A message broker architecture for adaptive data exchange in the IoT

Houssam Hajj Hassan, Georgios Bouloukakis, Luca Scalzotto, Nirmine Khaled, Denis Conan, Ajay Kattepur, Djamel Belaïd

A Message Broker Architecture for Adaptive Data Exchange in the IoT

Houssam Hajj Hassan*, Georgios Bouloukakis*, Luca Scalzotto†, Nirmine Khaled*,
Denis Conan*, Ajay Kattepur‡, Djamel Belaïd*
{houssam.hajj_hassan, georgios.bouloukakis, nirmine.khaled, denis.conan, djamel.belaid}@telecom-sudparis.eu,
luca.scalzotto.94@gmail.com, ajay.kattepur@ericsson.com
*Télécom SudParis, Institut Polytechnique de Paris, France
†Injenia S.r.l., Italy
‡Ericsson AI Research, India

***Abstract*—In today's IoT environments, message brokers play a pivotal role in facilitating data exchange between IoT devices and applications. Existing message broker implementations offer different configuration options for IoT systems designers for performance tuning. However, designers still have to manually configure the message broker to find the best parameters combination that satisfies the requirements of the deployed applications. In addition, runtime changes might lead to performance degradation and require reconfiguration. This paper presents a publish/subscribe message broker architecture for enabling adaptive data exchange in IoT environments. Core software components are proposed for (i) refining *per-subscription* data flows based on the applications deployed in the environment, (ii) dynamically assigning drop rates or priorities to data flows according to the requirements of these applications, and (iii) enabling the adaptation of data flows based on dynamic changes in the environment or evolving applications' requirements. The proposed architecture is enriched with automated planning capabilities for providing such adaptation of data flows. To demonstrate the applicability of our architecture, we implement the PlanEMQX prototype. Our experimental evaluation shows improvements of 20% of response time for time-sensitive data flows.**

***Index Terms*—IoT, Data Exchange, Adaptive Systems, Automated Planning, QoS.**

I. Introduction

The proliferation of IoT devices has led to highly interconnected and data-rich environments. Today's smart spaces are embedded with sensors and IoT devices to enable data-driven decision-making and facilitate automation through dedicated applications. For instance, buildings may be equipped with sensors to monitor temperature, humidity, and air quality, enabling building managers to optimize energy consumption and improve occupants' comfort [1]. Such applications often define Quality-of-Service (QoS) requirements [2]. For example, a fire detection application may have strict latency bounds that have to be respected. On the other hand, applications used for monitoring purposes (e.g., thermal comfort applications) are more latency and loss tolerant. In addition, space administrators wish to leverage existing sensors and IoT devices to build multi-purpose applications sharing the same data flows [3]–[6]. For example, an HVAC control application may receive the same flow of data from temperature and occupancy sensors as an evacuation planning application used for fire emergencies.

In IoT environments, communication between devices and applications happens through a data exchange system that typically relies on the publish/subscribe paradigm [2], [7]. For this purpose, message brokers are deployed for data processing and dissemination from devices to applications. At the middleware layer, the performance of an IoT system heavily depends on how the message broker handles data flows [2], [7]. Therefore, IoT systems designers often need to tune the deployed message broker(s) to ensure that the QoS requirements of applications are satisfied. This involves adjusting various parameters to guarantee that the broker can handle the data flows and deliver them timely, with the desired reliability (e.g., by assigning priorities, buffer capacities).

Existing message brokers [8]–[11] offer a variety of features for QoS tuning purposes, such as support for delivery and order guarantees [12], and priority queues to prioritize topics [10]. However, IoT systems designers usually have to tune their systems manually. This necessitates trying different combinations of parameters to find the optimal configuration. Clearly, this long and tedious process becomes even more challenging when we consider the changes that might happen in dynamic IoT infrastructures: adding/removing IoT devices/applications, subscriber churn, overloaded system, etc. In addition, handling data flows belonging to the same topic but shared by different applications is usually complex. Hence, the architecture of traditional message brokers does not enable autonomous and self-adaptive data exchange. We thus need to design new architectures for brokers that (i) offer automated configuration and tuning capabilities, (ii) handle and control data flows at the *subscription* level, and (iii) automatically detect and adapt to changes in the IoT infrastructure.

In this paper, we propose refining the architecture of traditional pub/sub message brokers to enable automatic configuration and adaptation in IoT environments based on the QoS requirements of subscribing applications. Namely, we present the architecture of a message broker capable of automatically controlling flows for efficient data exchange, and handling dynamic situations. This is achieved via (i) an *automated configuration planner* that generates configuration and adaptation plans to be executed, and (ii) the implementation of runtime components for supporting the management and control of *per*

subscription data flows through applying drop rates and priorities to flows. Finally, we present a prototype implementation—PlanEMQX—based on our proposed architecture, and provide a performance analysis showing how PlanEMQX is able to efficiently control flows depending on applications' requirements. The main contributions of this paper are:

- A pub/sub-based software architecture for adaptive data exchange in IoT environments.
- A refinement of topic-based pub/sub schemas by relying on rule-based techniques and automated planning.
- A prototype implementation of the proposed architecture using state-of-the-art technologies (https://github.com/satrai-lab/planemqx).

