PPS: A Publish-Process-Subscribe Middleware for Predictive Supply Chains

Amir Jabbari[1(✉)], Gowri Ramachandran[1], Sidra Malik[2], and Raja Jurdak[1]

[1] School of Computer Science, Queensland University of Technology, Brisbane, Australia
e.amirjabbari@gmail.com, {g.ramachandran,r.jurdak}@qut.edu.au
[2] Data61, CSIRO, Eveleigh, Australia
sidra.malik@data61.csiro.au

Abstract. IoT deployments collect sensor data in supply chains, industrial monitoring, and smart environments. Raw sensor data is sent to remote dashboards or the cloud. Raw data have limited insights compared to processed data, which enables valuable insights through predictive analytics and rule-based alerts. Existing data-sharing middleware, like publish-subscribe frameworks, lacks on-the-fly data processing. This forces users to handle raw data. Built-in data processing in middleware reduces user process management overhead and provides near real-time insights. We propose a novel Publish-Process-Subscribe (PPS) middleware that provides on-the-fly data processing abilities while preserving the loose coupling benefits of the publish-subscribe paradigm. Our middleware simplifies the process management efforts for the end users by allowing them to register processes and map them to real-time data streams. Besides, PPS dynamically allocates computing resources based on the data rate and perceived processing demands. As a result, the end-users receive processed data in near real-time without manually setting up processing nodes. Experimental results show that our PPS offers significant benefits at the cost of minimal performance overhead compared to the traditional publish-subscribe middleware. We conclude that the significant reductions in replicated computations among subscribers outweigh the overheads of our framework, rendering it an attractive option for predictive supply chains.

Keywords: IoT Applications · Broker · Publish-Subscribe · Supply Chain Monitoring · Predictive Insights

1 Introduction

Current supply chains operate globally, including numerous interconnected organisations, extensive distribution, and complex transactions. There is an increasing need to improve efficiency by adopting modern technologies and logistics due to concerns about supply-demand balancing, traceable oversight,

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
A. Zaslavsky et al. (Eds.): MobiQuitous 2023, LNICST 594, pp. 125–144, 2024.
https://doi.org/10.1007/978-3-031-63992-0_8

just-in-time processes and adaptive supply chains. Emerging supply chains use sensor-based logistics in an industry-wide trend of transforming non-digital supply chains into digital ones with near real-time information covering asset conditions and locations. Sensor-based technology enables supply chains to link physical assets with their virtual counterparts, enabling automatic traceability. This traceability system uses a temporal data model to track the raw materials and their relevant events through the supply chain.

Supply chains are transitioning to proactive predictions from traditional reactive methods. Therefore, supply chains must support prediction to ensure they receive on-the-fly insights and make proactive decisions. Reactive models, which respond to events after they occur, may need to be revised in such dynamic environments. Proactive predictions, on the other hand, allow supply chain managers to anticipate future events and take preventive actions to optimise their operations [22]. Therefore, there is a need for immediate processing of the shared raw temporal data to extract predictive and analytic insights. Internet of Things (IoT) devices publish this temporal data as messages through data sharing middlewares following either a publish-subscribe or request-reply messaging model [19,23,25]. In the supply chain context where many IoT devices publish many messages simultaneously, the Publish-Subscribe messaging model is widely utilised and well established due to their efficiency in low communication overheads and support of loosely coupled many-to-many messaging [25,30]. In this model, message publishers publish messages under topics, and subscribers must subscribe to topics to receive updates. Unlike Publish-Subscribe, the Request-Reply messaging model is more suitable for resource-constrained systems as there must be a request per message to get replies [23]. However, no matter the size of the system, none of these messaging models offers built-in support for processing published messages. Moreover, existing cloud or edge-based data sharing applications [6,15,18] that are developed on top of the mentioned messaging models offer data processing as an external feature. To gain insights, the user must manually activate data processing features in a reactive manner. Therefore, existing applications also lack automated processes that help predictive supply chains to gain insights proactively.

Shared data within the Publish-Process-Subscribe (PPS) middleware must be processed to extract insights in a near-real-time fashion [7]. Insights are key components for predictive supply chains, and a series of automated actions are required to gain insights from published messages and gathered data. In a supply chain, many IoT-based sensors produce data simultaneously, and existing technologies can generate analysis on each of them separately by adopting edge computing [20]. However, the shared data of multiple sensors carry insights when combined and processed for the fusion of the computations. These computations must be executed immediately to gain near-real-time insights. Therefore, there is a need for computational infrastructure that supports a vast number of processes. However, the lack of data processing within the data-sharing middleware makes the message subscribers hire or invest in computing units that can execute complex algorithms on shared data [9]. Due to the high expenses of building com-

putational infrastructure and the complexity of computations required to gain predictive and analytic insights, most supply chain nodes tend to offload these computations to other computing nodes, such as third-party cloud computing platforms. However, offloading computations to another computation provider increases inefficiencies, operational costs, and delays. Also, different subscribers must replicate the execution of the same substantial computations on the same shared data to extract insights individually.

To address the above challenges, we propose PPS, a data sharing and processing middleware that provides automated processing features within the messaging model and delivers insights instead of raw data. We design a data-sharing and processing middleware for publish-subscribe systems by executing computations on data to generate insights to allow the data consumers (or subscribers) to receive processed insights in near real-time from the data broker. Our middleware automates data processing by dynamically scheduling processing tasks on computing units based on the perceived load. The contributions of this paper are:

- We design a built-in data processing feature within the publish-subscribe data-sharing middleware where published raw data is processed on the fly at the PPSManager (so-called broker in traditional publish-subscribe architecture).
- We develop a load-balancing algorithm to dynamically allocate and deallocate computing nodes to prevent bottlenecks at PPSManager.
- We evaluate PPS using a proof-of-concept through a simulated supply chain use case. Our evaluation results indicate that PPS significantly reduces the total number of computations needed to gain insights while incurring minor increases in broker's CPU and Memory usage.

