA, both protocols are protected against unauthorized actors ($T11$) and data repudiation ($T17, T19$). However, there is a risk of eavesdropping ($T1, T2$) in Trinity.

Trinity's blockchain network is only available for brokers, it supposes a minimal amount of trust between brokers, because of this trust, data are not encrypted on-chain. Thus, brokers have access to every published data, even data from a topic for which they have no subscribers. In SUPRA, on-chain data are encrypted and only the receiver can read it, which removes eavesdropping. Managers trust the blockchain as a whole, but they do not trust other managers.

Both protocols don't implement tools against flooding and deny of service-based attacks ($T14, T15, T16, T17$). SUPRA and Trinity also share a risk of subscription leak ($T12$), if a manager, or a broker, is hacked. These leaks will only concern the local subscribers, in Trinity's case, or the manager's subscription, in SUPRA's case, but not every subscriptions in the network. Trinity implements a QoS system on published data in [13], this feature is not present in SUPRA but can be added in a future work.

SUPRA and Trinity do not implement routing protocols (layer 3) in their specifications, so we think that it is not meaningful to compare them against route poisoning attacks ($T7$).

# 6 Proof of concept and experimentation

We developed a proof of concept of SUPRA [2] compatible with Transchain's blockchain with the purpose of compare message delivery delay between SUPRA and Trinity. We did not implement the judge and the fail-over system, and published data have a fixed size of 64 bytes.

With SUPRA, we simulate a 10% error rate for the off-chain channel, and a message is resend once off-chain before being sent on-chain. This is a very pessimistic scenario for SUPRA as in practice, transmission over the Internet is more reliable.

We chose a fixed data publication rate and the delay between message retransmissions (for SUPRA) is equal to the delay between two consecutive data publication. We have two publication rates, one of 10 data per second and the other of 100. To make measurements with Trinity, we reuse our code for SUPRA but we forced the usage of the on-chain channel, and we remove the error rate.

We did 15 experimentations with each publication rate where we measured the delay, for each message, between the first transmission by the publisher and its delivery by the subscriber. A message is delivered by the subscriber when every messages before it has been delivered, *i.e.*there is no missing signatures. Each experimentation has 2000 data publications, which makes 30 000 data data publications in total. Our results are shown in Figures 5 and 6.

We observe two things. First, that SUPRA looks faster than Trinity, regardless of the data publication rate. This is logical because most of SUPRA messages are sent directly to the subscriber, and we do not need to wait for the

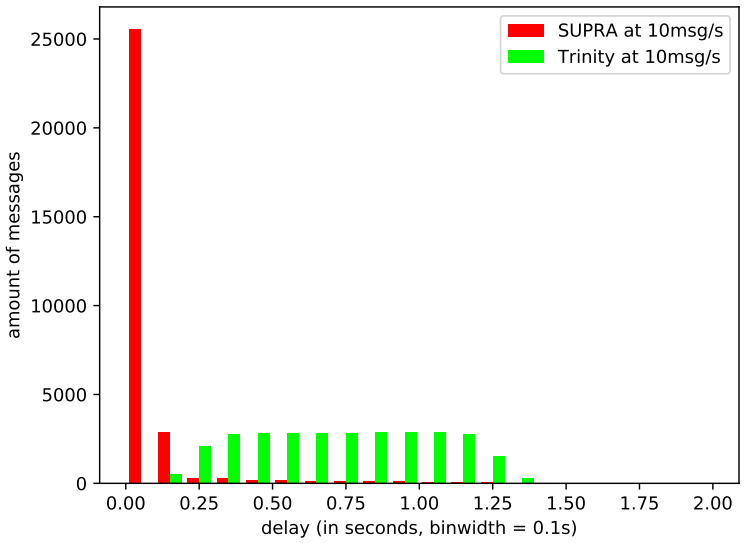


Figure 5: Message distribution based on delay for Trinity and SUPRA at 10 data per second

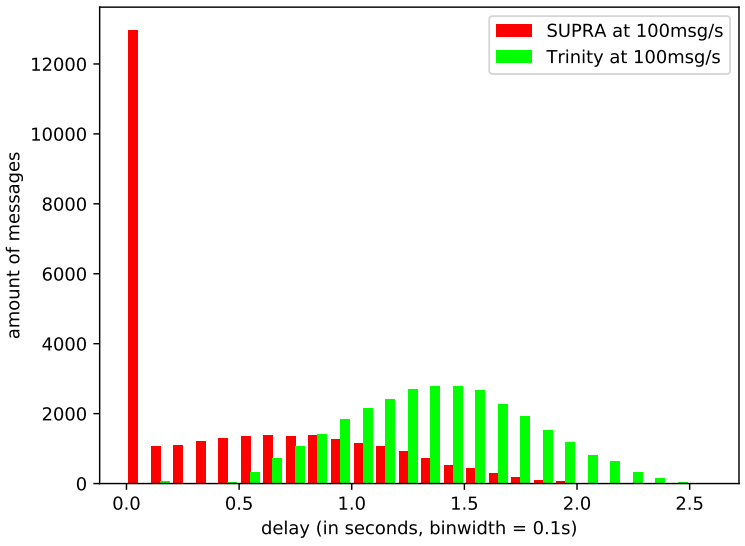


Figure 6: Message distribution based on delay for Trinity and SUPRA at 100 data per second

message availability on-chain. Second, we see that the publication rate has an impact in message delay for SUPRA and Trinity.

