sustainability

Article

The PeRvasive Environment Sensing and Sharing Solution

Paolo Bellavista [1,*], Carlo Giannelli [2] and Riccardo Zamagna [1]

[1] DISI—Department of Computer Science and Engineering, University of Bologna, 40136 Bologna, Italy; riccardo.zamagna@studio.unibo.it

[2] DMI—Department of Mathematics and Computer Science, University of Ferrara, 44122 Ferrara, Italy; carlo.giannelli@unife.it

* Correspondence: paolo.bellavista@unibo.it; Tel.: +39-051-209-3866

Academic Editor: Sehyun Park
Received: 6 March 2017; Accepted: 6 April 2017; Published: 11 April 2017

Abstract: To stimulate better user behavior and improve environmental and economic sustainability, it is of paramount importance to make citizens effectively aware of the quality of the environment in which they live every day. In particular, we claim that users could significantly benefit from cost-effective efficient Internet-of-Things (IoT) solutions that provide them with up-to-date live information about air pollution in the areas where they live, suitably adapted to different situations and with different levels of dynamically selected granularities (e.g., at home/district/city levels). Our PeRvasive Environment Sensing and Sharing (PRESS) project has the ambition of increasing users' awareness of the natural environment they live in, as a first step towards improved sustainability; the primary target is the efficient provisioning of real-time user-centric information about environmental conditions in the surroundings, and in particular about air pollution. To this purpose, we have designed, implemented, and thoroughly evaluated the PRESS framework, which is capable of achieving good flexibility and scalability while integrating heterogeneous monitoring data, ranging from sensed air pollution to user-provided quality perceptions. Among the elements of technical originality, PRESS exploits extended Kura IoT gateways with novel congestion detection and recovery mechanisms that allow us to optimize bandwidth allocation between in-the-field PRESS components and the cloud. The reported performance results show the feasibility of the proposed solution, by pointing out not only the scalability and efficiency of the adopted message-based solution that uses Message Queue Telemetry Transport (MQTT) and WebSockets, but also the capability of PRESS to quickly identify and manage traffic congestions, thus, ensuring good quality levels to final users.

Keywords: environmental monitoring; heterogeneous monitoring data; scalability; dynamic extensibility; Traffic Congestion Management; MQTT; WebSockets

1. Introduction

It has started to be widely recognized that to increase the environmental and economic sustainability of user behavior, it is of paramount importance to make users aware of the quality of the environment they live in every day. In fact, based on enhanced awareness of the current and actual state of ecological and societal aspects of their environments, citizens are able to make better decisions, for e.g., pushing them towards a more responsible and informed freedom of choice, which possibly includes the modification of personal behaviors and also demanding more sustainable policies from the municipal/regional/national authorities.

In particular, we claim that citizens could significantly benefit from up-to-date live information about air pollution in the areas where they live in, with different home/district/city granularities.

For instance, by providing users with current data about air pollution in their district in a pervasive and time-continuous manner, they will be able to assess the overall quality of their neighborhood, thus, possibly triggering more environment-friendly habits and pushing them to ask municipalities to undertake stronger actions for air quality improvements. Similar considerations could apply and have an impact on the real estate market, influencing the value of homes based on current/average/peak air pollution levels. In addition, the actions of smart mobility managers can be influenced by this collective awareness in order to induce and incentivize specific actions at home/district levels. To enable the above scenario by benefitting from already deployed and dynamically discovered sensors, there is the need to consider air pollution data sensed from a plethora of heterogeneous sensing devices, as well as to get an aggregated view of the data sensed in a given geographical area and in a specific time period. In this manner, while sensors located in given localities will be useful to gather information on the private surroundings of specific users, with the correct levels of visibility/granularity, a more aggregated and summarized view will positively affect the knowledge of air quality in wider areas of an overall smart city.

