UIP2SOP: A Unique IoT Network applying Single Sign-On and Message Queue Protocol

Lam Nguyen Tran Thanh[1], Nguyen Ngoc Phien[2*], The Anh Nguyen[3], Hong Khanh Vo[4], Hoang Huong Luong[5], Tuan Dao Anh[6], Khoi Nguyen Huynh Tuan[7], Ha Xuan Son[8]
VNPT Information Technology Company, Ho Chi Minh city, Vietnam[1]
Center for Applied Information Technology, Ton Duc Thang University, Ho Chi Minh City, Viet Nam[2]
Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam[2]
Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VSB-Technical University of Ostrava, Ostrava, Czech Republic[2]
FPT University, Can Tho City, Viet Nam[3,4,5,6,7]
University of Insubria, Varese, Italy[8]
Corresponding Author: Ton Duc Thang University, Ho Chi Minh City, Vietnam[*]

Abstract—**Internet of Things (IoT), currently, plays an importance role in our life, also, this is one of the most rapidly developing technology trends. However, the present structure has some limitation - one of these is the communication via client-server model - the users, devices, and applications using IoT services where all the connection/requirement is managed at IoT service providers. On the one hand, the IoT service providers (e.g., individual, organization) have different method to manage their devices, services, and users. Thus, the unique standard (i.e., communication method among the service providers and between client server) is still the challenge for the developers. On the other hand, Message Queuing Telemetry Protocol (MQTT) that is one of the most popular protocols in IoT deployments, has significant security and privacy issues by itself (e.g., authentication, authorization, as well as privacy problem). Therefore, this paper proposes UIP2SOP - an unique IoT network by using Single Sign-On (SSO) and message queue to improve the MQTT protocol's security problem. Besides, this model allows the organizations to provide the IoT services to connect into a single network but does not change the architecture of organization at all. The evaluation section proves the effectiveness of our proposed model. In particular, we consider the number of concurrent users publishing messages simultaneously in the two scenarios i) internal communication and ii) external communication. In addition, we evaluate recovery ability of system when occurred broken connection. Finally, to engage further reproducibility and improvement, we share a complete code solution is publicized on the GitHub repository.**

Keywords—Internet of Things (IoT); MQTT; OAuth; Single Sign-On; Kafka; message queue

I. Introduction

The Internet of Thing (IoT) services/applications grown and play a vital role in our life such services/applications have applied in most fields such as smart cities, healthcare, supply chains, industry, and agriculture. In fact, by 2025, the whole world have approximately 75.44 billion IoT-connected devices [1], [2]. However, these devices is own by the different individuals or organizations, its characteristic (e.g., capacity, communication, compuation ability) is totally different. Therefore, the ability to connect IoT service providers as well as the security issues are still an open problem and need more considerable from the developers.

The most of IoT systems, currently, are a centrally designed according to a client-server architecture [3], [4], the individual/organization requires all devices and users to authenticate exchange information through one or more of the its servers. This architecture is suitable when the number of devices is limited - one advantage can easily aware is this model can easily setup and deploy in the real environment. However, when the system is extended with a millions of users/devices join the IoT network, these may create a billion transaction among them in the short time, we should consider the latency or even the deadlock issues.

For the IoT devices, the current architecture have the limited network connectivity, power, and processing capabilities [5], [6], so there is a specific requirement for separate machine-to-machine (M2M) protocols, unlike traditional communication protocols. The five most prominent protocols used for IoT devices (i.e., communicate among them and communicate with the upper layers) are Hypertext Transfer Protocol (HTTP), Constrained Application Protocol (CoAP), Extensible Messaging and Presence Protocol (XMPP), Advanced Message Queuing Protocol (AMQP), and Message Queuing Telemetry Protocol (MQTT) [7], [8]. For the communication requirements in limited networks (constrained networks), MQTT and CoAP are more reasonable to be used [9]. Besides, we found that the MQTT protocol has faster creation time, and transmission time of the packet is twice as fast as the CoAP protocol [10]. Furthermore, for developers of low bandwidth and memory devices, MQTT is the most preferred protocol [11]. Therefore, this paper applies the MQTT protocol to develop UIP2SOP platform.

Nevertheless, the current MQTT model has some drawback especially in security and privacy issues, namely data confidentiality, availability, integrity, and privacy [12]. This model only provides identity, authentication, and authorization for the security mechanism [13] but it is very simple. Lundgren et al. [14] indicate that a simple ruby script is used to subscribe to topic # of any random MQTT server that is public on the Internet, and the obtained result reveals the data, including the device's GPS location, without any authentication, which is a severe security risk. In the MQTT protocol, the Client ID is unique, and MQTT Broker uses this Client ID to identify the client and its status. From this feature, an attacker can take advantage of performing a denial-of-service (DoS) attack. This is considered a risk in terms of the availability of the MQTT protocol. As the studies in [12], [15] show, if the attacker subscribes to topics with the client ID, the victim encounters denial of service status, and all information sent to the victim is forwarded to the attacker.

Regarding the authentication mechanism, MQTT supports authentication by username and password pairs, but the authentication mechanism is optional and not encrypted. The MQTT client authenticates itself by sending the username and password plaintext in the CONNECT package. Attacker attacks are made quickly by blocking packets [10]. Besides, the MQTT protocol allows any user to subscribe to a broadcast topic without any authentication, and anyone who has it can easily subscribe to any MQTT server available on the Internet [13]. According to a survey of article [16] about Shodan - the world's first IoT search engine for Internet-connected devices shows that there are 67,000 MQTT servers on the public Internet, with most of them without authentication. MQTT does support an authorization mechanism to access a specific topic based on the access list (ACL). This access list must be predefined in the MQTT broker config file[1] and we must restart the MQTT broker service to apply a new access-list configuration. This is inconvenient and difficult to expand, especially for systems with billions of devices, and these devices can only have the right to act for a specified period on specific topics. The problem of access control and authorization is a significant challenge [17].

The security problems in the MQTT protocol are also in the internet connection systems aspect. In particular, due to user behavior, the MQTT's security flaws are generally vulnerable to attack by the malicious users. Users often ignore this aspect, especially privacy, until the loss of critical data [18]. Statistics from [19] show that a significant proportion of the users are not fully aware of where their pieces of information are shared. Hence, with IoT systems having a huge number of users and devices (e.g., a millions), it is quite challenging to manage all user's behavior, especially when users among IoT systems exchange information with each other.

To address the risk of security and availability of MQTT, UIP2SOP manages users, things and channels (topic). This model allows the users to own, use, and exchange messages through ensuring precisely on which channels they are sharing information. To improve the Authentication and Authorization protocol of the MQTT protocol, we propose a combination of MQTT and OAuth protocol by adding a centralized authentication management system (Single Sign-On system). Finally, we use Kafka to build a message queue system that connects discrete IoT systems. To engage further reproducibility or improvement, we share the completely code solution which is publicized on the our Github[2].

[1]https://mosquitto.org/man/mosquitto-conf-5.html
[2]https://github.com/thanhlam2110/uip2sop_platform

The rest of the paper is organized as follows. We provide the background and the related work in the next two sections. Section 4 introduces architecture of UIP2SOP and its prototype system in Section 5. In Section 6, we discuss our evaluation in the different scenarios. Finally, we conclude the key points paper and discuss further work directions.