Indeed, at a rate of 10 publications per second, Trinity's messages are delivered between 0.10 and 1.3 seconds after the first sending and at a rate of 100 the delivery takes between 0.5 and 2.4s. This difference is linked with the blockchain's workload. Every message in Trinity go through the blockchain, it means that increasing Trinity's message rate will increase the work needed to generate a new block because there is more transaction to check. Remark that the use of the Traschain Blockchain is actually an advantage for Trinity as it can handle an important workload, whereas the same experiment using Ethereum for instance would have been much worse for Trinity.

For SUPRA, we set the error rate at 10% which means that 10% of the data will be re-transmitted off-chain and 10% of this 10% will be sent on-chain, which correspond to around 1% of the total amount of data. This means that around 90% of the messages should be received almost instantly, because the UDP delay is negligible, 9% should be slightly delayed because of the off-chain re-transmission, and 1% should be greatly delayed because of the on-chain transmission. This distribution is visible on figure 5. On this figure, we can observe that around 25 000 messages where received between 0.0 and 0.1s and around 2 500 messages are received between 0.1 and 0.2s. At 10 messages per seconds, a messages is resent off-chain after 0.1 seconds, if not acknowledged. Messages received between 0.0 and 0.1s are the messages received with the first sending and messages received between 0.1 and 0.2s are the messages received after the off-chain re-transmission.

We can also observe a tail in SUPRA's messages distribution. In figure 5, it starts at 0.2s and ends at 0.8s. In figure 6, it is more visible because it starts at 0.1s and ends at 2s. These tails are the result of the data publication rate and message's on-chain sending. While a subscriber is waiting for a missing message to be available on-chain, it keeps receiving new publications (that arrives faster because they are sent off-chain) and put them in stand-by because they can be delivered only after the on-chain message is retrieved. The tails on Figures 5 and 6 corresponds to those messages. Increasing the publication rate will increase the amount of messages received by the subscriber before finding the missing message on-chain.

# 7 Conclusion and future works

Blockchain data immutability has the potential to solve trust issues in distributed system. Blockchain can work as a trusted environment where everyone can check the information. This idea is the base idea for several blockchain-based communication protocols [9, 13, 14]. The main issue with these solutions is that they are over centered on the blockchain: every communication go trough the blockchain and this is not scalable. In this article, we presented SUPRA, which is a blockchain-based publish/subscribe protocol. It relies on the blockchain only to build trust between managers and resolve message delivery issues. This scal-

able protocol is used between managers to exchange data in a secured, traceable and autitable way using the publish/subscribe paradigm.

SUPRA can still be improved and we have two ideas on how to do it. The first idea is to add QoS rules. QoS rules are a feature presents in Trinity [13], with the help of smart-contracts. To add this feature, publishers could present a list of rules for the data during the subscription. When the subscriber detects an issue, it could use the Judge and the trial module to resolve it. The main issue for the moment with this system is that you need to reveal a decrypted version of the data to prove the error, if the error is about the value of the data itself. Otherwise, the judge cannot check the issue because the data is encrypted.

The second idea is to export our communication paradigm in other communication protocols. For instance, TCP [1] is a protocol that allows bidirectional communications, a property publish/subscribe protocols are not designed for. By merging SUPRA and TCP's message formats, we could create a bidirectional, secured, and auditable communication protocol.

# References


[1] Transmission Control Protocol. RFC 793, September 1981.

[2] Supra proof of concept. Waiting for the paper's acceptation before making the proof of concept public., 2021.

[3] Ken Borgendale Andrew Banks, Ed Briggs and Rahul Gupta. Mqtt version 5.0. `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`, 2019. Accessed: 2020-06-09.

[4] andyp. Mqtt for sensor networks – mqtt-sn. `https://mqtt.org/tag/mqtt-sn`, 2013. Accessed: 2020-06-09.

[5] Syed Muhammad Danish, Marios Lestas, Waqar Asif, Hassaan Khaliq Qureshi, and Muttukrishnan Rajarajan. A Lightweight Blockchain Based Two Factor Authentication Mechanism for LoRaWAN Join Procedure. *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2019.

[6] Patrick Th Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[7] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008.

[8] R. Kawaguchi and M. Bandai. A distributed mqtt broker system for location-based iot applications. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–4, 2019.

[9] Pin Lv, Licheng Wang, Huijun Zhu, Wenbo Deng, and Lize Gu. An IOT-oriented privacy-preserving publish/subscribe model over blockchains. *IEEE Access*, 7:41309–41314, 2019.

[10] Oscar Novo. Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT. *IEEE Internet of Things Journal*, 5(2):1184–1195, 2018.

[11] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[12] Jon Postel. User datagram protocol. RFC 768, 1980.

[13] Gowri Sankar Ramachandran, Kwame-Lante Wright, and Bhaskar Krishnamachari. Trinity: A Distributed Publish/Subscribe Broker with Blockchain-based Immutability. pages 1–8, 2018.

[14] Gowri Sankar Ramachandran, Kwame Lante Wright, Licheng Zheng, Pavas Navaney, Muhammad Naveed, Bhaskar Krishnamachari, and Jagjit Dhaliwal. Trinity: A byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence. *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency*, pages 227–235, 2019.

[15] Satoshi. Bitcoin: A peer-to-peer electronic cash system. pages 1–9, 2008.

[16] Transchain. Katena, 2019. Accessed: 2020-06-09.

[17] Anton V. Uzunov. A survey of security solutions for distributed publish/subscribe systems. *Computers and Security*, 61:94–129, 2016.