Section II provides an overview of related work, and Section III provides an overview of our proposed architecture. We then go into its implementation details by presenting the PlanEMQX prototype in Section IV. We provide a performance evaluation of PlanEMQX in Section V, and finally conclude the paper in Section VI.

II. Related Work

Several approaches have recently proposed architectures for enabling adaptive data exchange in IoT environments. For example, the authors of [13] focus on adaptation control patterns based on the MAPE-K loop, and the importance of selecting the correct pattern for ensuring an optimal performance of the system components in terms of energy consumption and data transfer. In [14], an architecture is proposed for deploying goal-driven self-adaptive systems through a monitoring component for detecting changes and triggering deployment planning when such changes occur. More recently, the use of machine learning-based techniques is becoming more prominent in self-adaptive systems. For instance, [15] provides an approach for architecting self-adaptive IoT systems LSTM networks for performance prediction, and reinforcement learning for selecting the best architectural pattern given the predictions of the forecasting engine. The authors of [2] provide a methodology and framework for finding an optimal configuration of a data exchange system in IoT environments by relying on automated planning. However, none of the proposed approaches provide effective solutions or architectures for handling data flows in dynamic situations.

More generally, there exist work that deals with enhancing QoS in pub/sub-based systems. For instance, in [16], the authors modify the architecture of MQTT-based systems by breaking the anonymity so that data consumers bypass the broker and send control flow messages directly to data producers (pause, un-pause, faster, slower, etc.). In [17], the authors conduct a sensitivity analysis to build a latency prediction model on a per-topic basis, and determine the number of brokers to deploy at the edge through an optimization problem for minimizing the number of brokers while meeting each topic's QoS requirement. In [18], the authors aim to manage QoS at the SDN controller. The proposed system is based on the pub/sub paradigm. Instead of a message broker, the authors consider a "Fog-like IoT gateway data aggregator", which implements the bandwidth allocation strategy for IoT flows and other traffic in network links.

Finally, although existing message brokers (e.g., EMQX [8], RabbitMQ [10], Mosquitto [9], HiveMQ [11]) offer different features and options to allow system tuning, there is still a need for manually testing different combinations of parameters to find the most efficient one given a specific IoT setup. In addition, existing brokers do not provide planning capabilities for adaptive message flows, especially in dynamic situations that involve changes in the IoT environment (e.g., adding devices/applications, changes in resources). Our approach is distinguished by providing an architecture for adaptive data exchange, and supporting dynamic environments through automated re-configuration with AI planning.

III. Adaptive Message Broker Overview

To highlight the need for a message broker with adaptive capabilities, we present the underlying motivation for designing and implementing our solution (§III-A). Then, we provide an overview of our proposed architecture for designing such an adaptive message broker (§III-B).

A. Motivation

Today's IoT environments deploy different groups of applications to provide services for occupants. We identify four application categories that can be deployed in smart environments: (i) *real-time (RT)*, e.g. a fire detection application; (ii) *transactional (TS)*, e.g. a meeting-room reservation tool; (iii) *IoT analytics (AN)*, e.g. an occupancy-based application using WiFi connectivity data; and (iv) *streaming (ST)*, e.g. a video surveillance application. These application categories define different QoS requirements [2], [19]; for instance, RT applications require strict latency bounds, while ST applications have high throughput demands. Existing message brokers lack support for an automatic configuration based on the QoS requirements defined by the applications and the available resources (processing, networking, etc) of the messaging system. This means that configuring the system requires a trial-and-error process to find the best parameters that satisfy the QoS requirements of deployed applications. This becomes more difficult when we consider dynamic IoT environments where the number of devices and applications is changing (e.g., subscriber churn). In addition, message brokers do not consider applications that may subscribe to the same topic but have different QoS requirements. For instance, if a fire detection application (of type RT) and a thermal comfort application (of type AN) subscribe to the same topic (e.g., "`temperature`"), no mechanisms exist for assigning different priorities or drop rates to messages published to "`temperature`", according to the QoS requirements of the receiving application. Therefore, there is a need for architectures of data exchange systems enabling (i) automated approaches that provide adaptive plans for message brokers based on the QoS requirements defined by the applications, and (ii) mechanisms for controlling individual data flows based on the QoS requirements of the receiving application.

B. Message Broker Architecture for Adaptive Data Exchange

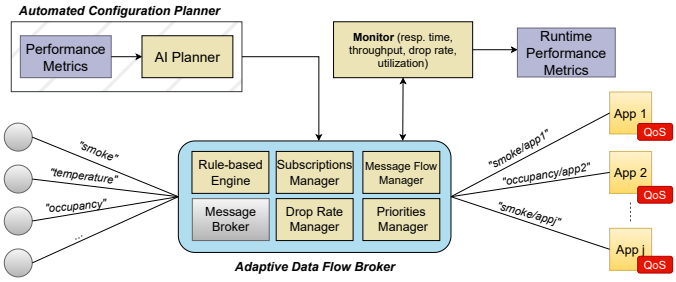

Fig. 1. Modular Message Broker Architecture for Adaptive Data Exchange

To enable data flow management in an adaptive manner, it is essential to refine message broker architectures relying on the traditional publish/subscribe (pub/sub) interaction paradigm [20]. Pub/sub is a distributed software system that is commonly used for content broadcasting/feeds, where multiple peers interact via an intermediate broker. Publishers produce messages (or events) characterized by a specific *filter* to the broker. Subscribers subscribe their interest for specific filters to the broker, who maintains an up-to-date list of subscriptions. The broker matches received messages with subscriptions and delivers a copy of each message to each interested subscriber. There are different types of subscription schemes: topic-based, content-based and type-based [20]. For instance, in a topic-based pub/sub, messages are characterized with a topic and subscribers subscribe to specific topics for receiving messages.

