2018-01-23

Scoping and Execution Monitoring for IoT Middleware

Fuentes Carranza, Juan Carlos

UNIVERSITY OF CALGARY

Scoping and Execution Monitoring for IoT Middleware

by

Juan Carlos Fuentes Carranza

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 2018

Abstract

Existing Internet of Things architectures rely on middleware (cloud services) to host coordination logic among devices. This middleware is based on Event Based Systems where the Broker architecture and the Publish/Subscribe design pattern are used to deal with heterogeneous environments and for decoupling purposes, being the MQTT protocol one of the most extensively used Event Based Systems for Internet of Things Solutions.

Two prominent security issues in these type middleware are: possible network interruptions between devices and the middleware, and potentially compromised devices. This thesis proposes Scoping and Execution Monitoring in Event Based Systems to cope with possible network disconnections, and to deal with misbehavior of faulty or compromised devices. I define a mathematical model for Event Based Systems where the interplay between Scoping and Execution monitoring is formalized, and empirically evaluate the performance of these security mechanisms.

Acknowledgements

I would first like to thank my supervisor Dr. Philip W.L. Fong. He always steered me to the right direction when I needed it, and his experience and advice during the last two years propelled my interest in research. I thank my parents, for teaching me to always aim for excellence, and who worked tirelessly to give my siblings and I the opportunities they never had. I thank my partner Clara Sanchez, for her constant and unconditional support, and for making me a better person day after day. I thank my siblings, for encouraging me to pursue my dreams, and for always being there when I need them. I am grateful to my teachers and fellow graduate students, to my friends in Calgary, and to my friends at home.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

The Internet of Things is an emerging domain characterized by the ability of devices to connect to the Internet for various purposes, including data aggregation and coordination among them. This concept has been widely adopted in different vertical markets, such as: Home Automation, Connected Cars, Health Care, Manufacturing, etc. The main component of an IoT architecture is the middleware.

The middleware makes available a notification service used by devices to communicate with one another, and hosts their coordination logic (i.e., rules that orchestrate interaction among devices). Additionally, the middleware may perform other tasks such as data aggregation, access control to restrict access to services and data provided by devices, etc. In this work, contributions are made to improve the communication and coordination logic aspects of the middleware.

The notification service in the middleware, is usually implemented as an Event Based System. These type of systems are commonly implemented by a combination of the broker architecture and the Publish/Subscribe design pattern, both of which have been widely used in distributed systems.

The broker architecture abstracts away the communication idiosyncrasy of low-level networking details, so that components in distributed systems connected to heterogeneous networking infrastructures can communicate with one another. Perhaps the most prominent example of a broker architecture is the World Wide Web [13, §2.3], where web browsers (i.e., clients) can communicate with different web servers (i.e., servers) through the combination of Internet gateways and the Internet infrastructure itself (i.e., the broker). In this scenario, clients and servers are not directly connected to the Internet, instead they are con-

nected to local area networks via different networking technologies (e.g., Ethernet, WiFi, 3G, LTE, etc.), and they rely on Internet providers that offer gateways to the Internet.

In the context of the Internet of Things, a wide variety of networking technologies are found. ZigBee, Z-wave, NFC, LPWAN, Thread and BLE are some examples of networking technologies used to enable communication among resource constrained devices. Additionally, IoT devices may also use conventional networking technologies (e.g., Ethernet, WiFi, 3G, LTE, etc.) in order to access the Internet. This clearly shows that heterogeneity is prominent in IoT systems, where massively distributed systems are built using IoT devices. For this reason, it is only natural to use the broker architecture in IoT middleware.

The Publish/Subscribe design pattern is used to decouple parties involved in a communication. This is achieved by the way in which publishers, subscribers and an intermediary entity, namely, the notification system interact with one another. Producers send events to the notification system where the subscribers previously shown their interest to specific events in form of subscriptions. In case a particular event is received by the notification system, and a subscription to this event is found, the notification system will notify the corresponding subscribers about this event. In this sense, a publisher does not address events to specific subscribers, nor does the publisher know their identities. Similarly, subscribers do not know the identity of publishers, since they issue subscriptions to the notification system. In this sense, publishers and subscribers can be added to the system, without major repercussions, which results in high scalability potential.

In contrast, in a typical client-server communication, clients need to know the identity or location of the servers in order to submit their requests. Consequently, every time a new server is added or removed, all clients need to be reconfigured, which represents a problem in terms of scalability.

In the Internet of Things, devices are constantly added or removed from the system, thus scalability considerations have to be taken into account. The decoupling features of

the Publish/Subscribe pattern offer better scalability potential, since devices do not need to learn the identities or locations of other devices.

Prior work on securing the broker architecture and the publish subscribe design pattern focus their attention on confidentiality of events [29], assume publishers to be honest [30], and are based on standard considerations such as access control [12]. However, two security issues that are prominent in the IoT, have not been considered.

Firstly, devices in IoT architectures are usually connected to the middleware via Internet, which represents a single point of failure. Since coordination logic is usually hosted in the middleware, any event that could interrupt communication between devices and the middleware (network device failure, DDoS attacks, etc.) represents a threat for the whole system. Although, a lot of interactions between IoT devices may be dispensable (e.g., if motion is detected by the motion sensor, turn on the lights), some interaction may actually be critical for safety or health purposes. For example, a smoke detector that interacts with a thermostat, in such a way that, if high CO_2 levels are measured by the smoke detector, the thermostat could turn the furnace off, to help prevent toxic CO_2 levels.

Secondly, it is unavoidable for IoT devices to be compromised. In October of 2016, hundreds of thousands of compromised IoT devices were used to execute a DDoS attack on Dyn, a domain name service provider. This attack disrupted access to major Internet services, such as, Twitter, Paypal and Spotify [6]. Another example occurred in a casino in North America, where a smart fish tank was compromised, allowing attackers to gain access to the local network [8]. When devices are compromised, they might cause deviation in coordination logic. For example, a compromised thermostat in a home automation system might report erroneous temperatures, which could trigger actions from other devices (e.g., the furnace could turn down the temperature of the house during winter).

In this research project, two security mechanisms are proposed: scoping and execution monitoring. Scoping is proposed as a countermeasure for network failures, and it refers

to hosting coordination logic in multiple interconnected broker nodes. Devices connected to a broker node create some form of network domain, called a scope, which can be used to delimit and constrain the visibility of events produced or consumed by them. Additionally, scopes act as publisher of internal produced events, and as consumers of outside notifications, which allows for interconnection of multiple scopes.

Each scope is capable of handling communication and coordination among internal devices (and scopes). Additionally, scopes can be distributed in different network areas, such that devices handling critical operations can be connected to a local scope, whereas another scope, hosted in the Internet, can be used for interactions between a broader group of devices. This way, even in the case that communication to the Internet is interrupted, critical operations may still persist.

Scoping was originally proposed by Fiege *et al.* [17, 19, 18], where it is used as mechanism to facilitate engineering and coordinations of components in event based systems. In their work, Fiege *et al.* define visibility of events in scopes in terms of a fixed visibility policy based on shared ancestors. In this thesis, scoping is considered from a security perspective, and allows for more tailored visibility policies.

The second security mechanism proposed in this work is execution monitoring. Execution monitoring is an enforcement mechanism that works by monitoring execution steps of some untrusted system. In [28], Schneider proposes execution monitoring to monitor execution of programs defined as a sequence of actions. In his work, execution monitors interpose themselves between the program being monitored, and the platform running the program. These execution monitors enforce security policies by terminating the program upon detecting a sequence of actions that violates them. An extension to Schneider's work is proposed by Ligatti *et al.* [23], where the execution monitors are not only capable of terminating programs, but also of modifying the stream of events that the monitored program sends to the underlying platform at run time.

In the context of a traditional broker architecture, an execution monitors can be used to monitor sequence of events propagated through one communication channel (e.g., from a device to the middleware). This enables individual components of a distributed system to be monitored, even if they cannot be controlled directly. Furthermore, by integrating scoping and execution monitoring, execution monitors can additionally be used to monitor communication channels between different groups of components (e.g., between two scopes).

In this work, I demonstrate how the combination of scoping and execution monitoring can be used to implement security policies to prevent leakage of sensitive information and, execution of potentially dangerous behavior.

The specific contributions of this thesis are the followings:

1. I formulated a model for Event Based Systems (Chapter 3), that incorporates Scoping an Execution Monitoring. This model allows the configuration of Visibility Control rules of scopes by including brokering policies.

2. I demonstrated how the two protection mechanisms, scoping with brokering policies, and execution monitoring, can be leverage to impose various forms of visibility control (Section 3.4).

3. I conducted a case study where I demonstrate the use of scoping and execution monitoring to enforce security policies (Section 3.5).

4. I extended Mosquitto [25], an open source implementation of the MQTT protocol [11], to support Scoping and Execution Monitoring (Chapter 4).

5. I mechanized the proposed model using PLT Redex [4] (Section 4.1).

6. I evaluated the performance of the extended version of Mosquito in terms of Message Throughput (Chapter 5)

Chapter 2

Background

In this chapter I introduce the concepts used as the basis of this thesis. Firstly, §2.1 introduces the general structure of an IoT architecture, in terms of its main components: Middleware, IoT Devices and Gateways. Then, in §2.2, I provide a description of Event Based Systems, which are based in the Event-Based cooperations model [19]. In §2.3 an overview of the MQTT protocol is given, which is one of the most extensively used protocols used in the Internet of Things. Finally, §2.4 presents the definition of an edit automaton, which is used in this research project to realize Execution Monitoring.

2.1 IoT Architecture

The Internet of Things (IoT) is the concept of connecting any type of device to the Internet. This includes cellphones, washing machines, thermostats, wearable devices, industrial machinery, and pretty much anything we can think of. This concept has caught the attention of numerous companies who are trying to come up with their own solutions for the different vertical markets of IoT such as: Home Automation, Automotive, Healthcare, Manufacturing, etc.

In the particular case of Home Automation systems, some of the most well known companies who have released products and services related to this market, including Google (Nest), Samsung (SmartThings) and Apple (HomeKit). Although their solutions are different, they all involve the cooperation between the following components: Middleware, IoT Devices, and Gateways.

Figure 2.1 illustrates the main components of an IoT architecture, and their connectivity. In the following sections, a description of each of these components is given.

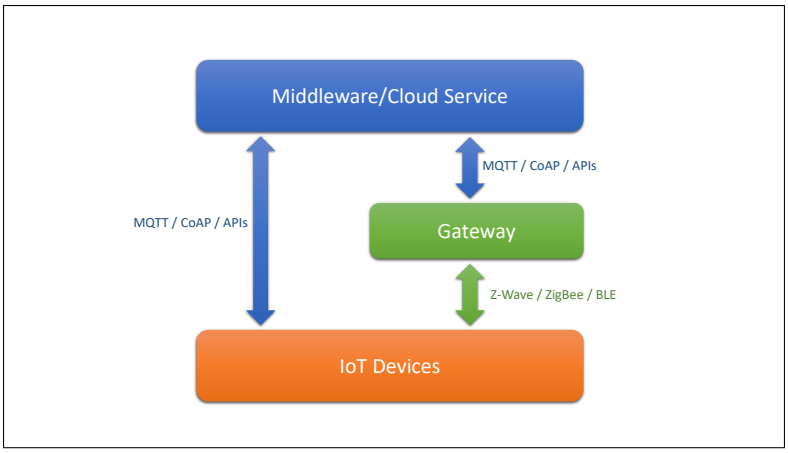

Figure 2.1: IoT Architecture

2.1.1 Middleware

The middleware is implemented as an Event Based System, with a combination of the Broker architectural style and the Publish/Subscribe design pattern [13], to host the cooperation logic among their devices. The Broker architectural style abstracts away the network idiosyncrasy of low-level network protocols such as ZigBee, Z-Wave, Thread, WiFi, etc. This enables devices to communicate to one another seamlessly by relaying information through a Broker. At the same time, this Broker makes available a notification system, that propagates information in the form of events from one device (the producer) to the rest of the devices (the consumers), so that they can react appropriately to these events.

The notification system is usually implemented by the Publish/Subscribe design pattern, due to its high decoupling features. Decoupling refers to the independence between publishers and subscribers, and is achieved by the way in which publishers, subscribers and the notification system interact with one another. Publishers send events to the notification system where the subscribers previously shown their interest to specific events in form of subscriptions. In case a particular event is received by the notification system, and a subscription to this event is found, the notification system will notify the corresponding subscribers about this event. In this sense, publisher does not address events to specific subscribers, nor does the publisher know the identities of the subscribers. Subscribers do not

know the identity of publishers, since they issue subscriptions to the notification system.

In IoT systems, the Publish/Subscribe design pattern is needed because otherwise communicating parties would need to know the existence (i.e., the network address or the identifier) of each other. Thus, every time a device was added or removed, all other devices would need to be reconfigured. With the Publish/Subscribe design pattern, communicating parties do not need to know of the existence of one another, which facilitates the addition and removal of devices. This feature offers high scalability potential to the systems.

2.1.2 IoT Devices

The term IoT Device refers to any type of device such as thermostats, light bulbs, door locks, medical equipment, industrial machinery, etc. that is capable to connect to the Internet. Devices are categorized in Sensors and Actuators:

Sensors. This type of devices gather information about their environment, some examples of sensors are: motion sensors and thermostats.

Actuators. This type of devices are used to execute actions, some examples of actuators are: door locks and light bulbs.

Although many devices are capable of connecting to the Internet by themselves, sometimes the resources available to some of them are very limited. For example, a battery powered motion sensor, with a limited supply of energy, cannot be connected to the Internet through common mechanisms such as WiFi, which are very power hungry. Instead, in constrained devices, different type of network protocols are implemented, which enable them to efficiently use their limited resources. Some examples of such protocols are: ZigBee, Z-Wave, Thread and BLE.

Addressee	Initiated by Consumer	Initiated by Producer
Direct	Request/Reply	Callback
Indirect	Anonymous Request/Reply	Event-Based

Table 2.1: Cooperation Models

2.1.3 Gateways

Network protocols used by constrained devices, are not capable to communicate directly with the Internet. In order to enable constrained devices to communicate with the middleware, gateways are used, which are devices that acts as interpreters between two or more different network protocols.

Although communication between gateways and the middleware is done via Internet, the use of an application level protocol is necessary. Two of the most popular standard application layer protocols used for the Internet of Things are: MQTT and CoAP. However, some companies prefer using proprietary API for this type of communications.

2.2 Event Based Systems

Fiege *et al.* [20] define an event based system as a system in which its components communicate by generating/receiving event notifications, where an event notification is a data representation that describes an occurrence of a particular event of interest, and components can be either consumers and/or producers. In their work, Fiege *et al.* propose a taxonomy of cooperation models, based on two important characteristics: 1) who is the initiator of the communication (the consumer or the producer), and 2) whether the addressee is known or unknown. This taxonomy distinguishes four cooperation models: Request/reply, Anonymous request/reply, Callback and Event-Based (see Table 2.1).

Request/reply: In this cooperation model, the consumer initiates the cooperation by requesting data and/or functionality from the provider, and in return, it expects

data and/or an specific task to be done. This request is directly addressed to the provider, and its identity is known. Additionally, in this cooperation model replies are mandatory.

Anonymous request/reply: In this cooperation model, consumers initiate the cooperation by requesting data and/or functionality. However, requests are not addressed to specific providers. Instead, requests are delivered to an arbitrary (possibly dynamically determined) set of recipients, and the identity of the recipient(s) is not know a priory by the consumer. Additionally, in this cooperation model one request may yield multiple responses.

Callback: In the callback cooperation model, consumers register at specific known providers to be notified whenever some condition is met. The provider is responsible of 1) maintaining the list of registered consumers, and 2) constantly evaluating if the notifications conditions are met, and notify registered consumers if necessary. In this sense, producers initiate the communication when the a notification condition is met, however, identities of the consumers are known a priori.

Event-Based: In the event based cooperation model, producers are the initiators of the communication. Producers generate notifications addressed to no particular recipient(s), whereas consumers express their interest on specific events in the form of subscriptions. Providers are not aware of the consumers, which relieves them from the task of maintaining a list of subscribers. Instead, all dependencies and coordination between providers and consumers are handled by an external entity (e.g., Broker). In this cooperation model, components are "self-focused" in the sense that they know how to react to input notifications, and publish updates about their own state, but never publish a notification with the intention of triggering an action.

Anonymous request/reply and event-based cooperation model are often confused, since both cooperation models achieve anonymity between producers and consumers. However, in the anonymous request reply cooperation model, the initiator of the communication (i.e., the consumer) expects data and/or functionality as a result of its request, which results in dependency on external provided data. In the event-based cooperation model, the initiator of the communication (i.e., the producer) does not publish information with the intention of triggering actions and/or receiving data.

In an event based system, its components make use of the event-based cooperation model which offers the highest level of decoupling between producers and consumers. This type of systems are usually implemented by the Publish/Subscribe design pattern.

2.3 The MQTT protocol

MQTT [11] is a messaging transport protocol that makes use of the Publish/Subscribe design pattern to provide one-to-many message distribution and decoupling features. In this design pattern subscribers express their interest on specific events in forms of subscriptions, and subscribers are the source of these events. According to [15], there exits three different types of subscriptions systems: Topic-Based, Content-Based and Type-Based.

The MQTT protocol subscription system is of type Topic-Based, where subscription on events is based on Topic Names. Topic Names in MQTT are UTF-8 encoded strings as specified in [31], excluding encoding of code points between U+D800 and U+DFFF. One special character used in topic names is the topic level separator (forward slash ”/”). This character is used to add structure to the Topic Names and divides them into multiple levels. However, the use of topic level separators is not mandatory. One example Topic Name could be “smoke-detector/co-level”.

In MQTT, publishers publish events using Topic Names, whereas subscribers express their interest on specific topics by using Topic Filters. A Topic Filter is just a regular Topic

Name as described before, with the (non mandatory) addition of special characters known as wildcards. Wildcards, allow subscribers to subcribe to multiple topics at once. The two different wildcards as defined in [11] are:

Multi Level Wildcard. The number sign (#) is used to match an arbitrary number of levels in a topic. For example, if a client (subscriber) subscribes to the Topic Filter ”smoke-detector/#”, it would receive messages published under the following Topic Names:

- smoke-detector/info/status
- smoke-detector/action/target-state
- smoke-detector/version

When only this wildcard is used as Topic Filter, it implies a subscription to all Topic Names.

Single Level Wildcard. The plus sign (+) is used to match a single level in a topic. For example, if a client (subscriber) subscribes to the Topic Filter ”smote-detector/+”, it would receive messages published under the following Topic Names:

- smoke-detector/
- smoke-detector/version

On the other hand, this client would not receive messages published under the following Topic Names:

- smoke-detector/info/status
- smoke-detector/action/target-state

Topic Names and Topic filters are mainly used to distinguish what event a certain subscriber should be notified about. However, an event in the MQTT protocol not only consists

Control Packet Name	Value	Description
Reserver	0	Reserved
CONNECT	1	Client request to connect to the Server
CONNACK	2	Connect acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish acknowlegment
PUBREC	5	Publish received
PUBREL	6	Publish release
PUBCOMP	7	Publish complete
SUBSCRIBE	8	Client subscribe request
SUBACK	9	Subscribe acknowlegment
UNSUBSCRIBE	10	Unsubscribe request
UNSUBACK	11	Unsubscribe acknowledgment
PINGREQ	12	PING request
PINGRESP	13	PING response
DISCONNECT	14	Client is disconnecting
Reserved	15	Reserved

Table 2.2: MQTT Control Packet Types

of a Topic Name, but it also has an associated payload. The payload is used to carry the actual application information to be used by the subscribers, and its data type is application dependent.

The MQTT protocol works by exchanging messages called control packets, where sixteen different control packets types are defined and are represented as a 4-bit unsigned value. Table 2.2 shows a list of these control packets. Finally, the MQTT protocol defines two different type of entities: MQTT Server (Broker), and MQTT Clients (Publishers and/or Subscribers). The Server is on charge of maintaining the list of subscriptions, receive event from the publishers, match event against subscriptions and send the corresponding notifications. Clients simply send and/or receive events.

2.3.1 Quality of Service levels

In chapter 4, the implementation of scoping and execution monitoring on top of an open source implementation of the MQTT protocol is described. To fully appreciate this implementation, particularly §4.2.2 and §4.3.3, it is paramount to understand in detail the Quality

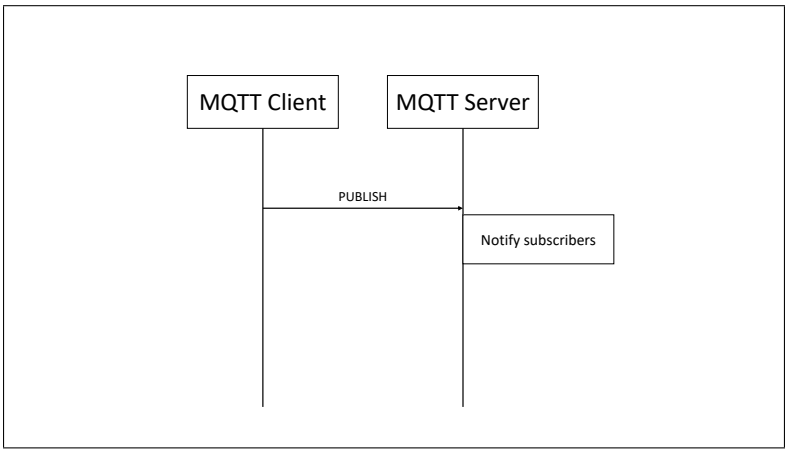

Figure 2.2: MQTT - QoS 0 - At most once delivery

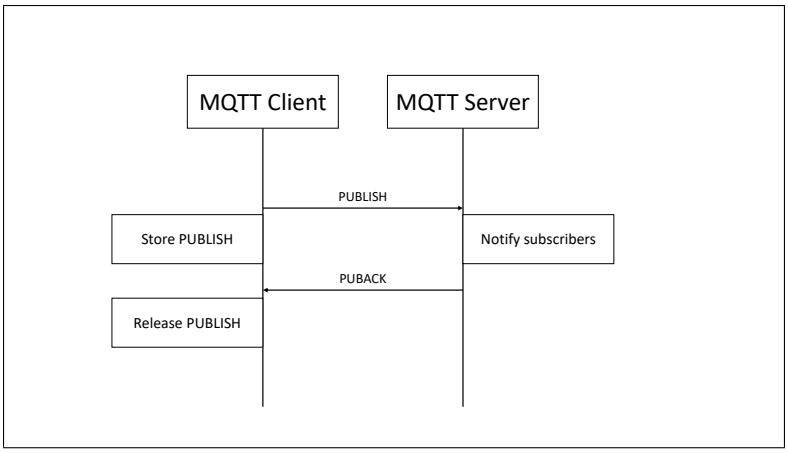

Figure 2.3: MQTT - QoS 1 - At least once delivery

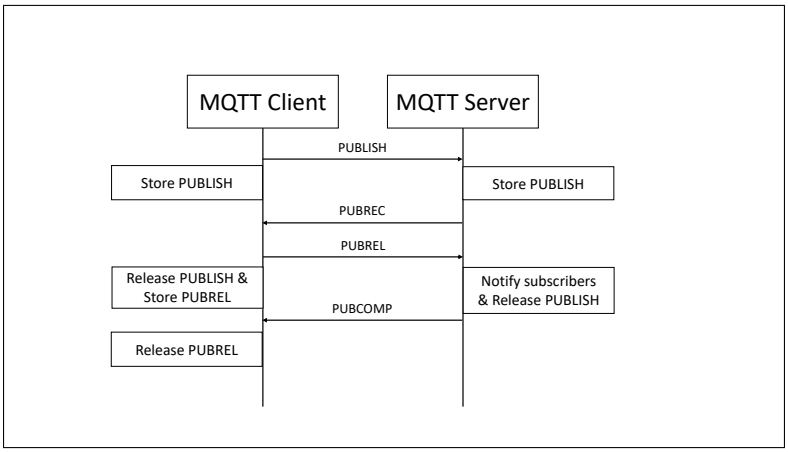

Figure 2.4: MQTT - QoS 2 - Exactly once delivery

of Service levels defined by the MQTT protocol.