The rest of the paper is organised as follows: Sect. 2 introduces the existing messaging middleware and discusses related works. Section 3 presents the detailed architecture definition for our proposed Publish-Process-Subscribe (PPS) framework, and Sect. 4 presents the system description. Section 5 presents the system's safety and liveness analysis, and Sect. 6 discusses the proposed PPS implementation and its evaluations. The last section concludes the paper with future work.

2 State of the Art and Related Work

There are a variety of platforms and studies conveying data processing opportunities in the IoT application context. Many rely on third-party cloud providers like Amazon [1] to ease the development of IoT applications and set up a serverless data-sharing environment. Using serverless computing, application developers delegate computational maintenance and concerns to cloud vendors [4]. However, data processing features in these serverless platforms are offered as extensions that must be activated and managed by clients after the shared data is received. Figure 1 shows different attitudes towards the raw data in data sharing middlewares. Assuming publish-subscribe as the default messaging model

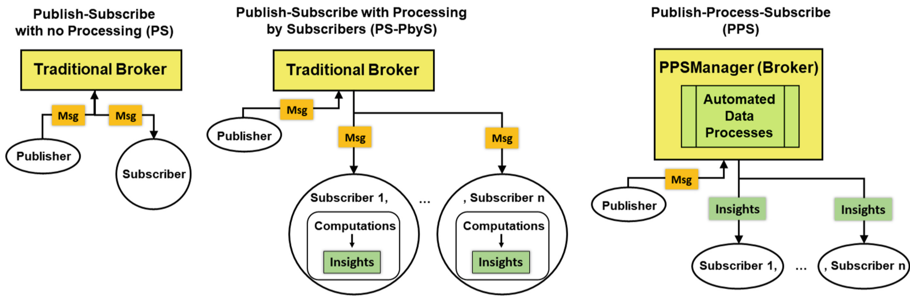

Fig. 1. Evaluation candidates: Traditional Publish-Subscribe (left), Publish-subscribe with Processing at Subscribers (centre), and PPS (right).

in this figure, all the existing studies and platforms forward data processing to the subscribers' side in a Publish-Subscribe with Processing by Subscribers (PS-PbyS) fashion. In serverless platforms, vendors act as subscribers. After receiving and storing, and collecting the shared data, vendors' computing units perform pre-defined processes on data based on the pre-defined settings [18]. However, in this reactive manner of processing shared data, the delay costs of gaining insights cause inefficiencies for predictive purposes. Furthermore, supply chain stakeholders, namely publishers and subscribers, must pay for these computational services.

Produced and shared data of IoT-based devices are valuable datasets for many applications. Machine learning models are trained and offered as paid services using these datasets. These models are used in applications like Digital Twin, where IoT and Artificial Intelligence are combined to interpret and visualise data [16]. Using Digital Twin, manufactured products or the whole IoT system's physical aspects are connected to the virtual prototypes to assess, validate and verify the system operation [24]. Raw datasets can be traded with computational infrastructure once the IoT data marketplace feeds the Digital Twin's input. Typical IoT data marketplaces' target is only to connect data buyers to sellers, and they lack data processing support [12,14,26,28]. However, the IoT data marketplace has the potential to offer data buyers like application developers the opportunity of exchanging computational infrastructure for collecting data with data sellers like supply chain stakeholders. In [20], the authors propose machine learning and edge computing support in the IoT data marketplace. Therefore, these computations can be executed at the edge or cloud and provide analytical and predictive insights based on the trained machine learning models on the gathered data in near-real-time. However, existing data marketplaces also lack an entity that facilitates and schedules these computations before data reaches the computing units. Therefore, they have the same architecture as the PS-PbyS approach in Fig. 1, where the shared data is delivered to the subscribers first, and computations are executed after.

Processing of shared data is one of the significant concerns in the decentralised applications of the Internet of Things (IoT) that require messaging protocols. The publish-subscribe messaging paradigm is frequently used in modern IoT and business deployments because of its resource efficiency and support for loosely-coupled messaging. In this paradigm, subscribers register their interest in an event, or a pattern of occurrences, in systems based on the publish-subscribe interaction and are then asynchronously alerted of events created by publishers. Many variations of the publish-subscribe paradigm have been presented, each tailored to a unique application or network architecture [8]. The common denominator underlying these versions is the complete decoupling of the communication entities in time, space, and synchronisation.

Publish-subscribe capabilities can adopt either the network and application layer or the content-based protocols [15]. In the first category, network layer protocols rely on the broker as an entity that runs on a server, like MQTT (Message Queuing Telemetry Transport) [25] and AMQP (Advanced Message Queuing Protocol) [11]. The broker receives the messages and routes them to the subscriber. On the other hand, content-based protocols are designed to run without the central broker. Therefore, they move overhead from the broker to the network, slowing down the network traffic when the throughput grows. Even though the content-based publish-subscribe models perform well under hardware limitations, they still need to sacrifice parameters like delivery nodes to decrease workload traffic and their performance compared to broker-based systems [15]. For instance, in [6], authors have proposed architecture in Edge-to-Cloud Continuum for adaptive data-driven routing on top of the CUPUS [3], their former content-based publish-subscribe implementation. However, to increase the performance metrics like less delay in data delivery, they limit the number of subscribers and prioritise them as top-K based on the producers' requirements. This reduced latency results from reduced delivery nodes and communication traffic and does not guarantee message delivery to all subscribers. Broker-based network layers rely on a central entity called the broker that can be integrated with cloud-based systems, like Kafka [17,29]. In Kafka's architecture, messages are divided by topics and must be published in different disc partitions for each topic. Each publisher chooses the topic and the partitions to publish to, then writes the messages in a round-robin method to the partitions. Subscribers should read messages from the partitions, and the number of subscribers to a topic cannot be more than the number of partitions. Otherwise, they might be unable to read the messages and become inactive [15]. Also, each read generates an offset commit request, which adds to the broker's workload. Workload increase in Kafka leads the system to ignore real-time data sharing and limit the subscriber's message consumption [21]. Kafka, on its own, lacks adequate support for stateful data processing. Therefore, external frameworks like Flink [5] must handle processes. However, setting up Flink data stream processing as a Kafka consumer object does not imply automated processing as there is a need for controlling and triggering the executions by a developer since Flink is lazy loaded [27]. Therefore, there is no guarantee that all subscribers in Kafka can benefit from the raw