II. Background

A. MQTT Protocol

MQTT (Message Queue Telemetry Transport) is a messaging protocol in a publish-subscribe model, using low bandwidth and high reliability. MQTT architecture consists of two main components: Broker and Client. MQTT Broker is the central server. It is the intersection point of all the connections coming from the client. The broker's main task is to receive messages from all clients and then forward them to a specific address. Clients are divided into two groups: publisher and subscriber. The publisher is the client that publishes messages on a specific topic. Subscribers are clients that subscribe to one or more topics to receive messages going to these topics.

B. OAuth Protocol and Single Sign-On

Oauth is an authentication mechanism that enables third-party applications to be authorized by the user to access the user's resources located on another application. OAuth version 2, an upgrade of OAuth version 1, is an authentication protocol that allows applications to share a portion of resources without authenticating via username and password as the traditional way. Thereby limiting the hassle of having to enter the username, password in too many places or register too many accounts for many applications that they cannot remember.

According to the OAuth document[3], there are four basic concepts, namely, Resource owners, Resource server, Clients, and Authorization server.

- **Resource owners**: are users who have the ability to grant access, the owner of the resource that the application wants to get.
- **Resource server**: a place to store resources, capable of handling access requests to protected resources.
- **Clients**: are third-party applications that want to access the resource shared by the resource owner, and before accessing, the application needs to receive the user's authorization.
- **Authorization server**: authenticates, checks the information the user sent from there, grants access to the application by generating access tokens. Sometimes the same authorization server is the resource server.

A token is a random code generated by the Authorization server when a request comes from the client. There are two types of tokens, the access token, and the refresh token. The access token is a code used to authenticate access, allowing third-party applications to access user data. This token is sent by the client as a parameter during the request when it is necessary to access the resource in the Resource server. The

[3]https://oauth.net/2/

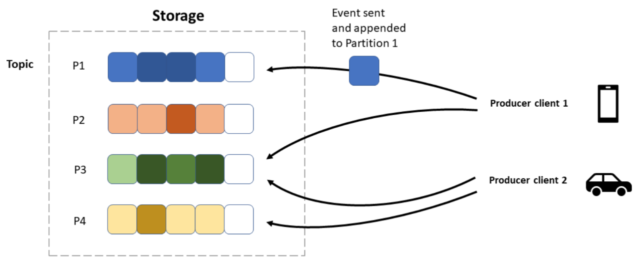


Fig. 1. The Activity Sample of Kafka [21]

access token has a reasonable time (30 minutes, 1 hour). When it expired, the client had to request the Authorization Server to get the new access token. The Authorization server also generates the refresh token simultaneously as the access token but with a different function. The refresh token is used to get the new access token when it expires, so the validity period is more extended than the access token.

Single Sign-On (SSO) is a mechanism that allows users to access multiple applications with just one authentication. SSO simplifies administration by managing user information on a single system instead of multiple separate authentication systems. It makes it easier to manage users when they join or leave an organization [20]. SSO supports many authentication methods such as OAuth, OpenID, SAML, and so on.

C. *Kafka*

Kafka is a distributed messaging system. Kafka is capable of transmitting a large number of messages in real-time. In case the kernel has not received the message, this message is still stored on the message queue and on the disk to ensure safety. Fig. 1 show a sample of Kafka [21].

Kafka includes the four components: producer, consumer, topic, and partition. Kafka producer is a client to publish messages to topics. Data is sent to the partition of the topic stored on the broker. Kafka consumers are clients that subscribe and receive messages from the topic. Group names identify consumers, whereas many consumers can subscribe to the same topic. Data is transmitted in Kafka by topic. Once it is necessary to transmit data for various applications, it can create many different topics. Partition is the data storage on a topic. Each topic can have one or more partitions. For each partition, the data is stored permanently and assigned an ID called offset. Besides, Kafka servers are also called a broker, and the zookeeper is a service to manage the brokers.

III. Related work

A. *OAuth and MQTT*

Paul Fremantle et al. [17] used OAuth to enable access control in the MQTT protocol. The paper results show that IoT clients can fully use OAuth token to authenticate with an MQTT broker. The article demonstrates how to deploy the Web Authorization Tool to create the access token and then embed it in the MQTT client. However, the article does not cover the control of communication channels, so when the properly authenticated MQTT client is able to subscribe to any topic on the MQTT broker, this creates the risk of data disclosure. The paper presents the combined implementation of OAuth and MQTT for internal communication between MQTT broker and MQTT client in the same organization, but not the possibility of applying for inter-organization communication. Therefore, in our article, we implement a strict management mechanism for users, devices, and communication channels. We also present the mechanism of combining MQTT and OAuth protocols to authenticate users when communicating among organizations.

Benjamin Aziz et al. [22] investigated OAuth to manage the registration of users and IoT devices. These papers also introduce the concept of Personal Cloud Middleware (PCM) to perform internal communication between the device and a third-party application on behalf of the user. PCM is an MQTT broker that isolates and operates on a Docker or operating system. Each user has their PCM, and this can help limit data loss. However, Benjamin Aziz et al. also said that they do not have a mechanism for revoking PCM when users are no longer using IoT services, nor have they clearly stated the mechanism to ultimately connect PCMs to form a network for users of various organizations to communicate with each other.

B. *Kafka and MQTT*

A.S. Rozik et al. [21] found that the MQTT broker does not provide any buffering mechanism and cannot be extended. When large amounts of data come from a variety of sources, both of these features are essential. In the Sense Egypt IoT platform, A.S. Rozik et al. have used Kafka as an intermediary system to transport messages between the MQTT broker and the rest of the IoT system, which improves the overall performance of the system as well as provides easy scalability.

Moreover, in the previous studies [23], the authors presented Kafka Message Queue and MQTT broker's combined possibilities in Intelligent Transportation System. The deployment model demonstrates the ability to apply to bridge MQTT with Kafka for low latency and handle messages generated by millions of vehicles. They used MQTT Source Connector to move messages from MQTT topic to Kafka Topic and MQTT Sink connector to move messages from Kafka topic to MQTT topic as shown in Fig. 2.

Lam et al. [24] presented an architecture that combines MQTT broker and kafka message queue to connect different IoT service providers. This architecture allows individual service providers to communicate with each other easily without changing the existing architecture too much. In addition, Lam et al. [25] also evaluates power consumption, transfer speed, communication reliability, and security when using a combination of MQTT broker and kafka message queue. With Kafka's capabilities, we don't need to trade off transmission speed and reliability for power consumption (this is related to QoS-0 and QoS-2 levels).

In the implementation of the IoT Platform, we also adopt and extend this technique by building APIs that allow users to map their topics.