In the MQTT protocol, three Quality of Service (QoS) levels related to the delivery of events from publishers to subscribers are described. Illustrative examples of these QoS levels are shown in Figures 2.2, 2.3, and 2.4, where the sender is assumed to be a MQTT Client, and the receiver is assumed to be a MQTT Server.

QoS 0: At most once delivery. In this QoS level, the delivery of an event is not guaranteed, and it depends on the capabilities of the underlying network. In this level no response is expected from the receiver, and the event arrives once or not at all. An example of this QoS level is shown in Figure 2.2, where:

- The sender:
  - Sends a PUBLISH packet.
- The receiver:
  - Accepts the message and notifies subscribers with matching subscriptions.

QoS 1: At least once delivery. This quality of service level guarantees the event to be delivered to the receiver at least once. However, the receiver could receive multiple copies of the same event. An example of this QoS level is shown in Figure 2.3, where:

- The sender:
  - Sends a PUBLISH packet with a packet identifier.
  - Stores the PUBLISH packet as until it receives the corresponding PUBACK packet from the receiver with the same packet id.

- The receiver:
  - Accepts the message and notifies subscribers with matching subscriptions.
  - Responds the PUBLISH packet with a PUBACK packet with the same packet identifier.

QoS 2: Exactly once delivery. This quality of service level guarantees events to be delivered to receivers exactly once (loss or duplication events is unacceptable). An example of this QoS level is shown in Figure 2.4, where:

- The sender:
  - Sends a PUBLISH packet with a packet identifier.
  - Stores PUBLISH packet until it receives the corresponding PUBREC packet from the receiver with the same packet id.
  - Responds the PUBREC packet with a PUBREL packet using the same packet identifier than the original PUBLISH packet.
  - Stores the PUBREL packet until it receives the corresponding PUBCOMP packet from the receiver with the same packet id.
- The receiver:
  - Responds the PUBLISH packet with a PUBREC packet with the same packet identifier.
  - Stores the PUBLISH packet until it receives the corresponding PUBREL packet.

– Responds the PUBREL packet with a PUBCOMP packet with the same packet identifier than the original PUBLISH packet and notifies subscribers with matching subscription.

2.4 Edit Automata

Security automata have been widely used for monitoring the execution of programs. In [28], Schneider proposes the notion of security automaton, which is capable of interposing itself between an untrusted program and the platform on which it runs. This automaton is used to enforce security policies and runs in parallel with the monitored program, analyzing the sequence of actions the program executes. If the monitor recognizes a sequence that violates its policy, it terminates the program. Ligatti *et al.* [24] [23] propose an extension to the security automaton, called ***edit automata***, capable not only of terminating a program in case of a violation, but also of modifying the stream of events that the monitored program sends to the underlying platform at run time.

A system (program) is specified via a set of actions (or events) $\mathcal{A}$, and an *execution* σ is a finite sequence of actions. The set of all finite-length sequences of actions in the system is denoted by $\mathcal{A}^*$, having σ and τ as typical members of this set. The concatenation of two sequences is denoted by $\tau;\sigma$, and ε represents the empty sequence.

An edit automaton is described as a triple of the form (Q, q_0, δ), where Q is the possibly countably infinite set of states of the automaton, and q_0 represents its initial state, such that $q_0 \in Q$. The single-step transition relation in the automaton is denoted by $(q, \sigma) \xrightarrow{\tau}_E (q', \sigma')$, where σ represents the sequence of actions the monitored program wants to execute, q is the current state of the automaton, σ' and q' represents the sequence of actions and the new state of the automaton after it takes a single step. The input sequence σ is not observable to the rest of the world (i.e., it is only visible to the edit automata), and τ is an

observable sequence of at most one action observable to the rest of the world.

The total transition function δ of an edit automaton has the form $\delta : Q \times \mathcal{A} \to Q \times (\mathcal{A} \cup \{\varepsilon\})$. Given a current state q and an input action a, if $\delta(q,a) = (q',a')$, it specifies the next state q' of the automaton, and the action a' to be inserted into the output stream (i.e., a' is made observable) without consuming the input action a. On the other hand, if $\delta(q,a) = (q',\varepsilon)$, it represents that a should be suppressed, that is, a is consumed without being made observable.

Formally, given a single step $(q,\sigma) \xrightarrow{\tau}_E (q',\sigma')$ for the edit automata, and having a and a' representing typical members of $\mathcal{A}$, the transition relation is defined by the following set of transition rules:

- **E-Ins**. *Insert an action into the output stream.*
  - **Precondition:** $\sigma = a;\sigma' \wedge \delta(q,a) = (q',a')$
  - **Effect:** $(q,\sigma) \xrightarrow{a'}_E (q',\sigma)$
  - **Description:** To insert an action into the output stream, two conditions have to be met:

    1. a is the first action in the execution σ.
    2. Given the current state q of the automaton, $\delta(q,a) = (q',a')$ is defined, where q' is the next state of the automaton, and a' is the action to be inserted.

    Consequently, a' becomes observable to the world, and σ represents the remaining sequence of actions the monitored program wants to execute, which means that the input action a is not consumed.
- **E-Sup**. *Suppress one input action.*
  - **Precondition:** $\sigma = a;\sigma' \wedge \delta(q,a) = (q',\varepsilon)$
  - **Effect:** $(q,\sigma) \xrightarrow{\varepsilon}_E (q,\sigma')$
  - **Description:** An input action is suppressed, when the following conditions are met:

1. a is the first action in the execution σ.
2. Given the current state q of the automaton, $\delta(q,a) = (q',\varepsilon)$ is defined, this represents that the input action a should be suppressed.

Consequently, a is consumed from the input sequence of actions σ without being made observable, and σ' represents the remaining sequence of actions the monitored program wants to execute.

Edit automata is the cornerstone of execution monitoring as defined in this thesis, although a slightly different characterization is used in this work. In the next chapter, the formalization of the mathematical model for event based systems is given, where edit automata is revisited in §3.3.2.

My contributions to the security of IoT middleware are contained in the three following chapters. Chapter 3 introduces a mathematical model for event based systems, where the interplay between scoping and execution monitoring is formalized. Chapter 4 describes two implementations conducted during the research project. Finally, chapter 5 contains the results of the performance evaluation of the security mechanisms, based on one of the implementations described in chapter 4.

Chapter 3

Event Based Systems Model

This chapter introduces a highly configurable mathematical model for Event Based Systems. The model defines the interplay between both security mechanisms: execution monitoring and scoping. Execution monitoring is used to monitor the behavior of potential malfunctioning or compromised devices, and scoping enables the system administrator to tailor events propagation according to her needs.

This chapter is organized as follows. In §3.1 a description of the Threat Model and Security Assumptions of the research is given. §3.3 describes the operational semantics of the model. §3.4 illustrates how it is possible to configure the model to enforce different types of visibility control.

3.1 Threat Model and Security Assumptions

As described in [19], the event-based cooperation model involves three different types of entities: producers, consumers and a notification service. Publishers initiate the communication by sending information to the notification service in form of events, however these events are not addressed to any particular recipient (or set of recipients), instead, the notification service is in charge of relaying events from producers to consumers based on subscriptions. Consumers express their interest on specific events by issuing subscriptions in the notification service, and when a published event matches one consumer's subscription, the service system relays this event to the interested consumers. Since notifications are not directed to particular consumers, a consumer may receive events from many publishers. Additionally, these entities are connected to one another through some form of network connection.

In the context of Internet of Things, devices may act as publishers, or subscribers, or both. The notification services is made available by the middleware, and devices are connected to the middleware through some form of network connection. In the following paragraphs, the threat model and the security assumptions in which this research is based are described in terms of the aforementioned entities.

Middleware. The middleware is assumed to be trustworthy. All events relayed by the notification service embedded in the middleware are assumed to be integral.

Devices. Devices are corruptible. This corruption affects the publication of events, thus, events published by corrupted devices should not be trusted. On the other hand, subscriptions are registered on the middleware, which is trusted. Once registered, subscriptions are not affected by the corruption of devices.

Network Connection. It is also assumed that network connection devices and the middleware may fail (due to DDoS attacks, a faulty network device, etc.).

The goal of this research project is to 1) offer the ability to detect and react appropriately to potentially malicious events, and 2) preserve critical operations among devices in the event of an interruption of a network connection.

3.2 Security Mechanisms

In order to deal with the threat mode described in §3.1, a number of security mechanisms are incorporated in the model. In the following paragraphs, a high level description of such security mechanisms is provided.

Scoping. To respond gracefully to possible network failures, a notification service based on a network of interconnected scopes is proposed. Scopes can be seen as independent brokering nodes capable of handling communication among

smaller groups of devices. To maintain communication between devices in charge of critical operations (e.g., safety related devices), these devices could belong to the same scope. This particular scope could be hosted in the local area network with respect to its corresponding connected devices. A second scope, connected to the former scope, and hosted in a different network area (e.g., Internet) could be responsible of handling supportive interactions among a broader group of devices. This way, even if communication between the two scopes is interrupted (e.g., Internet connection is lost), all critical operations hosted in the local area network will persist.

The term scoping is inherited from previous work of Fiege *et al.* [17, 19, 18]. In their work, Fiege *et al.* use scopes in event based systems for engineering purposes, where clients (i.e., publishers and subscribers) and scopes are organized in a hierarchical structure. The visibility of events published in one scope is then delimited by the configuration of the hierarchical structure, in a similar fashion than the visibility of variables in a statically scoped programming languages is delimited by the scope in which they are declared. In this work, scoping is adopted not only as a security mechanism to counter possible network disconnections, but also to delimit the visibility of events, feature that can be used for confidentiality and integrity purposes.

Execution Monitoring. Execution monitoring is a special application of execution monitors [28]. It can be used to detect behavioral anomalies on publications and notification of events in order to react appropriately. Additionally, using Execution Monitoring, it is possible to induce two popular features in Event Based Systems: Event Filtering and Event Mapping.

3.3 Modelling Distributed Event Systems

In an event based system, publishers, subscribers and the notification service interact with one another by:

1. Issuing subscriptions to express interest on specific events (subscribe).
2. Canceling subscriptions (unsubscribe).
3. Transferring events from publishers to the notification service, and from the notification service to subscribers (transmit).
4. Queuing up notifications for all subscribers who issued a corresponding subscription (broker).

I categorized these interactions into two groups. The first group is composed of the subscribe and unsubscribe operations, and is called **Administrative tasks**. The second group is composed of the transmit and broker operations, and is called **Event transmission tasks**.

In this research, it is assumed that subscriptions are previously configured by the system administrator, as such, the model only takes into account transitions caused by **Event transmissions tasks**.

3.3.1 Preliminaries

Suppose $R \subseteq S \times S$ is a binary relation over S. Then $R(a,b)$ is written to assert that $(a,b) \in R$. The power set of S is written as 2^S.

3.3.2 Execution Monitoring

In this work, an ***edit automaton (EA)*** is defined as a quadruple $\langle \Sigma, Q, q_0, \delta \rangle$, such that Σ is a finite set of symbols, Q is a countable set of states, $q_0 \in Q$ is the initial state, and

$\delta : Q \times \Sigma \to Q \times \Sigma^*$ is the transition function. Given a current state q and an input symbol a, $\delta(q,a)$ is a pair (q',w), where q' is the next state, and w is a sequence of output symbols generated by the transition. If the output sequence is the empty string (ε), then the input event is "suppressed.", otherwise the output sequence is inserted into the output stream, making it observable to the world. Throughout this chapter, a, b and c denote typical members of Σ, and u, v and w denote typical members of Σ^*.

In §2.4, the original characterization of edit automata [24] was introduced, where the single-step transition of the autoamata is defined by two transition rules: E-Ins and E-Sup. E-Ins is used to insert an event into the output stream (i.e., make it visible) without consuming the input event, and E-Sup is used to consume and suppress the input event without making it observable to the world. One natural question to ask is whether or not both characterizations are equivalent. In preparation for this discussion, consider *EA* to be used to refer to the original edit automata definition, and EA^* is used to refer to the characterization introduced in this section.

Observe that *EA* is simply a special case of EA^*, where the output sequence of events is composed of at most one event. On the other hand, in EA, one input event in the automaton can be either suppressed without inserting a single event into the output stream, or it can be used to generate an arbitrary long sequence of events, where each event in the generated sequence requires one state transition in the automaton. This means that both characterizations of edit automata can be used to suppress events, or to generate sequence of events, with the only difference being the number of transitions required for the second task by each of them. In this sense, both *EA* and EA^* are equivalent. In this work, the characterization of EA^* is preferred because of the possibility of generating sequence of events on real time.

3.3.3 Ontology

A ***system schema (or simply a schema)*** χ is a quintuple $\langle \mathcal{CG}, \mathcal{EP}, \mathcal{BP}, sub, \sqsubseteq \rangle$:

- $\mathcal{CG}$ is a ***connection graph*** of the form $\langle \mathcal{D}, \mathcal{S}, \mathit{link} \rangle$:
  - $\mathcal{D}$ and $\mathcal{S}$ are two disjoint, finite sets of ***entities***. $\mathcal{D}$ is the set of ***devices***, and $\mathcal{S}$ is the set of ***scopes***. Each scope represents a broker. To denote $\mathcal{D} \cup \mathcal{S}$, $\mathcal{E}(\chi)$ is written.

    Devices represent IoT Devices in an Internet of Things system. The term "scope" is inherited from previous work [17, 19, 18], and each scope represents a broker, or more precisely, a server process that passes along messages from devices publishing events to other devices subscribed to such events. The term entity is used to refer to either a device or a scope, and each entity represents a node in the connection graph. Typically, each broker runs on a dedicated machine in the network, however multiple brokers may be executed in one single machine, and each of them is represented as an independent scope in the connection graph.
  - $\mathit{link} \subseteq \mathcal{E} \times \mathcal{E}$ is a binary relation over entities. It represents network connections. The binary relation *link* satisfies two additional requirements: (a) *link* is symmetric but irreflexive; (b) $\mathit{link} \cap (\mathcal{D} \times \mathcal{D}) = \emptyset$.

  In other words, the connection graph can be seen as a loop-free undirected graph, with vertices labelled as either devices or scopes, so that devices are never adjacent to one another. Furthermore, *link* induces four binary relations: $\mathit{publish} = \mathit{link} \cap (\mathcal{D} \times \mathcal{S})$ captures device-to-broker links, $\mathit{notify} = \mathit{link} \cap (\mathcal{S} \times \mathcal{D})$ captures broker-to-device links, $\mathit{bridge} = \mathit{link} \cap (\mathcal{S} \times \mathcal{S})$ captures broker-to-broker links, and $\mathit{monitored} = \mathit{link} \setminus \mathit{notify}$ captures links with a broker as the destination.

Intuitively, the connection graph is used to define devices, scopes, and connections between them, which represents IoT devices, brokers, and network connections in a real IoT system.

- The ***event policy*** $\mathcal{EP}$ is a quadruple $\langle \Sigma, Q, q_0, \Delta \rangle$:
  - Σ, Q, and q_0 are the components of an EA. Σ is the set of ***events*** (more precisely ***event topics***) that can be transmitted in the system.
  - $\Delta : \mathit{monitored} \to (Q \times \Sigma \to Q \times \Sigma^*)$ assigns an EA transition function to each monitored link.

  More specifically, the EA $M(x,y) = \langle \Sigma, Q, q_0, \Delta(x,y) \rangle$ is the EA that transforms the events sent from x to y.

  The event policy is used to define what execution monitors will be used, and the network connections each of them will monitor. This component represents a configuration defined by the system administrator, who is responsible for configuring the IoT system.

- $\mathcal{BP}$ is the ***brokering policy***, which is a structure of the form $\langle \mathcal{T}, \mathit{type}, \mathit{allow} \rangle$:
  - $\mathcal{T}$ is a finite set of ***link types***.
  - $\mathit{type} : \mathit{link} \to \mathcal{T}$ assigns a link type to each link.
  - $\mathit{allow} \subseteq \mathcal{T} \times \mathcal{T}$ is a binary relation defined over $\mathcal{T}$. If $\mathit{allow}(t_1, t_2)$, then a broker is allowed to pass along an event it receives from a link of type t_1 to a link of type t_2.

  The brokering policy $\mathcal{BP}$ induces a ternary relation $\mathit{propagate} \subseteq \mathcal{E} \times \mathcal{E} \times \mathcal{E}$, so that $\mathit{propagate}(x,y,z)$ iff $\mathit{link}(x,y)$, $\mathit{link}(y,z)$, and $\mathit{allow}(\mathit{type}(x,y), \mathit{type}(y, z))$. That is, $\mathit{propagate}(x,y,z)$ asserts that an event passing through link

(x,y) is allowed to be further propagated by the broker y through the link (y,z).

Intuitively, the brokering policy represents the visibility control rules configured by the system administrator, who uses the configuration to delimit the propagation of events in the system.

- $sub : notify \rightarrow 2^{\Sigma}$ assigns a set of events to each scope-to-device link. Intuitively, $sub(x,y)$ is the set of events subscribed by device y in scope x.

  This component of the schema is used to represent the subscriptions of all devices in the system.

- $\sqsubseteq$ is a partial ordering defined over the set of ***annotated tasks***. Intuitively, the dynamics of the system is modelled as the generation and discharging of tasks. These tasks are "queued up" in a work list within the system state for further processing. A ***task*** τ is defined via the following grammar.

  $$\tau ::= \mathsf{transmit}(x,y,a) \mid \mathsf{broker}(x,y,w)$$

  where $x,y \in \mathcal{E}$, $a \in \Sigma$, and $w \in \Sigma^*$. To denote the set of all tasks defined for schema χ, TK_{χ} is written.

  An annotated task is a construct of the form $\tau[t_{gen},t_{pub}]$, in which the task τ is annotated with two timestamps (i.e., natural numbers), (i) t_{gen}, the generation time of τ, and (ii) t_{pub}, the generation time of the event publication task from which τ is derived. The set of all annotated tasks for schema χ is denoted by AT_{χ}, and α is a typical member of AT_{χ}.

  By imposing the partial ordering $\sqsubseteq$ over AT_{χ} to indicate how tasks are prioritized, one can simulate different quality-of-service (QoS) concepts (see §3.3.6 for details). In particular, if $\alpha \sqsubseteq \alpha'$, then α will be processed before

α'. The annotation of tasks allows $\sqsubseteq$ to be formulated in terms of timestamps (e.g., FIFO).

In IoT systems, devices and brokers are connected to one another through different types of network technologies. The speed at which events can be propagated through the different network connections, depends on the underlying network technologies used. The partial ordering $\sqsubseteq$ is used to simulate different QoS assumptions (see §3.3.6), which are assumptions about the relative speed of network connections. In other words, the partial ordering $\sqsubseteq$ is used to account for different network speed in the network connections of the system.

3.3.4 System States

Given a schema χ, a ***system state*** γ is a triple $\langle t, ST, WL \rangle$:

- The system state tracks a global clock, for which $t \in \mathbb{N}$ is the current time. The clock is used within the model for producing timestamps.
- $ST : \mathit{monitored} \to Q$ is a function assigning an EA state to each link that is monitored by an EA. In particular, $ST(x,y)$ is the current state of $M(x,y)$, the EA guarding link (x,y).
- The ***work list*** $WL \subseteq AT_\chi$ is a finite set of annotated tasks. In the following, the predicate $\mathit{select}(\alpha, WL)$ asserts that α is a minimal element in WL according to $\sqsubseteq$. Note that for a given WL there may be multiple annotated tasks satisfying the *select* predicate. As usual, nondeterminism is implied in such cases.

Let Γ_χ be the set of all system states as defined above. γ denotes a typical member of Γ_χ. The ***initial state*** of a system is $\gamma_{init} = \langle 0, ST_{init}, \emptyset \rangle$, where $ST_{init}(x,y) = q_0$ for every

$(x,y) \in$ *monitored* (i.e., the initial state of all execution monitors is q_0).

3.3.5 State Transition

Given a schema χ, a state transition relation is defined by $\cdot \rightarrow_\chi \cdot \subseteq \Gamma_\chi \times \Gamma_\chi$. Intuitively, $\gamma \rightarrow_\chi \gamma'$ means that γ' is a successor state of γ. The transition relation is defined by the following set of transition rules, which specify the condition under which $\gamma \rightarrow_\chi \gamma'$, where $\gamma = \langle t, ST, WL \rangle$, and $\gamma' = \langle t', ST', WL' \rangle$. In the following specification, the following convention is followed by default, unless the rules explicitly say otherwise: $t' = t+1$, $ST' = ST$ and $WL' = WL$.

- **T-Publish**. *Generate an event publication task.*
  - **Precondition:** *publish*(x,y), and $a \in \Sigma$.
  - **Effect:** $WL' = WL \cup \{\, \mathsf{transmit}(x,y,a)[t,t] \,\}$
- **T-Notify**. *Consume an event notification task.*
  - **Precondition:** *select*(α, WL), $\alpha = \tau[t_{gen}, t_{pub}]$, $\tau = \mathsf{transmit}(x,y,a)$, and *notify*$(x,y)$.
  - **Effect:** $WL' = WL \setminus \{\, \alpha \,\}$.
- **T-Deliver**. *Transmit an event over a link, and apply execution monitor to the transmitted event.*
  - **Precondition:** *select*(α, WL), $\alpha = \tau[t_{gen}, t_{pub}]$, $\tau = \mathsf{transmit}(x,y,a)$, and *monitored*$(x,y)$.
  - **Effect:** Let $\delta = \Delta(x,y)$ and $(q,w) = \delta(ST(x,y),a)$. Then $ST'(x,y) = q$, and $WL' = WL_1 \cup WL_2$, where:

$$WL_1 = WL \setminus \{\, \alpha \,\}$$

$$WL_2 = \begin{cases} \{\, \mathsf{broker}(x,y,w)[t,t_{pub}] \,\} & \text{if } w \neq \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

- **T-Broker**. *Process a sequence of received events to create further transmissions.*
  - **Precondition:** *select*(α, WL), $\alpha = \tau[t_{gen}, t_{pub}]$, and $\tau = \mathsf{broker}(x,y,aw)$, such that $a \in \Sigma$ and $w \in \Sigma^*$.