shared data simultaneously and receive insights through automated processes in Flink. Furthermore, Flink's resource management is demanding, especially with varying data arrival rates. Therefore, dynamic resource allocation support is still challenging in systems that use this framework for stream processing [31]. On the other hand, Flink's development and operational complexity require a professional back-end team in supply chains to manage and control the system, which might be against the incentives of supply chains' stakeholders that lack a suitable computational infrastructure.

As shown in Fig. 1, existing data sharing middlewares and IoT-based applications lack a processing stage that automates data processing, task scheduling and computing node allocations. Also, the edge or cloud-based platforms lack delivering predictive/analytic insights from the shared data to subscribers, and they are the subscribers who reactively execute computations. These computations' operational costs are handed to each subscriber individually, and providing the processing requirements like resource allocations, and bottleneck prevention is all on the client's end. We propose PPS middleware, where data processing features are automated within the data sharing middleware, and insights are delivered to subscribers instead of raw data. The broker of this architecture, PPSManager, dynamically allocates and deallocates resources and maintains load balancing on the server's side. In the next section, we provide a detailed discussion regarding the architecture of PPS.

3 Architecture Definitions and Assumptions

In this section, the architecture of our proposed PPS middleware is discussed and definitions and assumptions are provided to describe the system.

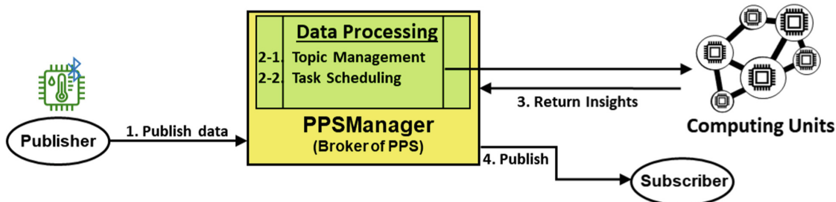

Fig. 2. Overview of the framework

3.1 Overview of Architecture

Figure 2 provides an overview of the steps to share and process published data within our proposed PPS architecture. Unlike the traditional broker-based publish-subscribe frameworks like Kafka [21], messages in our proposed architecture are published to a processing manager called PPSManager instead of

the broker. Besides satisfying all the services of a traditional broker, this processing manager facilitates the data processing requirements before sharing the outcome of processing (insights) with the subscribers. The key components of our proposed PPS data sharing and processing middleware indicated in Fig. 2 are defined below, followed by their interactions descriptions:

Publishers: are the IoT-based sensors, organisations and anyone who provides updates and information published as messages with a standard structure under specific topics. Publishers must register with the system, and this registration is also shared as a message within PPS middleware. All messages have topics that are available to other stakeholders within the application ecosystem, and interested subscribers can subscribe to them.
Topic: is descriptive metadata in a string format that can come in multiple levels. For example, topic "TTISensor/001" means the data is related to a temperature sensor with the sensor ID "001".
Subscribers: are nodes that subscribe to topics and receive all messages published under those topics. Similar to publishers, subscribers must also register with the system, and this registration is shared as a message within PPS middleware.
PPSManager: This trusted entity works as a broker in PPS architecture, with additional data processing responsibilities to manage topics. In a supply chain, predicting the expiry date of a product based on its current state would be an example of data processing. In this case, whenever a supply chain entity such as a warehouse or carrier publishes the data, the PPSManager receives the data and executes the"predictExpiryDate()" process and relays the outcome to the relevant subscribers. Using topics, PPSManager identifies, schedules, and allocates tasks on a random computing unit for further computations and insights extraction.
Computing units: are the participants registered to provide computational services to the framework. These computers register to the network. PPSManager maintains and manages the registered computers for further task allocations.
Insights: are predictive and analytic insights gained from executed computations on the shared raw data.

3.2 Assumptions

There are also key assumptions in the proposed PPS middleware architecture that are described below: **Assumption 1:** A correct PPSManager (broker) is a trusted entity that is always online. **Assumption 2:** A correct publisher always publishes the messages following the topic and the protocol structure recommended by the PPS. **Assumption 3:** PPS manages a sufficient number of computing units to meet the processing demands of publishers and subscribers. For simplicity, we assume the computing units belong to PPSManager. **Assumption 4:** Correct publishers trust the PPSManager to process the raw data and share the insights with the subscribers. **Assumption 5:** The computational tasks do not include malicious code, including malware. **Assumption 6:** The

computational tasks are terminated within a bounded delay. **Assumption 7:** All the communication between PPS, publishers, subscribers, and computational units are secured. All messages published by the correct publishers are delivered exactly the same as the messages received by the PPSManager. **Assumption 8:** If a subscriber receives a processed message, all the other interested subscribers must receive the same processed message within a bounded delay. **Assumption 9:** The order of the messages is preserved. Therefore, if message m is received before message m' by a correct subscriber, all subscribers will receive the same messages in the same order. **Assumption 10:** PPS do not charge the publishers and subscribers for the computation. We will investigate a pricing model for PPS in our future work. **Assumption 11:** We assume the maximum computation time for each computation. This can be calculated by running the computations repeatedly for n number of times at the PPSManager.