This paper refines the architecture of topic-based pub/sub by introducing automatic management of data flows for efficient data exchange based on the deployed applications' QoS requirements. Figure 1 presents the high-level architecture of an adaptive message broker that includes the components that must be designed along with existing message brokers implementations. In particular, we propose the design of an ***Automated Configuration Planner*** that is responsible for generating plans with details related to configuring the broker to satisfy the QoS requirements of subscribing applications. These plans are sent to the ***Adaptive Data Flow Broker*** where they are executed. Note that, the proposed architecture is generic and it can be adapted to any message broker implementation (e.g., EMQx, RabbitMQ, ActiveMQ, etc.).An overvew of the *Automated Configuration Planner* and the *Adaptive Data Flow Broker* is provided next.

a) Automated Configuration Planner: We leverage Automated Planning (AI Planning) [21], [22] as a decision-making tool for providing a message broker with automated configuration and adaptation capabilities. In particular, we use the Planning Domain Definition Language (PDDL) [23] to create domain models that represent the possible configuration parameters that can be used to tune the message broker (e.g., drop rates and priorities for data flows). Moreover, these domain models include the properties of the IoT system components (e.g., devices and applications), as well as the performance of the system under different situations (e.g. response times, message losses). Such metrics can either be derived from synthetic datasets or real traces (more details in §IV-A and §V-B). The models are then used as input to an AI planner that finds the optimal configuration for managing data flows by the broker. In dynamic situations that involve changes in the IoT infrastructure, the *Automated Configuration Planner* is triggered for providing reconfiguration plans.

b) Adaptive Data Flow Broker: The adaptive data flow broker is responsible for effectively handling IoT data flows (including applications receiving identical data flows) to satisfy the QoS requirements of applications based on the Automated Configuration Planner's output. This is achieved via four core software components. The *Subscriptions Manager* handles the applications' subscription requests. The *Message Flows Manager* creates *per subscription* data flows using a rule-based engine. Unlike traditional message brokers, our proposed broker does not control flows per topic. Instead, it defines *per subscription* flows, i.e., applications subscribing to the same topic will receive two separate data flows that can be managed and tuned differently, according to the QoS requirements of the subscribing application. We provide in §IV-B and §IV-C more details about how this mechanism is implemented. After successfully creating these data flows, our broker assigns drop rates and priorities to flows according to rates provided by the Automated Configuration Planner. Finally, the *Priorities Manager* handles the data flows according to the assigned priorities. This is done by creating priority queues that process messages according to the priorities defined (§IV-E).

IV. Adaptive Message Broker Implementation

This section presents the prototype implementation of the proposed message broker architecture for adaptive data exchange. Our prototype, called PlanEMQX, leverages the EMQX[1] open-source message broker. EMQX is widely used in IoT environments and provides appropriate scaling capabilities for such use cases, high performance, and low processing latency [24]. In addition, EMQX is equipped with a *Rule Engine* for real-time data processing and analysis on messages. As shown later, we use the Rule Engine for creating rules to control data flows according to configurations received from the Automated Configuration Planner.

PlanEMQX leverages different components to adapt message flows. In particular, as opposed to traditional message brokers, PlanEMQX manages flows *per subscription*, and not per topic. This allows for more flexibility in controlling flows belonging to the same topic that have to be delivered to multiple applications (by applying different drop rates, priorities, etc.). To enable automatic configuration and adaptation in dynamic situations, PlanEMQX relies on AI planning (§IV-A). We provide next a detailed description about the implementation of each of the core components of PlanEMQX.

A. Automated Configuration Planner

To determine the best configuration that satisfies the QoS requirements of applications, we leverage automated planning

[1]https://www.emqx.io/

techniques [21], [22]. AI Planning enables adaptive data flow management by defining domain models that capture changes that can happen in the IoT system (e.g., overloaded system, emergency scenarios). Adaptation plans can then be generated at runtime in response to dynamic situations. To express planning models, we use PDDL [23], [25], an action-centered language that provides a standard syntax to describe actions by their parameters, preconditions, and effects. PDDL divides the definition of a planning problem into two parts: the domain and the problem. The *domain* file contains a description of the actions that can be taken by the planner; in our case, the actions represent configurations of the IoT system. These actions may include prioritizing different application categories or applying drop rates to flows belonging to some categories. The *problem* file includes a description of the initial state of the system, as well as the desired goal state to be achieved—i.e., to satisfy the QoS requirements of IoT applications. Algorithms and techniques such as search-based or reasoning-based planning are used to find an optimal plan given the domain and problem descriptions. Examples of how these domain and problem files are created can be found at [26].

At design time, the PDDL domain is instantiated from provided templates using prior collected performance models. Of particular interest are the *values* of latency increments with configuration changes on drop rates or priorities. Before deployment, the PDDL problem file is instantiated with the subscribed applications. These domain and problem files are used to generate a PDDL plan, that includes configuration parameters to be set. At runtime, the plan steps are actuated via the runtime components. The priorities manager and subscriptions manager are invoked to control data flows. The required output is monitored at runtime and may be used to trigger a re-planning of the configurations. Re-planning would instantiate a new PDDL problem file to provide a new plan.