IV. UIP2SOP Platform

The UIP2SOP Platform is a set of APIs combined with system architecture such as Single Sign-On system, Kafka

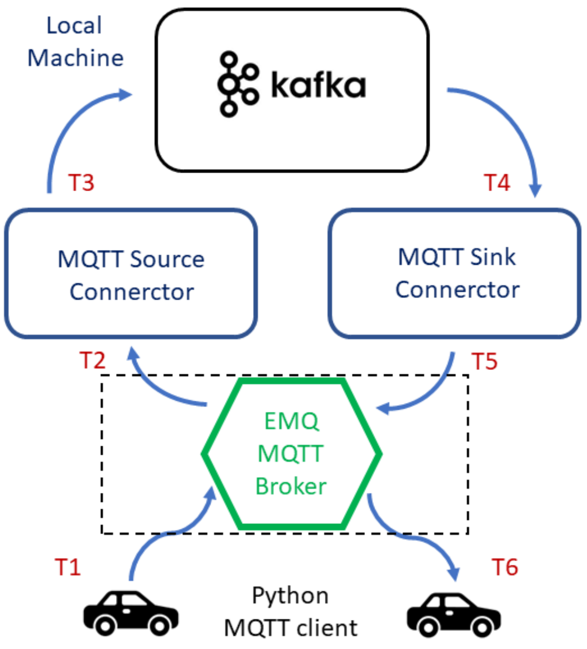


Fig. 2. Communication between MQTT Borker and Kafka [23].

Message Queue, and MQTT broker that provide the following capabilities:

- Authenticate information verifies the user info by granting OAuth access token and refresh token for the user.
- Management information exploits the tree model and the unlimited number of user levels created.
- Management of physical devices/ applications and communication channels participates in an IoT system.
- Sending and receiving messages locally defines a user, a thing, and a specific communication channel.
- Sending and receiving messages globally combines two various organizations through the Kafka message queue.

A. System Architecture

Fig. 3 presents an architectural proposal model of the IoT framework. Let's consider Organization A and Organization B are two separate organizations in the system. Each organization is a set of MQTT brokers that are interconnected to form a cluster MQTT broker. These MQTT brokers play two roles as follows: i) **Internal MQTT broker** is responsible for transporting messages communicating between users, IoT devices; ii) **External MQTT broker** is in charge of transporting messages communicating between two organizations The UIP2SOP Platform system includes the Single Sign-On server and the Single Sign-On service's database cluster to perform the following tasks: i) authenticate user according to OAuth protocol; ii) manage user registration information, channels (public and local), and things. Finally, the UIP2SOP Platform system uses the Kafka Message Queue to transport messages between two organizations' external brokers.

B. UIP2SOP Architecture

To meet the goals set out by the UIP2SOP Framework, we provide several definitions of the components involved in the system and the interaction of these components.

1) Users::
Users participate in an organization and use IoT services provided by that organization. They have two types (corresponding to the parameter field *"usertype"* in the database): a representation user and a normal user.

Each organization has only one representation user that created when an organization registers information of the organization with the UIP2SOP Platform. Representation users are not allow to send or receive messages through the MQTT broker or Kafka message queue. A representation user only creates an organization's public channel and the normal users use this channel to communicate with other organizations in the IoT network. Also, they manage the normal users of the organization.

The purpose in creating a representation user concept is to efficiently manage (e.g., send or receive) messages as well as the organization's (e.g., join or leave) the IoT network. All normal users have to send and receive public messages on the public channel that created and not allowed to create a public channel. Besides, the UIP2SOP Platform manager efficiently manages the entry and exit of an organization's IoT network via the organization's status (i.e., the `userstatus` parameter field in the database). When an organization leaves the IoT network or may be attacked, the UIP2SOP Platform administrator switches the `userstatus` from `ACTIVE` to `DISABLE`, which results in isolating the entire organization from the IoT network, and all normal users and devices within the organization cannot communicate with other organizations but can still communicate internally within the same organization.

Similarly, after the problem is resolved or wants to rejoin the IoT network, through a representation user, the organization can request the manager of the UIP2SOP Platform to change his or her state to `ACTIVE`. By constructing a user hierarchy model tree with the child's `user_parent_id` value equal to the parent user's username, our UIP2SOP Platform allows the creation and management of multiple users' levels and is not restricted depending on the characteristics of the organization. This tree-modeled hierarchy of users makes the UIP2SOP platform more suitable for companies, especially when it comes to authorize a specific user. The user hierarchical management model is shown in Fig. 4.

A normal user registers to use the IoT services of a particular organization. They can create things, channels and assign things to channels to manage which things allowed to send and receive messages on a predefined channels. Each user has a unique `user_id` value, conforming to the UUID[4] standard created by the API and managed by the UIP2SOP Platform (user is not aware of this value). When publishing or subscribing, the user must pass his access token obtained

[4]https://tools.ietf.org/html/rfc4122

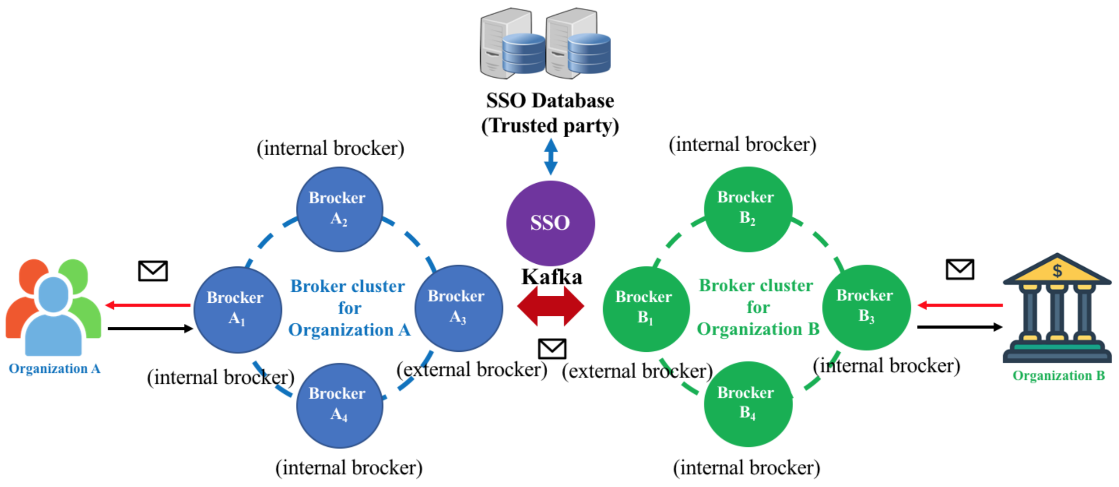


Fig. 3. UIP2SOP Platform.

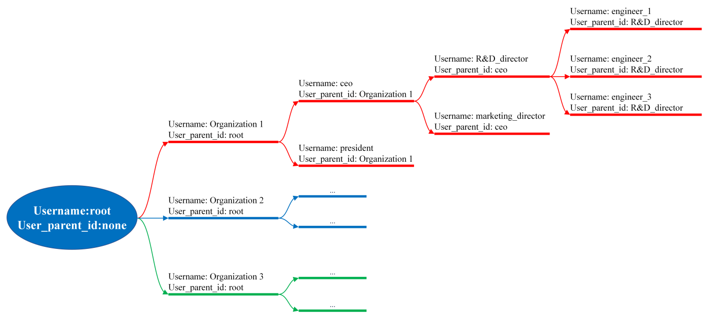


Fig. 4. Management user as a Model Tree.

from the Single Sign-On server, which contains the `user_id` information and is used as the MQTT protocol's `clientID` value. In this way, we have enhanced the authentication and authorization mechanisms for the user and minimize the risk of a denial-of-service attack when a hacker subscribes to a topic with the user's `clientID` affects the availability of the UIP2SOP Platform as we mentioned in Section I.