3.3 System Model

We consider a system with a broker $\mathcal{B}$ and a set of publishers $\mathcal{P}$ and subscribers $\mathcal{S}$. Each subscriber $s_m \epsilon \mathcal{S}$ needs certain processed data d_x for its application. To get d_x, the subscriber, s_m, must receive raw data, r_x from broker $\mathcal{B}$ and execute a computation c_x to receive the processed data d_x. Existing publish-susbcribe systems only allow the subscriber to receive r_x without any processing support for generating d_x on the fly. Allowing the broker to process r_x using the computation c_x, the subscriber can directly receive the processed data d_x without any processing locally. Traditional publish-subscribe systems allow subscribers to procure computation resources $\mathcal{R}$ to run computations. Assuming the subscriber is interested in receiving several processed data d_1, d_2, ..., d_x, ... d_n, it has to have sufficient computing resources to process all the data. Offloading the processing to the broker would reduce the compute management process for the subscriber while still providing access to the processed data. To that end, our work extends $\mathcal{B}$ with support for computations through a set of managed computing resources, $\mathcal{R}$.

In traditional publish-subscribe, each subscriber has to run the processing on shared r_x locally to extract d_x. This study calls this reactive manner of processing raw data PS-PbyS as Publish-Subscribe, with Processing by Subscribers. If the system has m subscribers, then the total number of computations needed to convert r_x to d_x for all the subscribers is m*c_x. Our proposed publish-process-subscribe processes c_x once per data item and directly shares d_x without needing local computations at the subscriber's end. As the complexity of c_x increases, our approach significantly saves the resource consumption of subscribers. Figure 3 compares our proposed PPS system and PS-PbyS approach regarding the number of computations executed to get processed data d_x as insights. As the message arrival rate increases, the total number of computations to obtain d_x differ significantly.

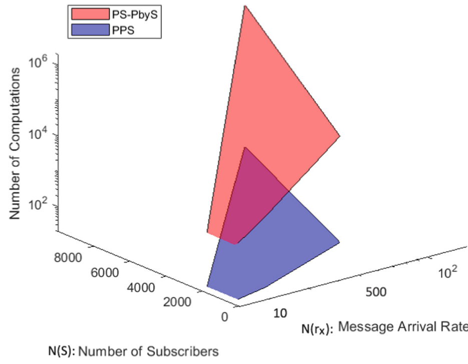

Fig. 3. Number of Computations Comparison in PPS and PS-PbyS

3.4 Existing and Newly Introduced Control Packets

Control packets are the exchanged packets between the server and clients. In this section, we introduce new control packets for the proposed PPS architecture to augment existing MQTT control packets [8].

Table 1. Existing Control Packets (✓) and Proposed Control Packets (⊕)

Control Packet	Description	Status
CONNECT	Connection request	✓
PUBLISH	Publish message	✓
SUBSCRIBE	Subscribe request	✓
UNSUBSCRIBE	Unsubscribe request	✓
ADVERTISE	Advertise topics	⊕
DISCOVER	Discover topics	⊕
QUERY	Query registered computing units	⊕

Using these control packets, publishers can ADVERTISE topics they use to publish messages, sell the data published under those topics in the data marketplace and help subscribers to DISCOVER these topics. Besides subscribers, data buyers can also DISCOVER topics and connect to sellers through PPSManager. QUERY control packets help the PPSManager to query a list of registered computing units for task scheduling and allocations.

Based on the mentioned assumptions and introduced control packets, the system description is provided below.

4 Architecture of PPS Middleware

Publishers and subscribers in this architecture must connect to the PPSManager the same as they connected to the broker in MQTT [25]. They use the same APIs to Publish and Subscribe to different messages, and Subscribers can Discover topics to investigate their interest in subscribing to new shared data. Publishers can Advertise topics of their publicly published messages, and buyers in the Data Marketplace can aim to trade their computational infrastructure with data [20]. As mentioned in assumption 8, computing units belong to the PPSManager and are registered separately to the networks based on their availability. There is a query of the registered computing units at PPSManager with connection details like their IP addresses. PPSManager has a wildcard subscription to all topics. Therefore, it receives all messages, investigates published messages by their topics, and allocates computation-related messages to these computing units. These messages include temporal data that is needed for insights extraction. PPSManager investigates these topics and allocates them as a task with an updated topic to the computing units. Computing units are picked randomly by PPSManager, and the required functions and algorithms are predefined on the computing units. Based on the topic of the allocated tasks, they automatically execute related functions and algorithms. The insights are published under the Advertised topics to subscribers, who can subscribe to the topics by discovering them using the proposed control packets in Table 1. Computations are bound to a deadline, and computing units must execute these computations before the deadline [13].

Messages in our proposed PPS are published under topics, and each topic can trigger specific functions to be executed on the information. For instance, in a supply chain, a distribution truck unloads products at different places. Multiple IoT sensors in the truck publish temporal messages regarding the container's temperature, current location and speed. Message publishers in this example are the Time and Temperature (TTI), Relative Humidity (RH), and Global Positioning systems (GPS) sensors. TTI sensors publish temperature-related messages and trigger functions required for calculating the remaining shelf life of a perishable product and spoilage rate prediction algorithms based on the shared temperature values. Latitude and Longitude details of the truck's current location published by Global Positioning systems (GPS) sensors trigger functions related to calculating the travelled distance and the algorithms for finding the optimal and nearest destination to unload the products. Executing these computations requires a computational infrastructure capable of running complex computations.