B. Subscriptions Manager

To control individual subscription flows, we create one topic per subscription. In particular, when multiple applications subscribe to the same topic, the Subscriptions Manager creates one topic per subscribing application. For instance, when *app1* and *app2* subscribe to "`smoke`", the topics "`smoke/app1`" and "`smoke/app2`" are created, and are associated with *app1* and *app2*, respectively. However, to avoid undermining the principles of the publish/subscribe paradigm, we implement this mechanism by having the IoT devices (publishers) and applications (subscribers) agnostic to the internal topics.

Figure 2 shows the mechanism used to manage subscription requests. Applications subscribe to a topic by sending a subscription request as they would normally subscribe to a topic in a message broker (step ❶). However, when PlanEMQX receives the subscription request, it creates a new topic containing the subscribing application's ID. This is achieved by using EMQX's Authentication API, which allows external systems to authenticate MQTT clients against custom authentication back-ends through a RESTful API. When an application sends a subscription request to PlanEMQX, the

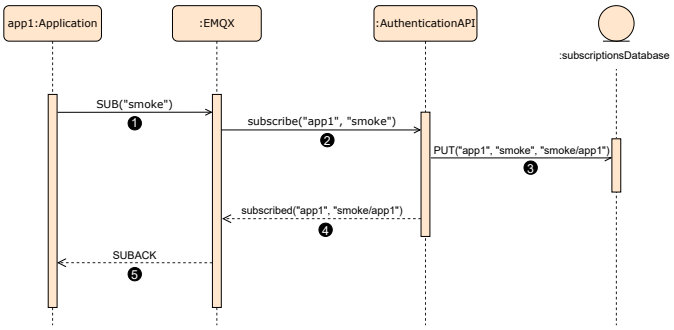

Fig. 2. PlanEMQX Subscription Process

request is forwarded to EMQX's Authentication API (step ❷). The request contains the application ID ("`app1`") and the topic the application wants to subscribe to ("`smoke`"). The API denies the subscription request to the topic specified ("`smoke`"); however, it generates another request to subscribe the application to a new topic. The new topic is created by concatenating the original topic name and the application's ID, separated by a "/". Thus, `app1` in Figure 2 would be subscribed to topic "`smoke/app1`" instead of topic "`smoke`". The API then stores in an SQL database the application ID, the original topic the application intended to subscribe to, and the topic that PlanEMQX created (step ❸); this information is needed to handle unsubscription requests. Finally, the API then sends the authorization to EMQX to subscribe the application to the newly created topic (step ❹), and EMQX sends back a `SUBACK` MQTT message to the subscribing application, indicating that the subscription has been successful (step ❺). The unsubscription process is similar to the subscription scenario: PlanEMQX forwards the applications's unsubscription request to EMQX's Authentication API. The API queries the SQL database to find the topic that the application is subscribed to, then forwards the application's request to unsubscribe with the correct topic to EMQX. PlanEMQX finally sends an `UNSUBACK` MQTT message back to the application.

Notice that PlanEMQX's mechanism for subscriptions ensures that applications remain agnostic to the internal topics created. This is a key element of our architecture; subscribing clients can use regular MQTT APIs without being aware of the inner workings of PlanEMQX.

C. Message Flows Manager

Because publishers are not aware of the internal topics used by PlanEMQX, we need a mechanism to associate messages published from the original topic (e.g., "`smoke`") to the newly created topics (e.g., "`smoke/app1`" and "`smoke/app2`"). To achieve this, we use rule-based techniques to rewrite messages to the newly created topics. In particular, we leverage EMQX's rule engine to create *topic rewrite* rules, which allow modifying the MQTT topic of a message once it is received by EMQX. The topic rewrite feature is implemented as a rule action in the EMQX rule engine.

When PlanEMQX receives a subscription request, the `Message Flows Manager` is triggered to create a topic rewrite rule for the topic that the application subscribes to.

This is achieved through an API call to EMQX, which consists of a request to create a rule, along with information about the type of action ("*rewrite*"), the topic to rewrite, and the topic(s) that messages need to be rewritten to. Thus, when PlanEMQX receives a message, the rule engine will be triggered to check for existing rules related to the topic of the received publication (Figure 3 step ❶). To do this, the topic in the message will be used to sequentially match the topic filter part of the rules created (step ❷). Finally, the message will be republished to the newly created topics (step ❸). Note how the first level of the created topics is "`unsorted`"; this hierarchy is needed to distinguish topics that need to be delivered to the Priorities Manager from the topics that need to be delivered to applications (more details in §IV-E).

D. Drop Rate Manager

By creating per-subscription data flows, PlanEMQX is able to control the flows belonging to each application independently. One of the mechanisms that PlanEMQX is equipped with to improve the performance of the IoT system (i.e., satisfy the QoS requirements of applications) is assigning drop rates to data flows in network congestion scenarios. Such drop rates are identified by the Automated Configuration Planner, and applied to loss tolerant applications only, according to the QoS requirements that they define. Applying drop rates alleviates the load on the network infrastructure and improves the overall throughput and latency for data flows, while keeping message losses under the limit specified by applications.