2) Things:
The things represent the physical devices info or the applications. To create the things, the owners need to call the API provided by the UIP2SOP Platform and pass in his/her valid OAuth token. If the user owns a valid token, the API generates the device's management information. The device's info includes two values `thing_id` and `thing_key`, corresponding to the device just created and return to the user. These two values are unique, created according to the `UUID` standard and used respectively with the username and password values used in the MQTT protocol. In practice, these two values are embedded in the physical device or application. Only the things that own the valid pair of `thing_id` and `thing_key`, can communicate with the MQTT broker since the other ones are not able to publish or subscribe directly to the MQTT broker. Instead, the things must go through the API layer. This layer API validates `thing_id` and `thing_key` sent by the things. Similarly, the user can create device management info to reduce the risk of denial-of-service attacks.

3) Channels and Assign Things to Channels:
In the UIP2SOP Platform, we propose, channels are the logical concept that governs topics where users and things publish and subscribe to messages. There are two types of channels: public channel and local channel.

The local channel is an MQTT topic that is created and managed by a normal user. Each normal user can create one or more local channels. Similarly, when creating a thing, the user who wants to create a channel must call the UIP2SOP Platform's API and pass in his valid Oauth token. If the user owns a valid token, the API generates the management information of the channel. This information includes value `channel_id` corresponding to the channel just created and return to the user. The `channel_id` value is unique, and according to the UUID standard, `channel_id` are returned to the user who created this channel. Each user only has his channel's informations and does not know the channel information of other users. Besides, the user has to assign things to this channels by calling the API and pass in `thing_id`, `channel_id` and his valid OAuth token. The purpose of this process is to only allow a thing with a valid `thing_id` and `thing_key` to publish and subscribe to messages on a predefined channel. From there, this help to avoid the client can subscribe to any topic. This enormously increases the authorization mechanism, which is a flexible way that the original MQTT protocol did not support. The assign things to channels mechanism also enhance security because, through our API, only things are mapped to the channel are allowed to publish and subscribe to messages on this channel.

The public channel is a Kafka topic are created by the representation user. Each organization has a unique public channel. All of the normal users of the organization have to communicate with another organization through the public channel.

4) Publish and Subscribe Message Locally: : After creating the things, the users (owners) have enough information including `channel_id`, `thing_id` and `thing_key` generated by the API layer of UIP2SOP Platform Proposal and returned to the user. Users embed three values `thing_id`, `thing_key` and their refresh token into the things (physical device or application). The process of embedding the above information into a thing is out of this article's scope, so we are not able to elaborate on it here. The things with the necessary information embedded performs the API that provided by the UIP2SOP Platform Proposal to publish the message within the organization.

```
{
    "token":"access token of publisher",
    "thingid":"1960ff8f-ec57-4d81-b6d2-cb1
       7e83016ad",
    "thingkey":"3415e387-1b3b-49aa-8113-40
       799005f3bc",
    "chanelid":"b7cd662c-7d15-4d6b-a8f0-d4
       51e2f368ea",
    "message":"Message local communication
       "
}
```

The information of token, `thing_id`, `thing_key` and `channel_id` are validated by the API services. If all information is correct, the message is sent to the MQTT broker; otherwise, the message is discarded. Similarly, for the Subscribe process, things also send information of token, `thing_id`, `thing_key`, `channel_id` to connect to the MQTT Broker through the API. If the information is not valid, the API are not allow the things to connect to the MQTT Broker.

5) Publish and Subscribe message publicly: : The process of publishing and subscribing to the message is described in Figure 5.

To support the communication among the users in the different organizations, our platform applies a public channel (or Kafka topic). In particular, we assign a local topic (MQTT topic) with a public channel (Kafka topic). For instance, organization A has a public channel with channel ID "95ce1a32-2136-417e-85b4-46b432f1c9ad". A user of organization A, called `''user-a''`, wants to send a message to any user of another organization, called `''user-b''`, he must go through this public channel and perform two steps as follows:

- Step 1: `''user-a''` creates a dedicated local channel to send messages to the public, assuming it is called `''send-public-a"`. This channel is created via the API that creates a local channel, as shown in Section 4.2.3. In fact, the API uses `channel_id`, but for brevity, we cover the channel's name.
- Step 2: `''user-a''` uses the UIP2SOP Platform's API to assign the local channel just created above to the public channel. This process is equivalent create the MQTT Source Connector. After the mapping complete, `''user-a''` publishes the message to the `''send-public-a"` channel. Finally, the message is automatically routed to the public channel. The body structure to call API are as follows:

```
{
''name": ''mqtt-source-for-user-a",
''config": {
   ''connector.class": ''io.confluent.
      connect.mqtt.MqttSourceConnector",
   ''tasks.max": ''1",
   ''name": ''mqtt-source-for-user-a",
   ''mqtt.server.uri": ''tcp://13.212.194.
      253:1883",
   ''mqtt.topics": ''send-public-a",
   ''kafka.topic": ''95ce1a32-2136-417e-85
      b4-46b432f1c9ad",
   ''mqtt.clean.session.enabled": ''true",
   ''mqtt.connect.timeout.seconds": ''30",
   ''mqtt.keepalive.interval.seconds": ''3
      0",
   ''confluent.topic.bootstrap.servers":
      ''localhost:9092",
   ''confluent.topic.replication.factor":
      ''1",
   ''mqtt.qos": ''0".
  }
}
```

The UIP2SOP Platform builds the Kafka consumer service, which receive the message sent by `''user-a''`, check the destination address (defined in body of public message), then forward it to the public channel of the `''user-b''`. At that time, the message is on the public channel (Kafka topic) of `''user-b''`. Therefore, to receive the message, the `''user-b''` must previously create a local channel (MQTT topic), called `''receive-public-b''` and map it to the public channel of `''user-b''`. This is equivalent create an MQTT sink connector. The `''user-b''` uses an API provided by UIP2SOP Platform to create MQTT sink connector. The body structure creates MQTT sink connector is shown follow:

```
{
    ''name": ''receive-public-b",
    ''config": {
        ''connector.class": ''io.confluent
           .connect.mqtt.
           MqttSinkConnector",
        ''tasks.max": ''1",
        ''mqtt.server.uri": ''tcp://172.31
           .46.150:1883",
        ''topics": ''receive-public-b",
        ''mqtt.qos": ''2",
        ''key.converter": ''org.apache.
           kafka.connect.storage.
           StringConverter",
        ''value.converter": ''org.apache.
           kafka.connect.storage.
           StringConverter",
        'confluent.topic.bootstrap.servers
           ": ''localhost:9092",
        ''confluent.topic.replication.
           factor": ''1".
    }
}
```

The body structure when using API public publishing is as follows:

```
{
    ''token": ''access token of publisher"
       ,
    ''source": ''user-a",
```