Shared data published by each publisher provides information regarding that sensor only. For instance, temperature sensors include temperature-related values only, and if computations need to extract insights from these values, they must combine them with other published messages. Therefore, if computations are executed on a single publisher's data, they provide limited analysis of that sensor. However, to extract insights, there is a need to execute computations on the fusion of data from different publishers. Therefore, the processing must

happen within the middleware, and shared data must be combined to result in insights. As existing data-sharing middlewares lack a processing layer within their messaging models, subscribers must collect and process shared data individually. Executing computations after raw shared data are received by subscribers results in insight in a reactive manner. In the mentioned existing data sharing middlewares that lack built-in data processing support, subscribers must collect, analyse and process data separately. Therefore, considering n subscribers subscribing to the same topic in a publish-subscribe middleware, subscribers execute n same processes on the same shared message separately. On the other hand, if shared data like the temporal published messages by sensors are not processed instantly, the computation results are useless for predictive supply chains. Our Publish-Process-Subscribe (PPS) data sharing and processing middleware addresses this issue with a built-in processing layer where shared data are processed automatically within the messaging middleware. PPS trades off insights and reduced computation replication with a bounded delay for predictive supply chains.

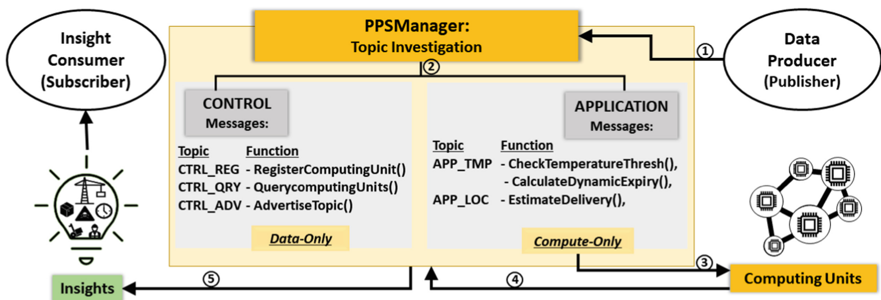

Fig. 4. Message Topic Investigation

4.1 Topic Management

PPSManager in PPS middleware architecture provides data processing facilities. Therefore, PPSManager must investigate topics to distinguish messages that carry temporal data as data extraction materials from messages that include general information like a new node registration in the system. As indicated in Fig. 4, we categorise messages into two separate categories, the Control Messages and the Application Messages.

Control Messages are Data-only messages representing statistical information published to the network to make an announcement. These messages are published to the PPSManager for certain actions like registering a computing node or a new subscriber. There is no need to execute complex computations to extract insights from these messages, so they will be forwarded directly to their topic subscriber after running the required actions.

On the other hand, *Application Messages* are Compute-only messages, indicating that these messages include vital information that must be executed to gain insights. After investigating the topic of each message, PPSManager treats these messages as tasks and allocates them to random computing units for insights extraction. Each topic triggers specific functions and algorithms for execution by computing units. For instance, as shown in Fig. 4, messages published under APP_TMP show PPSManager that they must be treated as tasks, and the required functions to be executed on them are to check the temperature threshold and calculate the dynamic expiry date. Each Data-Only message includes the payload section, following the JSON message format and JSON payload structure. Payloads of each message contain the required values for computations, and tasks are the required computations that must be executed on these values.

4.2 Task Scheduling and Load Balancing

Each message's topics help PPSManager determine whether this message needs computations or must be forwarded to its topic subscribers directly. Messages that include temporal data in their payload are allocated as tasks. These tasks are scheduled instantly as jobs that need to be processed, and the priority for tasks to be allocated is first in, first served. In our proposed task scheduling algorithm for PPS, every time a new message arrives, PPSManager queries registered computing units to list available computing units and allocates tasks to them in a round-robin fashion. Task scheduling is done in the message arrival order, and allocations on the computing units start with the first available computing unit. Equation 1 determines the required number of computing units (N(C.U.)) for executing tasks. In this Equation, C_{Dem} and C_{Cap} are Computation Demand and Computation Capacity. Computation Demand(C_{Dem}) is calculated with R, the message arrival rate indicating the number of messages received per Second, multiplied by the number of functions each topic needs to compute, N_f. Computation Capacity C_{Cap} is also the number of computations that the system can handle per second.

$$N(C.U.) = C_{Dem}/C_{Cap} \tag{1}$$

$$C_{Dem} = R * N_f \tag{2}$$

In [13], the authors implemented an algorithm for task scheduling where computing units are scheduled based on the deadline. The initial allocations must satisfy the required number of computing units, and once the deadline is at risk, a new computing unit is injected to assist with the computations. Likewise, in our PPS middleware, as allocations are random and instant, the first random set of computing units that can execute all functions of computations within the bounded time is chosen for the job. In this best-fit-first method, resources must execute computations in a time-bounded manner. The maximum execution time

in demand capacity derived from the Eq. 2, ensures we avoid under-provisioning the initial number of resources. We use ET_{max} as a restriction to guarantee that the service rate is faster than the data arrival rate.

Algorithm 1: Task Scheduling and Load Balancing.