PlanEMQX receives the drop rates that should be assigned to each data flow (if any) from the Automated Configuration Planner. Then, the Drop Rate Manager creates the necessary rules for filtering messages and, in some cases, dropping a percentage of messages from certain flows. An example of such rules is shown in Listing 1. These rules are then executed in the EMQX rule engine when messages are received.

```
SELECT qos, payload,
    hexstr2bin(sha256(id)) < hexstr2bin(thresholdValue)
AS republish
FROM smoke/app1
WHERE republish = true
AND payload != 'unsubscribe'
```

Listing 1. Rule for dropping messages

To decide whether to drop a message or not, we compare the SHA-256 hash of the ID of the received message with a threshold value. The threshold is defined as the maximum value that the SHA-256 can take ($2^{256} - 1$), multiplied by the percentage of messages that are allowed to go through. For example, if 5% of messages have to be dropped from data flows belonging to the topic "`smoke/app1`", the SHA-256 of the ID of messages published to "`smoke/app1`" will be compared to the threshold value $0.95 \times (2^{256} - 1)$. All messages that satisfy the condition $sha256(id) < threshold$ will be forwarded; other messages will be dropped.

E. Priorities Manager

Prioritisation per subscription flow is implemented by the Priorities Manager, which enables setting up priorities identified by the Automated Configuration Planner. The manager implements a set of priority queues where incoming messages are queued according to the priority assigned to them. As mentioned in §IV-C, all messages received from publishers are rewritten to topics starting with "`unsorted/`". The manager receives all such messages through the topic filter "`unsorted/#`" (Figure 3 step ❹). When a message is received, the manager identifies the application that the message belongs to. Recall from §IV-B that all topics used by PlanEMQX are tagged with the application's ID. Then, the manager queues the message in one of the queues according to the priority assigned to the application (step ❺). Meanwhile, another process is responsible for polling the queues (in the order of their priority) and republishing the messages to EMQX; the republished messages are published without the "`unsorted/`" tag. The messages will finally be forwarded to the corresponding application.

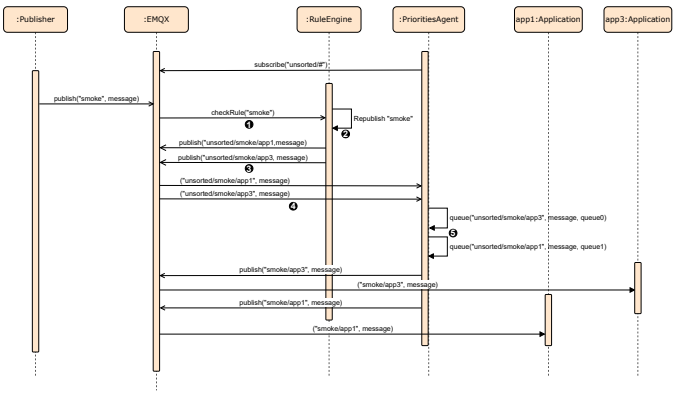

Fig. 3. PlanEMQX's Message Flows and Priorities Handling Process

V. PlanEMQX Evaluation

This section presents the evaluation of PlanEMQX. We start by describing our prototype implementation of PlanEMQX (§V-A), then we proceed to show how PDDL files are generated and used by the AI planner to automatically generate configuration plans for PlanEMQX (§V-B). Finally, we provide a performance evaluation of PlanEMQX by showcasing how our architecture enables improving the performance of IoT data flows (§V-C). The code used for setting up PlanEMQX and running the following experiments is publicly available on https://github.com/satrai-lab/planemqx.

A. The PlanEMQX Prototype

As described in §IV, PlanEMQX is implemented on top of the EMQX message broker and is further enhanced with the components presented in §III-B. We implement the *Subscriptions Manager*, the *Message Flows Manager*, the *Drop Rate Manager*, and the *Priorities Manager* as Java processes that interact with EMQX through API calls. In addition, the EMQX Rule Engine is used by the *Subscriptions Manager* and the *Message Flows Manager* to automatically create rules when applications suscribe to topics. To generate configuration and adaptation plans, we use the Metric-FF [27] AI planner,

which is an extension of the fast-forward planning system that supports reasoning with numerics. Metric-FF uses forward chaining heuristic state space planning to find an optimal solution given a set of actions and goal state to be reached. As specified in [28], Metric-FF passes most benchmarks and is able to solve problems at scale within a few seconds. When the IoT system is first deployed, the *Automated Configuration Planner* is triggered to generate a configuration plan for satisfying the QoS requirements of applications (i.e., by creating subscription flows, and setting appropriate priorities and drop rates). When changes occur (captured through the *Monitor*), re-planning is performed to adapt to the new settings.

Publishers and subscribers connect to PlanEMQX using the MQTT Paho library [29]. To simulate the networking infrastructure between the different components of the IoT system, we use Containernet[2] [30], a fork of the Mininet network emulator [31] that supports the use of Docker containers as hosts. We consider that the bottleneck of the data exchange system lies in the bandwidth of the network link between PlanEMQX and the applications, i.e. the subscribers.