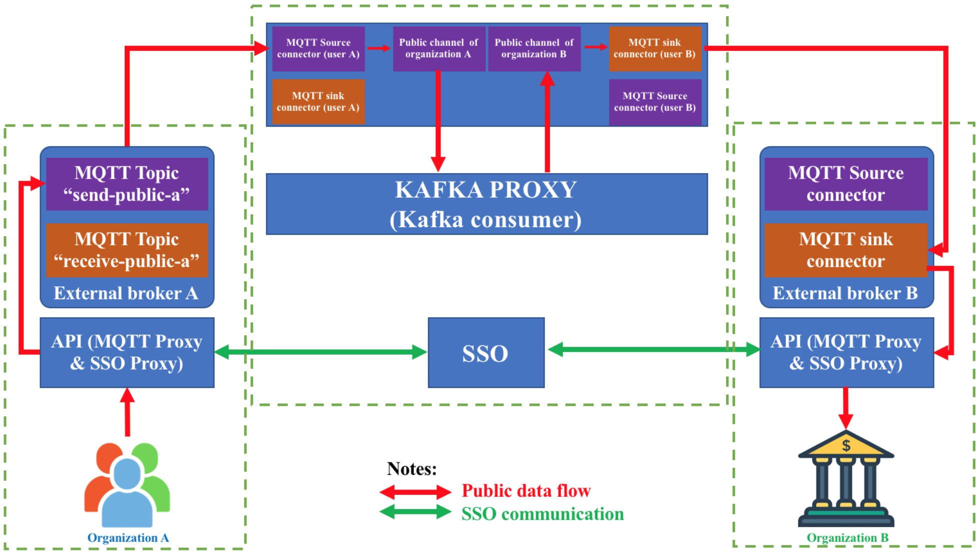

MQTT Source connector (user A)
Public channel of organization A
Public channel of organization B
MQTT sink connector (user B)
MQTT sink connector (user A)
MQTT Source connector (user B)
KAFKA PROXY (Kafka consumer)
MQTT Topic "send-public-a"
MQTT Topic "receive-public-a"
External broker A
API (MQTT Proxy & SSO Proxy)
SSO
MQTT Source connector
MQTT sink connector
External broker B
API (MQTT Proxy & SSO Proxy)
Organization A
Notes:
Public data flow
SSO communication
Organization B

Fig. 5. Process Publish a Message to the Public.

```
    ``destination": ``user-b",
    ``message": ``Hello user-b"
}
```

V. Implementation

To demonstrate the practical implementation of the proposal platform, we have built a prototype of the system. We do this to answer the following research questions:

- Can UIP2SOP Platform be realizable?
- Are there any specific problems with the implementation?
- How does this performance compare to an existing system?

A. Database

As explained in Section V, the IoT Platform allows to manage the information of users, things, channels and implement assign things to channel. In practical implementation, we use MongoDB as a database management system belonging to NoSQL. To realize the model tree (select, update, delete), we have used the Aggregation technique provided by MongoDB[5].

In the prototype system implementation we have seven data collections with the following roles:

Collection of users: store information of normal user and representation user. This collection of users are configured for the Single Sign-On system to read the information and authenticate with the parameters `username` and `password`. Our API layer (SSO proxy) also checks parameter `user_status`, if `user_status` is `ACTIVE` then API return Oauth access token and refresh token for user. We have designed the API that does not allow the users to change his or her `user_status`, only his or her parent user can do this. Thanks to the user management design as tree model, the parent user can quickly change the status of its entire child user's status, which helps to quickly isolate all users when something goes wrong. The parameter `user_parent_id` is used to indicate the parent of the user. In our design the child user has information `user_parent_id` equal the `username` of its parent user.

Collection of things: store information of physical devices or applications managed by a normal user. The API layer (MQTT proxy) checks the parameter `thing_status`, if `thing_status` is `ACTIVE` then the things is allowed to connect to the MQTT broker. User can change the status of things which is managed by the himself. We have also designed an API that allows to change the status of the user resulting in changing the status of all of the user's things.

Collection of local channel: store local channel information (MQTT topic) managed by the normal user. API layer (MQTT proxy) checks the parameter `channel_status`, if `channel_status` is active then the thing is allowed to publish messages to the MQTT broker on this channel. User can change the status of a channel managed by the himself. We have also designed an API that allows to change the status of the user resulting in changing the status of all of the user's channels.

Collection of `things_map_channel`: stores information of thing that has been assigned to channels by a normal user to the channel. API layer (MQTT proxy) checks the parameter `map_status`, if `map_status` is active then the thing is allowed to publish the message to the MQTT broker on this channel. A user can change the status of `things_map_channel`, which is managed by himself. We have also designed an API that allows to change the status of the user resulting in changing the status of all of the user's `things_map_channel`.

Collection `MQTT_source_connector`: stores information MQTT source connector created by normal user. API layer (MQTT proxy) checks parameter `mqtt_source_status`,

[5]https://docs.mongodb.com/manual/reference/operator/aggregation/graph-Lookup/

if `mqtt_source_status` is active then the message can be transfered from MQTT topic to Kafka topic. User can change the status of `MQTT_source_connector` managed by himself. We have also designed an API that allows to change the status of the user resulting in changing the status of all of the user's `MQTT_source_connector`.

Collection `MQTT_sink_connector`: stores the MQTT sink connector information generated by a normal user. API layer (MQTT proxy) check parameter `mqtt_sink_status`, if `mqtt_sink_status` is active then user will receive message from MQTT topic sent by Kafka topic. User can change the status of `MQTT_sink_connector` managed by himself. We have also designed an API that allows to change the status of the user resulting in changing the status of all of the user's `MQTT_sink_connector`.

B. Single Sign-On

In the prototype system, we use the open source CAS Apereo[6] to provide the Single Sign-On service. CAS Apereo supports many protocols for implementing Single Sign-On services such as OAuth, SAML, OpenID, etc. The protocol that UIP2SOP Platform used to communicate with the Single Sign-On server is OAuth. In our implementation, the clients are not allowed interact directly with the CAS server but instead we provide the API for request OAuth token. For example, the API request token has the following body:

```
{
    ``clientid": ``exampleOauthClient",
    ``clientsecret": ``
       exampleOauthClientSecret",
    ``username": ``thanhlam",
    ``password": ``12345678".
}
```

Fig. 6 describes an example of a the User with `DISABLE` status unable to request an `Access token`.

C. Mosquitto MQTT Broker

Mosquitto is an open source to implement MQTT broker that allows to transmit and receive data according to MQTT protocol. Mosquitto is also part of the Eclipse Foundation, the project iot.eclipse.org[7]. Mosquitto is very light and has the advantages of fast data transfer and processing speed, high stability.

D. Prototype System Deployment Model

We deploy prototype system as shown in Fig. 7.

The Prototype system we deployed on Amazon EC2 infrastructure consists of seven servers as shown in Table I.

Please add the following required packages to your document preamble:

In Table I, Organization 1 consists of two servers. The first server deploys the MQTT Broker 1a service for local communication. Second server, deploying MQTT Broket 1b service for public communication. In addition, we deploy MQTT Proxy API service and SSO Proxy to provide APIs for users and devices to communicate with MQTT brokers and SSO through APIs in the first server. The implementation of organization 2 is similar to Organization 1.

Management Central is a single server deploying four services: Kafka, Kafka Proxy, Single Sign-On and database. Single Sign-On service creates access token and refresh token for users according to OAuth protocol. The Kafka service acts as the message queue to transport messages between Organization 1 and Organization 2. The Kafka Proxy service acts as the Kafka consumer to receive messages from Organization 1's public channel and forward it to the Organization's public channel 2. Database service to store information of users, things, channels.