- **Effect:** Let $Z = \{z \in \mathcal{E} \mid \mathit{bridge}(y,z) \vee (\mathit{notify}(y,z) \wedge a \in \mathit{sub}(y,z))\}$. Then $WL' = WL_1 \cup WL_2 \cup WL_3$, where:

$$WL_1 = WL \setminus \{\alpha\}$$

$$WL_2 = \begin{cases} \{\mathsf{broker}(x,y,w)[t,t_{pub}]\} & \text{if } w \neq \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

$$WL_3 = \{\mathsf{transmit}(y,z,a)[t,t_{pub}] \mid z \neq x \wedge z \in Z \wedge \mathit{propagate}(x,y,z)\}$$

3.3.6 Quality-of-Service Assumptions

The partial ordering $\sqsubseteq$ is used for simulating different QoS assumptions. These QoS assumptions are essentially assumptions about relative network speed in the system, as concurrency leads to nondeterminism in the ordering of events observed by an execution monitor. To illustrate how $\sqsubseteq$ can be used for reflecting QoS assumptions, the following example called ***normal QoS*** is specified through the definition of a partial ordering $\sqsubseteq_n$.

Intuitively, normal QoS is intended to capture the following:

1. Brokering tasks ($\mathsf{broker}(\_,\_,\_)$) are carried out in the broker process, and thus it is executed "intantaneously."

2. Devices are connected to brokers via fast network connections (e.g., LAN). The order of event publication by different devices connected to the same broker is therefore preserved.

3. The network connections between brokers (i.e., bridges) have unpredictable speed (e.g., WAN). Although messages passing through a link will be delivered in the same order in which they are transmitted, messages passing through parallel links will travel at unpredictable relative speed.

To capture the intuition above, the normal QoS is captured in a partial orderin $\sqsubseteq_n$. Suppose $\alpha = \tau[t_{gen}, t_{pub}]$ and $\alpha' = \tau'[t'_{gen}, t'_{pub}]$ are two annotated tasks. The partial ordering $\sqsubseteq_n$ over annotated tasks is defined, so that $\alpha \sqsubseteq_n \alpha'$ when one of the following holds:

Norm-1 τ is a brokering task and τ' is a transmission task.

Norm-2 $\tau = \mathsf{transmit}(x, y, a)$ and $\tau' = \mathsf{transmit}(x, y, b)$ for some $x, y \in \mathcal{E}$ and $a, b \in \Sigma$, and $t_{gen} \leq t'_{gen}$.

Norm-3 $\tau = \mathsf{transmit}(x, z, a)$ and $\tau' = \mathsf{transmit}(y, z, b)$ for some $x, y \in \mathcal{D}$ and $z \in \mathcal{S}$, and $t_{gen} \leq t'_{gen}$.

The following are some observations about the definition above:

- Since **Norm-1** ensures that brokering tasks receive higher priority than transmission tasks, and **T-Broker** introduces at most one brokering task after consuming a brokering task, it is a state invariant that there is at most one brokering task in the work list. Thus there is no need to impose further ordering among brokering tasks.

- The effect of **Norm-1** is that, when an event is delivered to a broker, brokering will occur "instantaneously," leading to the generation of further transmission tasks in the work list. Event transmission will only resume after the brokering of a delivered event is complete.

- **Norm-2** ensures that a communication link delivers events on a first-come-first-serve basis. In other words, when events travel through a communication link, they are delivered in the same order in which they are transmitted.

- **Norm-3** ensures that, when multiple publishers are linked directly to a broker, the events they publish will arrive at that broker in the order of publication.

Other than the guarantees above, the relative speed of the communication links may vary nondeterministically, and arbitrary interleaving may occur to event transmissions.

3.4 Visibility Control

This chapter demonstrates how the two protection mechanisms, brokering control and execution monitoring, can be leverage to impose various forms of visibility control.

3.4.1 Visibility Control via Brokering Policies

The most liberal brokering policy is $\mathcal{BP}_0 = \langle \mathcal{T}_0, type_0, allow_0 \rangle$, where $\mathcal{T}_0$ is a singleton set $\{t_0\}$, $type_0$ maps every link to t_0, and $allow_0 = \{(t_0, t_0)\}$. This trivial brokering policy allows every event received from a link to be forwarded to another link. This set-up is essentially the bridge feature of Mosquitto [25]: a bridge connects two MQTT brokers, so that the events received by one broker are made visible to the other broker. This liberal brokering policy, however, suffers from the following shortcoming. The brokers connected by bridges form a single scope of events. There is no regulation of what events are visible to which subscribers, making it difficulty to confine the visibility of sensitive events (e.g., personal health alerts). Imposing brokering policies more restrictive than $\mathcal{BP}_0$ allows us to regulate the flow of information, as we illustrate in the following.

3.4.1.1 Information Flow Control

Brokering policies allow us to impose a form of information flow control in the style of the Bell-LaPadula (BLP) model [22]. The basic idea of the BLP scheme is that information sources (e.g., publishers) and information consumers (e.g., subscribers) are labelled with security labels (e.g., unclassified, confidential, secret, top secret). These security labels form a lattice structure, and is thus partially ordered. BLP is essentially an access control model that forbids "reading up" and "writing down" [27]. So information may only flow from "low" information sources to "high" information consumers.

In the context of Event Based Systems, an event publication is a "write," and an event subscription can be considered a "read." Rather than assigning security labels to entities, a more uniform and flexible approach is used to assign security labels to links. The key idea is that, when a broker y receives an event from a link (x,y), y is allowed to propagate the event to a subsequent link (y,z) if the security label of the second link is at least as high as the security label of the first link. Consequently, the semantics of assigning a security label t to a link is that events flowing through that link comes from sources with security labels lower than or equal to t. When an event is transmitted through the system, it goes through links with monotonically increasing labels $t_1 \leq t_2 \leq t_3 \leq \ldots$. More specifically, one can configure the brokering policy $\mathcal{BP} = \langle \mathcal{T}, type, allow \rangle$ as follows to control the flow of information within the system.

BLP-1 Let $(\mathcal{T}, \leq)$ be a partially ordered set of security labels.

BLP-2 The function *type* assigns a security label to each link, in such a way that $type(x,y) \leq type(y,z)$ whenever $y \in \mathcal{D}$. That is, a notification link of a device y must have a security label no higher than the security label of every publication link of y. In other words, the device is "reading down" through the notification link, and "writing up" through the publication link.

BLP-3 Define *allow* so that $allow(t_1,t_2)$ if and only if $t_1 \leq t_2$. That is, an event flowing through a link (x,y) with a security label l will only be propagated to a link (y,z) with a security label h at least as high as l.

With the scheme above, successive links that transmit an event will have monotonically increasing security labels. This observation is formalized as follows.

A sequence $x_0 x_1 \ldots x_n$ of entities in $\mathcal{CG}$ is called a ***flow path*** if (a) $(x_i, x_{i+1}) \in link$ for $0 \leq i < n$, (b) for $0 < i < n$, if $x_i \in \mathcal{S}$, then $x_{i-1} \neq x_{i+1}$ and $propagate(x_{i-1}, x_i, x_{i+1})$. A flow path is a ***flow route*** when none of the entities other than the two ends (i.e., x_1, x_2,

$\ldots, x_{n-1}$) is a device. A flow path is a potential path of information flow through the system. Intermediary entities along a flow path can be devices which read a message and then propagate information by publishing a correlated message. A flow route is a flow path for which the intermediary entities are all scopes.

With the way *allow* is defined (**BLP-3**), $type(x_{i-1}, x_i) \leq type(x_i, x_{i+1})$ when a broker x_i relays a message from link (x_{i-1}, x_i) to link (x_i, x_{i+1}). Thus a message passes through links of monotonically increasing security labels as it travels through a flow route. A flow path is essentially the concatenation of flow routes for which the concatenation points are devices. The definition of *type* (**BLP-2**) ensures that monotonicity is preserved by such concatenation.

The administrator of the system may have some preconceived ideas about what flow paths (resp. routes) are permitted. An important validation task is to ensure that the configuration of the connection graph and the brokering policy does not violate her expectation. Given a connection graph $\mathcal{CG} = \langle \mathcal{D}, \mathcal{S}, link \rangle$, one can use a variant of the Floyd-Warshall algorithm [14, §25.2] to compute whether there is a legitimate flow path between each pair of entities (more precisely, between each pair of links). The algorithm runs in $O(M^3)$ time, where $M = |link|$. The algorithm can be adopted to identify either flow paths or flow routes. Such an analysis allows us to debug the topology of the connection graph and the assignment of security labels, so as to ensure that devices that are supposed to communicate with one another can do so, and flow paths that are not supposed to exist are not accidentally enabled. Such an algorithm is described in full details in appendix A.

3.4.1.2 Component Architecture

The modular distributed event-based systems proposed in [17, 18, 19] is the idea that brokers and devices are organized into a hierarchy. This hierarchy is used to promote modularity in the engineering and designing of distributed systems, based on event based systems. Devices connected to one broker are bundled together to form a component, which delim-

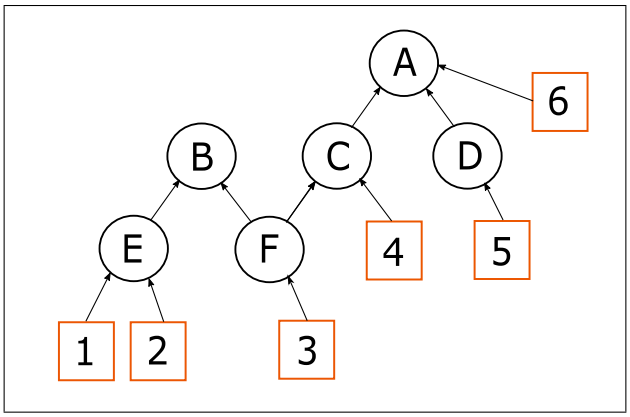

Figure 3.1: Component Architecture

its and constrains the visibility of events produced or consumed by them. Additionally, components act as publisher of internally produced events, and as consumers of outside notifications. This allows for further bundling, such that components can be bundled together to form higher-level components. This hierarchical grouping of components to form higher level components allows distributed systems to be constructed in a modular manner.

The component architecture hierarchy is specified through a parenthood relation, *parent* $\subseteq \mathcal{E} \times \mathcal{E}$. Intuitively, $parent(x,y)$ asserts that y is a parent of x. Two further restrictions apply to the specification of *parent*. First, a device is never a parent of any entity. Second, parenthood chains never form a cycle, not even a loop (i.e., a loop arises when an entity is its own parent).

Each broker bundles together a number of publishers, subscribers and children brokers, in order to provide a communication channel among them. Figure 3.1 illustrates a directed graph representation of a component hierarchy using six brokers (A, B, C, D, E and F) and six devices (1, 2, 3, 4, 5 and 6). In Figure 3.1, an arrow coming from an entity x to another entity y represents the relation $parent(x,y)$. Messages published in a broker x are visible in another broker y, if and only if x and y share a common ancestor in the parenthood hierarchy. For example, an event published by device 1 should be visible in brokers B, E and F, whereas an event published by device 3 should be visible in all brokers. To simulate

this model, it is possible to configure the brokering policy $\mathcal{BP} = \langle \mathcal{T}, type, allow \rangle$ as follows:

- Let $\mathcal{T} = \{ \mathsf{up}, \mathsf{down} \}$.
- $link = \{(x,y) \in \mathcal{E} \times \mathcal{E} \mid \text{either } parent(x,y) \text{ or } parent(y,x)\}$
- The function *type* reflects the parenthood relation, so that $type(x,y) = \mathsf{up}$ if $parent(x,y)$, and $type(x,y) = \mathsf{down}$ otherwise.
- Define *allow* so that $allow(t_1,t_2)$ holds unless $t_1 = \mathsf{down}$ and $t_2 = \mathsf{up}$. Effectively, a message can move "up" the hierarchy, and then "down", but never move "up" again after it has moved "down." In other words, a broker can propagate a message form a parent to a child (i.e., $allow(\mathsf{down}, \mathsf{down})$), from a child to another child (i.e., $allow(\mathsf{up}, \mathsf{down})$), and from a child to a parent (i.e., $allow(\mathsf{up}, \mathsf{up})$), but never from a parent to another parent (i.e., $allow(\mathsf{down}, \mathsf{up})$).

The above scheme ensures that events published in a scope x are visible to subscribers in another scope y if and only if x and y share a common ancestor in the hierarchy. To see this, observe that a flow route from one device to another will only go from "up" to "down" but not vice versa. This essentially means that the flow route will first reach a common ancestor of the two devices before reaching its destination. Thus the sharing of a common ancestor determines visibility.

A directed acyclic graph (of which the parenthood hierarchy is a special case) forms a ***conjoined forest*** whenever the following condition holds: There is at most one directed path from any given node to any other node in the graph. Figure 3.2 illustrates an example of a directed acyclic graph that is not a conjoined forest, whereas Figure 3.3 illustrates an example of a conjoined forest. Notice that all conjoined forest are directed acyclic graphs, but the opposite does not hold. Intuitively, in a conjoined forest, even though "trees" may share branches, duplicate event delivery and circular transmission are not possible. In the following, we consider only parenthood hierarchies that are conjoined forests.

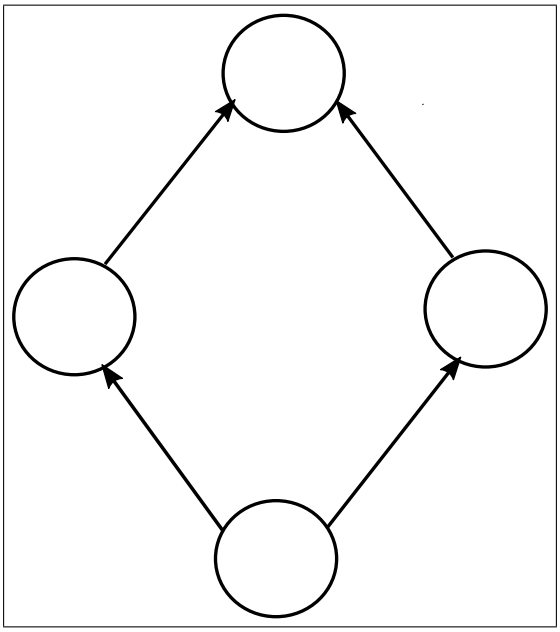

Figure 3.2: Directed Acyclic Graph

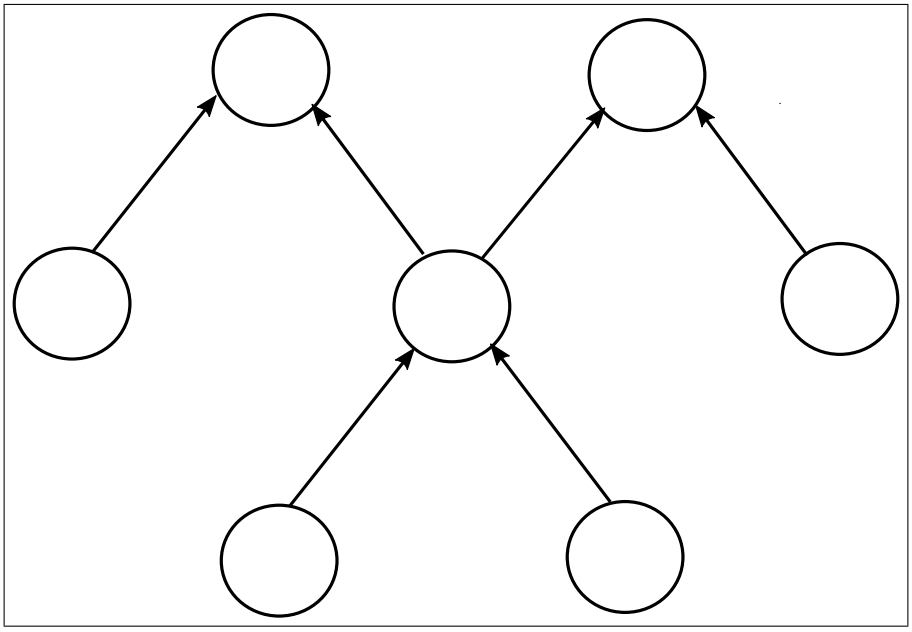

Figure 3.3: Conjoined Forest

3.4.1.3 Discussions

In §3.3.3, the brokering policy is introduced as a mechanism to define the visibility of events based on types associated to links. A different approach to define visibility could be articulated in terms of links, such that the administrator was able to define what links are able to propagate events to other specific links in a link by link fashion. In this work, the type-based brokering policy is used because of following considerations.

Firstly, it is more likely that the system administrator would want to work with the more abstract notion of types than to work with brokering policies on a link by link basis. Under this link-based formulation, every time a link is introduced, the administrator would have to reformulate the brokering policy. On the other hand, with the type-based formulation, when a new link is introduced, no change to the brokering policy is needed. All it takes is for the administrator to assign a type to the new link.

Secondly, the link-by-link formulation is subsumed by the type-based formulation. If every link in the system is assigned a distinct type, then the link-based formulation can be "emulated" by the type-based formulation. In this sense, any brokering policy defined under the link-by-link formulation, can also be defined under the type-based formulation, which means that no expressive power is lost by working with types rather than links.

3.4.2 Visibility Control via Execution Monitoring

Event filtering and event mapping are popular visibility control mechanisms in distributed event-based systems. They are featured in the model of Fiege *et al.* [17, 19], and implemented in Mosquitto [25]. We demonstrate that these two mechanisms are special cases of execution monitoring.

3.4.2.1 Event Filtering

Some events are used for coordination logics that belong to the internal working of a distributed software component. Publications of such events should not be visible outside of

the component because of confidentiality considerations, and subscriptions of these events should be invisible to the component's clients because of integrity considerations.

Suppose $E \subseteq \Sigma$ is a subset of events that are allowed to pass through a link. Define the ***event filtering transition function*** $filter(E)$ so that $\delta(q_0, a) = (q_0, a)$ if $a \in E$, and $\delta(q_0, a) = (q_0, \varepsilon)$ if $a \notin E$. By setting $\Delta(x, y) = filter(E)$, only events in E will be transmitted along the link.

Event filtering leads naturally to the notion of event "import" and "export" for component architectures such as the one presented in §3.4.1.2. A scope x can present an ***interface*** to each parent scope y. The interface consists of a set E_{import} of events that x is willing to import from y, and a set E_{export} of events x is willing to export to y. If the publication of an event in E_{export} is visible in x, then it will also be visible in y. If the subscription of an event in E_{import} is visible in x, then it will also be visible in y. To achieve the above, we set $\Delta(x, y) = filter(E_{export})$, and $\Delta(y, x) = filter(E_{import})$.

3.4.2.2 Event Mapping

Event mapping is the transformation of events from one naming scheme to another naming scheme when they are transmitted along a link. Event mapping is particularly useful for presenting an alternative interface of a distributed software component to a client component, often for "gluing" purposes or for information hiding. Suppose $f : \Sigma \to \Sigma$ is a function that renames events. Define the ***event mapping transition function*** $mapper(f) : Q \times \Sigma \to Q \times \Sigma^*$ such that $mapper(f)(q_0, a) = (q_0, f(a))$. That is, the EA remains in state q_0 at all time, and output $f(a)$ when the input event is a. Setting $\Delta(x, y)$ to $mapper(f)$ will cause all events passing through link (x, y) to be renamed according to f.

3.4.3 RBAC Based Visibility Control

The combination of publish/subscribe middleware and Role-Based Access Control (RBAC), proposed in [12], is used to delimit the events that clients (i.e., publishers and subscribers)

are able to publish and subscribe to. The idea is that publishers and subscribers are assigned to roles, and each role is associated to a set of events that it can publish, and to a set of events that it can subscribe to. In this section, it is shown how the different components of the model can be configured to support this form of RBAC, where roles are represented by scopes.

Let R be the set of roles in the system, then the set of events that members of each role $r \in R$ can publish is denoted by $pub(r)$, and the set of events that members of each role $r \in R$ can subscribe to is denoted by $sub(r)$, such that $pub(r) \subseteq \Sigma$ and $sub(r) \subseteq \Sigma$. The following paragraphs describe the configuration of the model used to delimit event publication and subscription based on roles.

Firstly, the connection graph is configured as follows:

RBAC-CG-1 $\mathcal{D}$ is defined according to the system administrator needs, to represent the set of all devices in the system.

RBAC-CG-2 Define the set of scopes $\mathcal{S}$ as follows:

- $\mathcal{S}^{pub} = \{r^{pub} | r \in R\}$
- $\mathcal{S}^{sub} = \{r^{sub} | r \in R\}$
- $\mathcal{S} = \mathcal{S}^{pub} \cup \mathcal{S}^{sub} \cup \{Bus\}$

To simulate each role $r \in R$, two scopes r^{pub} and r^{sub} are defined in the connection graph. The scope r^{pub} is used to delimit the events published by devices belonging to role r, whereas the scope r^{sub} is used to delimit the events that these devices can be notified about. Additionally, the scope *Bus* is used by all other scopes for event propagation purposes.

RBAC-CG-3 Let $\mathcal{S}^{roles}$ be the set of all scopes in the system except for the scope *Bus*, that is, $\mathcal{S}^{roles} = \mathcal{S}^{pub} \cup \mathcal{S}^{sub}$. Then, define the binary relation *link* as follows:

$$link = \{(s, Bus) | s \in \mathcal{S}^{roles}\} \cup \{(Bus, s) | s \in \mathcal{S}^{roles}\} \cup link^{roles}$$

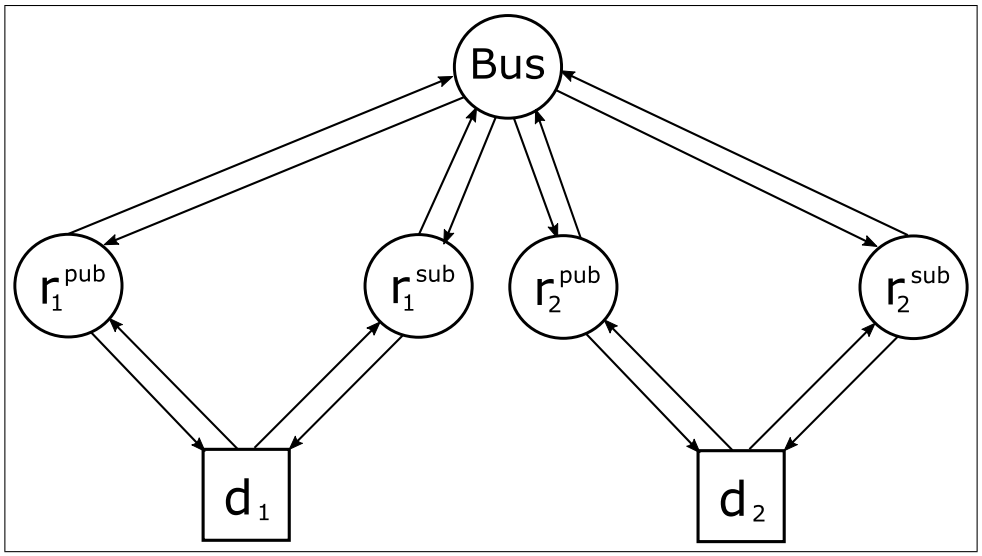

Figure 3.4: RBAC - Connection Graph Configuration

This means that all scopes representing roles are connected to the scope *Bus*. To represent membership of devices to roles, devices belonging to a role $r \in R$ have to be connected to its corresponding scopes r^{pub} and r^{sub}. Define $link^{roles}$ to represent this, for example:

$$link^{roles} = \{(d, r^{pub}), (r^{pub}, d), (d, r^{sub}), (r^{sub}, d)\}$$

Where $d \in \mathcal{D}$ and $r^{pub}, r^{sub} \in \mathcal{S}$ represent the role r. This configuration of $link^{roles}$ means that d is assigned to the role r.