Input: **D** ← Deadline,
M ← A set of Registered Computing Units,
T_{int} ← Sensors' Delay Interval per Message,
Result: Allocate Computing Units
1: calcluate **R: Rate of Message Arrival** using Sensors' Delay Interval T_{int};
2: calcluate **Total Computing Units Needed** using the Equation 1 ;
3: **for** <each i ∈ Tasks> **do**
4: **for** <each j ∈ Total Computing Units Needed> **do**
5: Allocate **M[j]** to execute **Task[i]**;
6: **for** Duration of T_{int} Publish Message Interval **do**
7: Idle **M[j]**;
8: **end for**
9: **if** No result for **Task[i]** in **D then**
10: Allocate **Task[i]** to M[j+1];
11: Terminate M[j];
12: **end if**
13: **end for**
14: **end for**
15: **return** the Set of Allocated Computing Units

5 Safety and Liveness Analysis

While the liveness property implicitly ensures that the system will advance and that "something positive will always happen", the safety property establishes that "something wrong will never happen" [2]. In this part, we demonstrate our proposed PPS's liveliness and safety qualities.

5.1 Safety Properties

Publisher's safety: A correct PPSManager can guarantee the published messages are delivered to topic subscribers. **Messages' safety:** All correct subscribers receive the same messages, including the same information and computational results. Also, all the correct subscribers receive the messages in the same order it has been delivered to all other subscribers. **PPSManager's safety:** All correct computing units are online. PPSManager can Query the registered computing units and allocate tasks on them, which must be executed within a bounded delay. If a task is not completed by a deadline, that task is allocated to another computing unit immediately.

There are always enough computing units available for executing computations. The minimum number of available computing units ($minN_{CU}$) for task allocations, resulting from a QUERY control packet by the PPSManager must match this Eq. 3:

$$minN_{CU} = n(T_{adv} \cup T_{discov}) * C_{Dem} \quad (3)$$

Where T_{adv} is the set of Advertised topics by publishers and T_{discov} is the set of Discovered topics by subscribers. Computation Demand (C_{Dem}) also results from Eq. 2.

5.2 Liveness Properties

Messages' liveness: Messages published correctly by the correct publishers will be delivered to the message's topic subscribers within a bounded delay. **Computing units' liveness:** Correct computing units are always online, and in each Query, there are at least an equal number of computing units with the number of topics. **Computations Liveness:** Computations are bounded by a deadline, and if computing units fail to execute computations or go offline, computations are allocated to another computing unit.

6 Implementation and Evaluation

This section discusses the implementation and evaluation of the proposed Publish-Process-Subscribe (PPS) middleware.

6.1 Experimental Setup

We implemented the PPS framework on a local testbed using Mosquitto, an open-source message broker that implements the MQTT protocol Broker. MQTT is one of the widely used publish-subscribe communication protocols in IoT applications. The broker-specific functionalities are implemented on top of MQTT, called PPS-Managers in our middleware. To obtain a real-world experiment, we used Teensy 2.0 development boards, DHT11 sensors to produce temporal temperature and humidity data, and HC-05 Bluetooth transceivers to communicate with a Raspberry Pi to publish data. An Amazon EC2 instance was also set up to work as the PPSManager, and the computations were allocated on an Apple M2 chip with an 8-core CPU and 10-core GPU with 100GB/s memory bandwidth for both PPS and PSPbyS scenario. Application-specific software were implemented in Arduino, Python and NodeJS.

6.2 Use Case Application of PPS Data Sharing and Processing Middleware

To justify the usability of our proposed PPS platform, we rely on a real-life need for data processing within a data-sharing middleware where the remaining life of perishable products in a supply chain distribution must be calculated. As mentioned in Sect. 2, [10] proposed a Decision Support System (DSS) to efficiently manage and prevent perishable loss by incorporating established functions for food quality assessment in dynamic settings. They used this equation:

$$KQ = \frac{Q_0 - Q_L}{k}$$

to calculate the remaining life of a perishable, where the Q_0 presents the current quality of the product and Q_L is the quality limit. K is the spoilage rate which is calculated as:

$$k = k_{ref} * exp^{\frac{EA}{R} * \frac{1}{T_{ref}} - \frac{1}{T}}$$

in which T is the given temperature and k_{ref} indicates the spoilage rate at a reference temperature T_{ref}. EA and R are also the activation energy and gas constant [10].

In our scenario, a truck distributes perishable products to different stores. There are sensors in the truck's container that publish temporal temperature data and GPS sensors to publish the truck's current location. Published temperature values are used in the mentioned equations to calculate the remaining life of perishables. Location parameters, namely latitude and longitude values, are also published to indicate the updated location of the distribution truck. Messages, including temperature and location data, are published as Application messages and trigger temperature and location-related functions. For instance, the temperature threshold is assessed (by CheckTemperatureThresh() function), and the dynamic expiry date (by CalculateDynamicExpiry() function) is calculated automatically using the payload values and mentioned equations. However, if the shared data gets processed within the proposed PPS data-sharing and processing middleware, it can benefit a proactive supply chain with near-real-time insights. Otherwise, in a reactive method like the MQTT, where computations are pushed at the subscribers' end, computational results only help supply chain stakeholders perform reactive actions.

6.3 Performance Evaluation

In this section, experimental results and evaluations are discussed. We conducted experiments to evaluate our PPS middleware performance and to compare it with the existing PS-PbyS approach. To compare and contrast PPS and PS-PbyS approaches in a similar environment, we evaluated an ideal setup for the PS-PbyS approach where the subscriber has a powerful computational infrastructure and executes computation immediately after messages are delivered.

In these evaluations, based on the perishable products use case Sect. 6.2, DHT11 sensors are used to generate temporal temperature and humidity values, and using a Raspberry Pi as a publisher, messages with a payload including these values are published. These messages are published under a specific "Temperature" topic, and this topic triggers related functions to execute. Temperature and humidity-related functions include the calculation of perishables' dynamic expiry and the remaining life, as well as the Moving Average function to predict the next expected temperature and the remaining life based on the previously published messages. The minimum execution time of the mentioned functions is 251 Milliseconds after more than 10000 runs, and the number of messages sent for each message arrival rate has been 5000 messages sent to the broker (PPSManager in PPS) per each arrival rate run.