VI. Evaluation

A. Environment

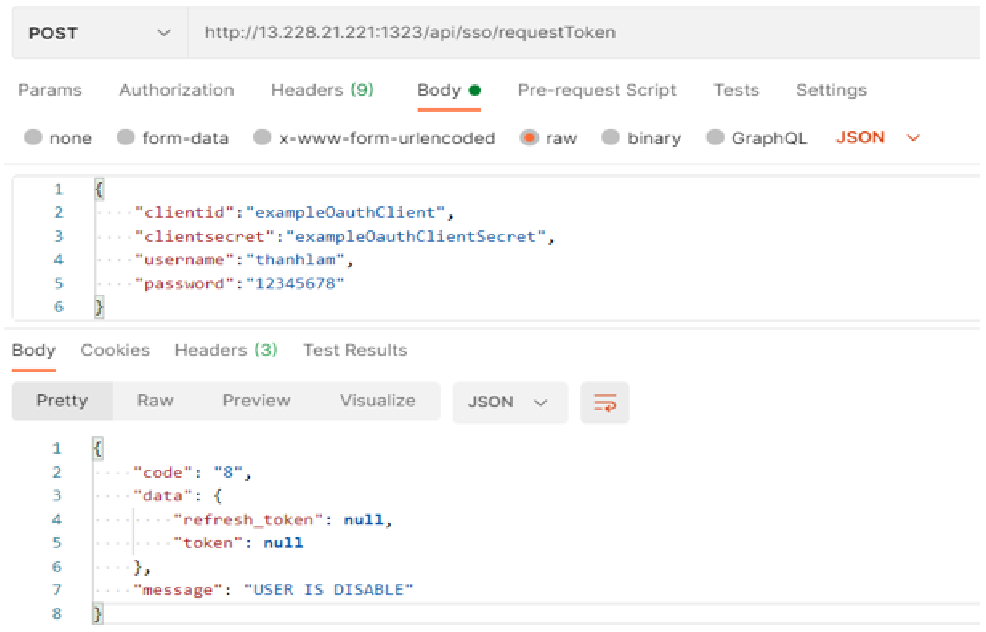
After completing the deployment of the prototype system on the Amazon EC2 infrastructure, we conduct performance test scenarios of the IoT Framework. We check the number of concurent users that can publish messages simultaneously for two cases of internal communication and external communication. We measure the time it takes to create a public channels. The test tool we use is the Apache Jmeter[8].

Apache Jmeter is an open source, written 100% in Java, usable for performance testing on both static resources, dynamic resources and Web applications. It can be used to simulate a large number of virtual users, large requests on a server or a group of servers, a network or an object to test for load capacity or analyze the response time. Apache Jmeter provides the ability to test different applications, servers and protocols such as: Web-HTTP, HTTPs, SOAP / REST Webservice, FTP, LDAP, etc. Since our UIP2SOP Platform Protocol provides API as REST, Jmeter is a perfect fit for system load testing. Firstly, Jmeter makes requests and sends them to the server according to the predefined method, in this case it is REST. Then, it receives responses from the server, collects them and displays information in the report. Jmeter has many report parameters, but when performing system load test, we are mainly interested in two parameters: throughput and error, where the former is the number of requests processed by the server in a second and the latter is the percentage of the number of failed requests over the total number of requests. For our UIP2SOP Platform, the system crashed because the API layer is overload and can't handle request lead to crash API service.

B. Local Communication Test Cases

For the local communication test scenario, we compare the case where the user publishes the message through the MQTT Proxy API, i.e., the authentication process with Single Sign-On, checked by the MQTT Proxy and SSO Proxy, then passed to the local MQTT broker and finally received by the MQTT subscriber with case the user publishes the message directly to the MQTT broker. For two cases in the internal communication test scenario, we use Jmeter to call the API publish message locally then record the number of concurrent users and the

[6]https://apereo.github.io/cas/6.3.x/index.html
[7]https://iot.eclipse.org/
[8]https://jmeter.apache.org/



```
http://13.228.21.221:1323/api/sso/requestToken
```

```
{
    "clientid":"exampleOauthClient",
    "clientsecret":"exampleOauthClientSecret",
    "username":"thanhlam",
    "password":"12345678"
}
```

```
{
    "code": "8",
    "data": {
        "refresh_token": null,
        "token": null
    },
    "message": "USER IS DISABLE"
}
```

Fig. 6. User with `DISABLE` status is not Able to Request an Access Token.

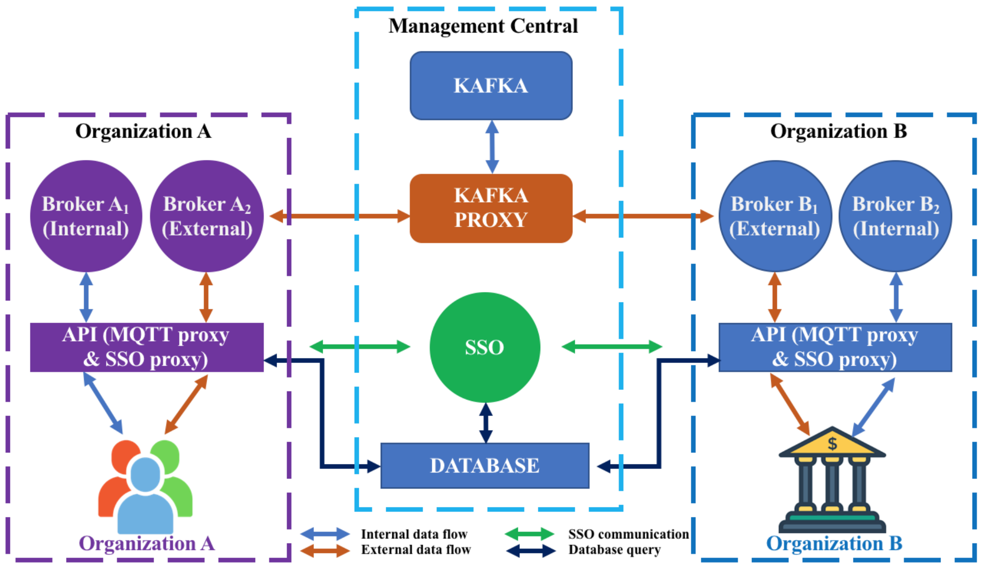


Fig. 7. Prototype System Deployment Model.

TABLE I. List of Infrastructure info in the Prototype

Zone	Server	Role	Server configuration
2*Organization A	Broker 1a	Deploy service MQTT broker for local communication Deploy service API (MQTT Proxy & SSO Proxy)	CPU 1 RAM 1 GB
	Broker 1b	Deploy service MQTT broker to serve public communication	CPU 1 RAM 1 GB
2*Organization B	Broker 2a	Deploy service MQTT broker for local communication Deploy service API (MQTT Proxy & SSO Proxy)	CPU 1 RAM 1 GB
	Broker 2b	Deploy service MQTT broker for public communication	CPU 1 RAM 1 GB
Management Central	Management Central	Deploy Single Sign-On service Deploy the Kafka service Deploy Kafka Proxy service Deploy the Database service	CPU 2 RAM 8 GB

number of requests processed per second. For MQTT Broker we use Mosquitto. The test model for internal communication is shown in the Fig. 8 and the test results are shown in Table II (via UIP2SOP Platform) and Table III (via MQTT Broker).