Figure 3.4 illustrates an example of a configuration graph, where two roles r_1 and r_2 are represented by the scopes $r_1^{pub}, r_1^{sub}, r_2^{pub}$ and r_2^{sub}. Furthermore, device d_1 is member of the role r_1 while device r_2 is member of role r_2.

After having configured the connection graph, the visibility of events is defined with the following configuration of the brokering policy.

RBAC-BP-1 Let $\mathcal{T} = \{\,\mathsf{SubUp}, \mathsf{SubDown}, \mathsf{PubUp}, \mathsf{PubDown}, \mathsf{DevUp}, \mathsf{DevDown}\,\}$.

RBAC-BP-2 The function *type* assigns types to links as follows:

$$type(x,y) = \begin{cases} \mathsf{SubUp} & \text{if } x \in \mathcal{S}^{sub} \wedge y = Bus \\ \mathsf{SubDown} & \text{if } y \in \mathcal{S}^{sub} \wedge x = Bus \\ \mathsf{PubUp} & \text{if } x \in \mathcal{S}^{pub} \wedge y = Bus \\ \mathsf{PubDown} & \text{if } y \in \mathcal{S}^{pub} \wedge x = Bus \\ \mathsf{DevUp} & \text{if } x \in \mathcal{D} \wedge y \in \mathcal{S}^{roles} \\ \mathsf{DevDown} & \text{if } y \in \mathcal{D} \wedge x \in \mathcal{S}^{roles} \end{cases}$$

RBAC-BP-3 Define *allow* to be:

$$allow = \{(\mathsf{PubUp}, \mathsf{SubDown}), (\mathsf{DevUp}, \mathsf{PubUp}), (\mathsf{SubDown}, \mathsf{DevDown})\}$$

The above configuration of the brokering policy, defines six types of links (**RBAC-BP-1**). By the way in which the function *type* is defined (**RBAC-BP-2**), links of types DevUp and DevDown are used to identify links between devices and scopes representing roles. Links of types PubUp and PubDown identify links between scopes used to delimit publication of events, and the root scope *Bus*. Similarly, links of types SubUp and SubDown identify links between scopes used to delimit event notification, and the root scope *Bus*. Finally, the configuration of *allow* (**RBAC-BP-3**) makes sure that devices can only publish events to publishing scopes (i.e., $allow(\mathsf{DevUp}, \mathsf{PubUp})$), then these events are propagated from publishing scopes to subscribing scopes through the *Bus* scope (i.e., $allow(\mathsf{PubUp}, \mathsf{SubDown})$), and finally delivered from subscribing scopes to devices (i.e., $allow(\mathsf{SubDown}, \mathsf{DevDown})$). This configuration disables any other type of propagation, including direct communication of devices (i.e., propagation form a type DevUp to a type DevDown is not allowed).

Figure 3.5 illustrates an example of the configuration graph in figure 3.4 with a brokering policy defined as described in the previous paragraphs.

The brokering policy makes sure that event propagations follow specific flow routes. However, to impose control over what events are allowed to be propagated through the

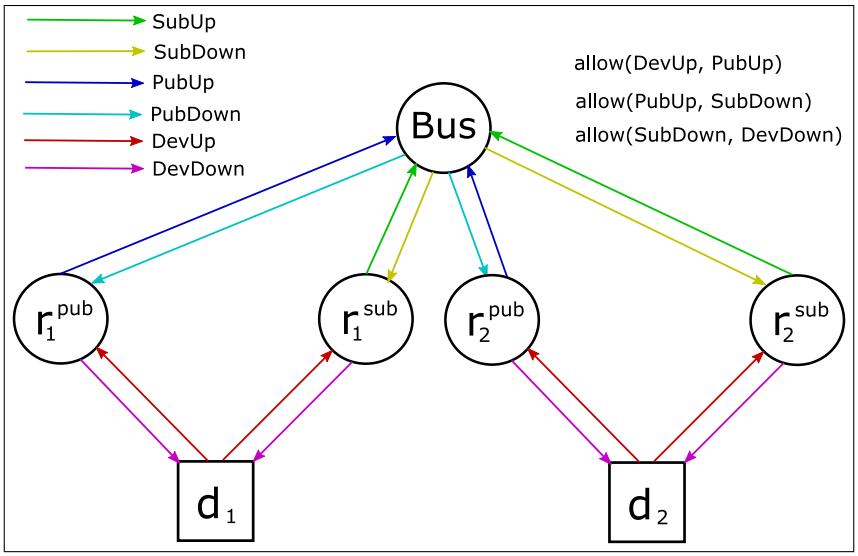

Figure 3.5: RBAC - Brokering Policy Configuration

allowed flow routes, execution monitoring is configured through the event policy. Particularly, the ***event filtering transition function*** defined in 3.4.2.1 is used for this purpose.

Consider $Role : \mathcal{S}^{roles} \rightarrow R$ to be a function that maps scopes used to represent roles, to their corresponding role, such that $Role(r^{pub}) = Role(r^{sub})$ is the role represented by scopes r^{pub} and r^{sub}. Then, the function Δ of the event policy is defined as follows:

$$\Delta(x,y) = \begin{cases} filter(sub(Role(y))) & \text{if } y \in \mathcal{S}^{sub} \wedge x = Bus \\ filter(pub(Role(x))) & \text{if } x \in \mathcal{S}^{pub} \wedge y = Bus \end{cases}$$

Intuitively, this configuration of the event policy means that events propagated from publishing scopes to the root scope *Bus* are filtered, so that only events publishable by their corresponding roles are allowed. Events propagated to subscribing scopes from the *Bus* scope are filtered in a similar fashion, in order to comply to the subscribing restrictions of their corresponding roles.

Figure 3.6 illustrates an example of a final configuration of the connection graph, the brokering policy and the event policy as described in this section. This configuration allows devices to be assigned to one role by simply connecting them to its corresponding scopes.

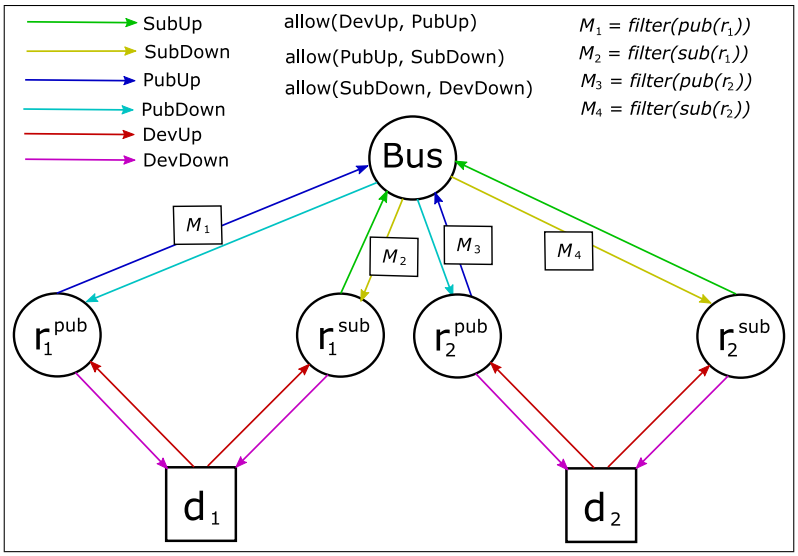

Figure 3.6: RBAC - Final Configuration

However, there are some issues related to this configuration of the model. Firstly, if devices are members of multiple roles, it implies that multiple flow routes are available between different devices, which could result in events being delivered multiple times to subscribers. Secondly, because in this configuration devices and scopes are connected in such a way that they form a cycle, then it is unavoidable for devices to listen to their own publications. To address these issues, the following restrictions are imposed:

1. Every device must belong to exactly one role. However, new roles can be created to accommodate event publication and subscription according to the system administrator needs.

2. Roles cannot subscribe to events publishable by them, that is: $\forall r \in R$, $pub(r) \cap sub(r) = \emptyset$

In this section, it was demonstrated how the model can be instantiated to enforce a form of RBAC inspired by [12], this is achieved through the use of scopes to model authorization principals (i.e., roles), and with the combination of the brokering policy and the event policy.

Device	Consumes	Produces
MD	none	MD_motion MD_no_motion
SC	none	none
DL	DL_unlock DL_lock	none
DB	MD_motion MD_no_motion	AccessRequest
SP	AccessRequest	AccessGranted AccessDenied DL_unlock

Table 3.1: Events Produced/Consumed by Devices

3.5 Scoping and Execution Monitoring Case Study

In this section, an example use case illustrates how the model can be used to enforce security policies for IoT applications. In this example, two security policies are enforced via scoping and execution monitoring taking into account the following security considerations. Firstly, events published by some devices are considered to disclose sensitive information. Therefore, they should be treated with special care. Secondly, the existence of potentially compromised devices that behave in a way that violates certain behavior protocols is assumed. Thus, mechanisms to protect the system against potentially malicious events have to be implemented. Finally, this case study also demonstrates how scoping can be used to delimit the visibility of events in order to ensure the protection of critical devices, and to create private domains where devices that do not require interaction with public networks (e.g., the Internet) can interact with one another.

Consider the following scenario. A home owner has acquired a number of smart devices to implement automated tasks at home. Among these devices, she acquired: one motion detector (MD), one security camera (SC) [1], one door lock (DL), and one door bell (DB). Additionally, the home automation system is able to communicate with her smart phone

[1] It is assumed that the home owner has access to the security camera image through her smart phone, and the security camera is used only to help the homeowner to make decisions (i.e., grant access or deny access), but no direct interactions between the security camera and other devices are considered in this case study.

Event	Description
MD_motion	Motion detected by the Motion Detector
MD_no_motion	No motion detected by the Motion Detector
DL_unlock	Commands the door lock to unlock the door
DL_lock	Commands the door lock to lock the door
AccessRequest	Indicates that access has been requested
AccessGranted	Indicates that access has been granted
AccessDenied	Indicates that access has been denied

Table 3.2: Events Description

(SP) via Internet at all times. Finally, each of the smart devices may consume and produce a number of events. The relationship between what events are produced and consumed by each device is shown in Table 3.1, and a description of each event is given in Table 3.2.

Events related to the motion detector are used to detect the presence of people at home, such that, if motion is detected (i.e., MD_motion), it is inferred that someone is in the house, and if no motion is detected (i.e., MD_no_motion), it means that the house is empty. Because this information is very sensitive, special measures have to be taken for events published by the motion detector.

Another critical event that has to be considered is the DL_unlock event. This event is used to command the door lock to unlock the door, thereby granting access to the house. If no security measures are taken, this event could be potentially used by a compromised device to unlock the door. For this reason, it is paramount to have a mechanism to discriminate between legitimate DL_unlock events allowed by the homeowner, and potentially malicious events of this type (e.g., published by compromised devices).

The door bell is a special device that works with the motion detector to send notifications to the homeowner, such that, if nobody is at home (i.e., MD_no_motion) and someone rings the bell, an access request notification event (i.e., AccessRequest) is published by the door bell. However, if someone is at home (i.e., MD_motion), and the door bell is rung, no events are published by the door bell. In response to the access request notification (i.e., AccessRequest), the homeowner may grant access (i.e., AccessGranted) and unlock the

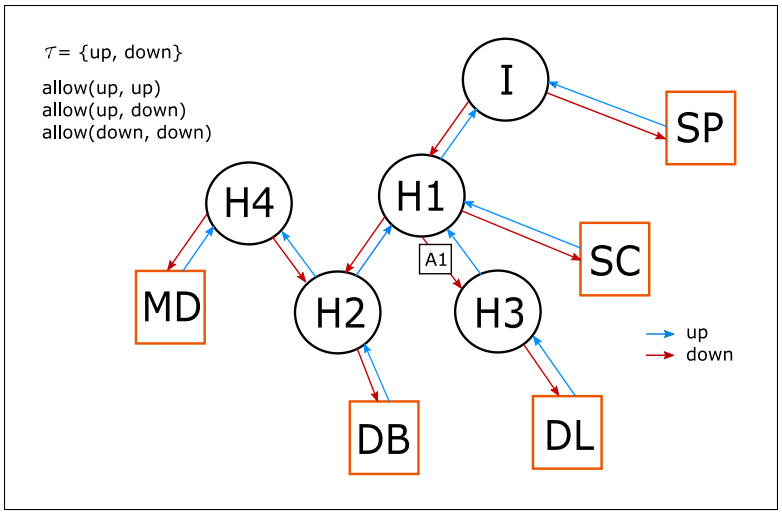

Figure 3.7: System configuration

door (i.e., `DL_unlock`), or simply deny access (i.e., `AccessDeny`).

In this sense, legitimate `DL_unlock` events are those that are formed by the sequence `AccessRequest`, `AccessGranted`, `DL_unlock`, because this sequence of events implies that the homeowner was notified about one access request, and in response she explicitly granted access and unlocked the door. Finally, considering the above discussion, the following security policies are imposed:

Policy 1. Prevent sensitive information (i.e., `MD_motion` and `MD_no_motion`) from leaking to the Internet.

Policy 2. Prevent the door from unlocking (i.e., `DL_unlock`) unless an access has been previously requested (i.e., `AccessRequest`) and granted (i.e., `AccessGranted`).

Figure 3.7 illustrates a possible configuration of the system, with visibility control based on the component architecture (see §3.4.1.2), and six scopes serving different purposes. Scope "I" represent a scope hosted on the Internet, which enables interactions between home devices, and the smart phone (SP) at any time. Scope "H1" can be seen as a gateway, used to potentially filter events coming from the Internet into the local network, and events propagated from the local network to the Internet. Scope "H4" is used to create a private

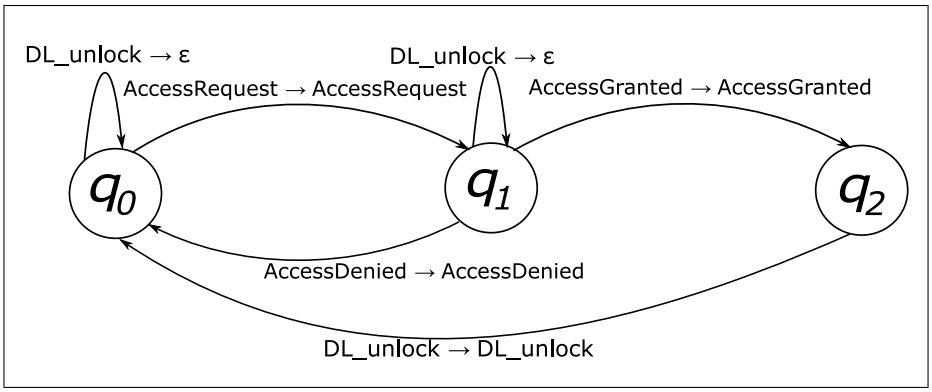

Figure 3.8: Automaton 1

domain, where entities that do not require interaction with the Internet can interact with one another. This private domain is created thanks to the visibility control based on the component architecture, where visibility of events published in one scope is limited to scopes sharing a common ancestor. This visibility control is enforced by not allowing propagation of events from "down" links (i.e., red arrows) to "up" links (i.e., blue arrows). In this particular case, events published in scope "H4", are only made visible to scope "H2". Scope "H2" is the only scope that allows its hosted devices to communicate with any other device in the system, including devices hosted in the private domain, and devices hosted on the Internet. Notice that this is possible because "H2" shares a common ancestor with every other scope in the system. Finally, "H3" isolates the door lock (DL), making sure that any interactions between any other device and the door lock are monitored by at an execution monitor (i.e., the execution monitor "A1").

Notice that events published by the motion detector (MD) are delimited to the private domain of the home automation system. In this way, the current configuration of the system is used to enforce **Policy 1**.

The configuration of the system also facilitates the enforcement of **Policy 2**, since the door lock is isolated in scope "H3", which is guarded by the execution monitor **Automaton 1 (A1)**.

Figure 3.8 illustrates **Automaton 1**. A1 is designed to enforce **Policy 2**, where events

of type DL_unlock are suppressed unless the sequence AccessRequest, AccessGranted precedes them. In Figure 3.8 transitions between two states q and q' have labels of the form $a \rightarrow w$, which represents one entry in the transition function $\delta(q,a) = (q',w)$, that is, the left side of the arrow represents the input event being processed by the automaton, and the right side of the arrow represents the sequence of events that are made visible to the world (i.e., scope "H3" in this case). This automaton has three states, the initial state q_0, and states q_1 and q_2. State q_0 represents that the AccessRequest event has not been detected by the automaton. In this state, events of type DL_unlock are suppressed since they are not legitimate, and upon detecting an event of type AccessRequest, a transition to state q_1 is made. State q_1 represents that the event AccessRequest was detected, but the AccessGranted event has not been detected yet. In this state, events of type DL_unlock are also suppressed since they are not legitimate. This state has two possible transitions: 1) if the event AccessDenied is detected, it implies that the homeowner denied access to the house, which results in a transition to the initial state q_0 to restart the detection of legitimate DL_unlock events, and 2) if the event AccessGranted is detected, it implies that the next DL_unlock event is legitimate, thus a transition to state q_2 is made. State q_2 represents that the sequence AccessRequest, AccessGranted was detected, which implies that the next event of type DL_unlock is legitimate. Upon detecting the event DL_unlock, the automaton makes this event visible to the world, and goes back to the initial state, to restart the detection of further legitimate DL_unlock events. With this automaton, the implementation of both security policies is completed.

This chapter formally described a mathematical model for Event Based Systems that includes both security mechanisms: Execution Monitoring and Scoping. These mechanisms where leveraged to impose various forms of visibility control, including some form of information flow control based in a popular access control model. Finally, a case study was presented to demonstrate the use of the model, to implement security relevant policies

for IoT applications. In the next chapter a description of two implementations conducted during the research project is given.

Chapter 4

Implementation of Scoping and Execution Monitoring

In this chapter, I describe two important implementation efforts conducted during this research project. The first implementation, refers to the mechanization of the mathematical model described in §3.3 using PLT Redex [4]. The second implementation is an extension to the open source software Mosquitto [25] to support Scoping and Execution Monitoring.

The mechanization of the model using PLT Redex is briefly described in §4.1. In §4.2, the background necessary to appreciate the extension to Mosquitto is given. Finally, §4.3 describes the implementation of Scoping and Execution Monitoring on top of Mosquitto.

4.1 Model mechanization using PLT Redex

Formalization of models is a very challenging task, since the language in which operational semantics of the models are described are very different to what we normally use on a daily basis. It is only natural that mistakes are made in the formalization process, even for the most seasoned researchers.

Similar to the process of developing systems with regular programming languages, where debugging tools help developers test their code. In the process of formalization of models as state transition systems, there exist similar debugging tools to help "mechanize" the modelling process. One such tool is PLT Redex.

PLT Redex [4, 16] is a language used for designing and debugging operational semantics. With this language it is possible to quickly come up with executable semantic models. These executables helped during the formalization process in this work, to spot bugs and gain more insight about the model.

The model was machanized in a modular fashion, starting with a simple event based

system, where no security mechanisms were included. Then, the implementation was extended to include scoping, with visibility control based on the component architecture 3.4.1.2. The final version of this implementation was extended to include execution monitoring and scoping.

The magnitude of the work done for this implementation was measured in term the number of lines written, for a total of 820 lines of code. These lines include the model definition, as well as a number of test cases.

4.2 Mosquitto Implementation Preliminaries

This implementation is based on the source code of Mosquitto version 1.4.10, which was the latest version when the project started, although the current latest released version of Mosquitto is 1.4.14, released on July 11, 2017.

Mosquitto is an open source message broker that provides a lightweight implementation of the MQTT protocol [11]. Additionally, this software supports a number of features that makes it ideal for this project:

1. Compared to other open source MQTT brokers, Mosquitto is a minimalist implementation of the MQTT protocol. It consists of 59 C language source files (.h and .c files), with a total of 20,289 lines of code (for the MQTT server implementation). Due to its relatively small size, identification of crucial processes in the source code required minor efforts.

2. Mosquitto provides a feature called ***Bridge***, for interconnecting Mosquitto servers to simulate a large, distributed Broker. Scoping was implemented on top of this facility, which accelerated the implementation process significantly.

3. Mosquitto also implements two lightweight clients: ***mosquitto_sub*** and

mosquitto_pub. These clients, although simple, were used for automated testing purposes.

The implementation of the two security mechanisms in Mosquitto involved a number of C language source files where internal processes of Mosquitto were modified. New definitions were added to the internal structures of mosquitto defined in header files. Finally, the compilation process of Mosquitto was updated via Makefiles. The magnitude of the work done for the implementation was measured in terms of the diff between the extended version code base, and the original Mosquitto code base, and it consists of 1,358 lines.

In order to appreciate the work done to implement the two security mechanisms, it is necessary to revise two fundamental concepts of the original version of Mosquitto: The Mosquitto bridging facilities, and the Mosquitto Message Flow.

In this section, a description of the Bridging facilities of Mosquitto is described in §4.2.1. In §4.2.2 the message flow followed by Mosquitto upon receiving a message in the three QoS levels is explained.

4.2.1 Mosquitto Bridging

In Mosquitto, a bridge is a feature that enables interconnections between two (or more) Mosquitto Servers. This is done via the Mosquitto configuration file [2], where the topics to be shared by the interconnected servers are defined. In its most extensive form, two interconnected Mosquitto Servers (or Brokers) *x* and *y* may share events published in all topics and in both directions (from *x* to *y* and vice versa). It means that the visibility of the shared events is extended, so that they are visible in all the interconnected Mosquitto Servers. For the rest of the chapter, it is assumed that bridges are always configured to share all topics in both directions.

The bridge connection is configured only by one of the interconnected brokers. Say for example that broker *x* established the bridge connection to broker *y*. In such case, *y*

treats x like a regular MQTT client, who is subscribed to the topic "#", which is a wildcard that indicates "all topics". On the other hand, x is aware that y is a bridge, and whenever x receives an event from a different source, it further propagates this event to y.

4.2.2 Mosquitto Message Flow

The MQTT standard [11] mandates the implementations of the following three Quality of Service levels:

QoS 0: At most once delivery. In this QoS level, the delivery of an event is not guaranteed, and no response is expected from the receiver. Events delivered with this QoS level arrive once or not at all. This QoS level is also known as "fire and forget".