B. Performance Metrics Dataset Generation and Validation

To compose the PDDL domain and problem files used by the Automated Configuration Planner, we need to get information about (i) the entities present in the IoT environment (devices and applications), and (ii) the performance of the IoT system (e.g., response time, loss rates) in different situations (e.g., overloaded system, changing number of subscriptions). For this purpose, datasets can be created through the monitoring component that collects performance metrics at runtime. However, it is not evident to obtain such datasets prior to deploying and running the system. Therefore, we use the EDICT [32] simulator to generate datasets that are used by the Automated Configuration Planner to compose the PDDL domain and problem files and generate configuration plans.

To verify that the datasets obtained from EDICT accurately model the performance of a pub/sub IoT system, we evaluate the generated datasets against running the PlanEMQX prototype in the default settings (i.e., without assigning any priorities or drop rates). Figure 4 shows the average response time of data flows using (i) EDICT and (ii) the PlanEMQX prototype under a varying bandwidth. The results show that the metrics derived from EDICT match the results collected from the prototype. Nevertheless, we observe that the results of the prototype have a lower response time in general. This is because we are using Containernet to emulate the network infrastructure, and Containernet lacks the ability to emulate proper queueing delay when transmitting packets. More precisely, Containernet uses Linux TC to apply queueing disciplines to network interfaces: bandwidth limitations, packet loss, and network delay. In Linux TC, The bandwidth limitation constrains the volume of traffic (i.e., number of bytes) that can be sent per unit of time. However, it does not emulate the actual packet transmission delay due to the available bandwidth. Hence, Linux TC sends packets at the same speed regardless of packet size (i.e., the transmission delay is constant), which in turn affects the perceived queueing delay. That is, if the available bandwidth is enough to empty the queues, the queued packets do not experience the queueing delay due to the transmission of previous packets.

[2]https://containernet.github.io/

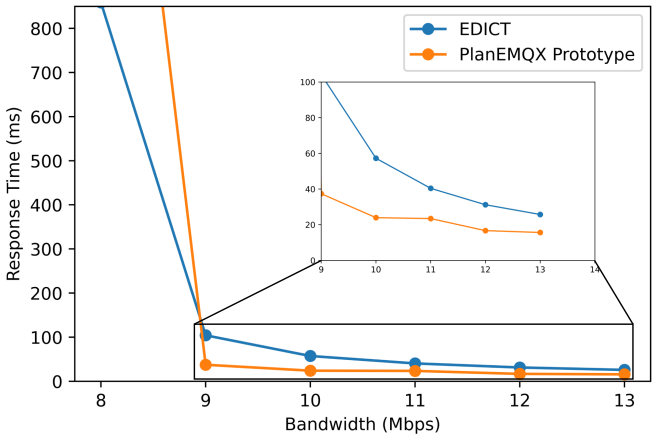

Fig. 4. Average response times: Queueing simulations *vs.* PlanEMQX

C. Performance Evaluation of PlanEMQX

To evaluate the performance of PlanEMQX, we simulate a smart environment with 30 IoT devices and 16 applications that belong to 4 application categories. PlanEMQX accepts publications to 30 topics, with a total number of 80 subscriptions. The properties of the IoT environment tested are presented in Table I. We consider that the available bandwidth between PlanEMQX and applications is 10 Mbps. When the system is set up, the Automated Configuration Planner is triggered to generate a configuration plan. As shown in Listing 2, the plan indicates that the highest priority should be given to RT applications, then to ST application, then TS applications, and finally to AN applications. In addition, the plan specifies that 2% of messages belonging to ST flows and 5% of messages belonging to AN flows should be dropped. We run several experiments to test the mechanisms implemented by PlanEMQX. For each experiment, publishers send messages with a specific message size and publication rate (according to the setup in Table I) for 5 minutes. We then collect metrics related to the response time of data flows, and the number of messages sent and received for each subscription.

```
1 : ff: found legal plan as follows
2 step    0: DROPPING_AN_5_ST_2 TOPIC_ALL APP_ALL
3         1: PRIORITIZE_RT_ST_TS_AN TOPIC_ALL APP_ALL
```

Listing 2. AI planner output

We start first by evaluating the effectiveness of the dropping mechanism of PlanEMQX. Table II shows the number of messages published by IoT devices to each application category, the number of messages consumed by each application category, and the percentage of dropped messages. The results show that the Drop Rate Manager is able to correctly apply the dropping rates according to the input of the configuration plan:

IoT system properties				
Category	*#topics*	*#subscriptions*	*Load (Mbps)*	*Available bandwidth (Mbps)*
AN	15	21	1.90	10
RT	18	21	2.33	
TS	11	18	2.05	
VS	16	20	1.61	
Total	30	80	7.9	10

TABLE I
EXPERIMENTAL SETUP

Category	*Published*	*Consumed*	*Dropped (%)*
AN	32100	30512	4.94
RT	33300	33276	0.07
TS	28500	28469	0.10
ST	30000	29433	1.85

TABLE II
DROPPED MESSAGES PER APPLICATION CATEGORY

4.94% of messages are dropped for AN flows, and 1.85% of messages are dropped for ST flows. The longer the IoT system is running, the more the drop rate percentages will converge to the values indicated by the Automated Configuration Planner. For other application categories, the drop rate is negligible, and the few messages dropped are due to packet losses at the network layer. These messages can be correctly delivered when re-transmission mechanisms are implemented.