Through the results obtained, we found that with a server configuration of 1GB RAM and 1 CPU, the UIP2SOP Platform provides the ability to connect 450 concurrent users with a processing speed of 23.9 requests/s without error. Correspondingly, when publishing messages directly to the MQTT broker, Mosquitto can provide the ability to connect 550 concurrent users with a processing speed of 49.6 requests/s. This is reasonable and the result is completely acceptable because our UIP2SOP Platform has added the inspection and authentication mechanisms in Section IV.

C. Public Communication Test Cases

For the test scenario for public communication between organizations, we use Jmeter to call the API publish message publicly then record the number of concurrent users and the number of requests processed per second. Test model of communication between two organizations is shown in Fig. 7 and detail process is shown in Fig. 5. Test results are shown in Table IV.

According to aforementioned results, we found that with a server configuration of 1GB RAM and 1 CPU, the UIP2SOP Platform provides the ability to connect 170 concurrent users with a processing speed of 33 requests/s without error. To the best of our knowledge, we do not find any similar implementation models, so in this section we only record the measured parameters. This test result is reasonable and this result is completely acceptable because when sending messages between two organizations, the UIP2SOP Platform must perform more processing and authentication steps as we have presented in Section IV. The processing speed and the number of concurrent users can be improved by deploying the UIP2SOP Platform on a higher configuration server. In fact, APIs of the UIP2SOP Platform such as MQTT Proxy, SSO Proxy, SSO service and Kafka services should also be deployed as cluster to enhance processing capacity and high availability.

D. Broken Connection Test Cases

In addition, we also have taken test scenario with broken connection between publisher and subscriber. We compare number of receive messages in case with and without using UIP2SOP platform when occured broken connection. The test model is shown in the Fig. 9.

The test result show that, in case without using UIP2SOP platform, subscriber only receive one message - the newest message that publisher send when occurred broken connection. This is the retain function of MQTT protocol. When we enable retain flag, MQTT broker is ability to keep only newest message that publish by publisher. This message is received by subscriber after it reconnects to MQTT broker[9]. However, when using UIP2SOP platform, subcriber can receive all message that published by publisher. This is ability of kafka message queue, therefore the system is guaranteed lost data.

[9]http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

E. Future Work

To develop a larger scenario and increase the number of devices/users authorized quickly, other security issues such as security, privacy, availability for objects are still the challenges. For the security aspect, further works will be deployed in different scenarios like healthcare environment [26], [27], [28], cash on delivery [29], [30]. For the privacy aspect, we will exploit attribute-based access control (ABAC) [31], [32] to manage the authorization process of the IoT Platform via the dynamic policy approach [33], [34], [35]. Finally, we will apply the blockchain benefit to improve the availability issues [36], [37], [38].

VII. Conclusion

The UIP2SOP Platform provides significant improvements in the security capabilities of the MQTT protocol by combining OAuth protocol (Single Sign-On), model of management user, things and channels. The procedure of assigning things to channels, strict management of local and public communication channels helps to minimize careless behavior of users when sharing data. The Kafka message queue system not only easily connects among the organizations providing IoT services to the IoT network without changing too much of the available IoT system architecture of each organization but also reduce lost data when occurred broken connection.

References

[1] T. Alam, "A reliable communication framework and its use in internet of things (iot)," *CSEIT1835111— Received*, vol. 10, pp. 450–456, 2018.

[2] V. Morfino and S. Rampone, "Towards near-real-time intrusion detection for iot devices using supervised learning and apache spark," *Electronics*, vol. 9, no. 3, p. 444, 2020.

[3] H. F. Atlam, A. Alenezi, M. O. Alassafi, and G. Wills, "Blockchain with internet of things: Benefits, challenges, and future directions," *International Journal of Intelligent Systems and Applications*, vol. 10, no. 6, pp. 40–48, 2018.

[4] O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[5] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the internet of things," *Transaction on IoT and Cloud computing*, vol. 3, no. 1, pp. 11–17, 2015.

[6] D. Weissman and A. Jayasumana, "Integrating iot monitoring for security operation center," in *2020 Global Internet of Things Summit (GIoTS)*. IEEE, 2020, pp. 1–6.

[7] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya, "Authorization mechanism for mqtt-based internet of things," in *2016 IEEE International Conference on Communications Workshops (ICC)*. IEEE, 2016, pp. 290–295.

[8] B. Mishra and A. Kertesz, "The use of mqtt in m2m and iot systems: A survey," *IEEE Access*, vol. 8, pp. 201 071–201 086, 2020.

[9] S. P. Jaikar and K. R. Iyer, "A survey of messaging protocols for iot systems," *International Journal of Advanced in Management, Technology and Engineering Sciences*, vol. 8, no. II, pp. 510–514, 2018.

[10] B. H. Çorak, F. Y. Okay, M. Güzel, Ş. Murt, and S. Ozdemir, "Comparative analysis of iot communication protocols," in *2018 International symposium on networks, computers and communications (ISNCC)*. IEEE, 2018, pp. 1–6.

[11] G. C. Hillar, *MQTT Essentials-A lightweight IoT protocol*. Packt Publishing Ltd, 2017.

[12] J. J. Anthraper and J. Kotak, "Security, privacy and forensic concern of mqtt protocol," in *Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur-India*, 2019.

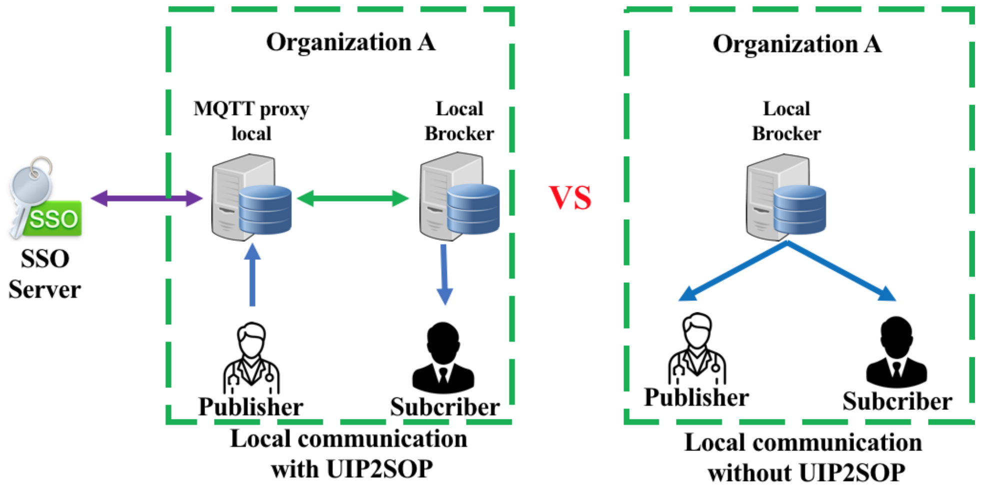


Fig. 8. Test Scenario for the Performance of Local Communication.