QoS 1: At least one delivery. This quality of service level guarantees the event to be delivered to the receiver at least once. However, the receiver could receive multiple copies of the same event.

QoS 2: Exactly one delivery. This quality of service level guarantees events to be delivered to receivers exactly once (loss or duplication of events is unacceptable).

For simplicity, and in preparation of the discussion ahead, a message a is defined to be a tuple $a = (mid, t, p, qos)$, where mid is a message id (e.g., an integer), t is the topic of the message, p is its corresponding payload, and qos is the QoS level. Also, whenever $a.mid$, $a.t$, $a.p$ or $a.qos$ is written, it implies that the message id, topic, payload or QoS level of the message a are being referred to, respectively.

Additionally, a simplified message representation of the control packets used during the communication between a publisher and the broker (as defined by the MQTT protocol specification) is used. These messages are: PUBLISH(mid, t, p, qos), PUBACK(mid), PUBREC(mid), PUBREL(mid), and PUBCOMP(mid).

In the following sections, the Message Flow (in terms of internal processing) of Mosquitto for the three QoS levels is described. Because the interest of this extension of Mosquitto is to transform the input message into a sequence of messages to be delivered, the description of the message flow is delimited to the queueing process of Mosquitto, from the moment the broker receives a message, to the moment it queue ups the corresponding notifications for the subscribers.

4.2.2.1 QoS 0 Message Flow

The QoS 0 is the most simple among the three QoS levels defined in the MQTT protocol specification. In this QoS level, a publisher x sends PUBLISH(mid, t, p, qos) to the broker. Upon receiving this message, the broker verifies for subscriptions to topic t, and notifies the corresponding subscribers. Notice that in this QoS level, no response from the broker is expected by the publisher.

Internally, the Mosquitto broker performs the following tasks to comply with the protocol:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.
2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.
3. Look for subscriptions to the topic t, and create a list of subscribers.
4. For each subscriber i in the list of subscribers:
    (a) Generate a new message id mid_i, and queue up a new message $a_i = (mid_i, a.t, a.p, qos_i)$ for delivery, where qos_i is the QoS level in which the subscription was issued.
5. Remove a from the list of stored messages.

4.2.2.2 QoS 1 Message Flow

In this QoS level, a publisher x sends PUBLISH(mid, t, p, qos) to the broker, and stores the message $a_x = (mid, t, p, qos)$. Upon receiving this message, the broker verifies for subscriptions to topic t, and notifies the corresponding subscribers. After having notified all subscribers, the broker sends PUBACK(mid) to x. When x receives PUBACK(mid), it proceeds to discard a_x.

Internally, the Mosquitto broker performs the following tasks to comply with the protocol:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.
2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.
3. Look for subscriptions to the topic t, and create a list of subscribers.
4. For each subscriber i in the list of subscribers:
    (a) Generate a new message id mid_i, and queue up a new message $a_i = (mid_i, a.t, a.p, qos_i)$ for delivery, where qos_i is the QoS level in which the subscription was issued.
5. Send PUBACK(mid) to x.
6. Remove a from the list of stored messages.

4.2.2.3 QoS 2 Message Flow

In this QoS level, a publisher x sends PUBLISH(mid, t, p, qos) to the broker, and stores the message $a_x = (mid, t, p, qos)$. Upon receiving PUBLISH(mid, t, p, qos), the broker stores the message $a = (mid, t, p, qos)$ in a list of stored messages, and sends PUBREC(mid) to x. In response to PUBREC(mid), x sends PUBREL(mid) to the broker. Upon receiving this last message, the broker verifies for subscriptions to topic t, and notifies the corresponding subscribers. After having notified all subscribers, the broker sends PUBCOMP(mid)

to x and proceeds to discard a from the list of stored messages. When x receives the PUBCOMP(*mid*), it proceeds to discard a_x.

Internally, the Mosquitto broker performs the following tasks to comply with the protocol:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.
2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.
3. Send PUBREC(*mid*) to x.
4. Upon receiving PUBREL(*mid*) from x. Look for subscriptions to the topic t, and create a list of subscribers.
5. For each subscriber i in the list of subscribers:
    (a) Generate a new message id mid_i, and queue up a new message $a_i = (mid_i, a.t, a.p, qos_i)$ for delivery, where qos_i is the QoS level in which the subscription was issued.
6. Send PUBCOMP(*mid*) to x.
7. Remove a from the list of stored messages.

4.3 Implementation

The implementation of the security mechanisms is presented in the following sections. §4.3.1 discusses how Scoping is implemented on top of the Mosquitto Bridging facilities. §4.3.2 describes the Execution Monitoring Framework implemented in Mosquitto to support custom made execution monitors. §4.3.3 illustrates the modifications done to the Mosquitto Message Flow to support execution monitoring. Finally, §4.3.4 shows

one example execution monitor implemented through the Execution Monitoring Framework. Throughout this section, **MOSQ_HOME** is used to refer to the base directory of the Mosquitto source code.

4.3.1 Implementing Scoping

For this extension of Mosquitto, the visibility of messages is based on the scoping and visibility definition given by Fiege *et al.* [17, 19]. As mentioned in §3.4.1.2, in this model, a hierarchy of brokers (or scopes) is specified though a parenthood relation, $parent \subseteq \mathcal{E} \times \mathcal{E}$, such that $parent(x,y)$ implies that y is a parent of x.

To simulate this parenthood relation in Mosquitto, the convention is that, given two interconnected brokers x and y, if x identifies y as a bridge, then y is a parent scope of x, which denotes the relation $parent(x,y)$. Devices are said to be children of the scope they are connected to (due to a subscription or publication of events). Once this relationship has been established, it is possible to identify four types of event propagation: **parent-parent** , **parent-child** , **child-parent** , and **child-child.**

1. **parent-parent**. The source of the input event is a parent scope, and the target is another parent scope.

2. **parent-child**. The source of the input event is a parent scope, and the target is a child scope/device.

3. **child-parent**. The source of the input event is a child scope/device, and the target is a parent scope.

4. **child-child**. The source of the input event is a child scope/device, and the target is another child scope/device.

To comply with the visibility rule impose by Fiege *et al.* [17, 19] (thats is, messages published in a broker x are visible in another broker y, if and only if x and y share a common

```
struct execution_monitor {
        void *state;
        enum exec_monitor_direction direction;
        int (*delta_function)(struct em_event_queue *queue,
           void **state, struct exec_monitor_event *event);
        char *name;
};
```

Figure 4.1: Execution Monitor Structure

ancestor), it was necessary to prevent event propagations of the type **parent-parent**.

The source file **MOSQ_HOME/src/subs.c** in Mosquitto is intended to process subscriptions made by clients in the server. In the final stage of the queuing process, after processing a new incoming event to the server, the function **_subs_process** in this source file enqueues events to all possible recipients, including subscribers and parent scopes. In this function two critical pieces of information can be found: 1) the id of the source entity of the event (source id), and 2) the list of recipients.

Using the source id and each recipient in the list of recipients, it was possible to identify the type of event propagation to be done per recipient. Whenever a **parent-parent** event propagation type was inferred, the queuing process for that particular recipient is canceled.

4.3.2 Implementing Execution Monitoring

To implement execution monitoring, a framework to support custom made execution monitors was embedded into Mosquitto. Users of this framework can implement their own execution monitors and configure what links are to be monitored with them. Execution monitors are implemented as execution monitor modules, which are small C programs (usually called shared libraries) loaded via dynamic linking into the framework.

In this framework, execution monitors are represented by the C structure shown in Figure 4.1, where *state* represents the current state of the execution monitor, *direction* is the direction in which the execution monitor is to be executed, *delta_function* is the transition

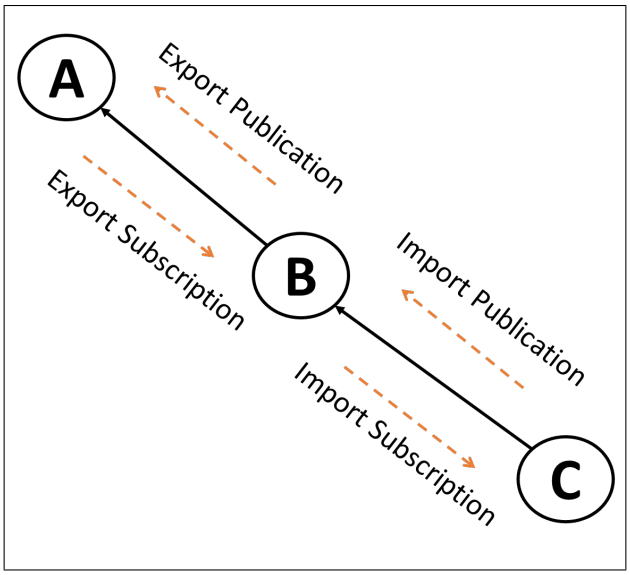

Figure 4.2: Events Flow Directions

function of the execution monitor, and *name* is a label which associates the current execution monitor with one execution monitor type (used when configuring execution monitors in the configuration file).

Execution Monitors are loaded from user created modules and associated with connection from one "working" broker, to any other entity the broker is interacting with. Consider for example Figure 4.2, where B represents the "working" broker. From this figure, it is possible to observe four different Event Flow Directions with respect to B:

Import Publication (im_pub). An event is received by B from a child entity.

Import Subscription (im_sub). An event is propagated from B to a child entity.

Export Publication (ex_pub) An event is propagated from B to a parent broker.

Export Subscription (ex_sub) An event is received by B from a parent broker.

These are the possible direction used in an execution monitor.

When an event is propagated between the working broker and any other entity, if there is an execution monitor associated with such connection and the direction of the execu-

```
struct em_vtable {
        int (*q_add)(struct em_event_queue *queue, struct
           exec_monitor_event *event);
        struct exec_monitor_event* (*q_peek)(struct
           em_event_queue *queue);
        struct exec_monitor_event* (*q_pop)(struct
           em_event_queue *queue);
        void* (*em_malloc)(size_t size);
        void (*em_free)(void *mem);
};
```

Figure 4.3: Framework Functions

tion monitor matches the current Event Flow Direction, then the transition function of the execution monitor is called.

The transition function receives the current state q of the execution monitor, an input event a, and a queue of events as inputs. The queue of events is used by the execution monitor to represent the output sequence w of the transition function, and it is initiated with the input event a as its first element (e.g. $w = a$). The transition function is responsible of updating the current state q of the monitor, to the next state q'. This represents the transition function $\delta(q,a) = (q',w)$.

The framework makes available the set of functions shown in Figure 4.3, where *q_add* is used to enqueue events, *q_peek* is used to look at the event at the front of the queue, *q_pop* is used to retrieve and remove the event at the front of the queue, and *em_malloc* and *em_free* are used for memory management purposes.

After having created one execution monitor module, users can use the extended configuration file of Mosquitto to load the module and associate it with any number of connections with entities. Figure 4.4 illustrate the new entries added to the configuration file as part of the framework. The **monitor_module** option is used to load the execution monitor module *module_full_path* with the name *execution_monitor_name*. The option **monitor** is used to associate an execution monitor of type *execution_monitor_name* with the entity *entity_id*,

```
monitor_module execution_monitor_name module_full_path
monitor entity_id execution_monitor_name direction
    configuration_file
default_monitor execution_monitor_name direction
    configuration_file
```

Figure 4.4: Extended Mosquitto Configuration File Entries

```
struct export_vtable {
        struct execution_monitor* (*constructor)(const char
            *monitor_name, enum exec_monitor_direction
            direction, const char *conf_file);
        int (*destructor)(const char *monitor_name, struct
            execution_monitor **monitor);
        int (*load_functions)(struct em_vtable *em_funcs);
};
```

Figure 4.5: Execution Monitor Functions

the Event Flow Direction *direction*, and the configuration file *configuration_file*. Finally, default monitors can be configured in the framework, so that entities with no specific execution monitors configured will be associated with the monitor set by the **defaul_monitor** entry.

Finally, to enable the framework to initiate the configured execution monitors, the execution monitor modules must make available a set of functions as describe in Figure 4.5. The function *constructor* is used to initiate a new instance of the execution monitor, and the function *destructor* is used to destroy one instance. Finally, the function *load_functions* is used to load the queueing and memory function of the framework into the execution monitor module.

4.3.3 Extended Mosquitto Message Flow

In order to support events queuing with execution monitoring, some modifications were implemented in the Mosquitto Message Flow for all QoS levels. We use the lists of tasks given in §4.2.2, and the notation described in §3.3 to illustrate the modifications included

in the extended version of Mosquitto.

4.3.3.1 Extended QoS 0 Message Flow

In the list of tasks shown in this section, the modifications to the original message flow for QoS 0 are described, and it is assumed that **source_id** is the id of the entity who initiated the message flow by "publishing" an event in the Mosquitto Server, and **broker_id** is the id of the Mosquitto Server. The extended version of Mosquitto performs the following tasks in order to comply with the QoS 0 protocol and include execution monitoring:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.
2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.
3. Let $\delta = \Delta(source\_id, broker\_id)$, and $(q, w) = \delta(ST(source\_id, broker\_id), a)$.

   Then $ST'(source\_id, broker\_id) = q$ is the new state of the execution monitor $M(source\_id, broker\_id)$.
4. For each event b in the sequence w (in a FIFO fashion):
   (a) Look for subscriptions to the topic $b.t$, and create a list of subscribers.
   (b) For each subscriber i in the list of subscribers:
      i. Let $\delta = \Delta(broker\_id, i)$ and
      $(q_i, u_i) = \delta(ST(broker\_id, i), b)$.
      Then $ST'(broker\_id, i) = q_i$ is the new state of the execution monitor $M(broker\_id, i)$
      ii. For each event $c = (mid_c, c.t, c.p, qos_{ib})$ in the sequence u_i (in a FIFO fashion), where mid_c is a new message id generated for c, and qos_{ib}

is the QoS level in which subscriber i issued a subscription to event b:

A. Queue up c for delivery.

5. Remove a from the list of stored messages.

In general terms, the modifications to the message flow of Mosquitto for QoS 0, involves interposing execution monitors between critical operations of the queuing process, so as to enqueue the resulting events for delivery to the corresponding recipients.

4.3.3.2 Extended QoS 1 Message Flow

In the list of tasks shown in this section, the modifications to the original message flow for QoS 1 are described under the same assumptions than in the previous section. The extended version of Mosquitto performs the following tasks in order to comply with the QoS 1 protocol and include execution monitoring:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.
2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.
3. Let $\delta = \Delta(source\_id, broker\_id)$, and $(q, w) = \delta(ST(source\_id, broker\_id), a)$.

   Then $ST'(source\_id, broker\_id) = q$ is the new state of the execution monitor $M(source\_id, broker\_id)$.
4. For each event b in the sequence w (in a FIFO fashion):

   (a) Look for subscriptions to the topic $b.t$, and create a list of subscribers.

   (b) For each subscriber i in the list of subscribers:

   i. Let $\delta = \Delta(broker\_id, i)$ and $(q_i, u_i) = \delta(ST(broker\_id, i), b)$.

Then $ST'(broker\_id, i) = q_i$ is the new state of the execution monitor $M(broker\_id, i)$

ii. For each event $c = (mid_c, c.t, c.p, qos_{ib})$ in the sequence u_i (in a FIFO fashion), where mid_c is a new message id generated for c, and qos_{ib} is the QoS level in which subscriber i issued a subscription to event b:

A. Queue up c for delivery.

5. Send PUBACK(mid) to x.

6. Remove a from the list of stored messages.

The modifications in the QoS 1 message flow, are identical to those made to the QoS 0 message flow. This is because both of the message flows share most of their code, being the only difference the PUBACK control packet sent as acknowledgment in the QoS 1 message flow.

4.3.3.3 QoS 2 Message Flow

In the list of tasks shown in this section, the modifications to the original message flow for QoS 2 are described under the same assumptions than in the previous sections. The extended version of Mosquitto performs the following tasks in order to comply with the QoS 2 protocol and include monitoring:

1. Read the incoming PUBLISH(mid, t, p, qos) message from x.

2. Store the message $a = (mid, t, p, qos)$ into a list of stored messages.

3. Send PUBREC(mid) to x.

4. Upon receiving PUBREL(mid) from x. Let $\delta = \Delta(source\_id, broker\_id)$, and $(q, w) = \delta(ST(source\_id, broker\_id), a)$.

Then $ST'(source\_id, broker\_id) = q$ is the new state of the execution monitor $M(source\_id, broker\_id)$.

5. For each event b in the sequence w (in a FIFO fashion):

    (a) Look for subscriptions to the topic $b.t$, and create a list of subscribers.

    (b) For each subscriber i in the list of subscribers:

        i. Let $\delta = \Delta(broker\_id, i)$ and $(q_i, u_i) = \delta(ST(broker\_id, i), b)$. Then $ST'(broker\_id, i) = q_i$ is the new state of the execution monitor $M(broker\_id, i)$

        ii. For each event $c = (mid_c, c.t, c.p, qos_{ib})$ in the sequence u_i (in a FIFO fashion), where mid_c is a new message id generated for c, and qos_{ib} is the QoS level in which subscriber i issued a subscription to event b:

            A. Queue up c for delivery.

6. Send PUBCOMP(mid) to x.

7. Remove a from the list of stored messages.

In general terms, the modifications to the message flow of Mosquitto for QoS 2 also involve interposing execution monitors between critical operations of the queuing process. However, in this case, it is necessary to take into account the intermediary control packets (e.g., PUBREC and PUBREL) that have to be exchanged between the broker and the source of the input event. Neglecting reception or delivery of these control packets, could result in events not being propagated properly.

```
map topic1 new_topic1
map topic2 new_topic2
map topic3 new_topic3
```

Figure 4.6: Event Mapping Configuration File

4.3.4 Example Execution Monitor Module

This section shows a brief example of how the framework of §4.3.2 can be used to implement one execution monitor based on Edit Automata. This example is not meant to be extensive, that is, not all the source code for the an actual implementation is provided.

Consider the Event Mapping Edit Automaton described in §3.4.2.2 with a transition function $mapper(f) : Q \times \Sigma \rightarrow Q \times \Sigma^*$ such that $mapper(f)(q_0, a) = (q_0, f(a))$. This Edit Automaton remains in the initial state q_0 at all time, however it is necessary to describe the event transformation function f. For that, it is possible to define the event transformation function in terms of the configuration file shown in Figure 4.6, where each entry represents the transformation from one event topic $topicX$, to a new event topic new_topicX.

The Execution Monitor Module constructor initiates all instances of the Event Mapping Execution Monitor with a NULL state (since the current state of the monitor is not actively used), and loads the configuration file to create a look up table T, such that each entry $T[topicX] = new\_topicX$ represents a transformation from $topicX$ to new_topicX. The destructor function frees up the memory used for the look up table.

The transition function, given an input event $a = (mid, topic, payload, qos)$, input state NULL, and an input queue representing the sequence $w = a$, can simply check if there exist an entry $T[topic]$ in the look up table. In case that such entry exists, the output sequence $w = a$ is replaced by $w = b$, such that $b = (mid, T[topic], payload, qos)$, this is, b is a copy of a with a new topic $T[topic]$.

Figure 4.7 illustrates a snippet of the source code used to implement the transition function defined in the previous paragraph. Notice that the structure *em_funcs* holds the

references to the functions provided by the framework. Finally, in this example, the lookup table is explicitly defined in the transition function, instead of being initialized based on a configuration file.

```
/*Lookup table structure definition*/
typedef struct {
    char *from_topic;
    char *to_topic;
} lookupTable;


/* The transition function */
int delta_function(struct em_event_queue *queue, void **state, struct exec_monitor_event *
    event){
    lookupTable T[] = {
        {"topic1", "new_topic1"},
        {"topic2", "new_topic2"},
        {"topic3", "new_topic3"},
    };
    int numRecords = 3;
    int i;
    int topiclen;
    struct exec_monitor_event *new_event = NULL;
    struct exec_monitor_event *tmp_event = NULL;

    for(i = 0; i < numRecords ; i++){
        //If there exists a map for the topic of the event
        if(!strcmp(T[i].from_topic, event->topic)){

            //Allocate memory for the mapped version of the event
            new_event = em_funcs->em_malloc(sizeof(struct exec_monitor_event));

            //Map the old topic using the lookup table
            topiclen = strlen(T[i].to_topic);
            new_event->topic = em_funcs->em_malloc(sizeof(char) *  (topiclen + 1));
            memset(new_event->topic, 0, sizeof(char) * (topiclen + 1));
            strcpy(new_event->topic, T[i].to_topic);

            //Copy the event of the input event to the mapped version of the event
            new_event->payload = em_funcs->em_malloc(event->payloadlen);
            memset(new_event->payload, 0, event->payloadlen);
            memcpy(new_event->payload, event->payload, event->payloadlen);
            new_event->payloadlen = event->payloadlen;

            //Remove original event from the queue and free up the corresponding memory
            tmp_event = em_funcs->q_pop(queue);
            if(tmp_event->topic) em_funcs->em_free(tmp_event->topic);
            if(tmp_event->payload) em_funcs->em_free(tmp_event->payload);
            if(tmp_event) em_funcs->em_free(tmp_event);

            //Add mapped event to the queue
            em_funcs->q_add(queue, new_event);

            return 0;
            }
    }
    return 0;
}
```

Figure 4.7: Transition function source code

Chapter 5

Performance Evaluation of Security Mechanisms

In this chapter, I present an empirical study conducted to measure the impact that scoping and execution monitoring have on the performance of event based systems. In similar studies [10] [21], two of the most standard measurements used to compare performance in networking technologies, are latency and message throughput. Latency is the time it takes to deliver one message from one designated point to another (e.g. from one publisher to one subscriber), whereas message throughput is the rate in which messages are delivered (e.g., number of messages per second), also from one designated point to another.

This study was performed using an open source implementation of the MQTT protocol, namely, Mosquitto [25]. The performance of Mosquitto was evaluated using six different configurations, and was measured in terms of message throughput. Message throughput was used as the standard measurement due to the following reasons: 1) Message throughput is commonly used to establish requirements and limits for IoT technologies [1][5], 2) preliminary experiments conducted to measure latency shown negligible differences between the original version of Mosquitto, and the extended version with the security mechanisms, 3) these latency experiments also shown to be very susceptible to latency peaks, which resulted in inconsistent results, sometimes even favoring the extended version of Mosquitto. Additionally, in similar experiments conducted by Babovic *et al.* [10], they recognize the importance of measuring message throughput to minimize error.

To measure message throughput, one experiment was performed under three different scenarios, and each scenario was executed for six rounds. Additionally, the experiment was repeated once per Mosquitto configuration. This chapter introduces some preliminaries in §5.1, where the configurations for Mosquitto are described. The experimental setup

is detailed in §5.2, including the description of the scenarios and rounds, as well as the hardware and software setup. Finally, results are discussed in §5.3.

5.1 Preliminaries

In the MQTT protocol, a topic-based subscription system is used [15], where subscribers demonstrate their interest on particular events based on a topic (or subject). Furthermore, events, are mainly regarded as a combination of a topic and a payload, where the payload contains the actual application data. For example, a publisher could publish the current temperature of the environment using the topic "current_temperature" and payload "15", which would imply a current temperature of 15 degrees.