Next, we compare the average response time per application category when PlanEMQX is used as a message broker against a default approach that deploys a message broker without automated planning capabilities. This is done by averaging the response time for individual messages belonging to each category (about 30000 messages per category,as shown in Table II). As Figure 5 shows, there is a significant improvement in the response time for RT and ST applications: PlanEMQX can identify that such applications have stricter latency bounds (based on their QoS requirements) through the Automated Configuration Planner. These improvements come at the expense of having higher response times for AN and TS applications, but this is deemed acceptable because these application categories are more delay tolerant.

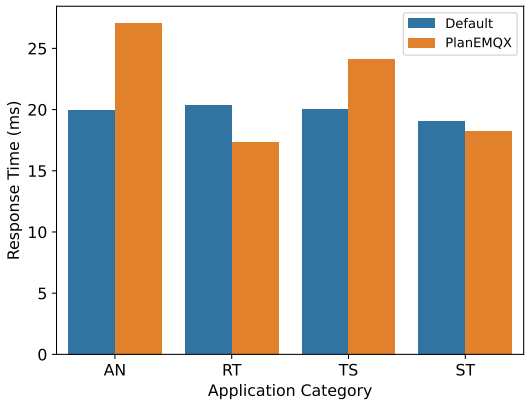

Fig. 5. Average response time per app. category: PlanEMQX *vs.* Default

We further analyse the response time results through box plots that show how the response times distributions when using the default approach, and how these distributions changes when PlanEMQX is used. Notice that not only the average response times are improved for time-sensitive applications (RT and ST), but also tail latency is lower for these application categories. For instance, the maximum response time for RT applications is decreased from 22.7 ms to 18.5 ms when using PlanEMQX. This is also the case for ST applications, where the maximum latency is reduced by 13%.

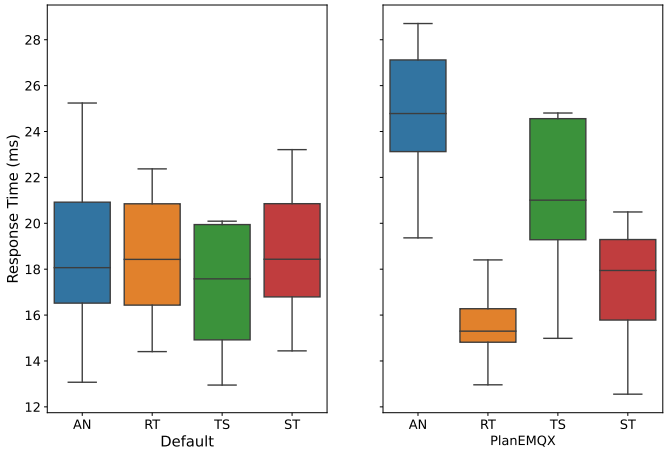

Fig. 6. Response times box plot - Default vs PlanEMQX

Finally, to highlight how PlanEMQX deals with applications that share the same data flows but have different QoS requirements, we plot in Figure 7 the average response time *per topic* (for topics shared by multiple categories) under the default approach and PlanEMQX. We observe that with PlanEMQX, for the same topic, we have different response times, depending on the requirements of the subscribing application. For instance, for "`topic5`", the response time for RT is 14.67ms, 15.75ms for ST, 27.12ms for AN. This demonstrates the effectiveness of assigning priorities per subscription. However, this is not the case with the default approach, which does not take into account the QoS requirements of applications. Therefore, when multiple applications belonging to different categories subscribe to the same topic, their response time is going to be randomly distributed.

D. Threats to Validity

Our approach leverages AI planning to find the best configuration parameters for satisfying the QoS requirements of IoT applications For this purpose, the actions to be taken by the AI planner (assigning priorities, setting drop rates) and their effects on the performance of the system should be defined at design time, either by relying on historical data, or by using synthetic data generated from simulators. Clearly, it is not feasible to include all possible combinations of actions to be taken by the AI planner because their effects might not be captured in the datasets used. Hence, the output of the planner can only be as good as the dataset used for generating the domain and problem files. Furthermore, we use rule-based techniques for managing data flows for different applications. This involves

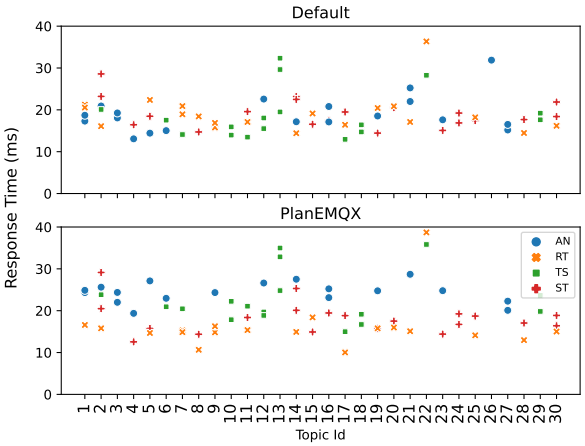

Fig. 7. Response time per topic - Default vs. PlanEMQX

creating and executing rules in the message broker used for the implementation of our solution (e.g., EMQX). If future updates of the message broker stop supporting rule engines, our approach will need to be modified to handle subscription and message flows in a different manner. Finally, since we propose a middleware-based architecture for ensuring QoS of applications, we do not consider network-related changes that might affect the performance of the system. Thus, in the experiments, we use a simulator for emulating the network infrastructure and setting up network link characteristics (e.g., bandwidth). However, in real-world environments, network conditions might not be as stable as they are in simulation settings, and may depend on external factors. To address this, existing network solutions may be implemented alongside our solution to guarantee optimal data exchange between the IoT system components.