TABLE II. Test Results of Internal Communication via UIP2SOP Platform

	50 CCU	150 CCU	250 CCU	350 CCU	450 CCU	550 CCU	650 CCU
Through put (request/s)	14	24.7	26.3	25.5	23.9	21.6	-
Error	0%	0%	0%	0%	0%	5.2%	-

TABLE III. Test Results of Internal Communication via the MQTT Broker

	50 CCU	150 CCU	250 CCU	350 CCU	450 CCU	550 CCU	650 CCU
Through put (request/s)	38	48.3	49.1	50.6	50.1	49.6	45.4
Error	0%	0%	0%	0%	0%	0%	28.3%

TABLE IV. Test Results of External Communication

CCU	25	50	75	100	125	150	175	200
Through put (request/s)	11	20.5	26.8	32.3	34.6	33.9	33	32.1
Error (%)	0	0	0	0	0	0	0	15.3

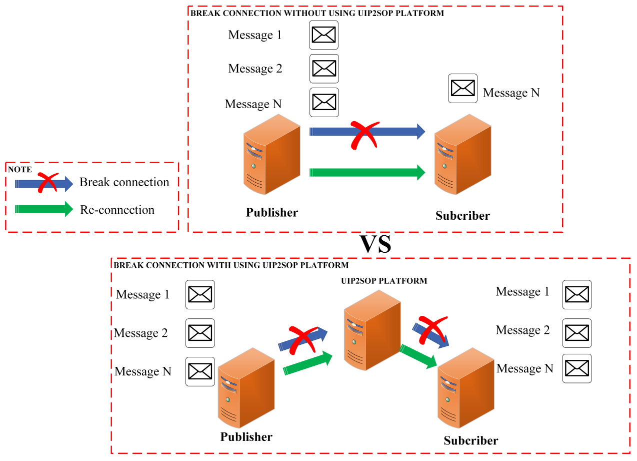


Fig. 9. Number of Receive Messages when System Recover after Broken Connection Issue.

[13] D. Mendez Mena, I. Papapanagiotou, and B. Yang, "Internet of things: Survey on security," *Information Security Journal: A Global Perspective*, vol. 27, no. 3, pp. 162–182, 2018.

[14] L. Lundgren, "Light-weight protocol! serious equipment! critical implications!" *Defcon 24*, 2016.

[15] S. Wang, K. Gomez, K. Sithamparanathan, M. R. Asghar, G. Russello, and P. Zanna, "Mitigating ddos attacks in sdn-based iot networks leveraging secure control and data plane algorithm," *Applied Sciences*, vol. 11, no. 3, p. 929, 2021.

[16] N. Zaidi, H. Kaushik, D. Bablani, R. Bansal, and P. Kumar, "A study of exposure of iot devices in india: Using shodan search engine," in *Information Systems Design and Intelligent Applications*. Springer, 2018, pp. 1044–1053.

[17] P. Fremantle, B. Aziz, J. Kopecký, and P. Scott, "Federated identity and access management for the internet of things," in *2014 International Workshop on Secure Internet of Things*. IEEE, 2014, pp. 10–17.

[18] L. Tawalbeh, F. Muheidat, M. Tawalbeh, M. Quwaider *et al.*, "Iot privacy and security: Challenges and solutions," *Applied Sciences*, vol. 10, no. 12, p. 4102, 2020.

[19] A. Subahi and G. Theodorakopoulos, "Detecting iot user behavior and sensitive information in encrypted iot-app traffic," *Sensors*, vol. 19, no. 21, p. 4777, 2019.

[20] V. Radha and D. H. Reddy, "A survey on single sign-on techniques," *Procedia Technology*, vol. 4, pp. 134–139, 2012.

[21] A. Rozik, A. Tolba, and M. El-Dosuky, "Design and implementation of the sense egypt platform for real-time analysis of iot data streams," *Advances in Internet of Things*, vol. 6, no. 4, pp. 65–91, 2016.

[22] P. Fremantle and B. Aziz, "Oauthing: privacy-enhancing federation for the internet of things," in *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 2016, pp. 1–6.

[23] Å. Hugo, B. Morin, and K. Svantorp, "Bridging mqtt and kafka to support c-its: a feasibility study," in *2020 21st IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2020, pp. 371–376.

[24] L. N. T. Thanh *et al.*, "Toward a unique iot network via single sign-on protocol and message queue," in *International Conference on Computer Information Systems and Industrial Management*. Springer, 2021.

[25] ——, "Toward a security iot platform with high rate transmission and low energy consumption," in *International Conference on Computational Science and its Applications*. Springer, 2021.

[26] H. X. Son and E. Chen, "Towards a fine-grained access control mechanism for privacy protection and policy conflict resolution," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 2, 2019.

[27] N. Duong-Trung, H. X. Son, H. T. Le, and T. T. Phan, "Smart care: Integrating blockchain technology into the design of patient-centered healthcare systems," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, ser. ICCSP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 105–109.

[28] ——, "On components of a patient-centered healthcare system using smart contract," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–35.

[29] H. T. Le, N. T. T. Le, N. N. Phien, and N. Duong-Trung, "Introducing multi shippers mechanism for decentralized cash on delivery system," *money*, vol. 10, no. 6, 2019.

[30] N. T. T. Le, Q. N. Nguyen, N. N. Phien, N. Duong-Trung, T. T. Huynh, T. P. Nguyen, and H. X. Son, "Assuring non-fraudulent transactions in cash on delivery by introducing double smart contracts," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 5, pp. 677–684, 2019.

[31] N. M. Hoang and H. X. Son, "A dynamic solution for fine-grained policy conflict resolution," in *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy*, 2019, pp. 116–120.

[32] H. X. Son and N. M. Hoang, "A novel attribute-based access control system for fine-grained privacy protection," in *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy*, 2019, pp. 76–80.

[33] S. H. Xuan, L. K. Tran, T. K. Dang, and Y. N. Pham, "Rew-xac: an approach to rewriting request for elastic abac enforcement with dynamic policies," in *2016 International Conference on Advanced Computing and Applications (ACOMP)*. IEEE, 2016, pp. 25–31.

[34] Q. N. T. Thi, T. K. Dang, H. L. Van, and H. X. Son, "Using json to specify privacy preserving-enabled attribute-based access control policies," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2017, pp. 561–570.

[35] H. X. Son, T. K. Dang, and F. Massacci, "Rew-smt: a new approach for rewriting xacml request with dynamic big data security policies," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2017, pp. 501–515.

[36] X. S. Ha, H. T. Le, N. Metoui, and N. Duong-Trung, "Dem-cod: Novel access-control-based cash on delivery mechanism for decentralized marketplace," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 71–78.

[37] X. S. Ha, T. H. Le, T. T. Phan, H. H. D. Nguyen, H. K. Vo, and N. Duong-Trung, "Scrutinizing trust and transparency in cash on delivery systems," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2020, pp. 214–227.

[38] H. X. Son, T. H. Le, N. T. T. Quynh, H. N. D. Huy, N. Duong-Trung, and H. H. Luong, "Toward a blockchain-based technology in dealing with emergencies in patient-centered healthcare systems," in *International Conference on Mobile, Secure, and Programmable Networking*. Springer, 2020, pp. 44–56.