In my experiment, I used four different execution monitors which read one input event a, and output a sequence of events w. The execution monitors used are: **simple_em, complex_em, hsup_em, and mutli_em.**

simple_em This execution monitor reads the topic of the input event a and outputs a sequence w consisting of one single event b, where the payload of b is a copy of the payload of a. The topic of b is obtained by mapping the topic of a using a lookup table. This execution monitor is used as a lower bound in terms of complexity. More specifically, the running time of this execution monitor is in constant time.

complex_em This execution monitor reads the input event a and outputs a sequence w consisting of one single event b. The topic of b is obtained by mapping the topic of a using a lookup table. The payload of b is generated byte by byte with random ASCII printable characters (0x20 to 0x7E), and has the same size than the payload of a. This execution monitor is used as an upper bound in terms of complexity. More specifically, the running time of this execution monitor is in polynomial time with respect of the size of the payload.

hsup_em This execution monitor suppresses every other input event a based on its current state. Particularly, the execution monitor has two possible states: "allow" and "suppress". Upon receiving the input event a, if the monitor is in state "allow", it will output a sequence $w = a$, and switch to state "suppress". On the other hand, if the monitor is in state "suppress", it will suppress the event a and switch to state "allow".

multi_em This execution monitor receives an input event a and duplicates it to produce the output sequence $w = aa$.

The experiment was conducted using six different configurations of Mosquitto. In the following section these configurations are described.

5.1.1 Mosquitto Configurations

In this study, two versions of Mosquitto are used: 1) The original Mosquitto without security mechanisms, and 2) The extended version of Mosquitto (see chapter 4) with security mechanisms.

The configurations of Mosquitto used during the experiment are defined by the version of Mosquitto, and the execution monitor (if any) used. Each configuration is also related to a factor Z, which represents the ratio of output events to input events for the execution monitor used for that configuration. For example, the execution monitor **simple_em** maps one input event to a sequence of only one event, which represents an output event to input event ratio of $Z = 1 \div 1 = 1$. However, in the case of the execution monitor **multi_em**, every input event is mapped to a sequence of two events, which represents an output event to input event ratio of $Z = 2 \div 1 = 2$. For configurations with no execution monitors, the factor Z is defined to be 1.

The configurations of Mosquitto used during the experiment are:

- Mosquitto without security mechanisms (ORI), with $Z = 1$.

- Mosquitto with security mechanisms but no execution monitors modules loaded (NEW), with $Z = 1$.
- Mosquitto with security mechanisms, using the execution monitor **simple_em** (SEM), with $Z = 1$.
- Mosquitto with security mechanisms, using the execution monitor **complex_em** (CEM), with $Z = 1$.
- Mosquitto with security mechanisms, using the execution monitor **hsup_em** (HSUP), with $Z = 0.5$.
- Mosquitto with security mechanisms, using the execution monitor **multi_em** (MULTI), with $Z = 2$.

5.2 Experimental Setup

The experiment was divided into three scenarios, and each scenario is executed for six rounds per Mosquitto configuration. Scenarios are illustrated in §5.2.1, where each scenario describes a different connectivity configuration between publishers, subscribers and scopes. Rounds represent the rate at which messages are published in the scopes, and are discussed in §5.2.2. Finally, §5.2.3 and §5.2.4 describes the hardware and software setup respectively.

5.2.1 Scenarios

As mentioned earlier, the experiment was divided into three scenarios: one scope, three scopes and five scopes. The purpose of these scenarios, is to measure the impact that an increasing the number of interconnected scopes have in terms of message throughput. In all scenarios, all publishers publish messages to one scope (PBroker), and all subscribers subscribe to events in another Scope (SBroker).

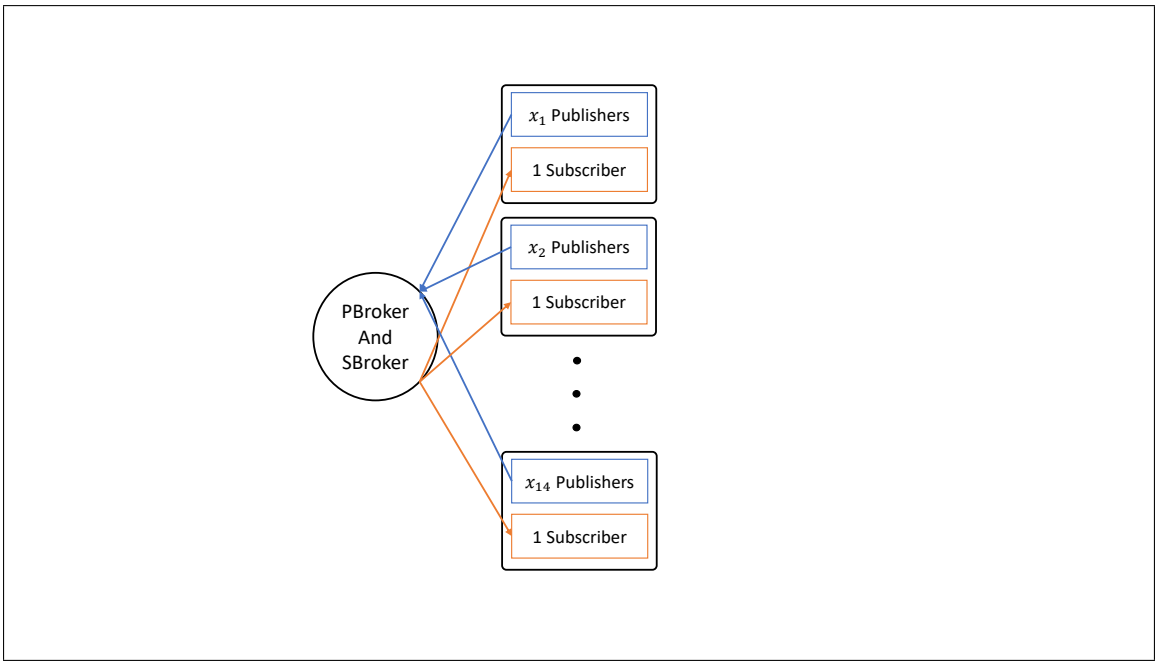

Figure 5.1: Scenario 1 - 1 Scope

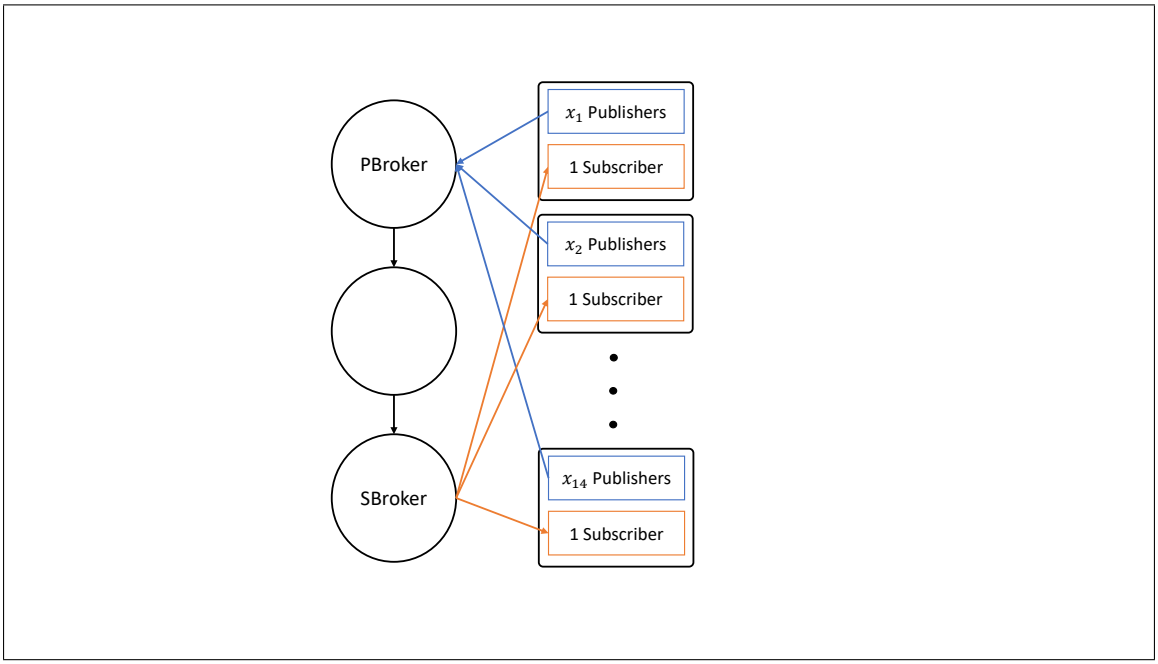

Figure 5.2: Scenario 2 - 3 Scopes

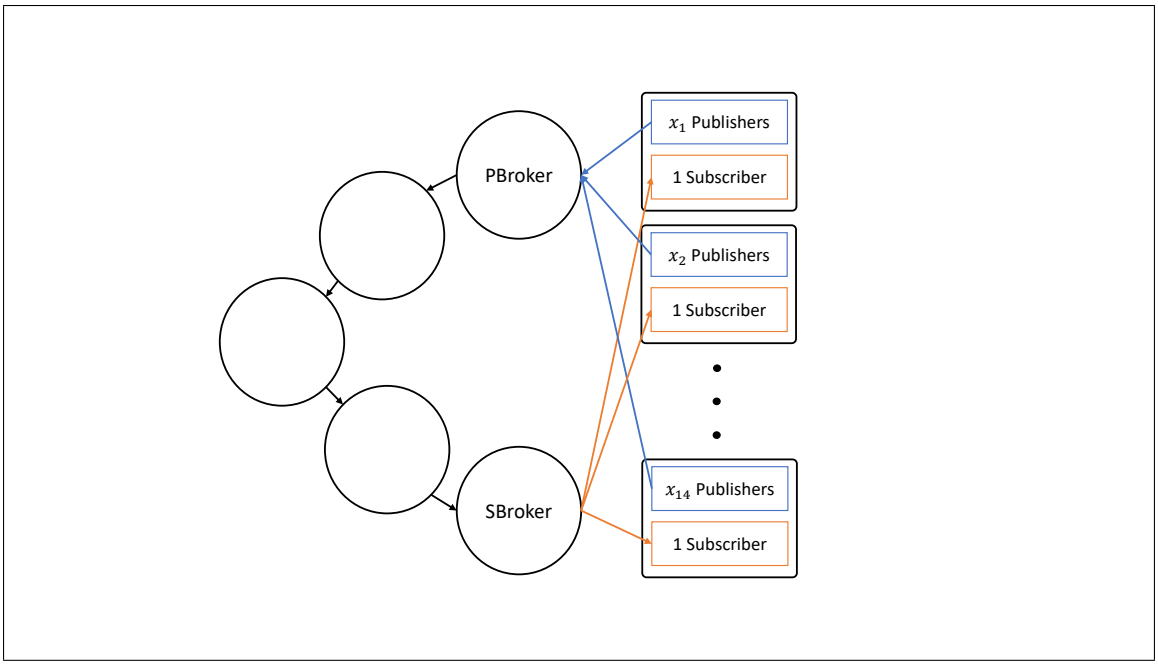

Figure 5.3: Scenarios 3 - 5 Scopes

Publishers and subscribers are spawned in fourteen different machines, where each machine launches one single subscriber, and an increasing number of publishers. In the following paragraphs, scenarios 1, 2 and 3 are described.

Scenario 1. Figure 5.1 illustrates scenario 1, where only one scope is used to relay events from publishers to subscribers. Publishers publish events in this scope, and subscribers are the recipients of such events, in such a way that each event is received by only one subscriber. The total number of publishers launched per machine is described in §5.2.2.

Scenario 2. Figure 5.2 illustrates scenario 2, where three scopes are used to relay events from publishers to subscribers. Publishers publish events in scope PBroker, and subscribers subscribe to events in SBroker, in such a way that each event is received by only one subscriber. The total number of publishers launched per machine is described in §5.2.2.

Scenario 3. Figure 5.3 illustrates scenario 3, where five scopes are used to relay events

from publishers to subscribers. Publishers publish events in scope PBroker, and subscribers subscribe to events in SBroker, in such a way that each event is received by only one subscriber. The total number of publishers launched per machine is described in §5.2.2.

In figures 5.1, 5.2, and 5.3, arrows represent the flow of events. Additionally, when using configurations with execution monitors (SEM, CEM, HSUP and MULTI), only one execution monitor per scope is configured.

5.2.2 Rounds

Scenarios were divided into rounds, and in each round a fixed number of events per second were published for a period of 10 minutes. The term ***Message Publish Rate*** is used to refer to the rate in which messages are published in one round. For simplicity purposes, the following abbreviations may be used throughout the rest of the chapter:

- **MPR:** Message Publish Rate.
- **mps:** Messages per second.
- **MT:** Message Throughput.

During each round, a different number of messages per second is published. The MPR in rounds 1, 2, 3, 4, 5, and 6 is 5K, 10K, 15K, 20K, 25K, and 30K mps respectively.

In all scenarios, fourteen machines were used to spawn publishers and subscribers. Each machine spawned an increasing number of publishers per round, and only one subscriber. The total number of publishers launched per machine was calculated according to the MPR of each round, so that each publisher would publish one message per second. For example, if the goal is to publish a total of 5K mps, then a total of (aprox.) 5000/14 publishers were spawned per machine.

Finally, in all instances of the experiment, a payload size of 175 bytes was used. This payload size was chosen to represent a payload carrying sensor data. One example of an MQTT message carrying information about the status of one battery can be found in [3], with a payload size of 97 bytes (97 characters). A size of 175 bytes was used as an upperbound.

5.2.3 Hardware Setup

In this study, fourteen machines were used to launch publishers and subscribers, and 5 machines were used to run Mosquitto Servers. These machines are virtual machines hosted in the following hardware:

- Chasis: IBM BladeCenter H type
- Storage: SAN = 600 GB + 300 GB + 2TB
- Visualization Software: WMware ESXi, 4.1.0, 800380
- Processor Type: Intel(R) Xeon(R) CPU X5660 @ 2.80GHz
- CPU Cores: 12 CPUs x 2.8 GHz

All virtual machines have the following specifications:

- OS: Centos release 6.9 (Final)
- CPU: Intel(R) Xeon(R) CPU X5660 @ 2.80GHz (1 core)
- Memory: 8 GB
- Disk Space: 14 GB

5.2.4 Software Setup

The Software used during the experiment is described in this section.

Mosquitto 1.4.10 (original): Open Source MQTT Server implementation. Used for Mosquitto ORI configuration.

Mosquitto 1.4.10 (with security mechanisms): Open Source MQTT implementation with added security mechanisms (Execution Monitors infrastructure). This is the extended version of Mosquitto implemented for this research project, and described in chapter 4. Used for Mosquitto NEW, SEM, CEM, HSUP and MULTI configurations.

Mzbench [7]: Load Testing Tool where users are able to write benchmarking scenarios for testing applications with different protocols. These scenarios use workers (smaller applications written in Erlang or Python) to perform the benchmarking tasks.

vmq_mzbench [9]: Mzbench worker for the MQTT protocol. Publishers and Subscribers used in the experiment are Mzbench scenarios which use the vmq_mzbench worker.

5.3 Results

In this section, the results of the experiment are presented in two different analysis. The first analysis corresponds to the ORI, NEW, SEM and CEM Mosquitto configurations, and the second one corresponds to the HSUP and MULTI Mosquitto configurations.

In the first analysis, I included comparable configurations of Mosquitto in terms of message throughput, whereas the second analysis two non-comparable configurations of Mosquitto are considered. To explain what comparable means in this context, the term ***Calculated Message Throughput*** is introduced in the following section.

5.3.1 Calculated Message Throughput

Calculated Message Throughput (CMT), is a calculated value, used as an upperbound in terms of Message Throughput for each Configuration/Scenario/Round combination. This value offers perspective to the results, and represents the maximum possible message throughput obtained under the assumptions that no network delay is present, and that events are processed by scopes instantaneously.

To illustrate, consider the configuration MULTI with a factor $Z = 2$, in the scenario 2 with 3 scopes, and round 3 with a MPR of 15K mps. Ideally, if the scopes executed actions instantaneously, and no network delay was present, it would mean that events are delivered from publishers to subscribers immediately, passing through all scopes. In the configuration MULTI, each scope processes events using the execution monitor **multi_em**, which means that for every input event, a sequence of two events is produced per scope. Since three scopes are used in scenario 2, a total of $2 \times 2 \times 2 = 8$ events would be delivered per published event. Then, considering a MPR of 15K mps, in this case the Calculated Message Throughput would be $15\text{K} \times 2^3 = 120\text{K}$ mps.

Following this intuition, given the factor Z of one configuration, in a scenario with m scopes, and a round with a MPR of x mps. The CMT is calculated using the following formula:

$$CMT = x * Z^m$$

Two configurations are said to be comparable, if they have the same CMT to MPR ratio under all scenarios and rounds. This conditions essentially means that two configurations are comparable if the have the same factor Z. Additionally, two different scenario/configuration combinations are said to be comparable if they have the same CMT to MPR ratio in all rounds.

In the following section, a brief description of how Message Throughput is calculated

from the empirical data is given.

5.3.2 Message Throughput

To measure message throughput, publishers and subscribers were launched in fourteen different machines as described in §5.2.2. Since machines are independent from one another, the following formula is used to calculate the total message throughput. Let $t(s_i)$ represent the total number of messages delivered to the ith subscriber during one round (10 minutes), where $1 \leq i \leq 14$ (for a total of 14 subscribers, 1 per machine). The message throughput (MT) for that round is calculated as follows:

$$MT = \frac{\sum_{i=1}^{14} t(s_i)}{10 * 60}$$

All MT's have a corresponding CMT, and a MT is said to be an ***Optimum Message Throughput (OMT)***, if it is equal to its corresponding CMT.

To offer one point of comparison between different scenario/configuration combinations, the ***Maximum Message Throughput (MMT)*** is defined to be the highest Optimum Message Throughput obtained throughout all the rounds of one scenario with the same configuration.

5.3.3 Analysis 1 - Scoping and Execution Monitoring

The purpose of this analysis is to measure the impact in terms of message throughput that scoping and execution monitoring have on event based systems. In this analysis, I compare the message throughput of the Mosquitto with configuration ORI against the message throughput of Mosquitto using configurations NEW, SEM, and CEM.

Figures 5.4, 5.5, and 5.6 illustrate the results used for this analysis. Each figure shows the results for one scenario, where the x axis represents the Message Publish Rate (MPR) used in each round, and the y axis represents Message Throughput (MT). Points in the graphs represent the Message Throughput obtained using one particular MPR. In the fol-

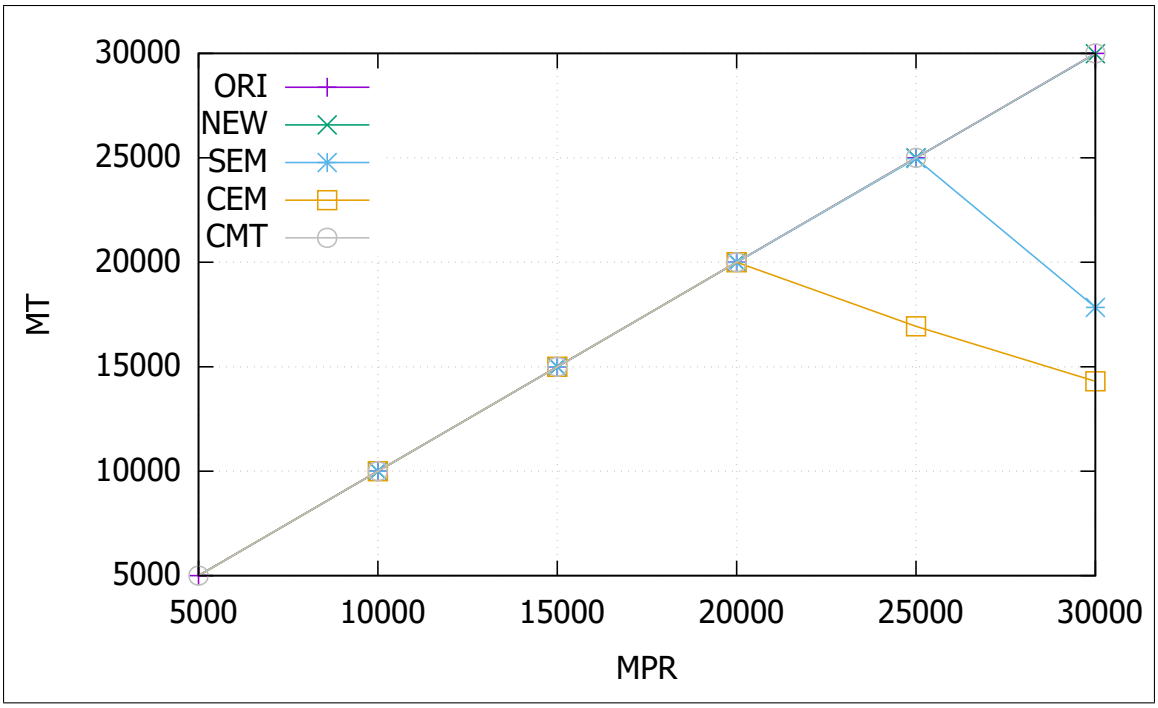

Figure 5.4: Experiment Results: Analysis 1 - Scenario 1

lowing paragraphs, each figure is analyzed independently and a summary of the most interesting remarks is given per figure. Conclusions for both analysis are left for §5.3.5.

Figure 5.4, shows the message throughput of configurations ORI, NEW, SEM and CEM in scenario 1. In this scenario, only one scope is used to relay events from publishers to subscribers. This scenario is used to minimize the impact of scoping in order to measure only the impact of execution monitoring. The results show that, when only one scope is used:

1. Using the original version of Mosquitto, it is possible to reach a Maximum Message Throughput of at least 30K mps (ORI).

2. Even after adding support for execution monitoring and scoping is added, the Maximum Message Throughput remains at 30K mps if none of these mechanisms is used (ORI vs NEW).