Figures 5 and 6 indicate the end-to-end delay of extracting insights by executing computations in both PS-PbyS approaches and our proposed PPS middleware. Due to the high latency of the messages being executed in the PS-PbyS approach and to better illustrate the distinctions, we plot the end-to-end delay in the logarithmic scale.

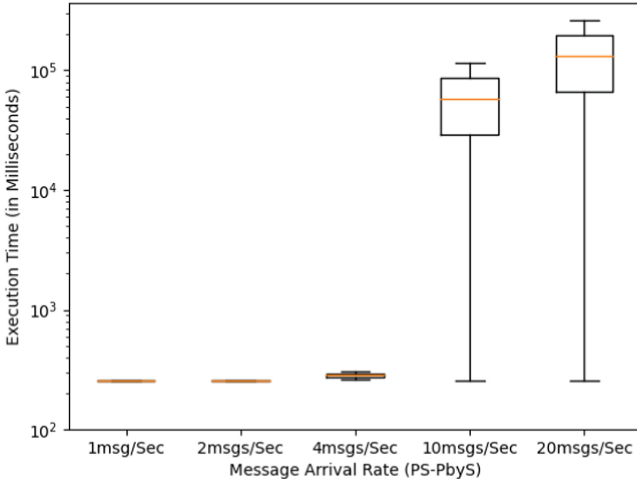

Fig. 5. The End-to-End Delay in PS-PbyS

While the message arrival rate is low, the end-to-end delay in PS-PbyS fashion and the PPS has the same trend with the minimum execution time fluctuations in milliseconds, within the one message per Second to the four messages per Second arrival rate. However, when the number of published messages increases, there is a significant change in the total execution time for 10 and 20 messages published per Second as the service rate falls behind the message arrival rate. This drastic change in execution time occurs when the load increases and more messages arrive at the subscribers' end. This enormous rise in delay for 10 and 20 messages published per Second leads to significant changes in the graph where the lower message arrival rates (1, 2 and 4 msgs/Sec in Fig. 5) seem flattened. In this experiment, one publisher produces the raw data and the subscriber in the PS-PbyS approach receives and processes the raw data. This subscriber then executes the same computations and functions that are evaluated in PPS to extract the required insights. However, as the PS-PbyS approach lacks load balancing, an inevitable bottleneck occurs as the message arrival rate increases.

In PPS, this bottleneck is avoided by obtaining the resource allocations in Algorithm 1, where the PPSManager first subtracts the required number of computing units using Eq. 3 from the list of registered computing units by the QUERY control packet in Table 1. As mentioned earlier in Sect. 4.2, this resource allocation follows a round-robin fashion and allocates tasks efficiently to the

minimum number of resources available. PPSManager uses this algorithm to choose a sufficient number of resources from the list of registered computing units, and after processing the topics Sect. 4.1, it allocates each message to an available computing unit and activates them to execute the computations. The number of activated computing units for task allocations is mentioned in Table 2. This Table 2 shows the results of dynamic and efficient resource allocations as the message arrival rate fluctuates.

Table 2. Computing Unit (CU) Distribution for Various Message Arrival Rate in PPS

Arrival Rate	Number of Activated Computing Units	Activated Computing Units
1msg/s	1	[CU1]
2msgs/s	1	[CU1]
4msgs/s	3	[CU1, CU2, CU3]
10msgs/s	5	[CU1, CU2, CU3, CU4, CU5]
20msgs/s	10	[CU1, CU2, ..., CU10]

Figure 6 shows the logarithmic scale of end-to-end delays in PPS. As the PPS handles the different message arrival rates by allocating the needed number of resources when a message is published and an event occurs, the end-to-end delay remains stable thanks to the PPSManager's load balancing. The number of activated computing units for each message arrival rate is also mentioned in Table 2, and the number of activated computing units to execute the tasks is the minimum and efficient number of computing units vital to prevent system bottlenecks. This minimum number of required computing units is derived from the earlier discussed in Eq. 1.

There are also some inevitable rises in performance while using PPSManager instead of a plain broker. These changes are investigated in Table 3, and as the results indicate, CPU and Memory usage increase once we add data processing features on top of the PPSManager. However, the network's bandwidth decreases significantly on the subscribers' end as the computations are executed before delivering insights. Therefore, by handling processes automatically, PPS middleware decreases throughput from the subscribers' end.

Table 3. Broker's Performance in PS-PbyS and PPS for 20msgs/s.

Data Processing Model	Average CPU Usage	Average Memory Usage	Broker's Network Bandwidth	Subs' Network Bandwidth
PS-PbyS	5.2%	0.1%	922.35 kbit/s	996.21 kbit/s
PPS	48.9%	1.5%	952.45 kbit/s	289.93 kbit/s

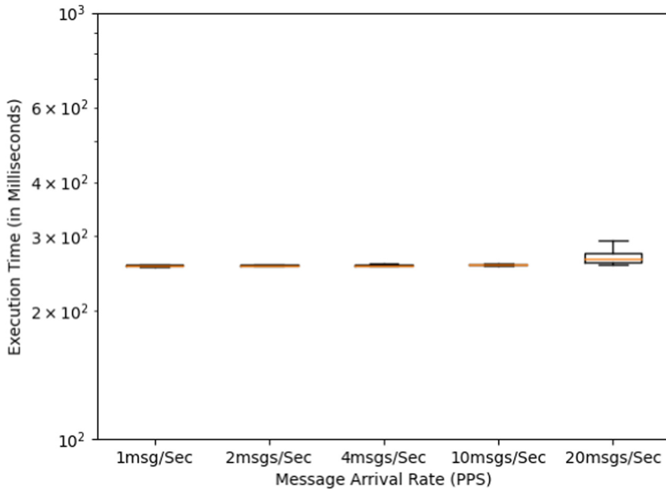

Fig. 6. The End-to-End Delay in PPS

7 Conclusion

We proposed a Publish-Process-Subscribe framework architecture, in which the processes are automated, and predictive and analytic computations are executed rigorously to provide insights in favour of modern predictive supply chains. An interesting direction for future work is to decentralise the system and to guarantee the trustworthiness of processes and computations that are executed in our processing layer. Another open requirement is privacy-preserving techniques in which we guarantee stakeholders that the data are only shared with whom it is meant to be and the computations and computational results do not risk privacy concerns of the data or data producer.