VI. Conclusion

This paper introduces a publish/subscribe-based architecture for adaptive data exchange in IoT environments. We refine traditional pub/sub message broker architectures by including an Automated Configuration Planner to generate configuration plans that take into account the QoS requirements of applications. The planner is also responsible for adaptation in dynamic situations to ensure that applications' requirements are satisfied despite changing conditions. We also design and implement runtime components alongside the message broker for efficiently controlling data flows by assigning priorities and drop rates identified by the automated planner. To evaluate our approach, we implement the PlanEMQX prototype by relying on our proposed architecture and using state-of-the-art technologies. Our experimental results show that PlanEMQX improves the end-to-end response time of time-sensitive applications, especially when applications sharing common data flows are deployed. Our future work includes using the performance metrics collected at runtime for performing a predictive analysis of changes that might occur, and proactively adapt the management of data flows to avoid degraded performance.

Acknowledgements

This work is partially supported by the Horizon Europe project DI-Hydro under grant agreement number 101122311.

References

[1] W. Zhang *et al.*, "Thermal Comfort Modeling for Smart Buildings: A Fine-grained Deep Learning Approach," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2540–2549, 2018.
[2] H. Hajj Hassan *et al.*, "PlanIoT: A Framework for Adaptive Data Flow Management in IoT-enhanced Spaces," in *SEAMS'23*.
[3] S. Chaturvedi, S. Tyagi, and Y. Simmhan, "Cost-Effective Sharing of Streaming Dataflows for IoT Applications," *IEEE Transactions on Cloud Computing*, vol. 9, no. 4, pp. 1391–1407, 2021.
[4] Q. H. Cao *et al.*, "A Trust Model for Data Sharing in Smart Cities," in *ICC'16*.
[5] G. Di Modica *et al.*, "IoT Fault Management in Cloud/Fog Environments," in *IoT'19*.
[6] P. Garcia Lopez *et al.*, "Edge-Centric Computing: Vision and Challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, 2015.
[7] G. Bouloukakis *et al.*, "PrioDeX: a Data Exchange Middleware for Efficient Event Prioritization in SDN-based IoT systems," *ACM Trans. Internet Things*, 2021.
[8] https://www.emqx.io/.
[9] https://mosquitto.org/.
[10] https://www.rabbitmq.com/.
[11] https://www.hivemq.com/.
[12] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper," in *DEBS '17*.
[13] H. Muccini *et al.*, "Self-Adaptive IoT Architectures: An Emergency Handling Case Study," in *ECSA'18*.
[14] F. Alkhabbas *et al.*, "A Goal-driven Approach for Deploying Self-adaptive IoT Systems," in *ICSA'20*.
[15] J. Cámara *et al.*, "Quantitative Verification-aided Machine Learning: A Tandem Approach for Architecting Self-adaptive IoT Systems," in *ICSA'20*.
[16] A. Sadeq *et al.*, "A Qos Approach For Internet Of Things (IoT) Environment Using MQTT Protocol," in *Proc. of the Intl. Conf. on Cybersecurity*, 2019.
[17] S. Khare *et al.*, "Scalable Edge Computing for Low Latency Data Dissemination in Topic-Based Publish/Subscribe," in *SEC'18*.
[18] P. Moraes *et al.*, "A Pub/Sub SDN-Integrated Framework for IoT Traffic Orchestration," in *Proc. of the 3rd Intl. Conf. on Future Networks and Distributed Systems*, 2019.
[19] 3GPP, Technical Specification Group Services and System Aspects, "Service Requirements for Cyber-physical Control Applications in Vertical Domains," 3GPP TS 22.104 (V17.2.0), Dec. 2019.
[20] P. Eugster *et al.*, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, Jun. 2003.
[21] M. Ghallab *et al.*, *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
[22] ——, *Automated Planning and Acting*. Cambridge University Press, 2016.
[23] M. Fox *et al.*, "PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains," *JAIR*, vol. 20, pp. 61–124, 2003.
[24] H. Koziolek *et al.*, "A Comparison of MQTT Brokers for Distributed IoT Edge Computing," in *ECSA'20*.
[25] P. Haslum *et al.*, "An Introduction to the Planning Domain Definition Language," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 13, no. 2, pp. 1–187, 2019.
[26] H. Hajj Hassan *et al.*, "Artifact: Implementation of an Adaptive Flow Management Framework for IoT Spaces," in *SEAMS'23*.
[27] J. Hoffmann, "Extending FF to Numerical State Variables," in *ECAI'02*.
[28] A. Gerevini *et al.*, "Planning with Numerical Expressions in LPG," in *ECAI'04*, 2004.
[29] https://eclipse.dev/paho/index.php.
[30] M. Peuster *et al.*, "MeDICINE: Rapid Prototyping of Production-ready Network Services in Multi-PoP Environments," in *NFV-SDN'16*.
[31] https://www.mininet.org/.
[32] H. Hajj Hassan *et al.*, "EDICT: Simulation of Edge Interactions across IoT-enhanced Environments," in *DCOSS-IoT'23*.