3. The impact of execution monitoring on message throughput depends on the

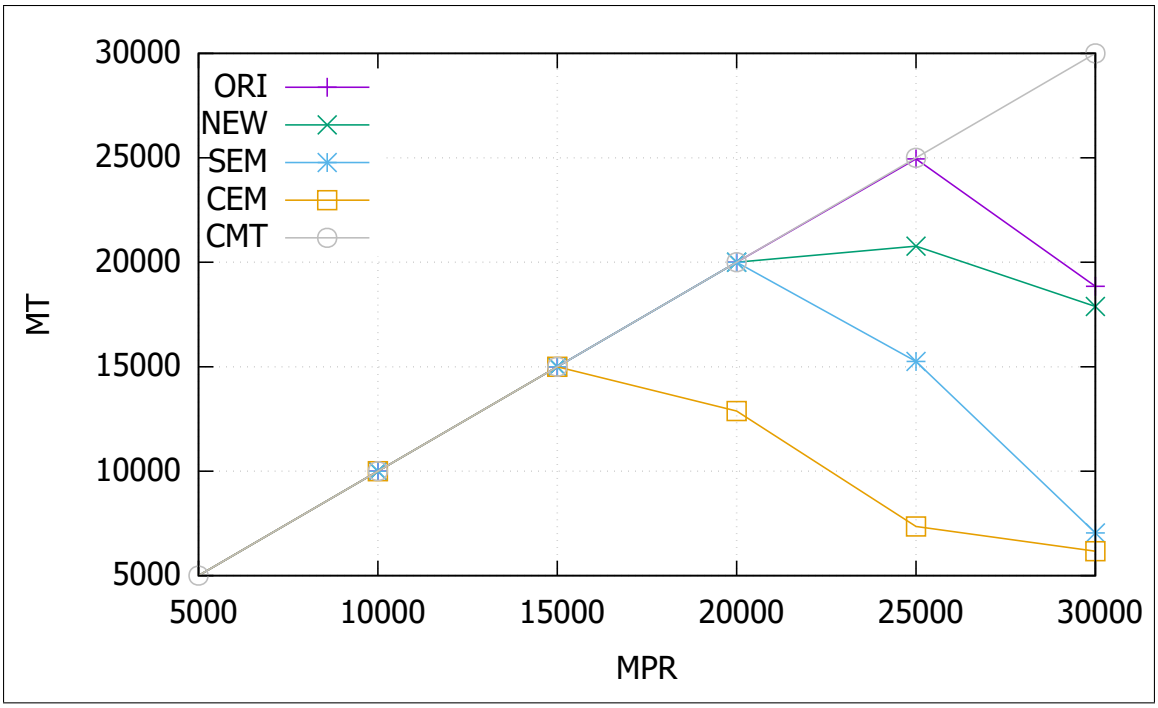

Figure 5.5: Experiment Results: Analysis 1 - Scenario 2

speed of the execution monitor used. This means, that the slower the execution monitor used is, the more impact in terms of message throughput it has (ORI vs CEM and SEM).

4. Even when execution monitors are used, it is possible to reach a Maximum Message Throughput of 20K mps (CEM).

Figure 5.5, shows the message throughput of configurations ORI, NEW, SEM and CEM in scenario 2. In this scenario, three scopes are used to relay events from publishers to subscribers. This scenario is used to measure the impact of execution monitoring on three interconnected scopes. It is worth to mention that in the case of configurations SEM and CEM, one execution monitor is used per scope. To interconnect brokers in configuration ORI, the bridging facility of Mosquitto described in §4.2.1 is used. The results show that, when three scopes are used:

1. A Maximum Message Throughput of 25K mps is reached in the original

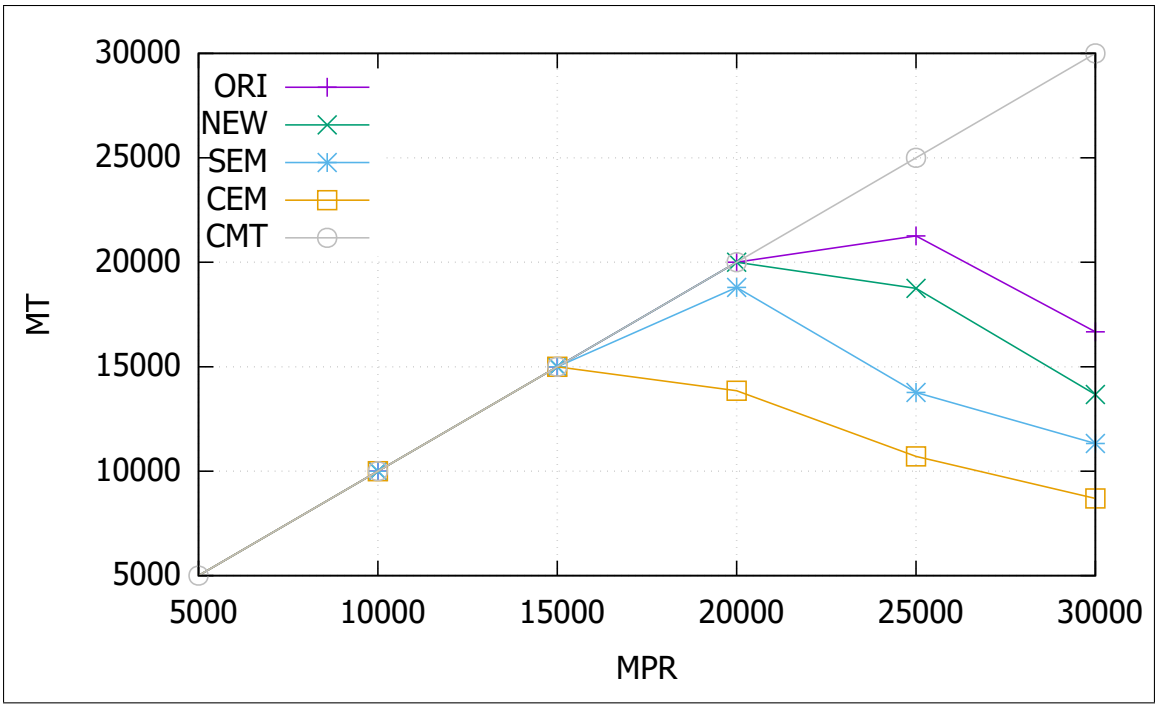

Figure 5.6: Experiment Results: Analysis 1 - Scenario 3

version of Mosquitto (ORI).

2. When support for execution monitoring and scoping is added, the Maximum Message Throughput drops from 25K mps to 20K mps (ORI vs NEW).

3. The impact of execution monitoring on message throughput depends on the speed of the execution monitor used. This means, that the slower the execution monitor used is, the more impact in terms of message throughput it has (ORI vs CEM and SEM).

4. Even when execution monitors are used, it is possible to reach a Maximum Message Throughput of 15K mps (CEM).

Figure 5.6, shows the message throughput of configurations ORI, NEW, SEM and CEM in scenario 3. In this scenario, five scopes are used to relay events from publishers to subscribers. This scenario is used to measure the impact of execution monitoring on five interconnected scopes. Juts like in scenario 2, in configurations SEM and CEM, one execution

monitor is used per scope, and bridges are used to interconnect brokers in configuration ORI. The results show that, when five scopes are used:

1. A Maximum Message Throughput of 20K mps is reached in the original version of Mosquitto (ORI).

2. When support for execution monitoring and scoping is added, the Maximum Message Throughput stays at 20K mps (ORI vs NEW).

3. The impact of execution monitoring on message throughput depends on the speed of the execution monitor used. This means, that the slower the execution monitor used is, the more impact in terms of message throughput it has (ORI vs CEM and SEM).

4. Even when execution monitors are used, it is possible to reach a Maximum Message Throughput of 15K mps (CEM).

5.3.4 Analysis 2 - Suppression and Multiplication of Events

The purpose of this analysis is to measure the impact that suppression and multiplication of events have in terms of message throughput. Intuitively, multiplication of events could result in over-flooding of the communication channels, which could potentially have a negative impact in message throughput. On the other hand, event suppression could help alleviate the traffic in one communication channel, favoring the transmission of events.

Figures 5.7, 5.8 and 5.9 show the results for event suppression, and figures 5.10, 5.11 and 5.12 show the results for event multiplication. For event suppression, the configuration HSUP was used, and the configuration MULTI was used for event multiplication.

In this set of results, the ratio of CMT to MPR depends on the number of execution monitors used (1 per scope) and their factor Z. For example, in the case of MULTI with $Z = 2$, when one, three, and five execution monitors are used, the ratios of CMT to MPR

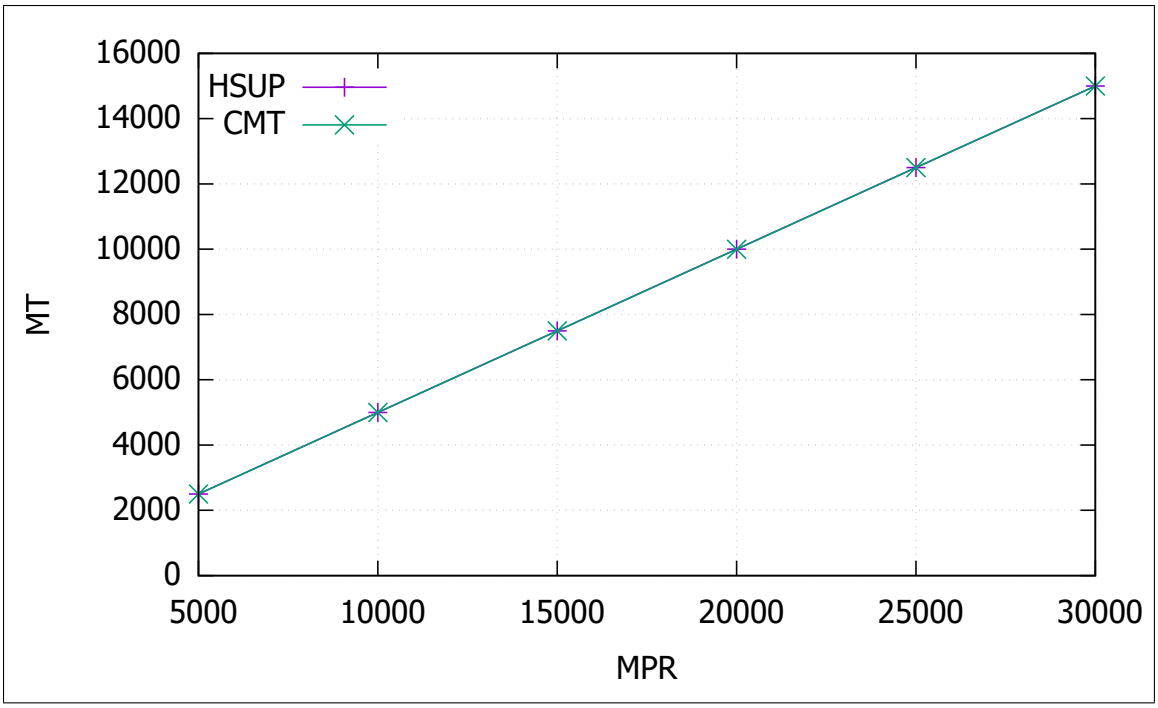

Figure 5.7: Experiment Results: Analysis 2 - Scenario 1, HSUP

are $2^1, 2^3$, and 2^5 respectively. This means that in this analysis, even scenarios using the same configuration are non-comparable, at least, not in terms of message throughput.

To add perspective to the results, the calculated message throughput (CMT) is used as reference in all figures. Each figure shows the results for one scenario/configuration combination, where the x axis represents the Message Publish Rate (MPR) used in each round, and the y axis represents Message Throughput (MT). Points in the graphs represent the Message Throughput obtained using one particular MPR.

A new term named ***Maximum Message Publish Rate (MMPR)*** is introduced as an alternative way to compare results. The MMPR is defined to be the MPR used to reach the Maximum Message Throughput in a scenario/configuration combination.

In the following paragraphs, important observations of each figure are highlighted, however, conclusions are left for §5.3.5.

Figures 5.7, 5.8 and 5.9, shows the relation between MT and CMT for configuration HSUP in scenarios 1, 2 and 3 respectively. In these figures, the following observations can

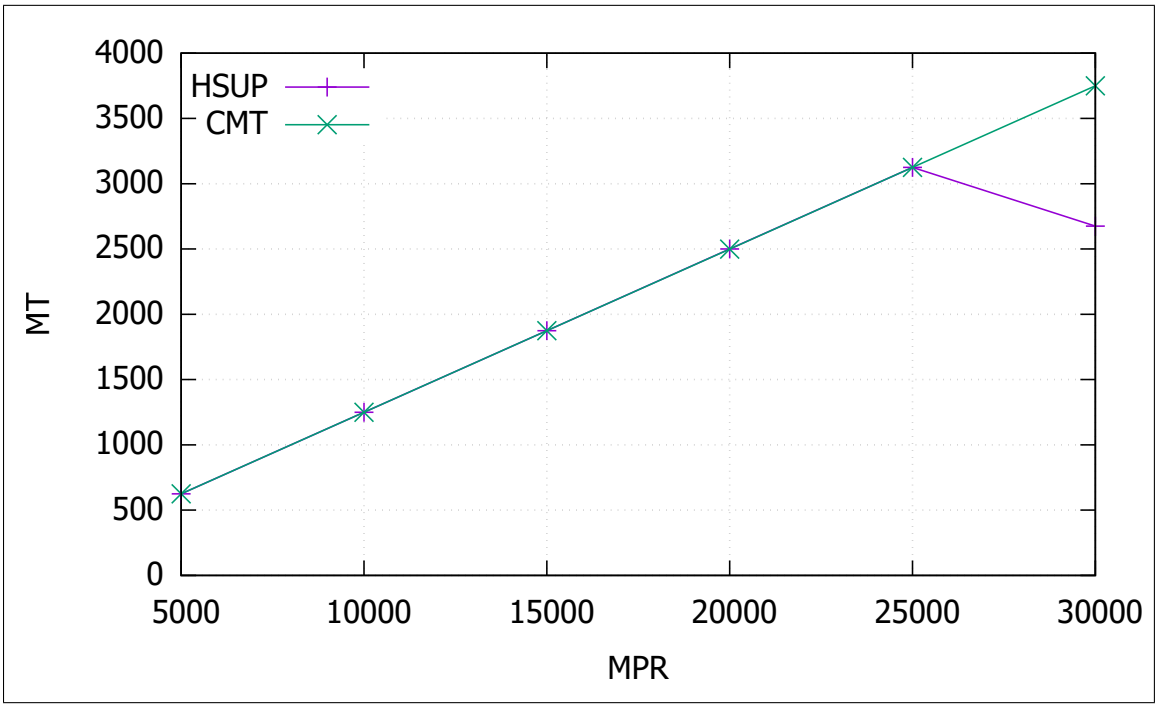

Figure 5.8: Experiment Results: Analysis 2 - Scenario 2, HSUP

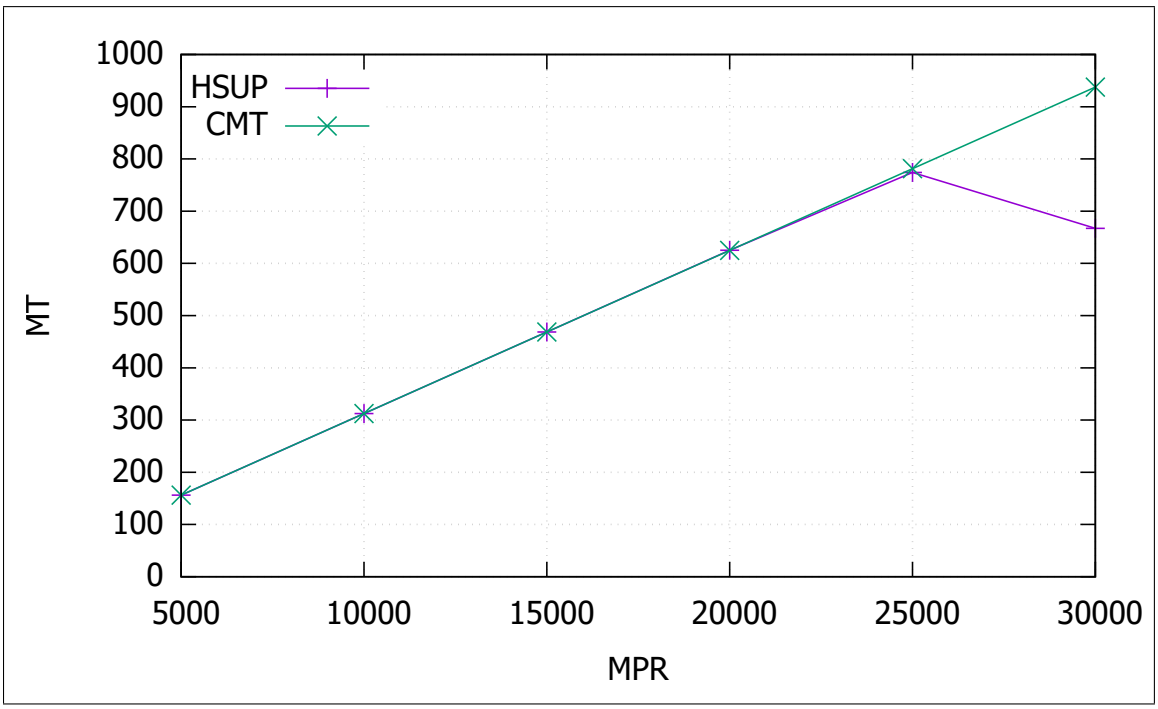

Figure 5.9: Experiment Results: Analysis 2 - Scenario 3, HSUP

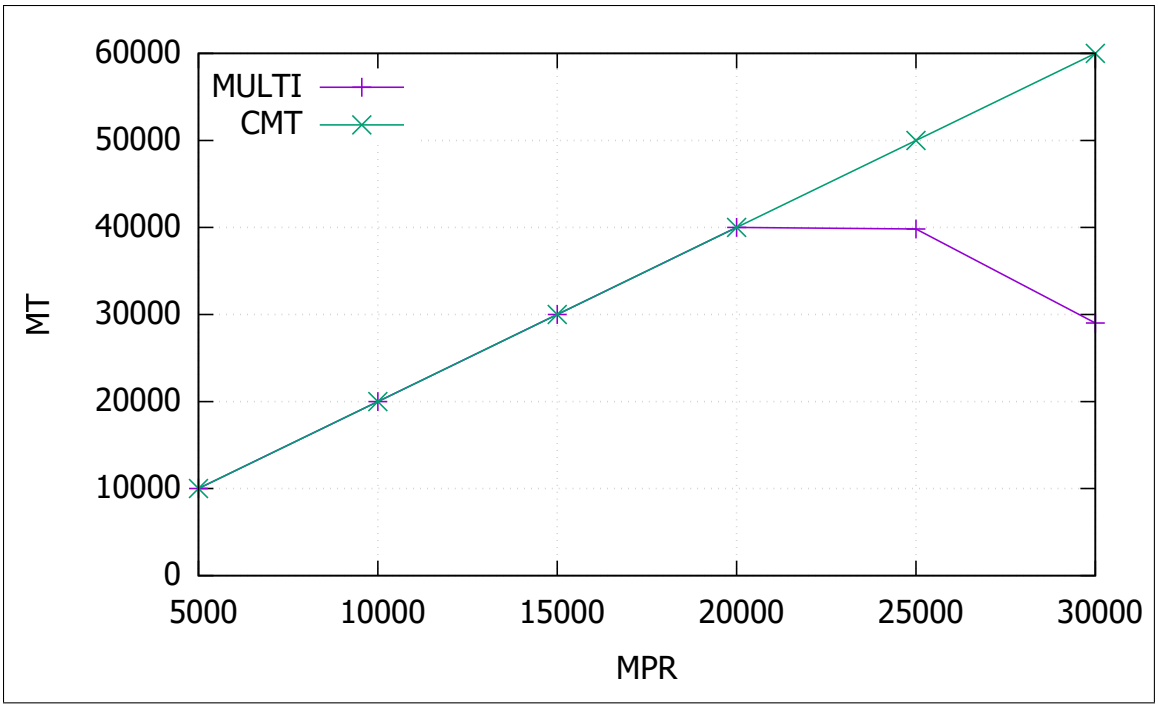

Figure 5.10: Experiment Results: Analysis 2 - Scenario 1, MULTI

be done:

1. Figure 5.7 shows that, throughout all rounds of scenario 1, it was possible to maintain an Optimum Message Throughput, despite of using execution monitoring. The MMPR in scenario 1 with configuration HSUP is 30K mps.

2. Figure 5.8 shows that, in most rounds of scenario 2, it was possible to maintain an Optimum Message Throughput, except for round 6 with a MPR of 30K mps. The MMPR in scenario 2 with configuration HSUP is 25K mps.

3. Figure 5.9 shows that, in most rounds of scenario 3, it was also possible to maintain an Optimum Message Throughput, except for rounds 5 and 6, with a MPR of 25K and 30K mps respectively. The MMPR in scenario 3 with configuration HSUP is 20K mps.

Figures 5.10, 5.11 and 5.12, shows the relation between MT and CMT for configuration MULTI in scenarios 1, 2 and 3 respectively. In these figures, the following observations

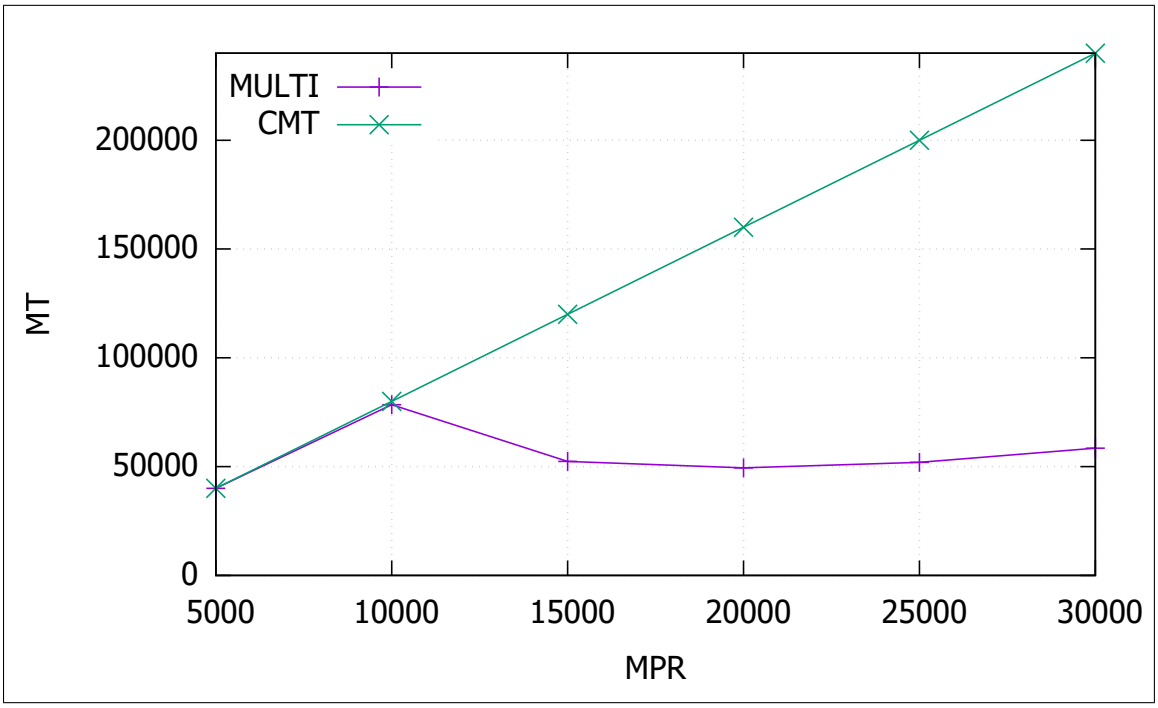

Figure 5.11: Experiment Results: Analysis 2 - Scenario 2, MULTI

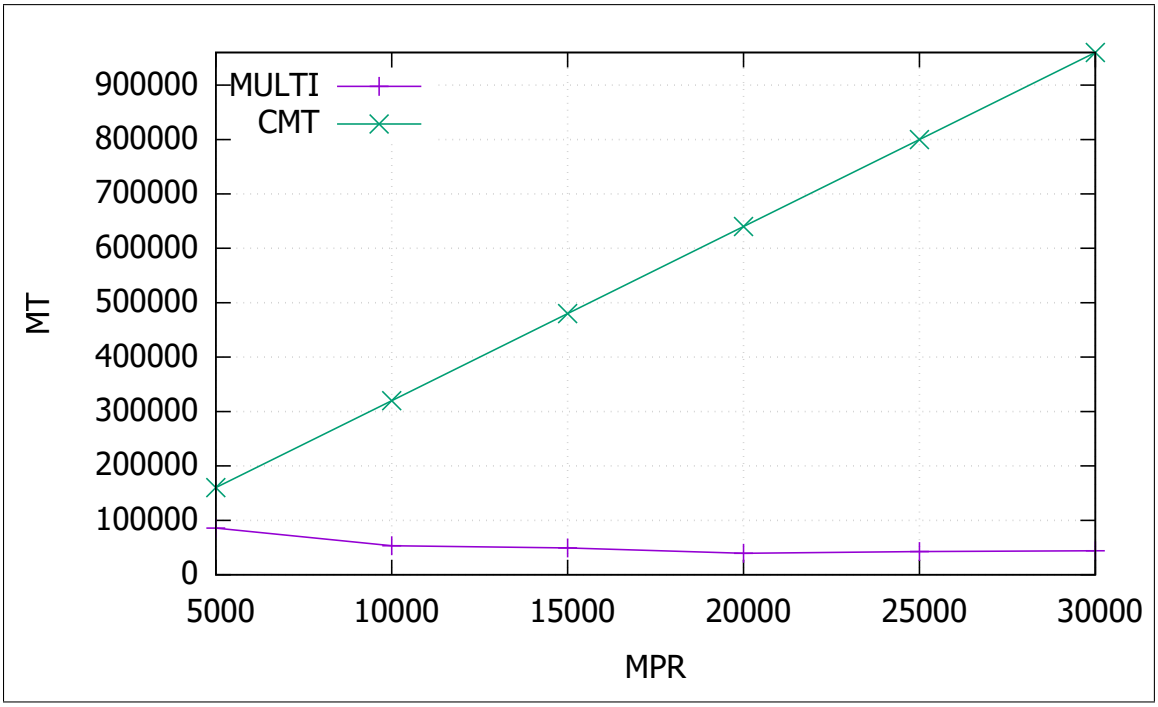

Figure 5.12: Experiment Results: Analysis 2 - Scenario 3, MULTI

can be done:

1. Figure 5.7 shows that, in most rounds of scenario 1, it was possible to maintain an Optimum Message Throughput, except for rounds 5 and 6 wit a MPR of 25K and 30K mps respectively. The MMPR in scenario 1 with configuration MULTI is 20K mps.