Acknowledgement. This work was funded and supported by iMOVE Australia Cooperative Research Centre (CRC).

References

1. Amazon Web Services. https://aws.amazon.com/
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
3. Antonić, A., Marjanović, M., Pripužić, K., Žarko, I.P.: A mobile crowd sensing ecosystem enabled by CUPUS: cloud-based publish/subscribe middleware for the Internet of Things. Futur. Gener. Comput. Syst. **56**, 607–622 (2016)
4. Baldini, I., et al.: Serverless computing: current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-5026-8_1

5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: stream and batch processing in a single engine. Bull. Tech. Committee Data Eng. **38**(4) (2015)
6. Čilić, I., Žarko, I.P.: Adaptive data-driven routing for edge-to-cloud continuum: a content-based publish/subscribe approach. In: González-Vidal, A., Mohamed Abdelgawad, A., Sabir, E., Ziegler, S., Ladid, L. (eds.) GIoTS 2022. LNCS, vol. 13533, pp. 29–42. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-20936-9_3
7. Cui, L., Gao, M., Dai, J., Mou, J.: Improving supply chain collaboration through operational excellence approaches: an IoT perspective. Ind. Manag. Data Syst. **122**(3), 565–591 (2022)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. (CSUR) **35**(2), 114–131 (2003)
9. Fahmideh, M., Zowghi, D.: An exploration of IoT platform development. Inf. Syst. **87**, 101409 (2020)
10. Fikar, C.: A decision support system to investigate food losses in e-grocery deliveries. Comput. Ind. Eng. **117**, 282–290 (2018)
11. Garcia, C.A., Naranjo, J.E., Garcia, M.V.: Analysis of AMQP for industrial Internet of Things based on low-cost automation. In: Iano, Y., Arthur, R., Saotome, O., Kemper, G., Padilha França, R. (eds.) BTSym 2019. SIST, vol. 201, pp. 235–244. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-57548-9_22
12. IOTA-Foundation: IOTA Data Marketplace. https://data.iota.org/#/demo/#list
13. Jabbari, A., Masoumiyan, F., Hu, S., Tang, M., Tian, Y.C.: A cost-efficient resource provisioning and scheduling approach for deadline-sensitive mapreduce computations in cloud environment. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 600–608. IEEE (2021)
14. Krishnamachari, B., Power, J., Kim, S.H., Shahabi, C.: I3: An IoT marketplace for smart communities. In: Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, pp. 498–499 (2018)
15. Lazidis, A., Tsakos, K., Petrakis, E.G.: Publish-subscribe approaches for the IoT and the cloud: functional and performance evaluation of open-source systems. Internet Things **19**, 100538 (2022)
16. Minerva, R., Lee, G.M., Crespi, N.: Digital twin in the IoT context: a survey on technical features, scenarios, and architectural models. Proc. IEEE **108**(10), 1785–1824 (2020)
17. Narkhede, N., Shapira, G., Palino, T.: Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly Media, Inc. (2017)
18. Nasirifard, P., Jacobsen, H.A.: A serverless publish/subscribe system. arXiv preprint arXiv:2210.07897 (2022)
19. Nelson, B.J.: Remote Procedure Call. Carnegie Mellon University (1981)
20. Sajan, K.K., Ramachandran, G.S., Krishnamachari, B.: Enhancing support for machine learning and edge computing on an IoT data marketplace. In: Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things, pp. 19–24 (2019)
21. Scott, D., Gamov, V., Klein, D.: Kafka in Action. Simon and Schuster (2022)
22. Seyedan, M., Mafakheri, F.: Predictive big data analytics for supply chain demand forecasting: methods, applications, and research opportunities. J. Big Data **7**(1), 1–22 (2020)
23. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (CoAP) (2014)

24. Sleuters, J., Li, Y., Verriet, J., Velikova, M., Doornbos, R.: A digital twin method for automated behavior analysis of large-scale distributed IoT systems. In: 2019 14th Annual Conference System of Systems Engineering (SoSE), pp. 7–12. IEEE (2019)
25. OASIS Standard: MQTT version 3.1.1 (2014).http://docsoasis-open.org/mqtt/mqtt/v3
26. Streamr: Streamr (2019) https://www.streamr.com/marketplace
27. Tan, L., et al.: Toward real-time and efficient cardiovascular monitoring for Covid-19 patients by 5G-enabled wearable medical devices: a deep learning approach. Neural Comput. Appl. 1–14 (2021)
28. Terbine.IO.: Terbine.IO (2019). https://www.terbine.io/
29. Vinka, E., Johansson, L., Kafka, A.: Cloudkarafka (2019). https://www.cloudkarafka.com
30. Vohra, D.: Apache Kafka. In: Practical Hadoop Ecosystem, pp. 339–347. Springer, Cham (2016). https://doi.org/10.1007/978-1-4842-2199-0_9
31. Will, J., Thamsen, L., Bader, J., Scheinert, D., Kao, O.: Get your memory right: the crispy resource allocation assistant for large-scale data processing. In: 2022 IEEE International Conference on Cloud Engineering (IC2E), pp. 58–66. IEEE (2022)