2. Figure 5.8 shows that, only in round 1 of scenario 2, has on Optimum Message Throughput. The MMPR in scenario 2 with configuration MULTI is 5K mps.

3. Figure 5.9 shows that, none of the Message Throughputs in scenario 3 with configuration MULTI are Optimum.

In the following section, a discussion of the results for both analysis is presented based on the observation given.

5.3.5 Discussion

The observations of Analysis 1, indicate that adding support for scoping and execution monitoring has a negligible impact in terms of message throughput in an event based system, as long as these mechanisms are not actively used. This essentially means that it is possible to support these mechanisms, and preserve the same message throughput. However, once execution monitoring and/or scoping are used, the message throughput starts to decline.

Take as a reference the Maximum Message Throughput of 30K mps of the original version of Mosquitto (ORI) in scenario 1. This Maximum Message Throughput represents the performance of a regular Event Based System with no security mechanisms. When execution monitoring alone was active, with the slowest execution monitor (scenario 1, CEM), the Maximum Message Throughput dropped from 30K mps, to 20K mps. When scoping

alone was active, with the most number of interconnected scopes (scenario 3, NEW), the Maximum Message Throughput also dropped from 30K mps to 20K mps. This indicates that a slow execution monitor has roughly the same impact than using a considerable number of interconnected scopes in terms of message throughput.

The worst case is represented by scenario 3 with configuration CEM, where 5 scopes are interconnected, and each of them runs a slow execution monitor. In this case, the Maximum Message Throughput drops from 30K mps, to 15K mps.

If these results are compared against the limits and requirements established for some IoT technologies, it is possible add even more perspective. For example:

1. Amazon established a limit of 9K publish requests per AWS account for their AWS Services (Table Message Broker Limits in the website) [1]. This means that one AWS account, can publish a maximum of 9K mps to their Message Broker.

2. Microsoft divides their IoT Hubs solutions into three tiers: S1, S2 and S3 [5]. The tier S3 is the most powerful of them, allowing an average message throughput of 208,333 messages per minute (around 3.5K mps).

Comparing these values against the Maximum Message Throughput of 15K mps obtained in the worst case, it is possible to observe that, even when execution monitoring and scoping are used, an event based system may still be used to satisfy limits and requirements imposed by real IoT technologies.

The observations of Analysis 2, indicate that event suppressing actually alleviates traffic in a communication channel, which has a positive impact in terms of message throughput. On the other hand, event multiplication has the opposite effect. This knowledge could be useful for system administrators, to help them make educated decision when setting up execution monitors in the system that involve suppression and multiplication of events. Particularly, event multiplication has to be treated with care, since abusing this feature

could result in an event based system not being able to keep up with the propagation of events, which could result in delays in the system.

Chapter 6

Conclusions, Related Work, and Future Work

6.1 Conclusion

This work introduced a mathematical model for Event Based Systems where the security mechanisms scoping and execution monitoring are formalized. The main components of this model offer the system administrator the ability to configure the structure of the system, control event propagation, dictate the behavior and localization of execution monitors, and simulate QoS assumptions to account for relative network speed.

In this work execution monitoring was implemented using edit automata [23], and it was shown to be a powerful mechanism for monitoring events sequences. However, execution monitoring can be used for other purposes, such as: implementing coordination logic among devices, and implementing event filtering and event mapping. Execution monitoring, along with scoping, enables coordination logic to be distributed in a number of middleware nodes (scopes), and enable the enforcement of different forms of visibility control.

Different forms of visibility control were demonstrated based on the two protection mechanisms, including information flow control based on the Bell-LaPadula model, and event visibility control based on Fiege *et al.* [17, 19, 18]. Additionally, a case study was presented, to demonstrate the use of scoping and execution monitoring in the context of a home automation system, where two security policies were enforced via the combination of scoping and execution monitoring.

During the research project, the mathematical model was mechanized using PLT Redex, which was an essential factor in the debugging process of the model. Once the mathematical model was complete, to evaluate the performance of execution monitoring and scoping on Event Based Systems, the open source software Mosquitto [25] was extended to support

these protection mechanisms.

An empirical evaluation of the performance was conducted, where the performance was measured in terms of message throughput. This empirical evaluation shown that, despite the impact that scoping and execution monitoring have on event based systems, the extended version of Mosquitto was able to satisfy limits and requirements imposed by current IoT technologies.

Finally, although the focus of this thesis was the middleware of IoT architectures, this work attacked two of the main security challenges in middleware for distributed systems based on the broker architecture, and the Publish/Subscribe design pattern: 1) Failure of network infrastructure, and 2) potentially compromised devices. The contributions of this thesis could be applied to similar architectures in different contexts with minimal modifications, however, the impact of scoping and execution monitoring in terms of performance has to be carefully considered.

6.2 Related Work

Scoping and event mapping in event based systems were introduced by Fiege *et al.* [17, 19, 18]. In their work, scoping and event mapping were formalized within a trace-based formalism adapted from temporal logic. Scoping particularly was proposed as a mechanism to facilitate engineering and coordinations of components in event based systems. In this thesis, scoping is used as a security mechanism used as a countermeasure to potential interruptions of network connections. Additionally, Fiege *et al.* define visibility of events in scopes in terms of a fixed visibility policy based on shared ancestors, whereas the model proposed in this work is more flexible, and allows for custom made visibility policies. This work also shows that event mapping is a special application of execution monitoring.

Schneider [28] classified the type of security policies enforceable with execution monitors implemented as secure automatons. In his work, Schneider focuses in the monitoring

of programs defined as a sequence of actions, where execution monitors interpose themselves between the program being monitored, and the machine running the program. The execution monitors enforce security policies, by terminating the program upon detecting a sequence of actions that violates them. Ligatti *et al.* [23] defined edit automata based on Schneider's work, by extending the capabilities of the security automaton. These new automata are capable not only of terminating the program, but also of modifying it at runtime. In this thesis work, execution monitoring is based on the edit automata proposed by Ligatti *et al.*, but it differs in the sense that the source of the monitored events are multiple distributed devices, instead of a single program.

A Composite Event Detection framework is propose by Pietzuch *et al.* [26] for Publish/Subscribe Systems. A composite event is regarded as a special type of event published when an event pattern occurs. For example, in the context of home automation systems, a home owner may be interested in receiving notifications if somebody rings the door bell when nobody is at home. With the Composite Event Detector proposed by Pietzuch *et al.*, which is implemented as an automaton, this pattern of events can be detected, so as to publish a representative composite event. The automata used for composite event detection has a close resemblance to the edit automata used to implement execution monitoring in this work. This thesis differs from the Composite Event Detection framework in the following. First, the automata used for composite event detection does not account for suppression of events, which is a necessary feature for implementing security policies. Second, in the Event Detection Framework, the Composite Event Detectors interact with the middleware as if they were one more publish/subscribe entity, thus, all events are visible in the middleware. In this thesis, execution monitors are interposed between two entities (i.e. between devices and the middleware), in such a way that events propagated from one entity to another are processed before they are delivered. Although, in appearance, this is a small difference, having the ability to intercept events is critical for security purposes. For example,

if a malicious event was to be propagated from one entity to another, the execution monitor can easily suppress it before it reaches its destination, preventing any possible harm to be done.

Other different efforts have been done in order to augment security in Publish/Subscribe systems. Srivatsa *et al.* proposed EventGuard [30], a framework used to protect Publish/Subscribe systems. In their work, Srivatsa *et al.* assume publishers to be honest, and all publications are assumed to be valid and correct. In contrast, this work assumes the existence of rogue publishers, capable of publishing malicious events. Singh *et al.* [29] proposed a secure version of the MQTT protocol named Secure MQTT (SMQTT). In their work, confidentiality of messages is enforced by applying cryptographic techniques based on Attribute Based Encryption. This works differs from SMQTT in that the main concern of SMQTT, is the confidentiality of events, whereas this work focuses in detection of potentially dangerous event sequences.

6.3 Future Work

Based on the contributions of this thesis, the following research opportunities arise:

1. In this work (§3.3.6), QoS assumptions are used to account for relative network speed. Different QoS assumptions may produce different ordering on event propagation. This implies that the order in which events are consumed by execution monitors highly depends on the underlying QoS assumptions, and their location. Important examples of future work derived from this observation are:

    (a) A clear identification of enforceable security policies based on different QoS assumptions, and identification of possible QoS independent enforceable security policies.

(b) Definition of a high level language to describe enforceable security policies, so as to compile policies into execution monitors that enforce them, taking into account the underlying QoS assumptions.

2. In §3.5, the security mechanisms were used individually to enforce security policies, however security policies that involved the combination of both mechanisms were not considered. Future work involves the exploration of what type of high-level security policies can be captured through the low-level configuration of brokering policies and execution monitors.

3. Configuring the mathematical model taking into account both security mechanisms might be a challenging task. Possible future work to facilitate this administrative task involves the design of a high-level specification language for articulating behavioral protocols. These protocols could then be compiled down to specific configuration of scopes, links, brokering policies, and execution monitors.

Appendix A

All Link-Pairs Flow Routes

In §3.4.1.1, it was suggested that a variant of the Floyd-Warshall algorithm, namely ***All Link-Pairs Flow Routes***, can be used to compute whether there is a legitimate flow route between each pair of entities in the system (more precisely, between each pair of links). The Floyd-Warshall algorithm is used to compute ***All-Pairs Shortest Paths***, given a weighted directed graph $G = \langle V, E \rangle$, with a weight function $w : E \rightarrow R$, that maps edges to real-valued weights. The Floyd-Warshall algorithm uses the adjacency matrix representation of G to compute all shortest paths between all pairs of vertices.

In the following sections, I describe how to adapt the Floyd-Warshall algorithm to compute all legitimate flow routes in a connection graph $\mathcal{CG} = \langle \mathcal{D}, \mathcal{S}, link \rangle$. In this variant, weights are substituted for boolean values, such that, $\top$ represents the existence of a flow route between two entities, and $\bot$ represents the absence of such flow route. This process is divided into two steps:

1. Transforming the connection graph $\mathcal{CG}$ into a directed graph $G = \langle V, E \rangle$ (§A.1).

2. Using the adjacency matrix representation of G to compute an answer matrix D, which represents all legitimate flow routes in $\mathcal{CG}$ (§A.2).

Finally, §A.3 shows how, given two entities x and y, it is possible to compute whether there is a legitimate flow route from x to y, based on the answer matrix D.

A.1 Transformation

Given schema $\chi = \langle \mathcal{CG}, \mathcal{EP}, \mathcal{BP}, sub, \sqsubseteq \rangle$ with a connection graph $\mathcal{CG} = \langle \mathcal{D}, \mathcal{S}, link \rangle$ and a brokering policy $\mathcal{BP} = \langle \mathcal{T}, type, allow \rangle$, we transform $\mathcal{CG}$ into a directed graph $G = \langle V, E \rangle$ as follows:

$$V = link$$

$$E = \{((x,y),(y,z)) \mid propagate(x,y,z) \wedge y \in \mathcal{S} \wedge x \neq z\}$$

That is, for each pair $(x,y) \in link$, a vertex $v = (x,y)$ is created. Additionally, for all tuples $(x,y,z) \in propagate$, the edge $((x,y),(y,z))$ is defined if 1) $y \in \mathcal{S}$, and 2) $x \neq z$. Note that $propagate(x,y,z)$ requires that $(x,y),(y,z) \in link$ and $allow(type(x,y), type(y,z))$.

In simple words, the set V represent links between entities, and the set E induces the single-hop flow routes in the system, where an edge $((x,y),(y,z)) \in E$ represents the flow route xyz. For the algorithm, the graph G is represented as an adjacency matrix [14, §22.1] $R = \left(r_{ij}\right)$, such that:

$$r_{ij} = \begin{cases} \top & (l_i, l_j) \in E \\ \bot & \text{otherwise} \end{cases}$$

Then, $r_{ij} = \top$ represents that there exists a single-hop flow route between links l_i and l_j, whereas $r_{ij} = \bot$ represents that no single-hop flow route exists between links l_i an l_j.

A.2 The algorithm

The ***All Link-Pairs Flow Routes*** algorithm considers the concept of ***intermediate*** links in a flow route. Given a flow route $p = l_1 \cdots l_k$, an intermediate link of p is any link in p other than l_1 or l_k. That is, all links in p are intermediate links, except for the first link and last link.

The algorithm is based on the following observation. Consider the graph $G = \langle V, E \rangle$, where $V = \{l_1, l_2, ..., l_M\}$. Now, consider a subset $\{l_1, l_2, ..., l_k\}$ of V for some $k \leq M$. For any pair of links $l_i, l_j \in V$ for which there exists a flow route $p = l_i \cdots l_j$, and whose intermediate links are drawn from $\{l_1, l_2, ..., l_k\}$, one of two cases must hold:

Case 1: l_k is not an intermediate link in the flow route p. In this case, all the intermediate links in the flow route p are drawn from the set $\{l_1, l_2, ..., l_{k-1}\}$. This intuitively means that, since l_k is not an intermediate link of p, and p is the flow route $l_i \cdots l_j$ with intermediate links drawn from the set $\{l_1, l_2, ..., l_k\}$, then p must be a flow route with intermediate links drawn from the set $\{l_1, l_2, ..., l_{k-1}\}$.

Case 2: l_k is an intermediate link in the flow route p. This means that p is composed of two segments: $p_1 = l_i \cdots l_k$ and $p_2 = l_k \cdots l_j$. Since intermediate links of p are drawn from the set $\{l_1, l_2, ..., l_k\}$, and p_1 and p_2 are segments of p, then the intermediate links of p_1 and p_2 are also drawn from $\{l_1, l_2, ..., l_k\}$. Furthermore, since l_k is not an intermediate link of either p_1 or p_2, then the intermediate links of p_1 and p_2 are in the set $\{l_1, l_2, ..., l_{k-1}\}$.

Observe that, in both cases, to answer if there is a flow route $p = l_i \cdots l_j$ with intermediate links drawn from the set $\{l_1, l_2, ..., l_k\}$, it is necessary to compute if there is a flow route between all pairs of links with intermediate links drawn from the set $\{l_1, l_2, ..., l_{k-1}\}$ beforehand.

Let $d_{ij}^{(k)}$ represent whether or not there exist a flow route between links l_i and l_j for which all intermediate links are in the set $\{l_1, l_2, ..., l_k\}$. When $k = 0$, only flow routes from l_i to l_j with no intermediate links are considered. Such flow routes are the single-hop flow routes represented by the edge $(l_i, l_j) \in E$, hence $d_{ij}^{(0)} = r_{ij}$. The following recursive definition obeys the above discussion.

$$d_{ij}^{(k)} = \begin{cases} r_{ij} & \text{if } k = 0 \\ (d_{ij}^{(k-1)} \vee (d_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)})) & \text{if } k \geq 1 \end{cases}$$

Since for any flow route, all intermediate vertices are in the set $\{l_1, l_2, ..., l_M\}$, the matrix $D^{(M)} = \left(d_{ij}^{(k)}\right)$ gives the final answer, such that $d_{ij}^{(k)} = \top$ if there exists a flow route from l_i to l_j and $d_{ij}^{(k)} = \bot$ otherwise.

Finally, based on the aforementioned recurrence, Algorithm 1 can be used to compute $D^{(M)}$. Where its input matrix is the adjacency matrix R as defined in §A.1, and $rosw[R]$ represents the number of rows (i.e., number of links) of R.

Algorithm 1 All Link-Pairs Flow Routes(R)

```
1: M ← rows[R]
2: D^(0) ← R
3: for k ← 1 to M do
4:    for i ← 1 to M do
5:       for j ← 1 to M do
6:          d_ij^(k) ← (d_ij^(k-1) ∨ (d_ik^(k-1) ∧ d_kj^(k-1)))
7:       end for
8:    end for
9: end for
10: return D^(M)
```

The algorithm runs in $O(M^3)$ where M is the number of links defined in the connection graph $\mathcal{CG}$, from which the input matrix R was inferred.

A.3 One Link-Pair flow route

Using the resulting matrix $D^{(M)}$, it is possible to verify if is there exists a flow route between any two given entities x and y. Since each vertex $l_i \in V$ is also a link of the form $l_i = (x, y)$, $first(l_i)$ is written to refer to the first entity in one link, and $second(l_i)$ is used to refer to the second entity, that is, $first(l_i) = x$, and $second(l_i) = y$. Given a matrix $D = D^{(M)}$, if there exist any cell d_{ij} in D such that $d_{ij} = \top$, $first(l_i) = x$ and $second(l_j) = y$, then there exist

at least one flow route from x to y. An exhaustive search over D is then used to compute the answer in Algorithm 2.

Algorithm 2 Link-Pair Flow Route(V, D, x, y)

```
1: M ← rows[D]
2: for i ← 1 to M do
3:    if first(l_i) = x then
4:       for j ← 1 to M do
5:          if second(l_j) = y ∧ d_ij = ⊤ then
6:             return ⊤
7:          end if
8:       end for
9:    end if
10: end for
11: return ⊥
```

Algorithm 2 runs in $O(M^2)$, where $M = |link|$, that is, the total number of links defined in the connection graph $\mathcal{CG} = \langle \mathcal{D}, \mathcal{S}, link \rangle$. To account for all entity pairs, this algorithm has to be repeated $|\mathcal{E}| \times |\mathcal{E} - 1|$ times, for a total complexity of $O((NM)^2)$, where $N = |\mathcal{E}|$. Considering the complexity of both Algorithm 1 and Algorithm 2, the complexity of the entire process is in $O(M^3 + (NM)^2)$.

Bibliography

[1] Aws service limits. http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html. [Online; accessed 2017-10-30].

[2] Mosquitto - broker configuration. https://mosquitto.org/man/mosquitto-conf-5.html. [Online; accessed 2016-11-24].

[3] Mqtt sernsor. https://home-assistant.io/components/sensor.mqtt/. [Online; accessed 2017-12-04].

[4] PLT Redex & SEwPR. https://redex.racket-lang.org/. [Online; accessed 2016-11-27].

[5] Scale your iot hub solution. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-scaling. [Online; accessed 2017-10-30].

[6] Massive cyberattack turned ordinary devices into weapons. http://money.cnn.com/2016/10/22/technology/cyberattack-dyn-ddos/index.html, October 2016. [Online; accessed 2017-12-16].

[7] Mzbench. https://github.com/machinezone/mzbench, 2017.

[8] A smart fish tank left a casino vulnerable to hackers. http://money.cnn.com/2017/07/19/technology/fish-tank-hack-darktrace/index.html, July 2017. [Online; accessed 2017-12-16].

[9] vmq_mzbench. https://github.com/erlio/vmq_mzbench, 2017.

[10] Z. B. Babovic, J. Protic, and V. Milutinovic. Web performance evaluation for internet of things applications. *IEEE Access*, 4:6974–6992, 2016.

[11] A. Banks and R. Gupta. Mqtt version 3.1. 1. *OASIS standard*, 2014.

[12] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM, 2003.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*, volume 1. J. Wiley & Sons, 8 1996.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[16] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[17] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *LNCS*, pages 309–333, Málaga, Spain, June 2002.

[18] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. pages 385–392, Madrid, Spain, Mar. 2002.

[19] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, Dec. 2002.

[20] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. In *The knowledge Engineering Review*, pages 359–388. Cambridge University Press, December 2002.

[21] C. A. Gutwin, M. Lippold, and T. Graham. Real-time groupware in the browser: testing the performance of web-based networking. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 167–176. ACM, 2011.

[22] L. J. LaPadula and D. E. Bell. MITRE technical report 2547, volume II. *Journal of Computer Security*, 4:239–263, 1996.

[23] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[24] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.

[25] R. A. Light. Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, 2(13), may 2017.

[26] P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE network*, 18(1):44–55, 2004.

[27] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1992.

[28] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[29] M. Singh, M. Rajan, V. Shivraj, and P. Balamuralidhar. Secure mqtt for internet of things (iot). In *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*, pages 746–751. IEEE, 2015.

[30] M. Srivatsa, L. Liu, and A. Iyengar. Eventguard: A system architecture for securing publish-subscribe networks. *ACM Transactions on Computer Systems (TOCS)*, 29(4):10, 2011.

[31] F. Yergeau. RFC 3629. *UTF-8, a transformation format of ISO*, 10646, 2003.