Within this perspective, our PeRvasive Environment Sensing and Sharing (PRESS) project supports citizens in becoming more aware of the natural environment they live in and, thus, providing the basis to press them for behavioral improvement through real-time information about air pollution and, more generally, environmental conditions related to their surroundings with the correct level of granularity. This high-level goal has required us to face many hard and still open technical challenges. First of all, there is the need to design a monitoring system able to manage very heterogeneous data, coming from many differentiated, existing, and legacy sources, for e.g., ranging from quantitative information about air pollution to qualitative considerations about perceived vehicle traffic, and also crime rate (environmental quality in a large sense). In addition, the PRESS framework must be able to easily gather, store, and process additional information that could be identified after first deployment, while not impacting its architecture and without any service interruption (extensibility during service provisioning). Moreover, the volume of managed data is likely to dramatically increase in the future, thus requiring a highly scalable solution able to easily manage growing workloads in an effective and efficient way, again, without imposing any service provisioning interruption. Finally, it is also required to consider that abrupt workload spikes can occur and could overcome the transmission/computational capabilities of the PRESS resources; this calls for introducing mechanisms and policies that can promptly and effectively detect congestions, and consequently take proper countermeasures to ensure high levels of service quality and continuity.

In particular, this paper originally focuses on the original aspects of the design and implementation of our PRESS framework and of some application examples on top of it. On the one hand, we have developed, deployed, and thoroughly evaluated our PRESS framework, which is based on the Spring technology framework and on the widespread Message Queue Telemetry Transport (MQTT) protocol, and verified its scalability when gathering and storing datastreams of significant volume. On the other hand, the paper describes how we have developed our PRESS IoT gateway on top of the open-source Kura solution to facilitate and optimize the gathering and delivery of monitoring data from sensors to the cloud. We have selected Spring as the basis of our IoT platform because Spring has widely demonstrated its good extensibility and flexibility; it provides developers with a wide set of features (in particular the Spring Integration project, as better detailed in Section 3.2), and this has made it a reference base technology for highly flexible Web applications. In addition, the MQTT protocol is nowadays the standard de facto for publish/subscribe protocols in IoT scenarios; it has significantly gained momentum within the academic and industrial community, not only thanks to its reduced overhead, but also because of its support to differentiated Quality of Service (QoS) levels, as described in Section 3.2. Moreover, the Kura project represents a very interesting innovative initiative, already with a good community of associated IoT developers, that primarily aims at simplifying the development of IoT gateways by providing basic mechanisms for the dynamic deployment of software components (see Section 3.1 for additional details).

Moreover, the paper also presents the adopted congestion detection and recovery mechanisms that allow for optimiation of bandwidth allocation between in-the-field components and our server-side applications, typically hosted on the cloud. Finally, it is worth noting that our PRESS framework can be easily and modularly extended to integrate a larger set of monitoring devices and a wider set of heterogeneous data, for e.g., ranging from traffic conditions to energy consumption, thus, further pushing the general concept of wide-spectrum awareness of the environmental situation for sustainable users' behaviors at home/district/city levels.

2. Design Principles

The PRESS solution stems from the exploitation of the Internet of Things (IoT) paradigm in dynamic and open environments. PRESS specifically addresses environmental concerns related to the lack of fine-grained information about the environment by taking into consideration its relevant variability in different hours of the same day and in different streets of the same neighborhood.

It is worth noting that the fine-grained monitoring of environmental conditions raises issues related to both the high heterogeneity of information from very differentiated sensors, for e.g., ranging from quantitative geo-localized air-pollution values measured by specialized high-cost sensors to qualitative traffic evaluations submitted by final users, and the great amount of collected data, considering that each user equipped with a sensor can potentially contribute to gathering and sending observation data. To address these issues, we claim the need to properly manage:

- **Data heterogeneity**. On the server-side, software components in charge of receiving and storing monitoring information must be able to manage multiple (and possibly any) types of data, independently of their associated semantics. In this manner, the impact of newly added information would be minimized, requiring only the enhancement of the Graphical User Interface (GUI) to show the new data, but not imposing modifications to any server-side core modules. Once data are stored, it is possible to adopt solutions aiming at extracting high-level semantic information from raw data (out of the scope of this specific paper), thus, making easier their joint exploitation [1,2];
- **Pluggability and extensibility**. The adoption of a flexible and extensible software architecture, at both client and server sides, easily allows new data types to be seamlessly gathered, managed, and stored. In this manner, it is possible to gather previously not considered information, for e.g., manually provided by users or available through new plug&play hardware, without the burden of recompiling and reconfiguring the participating clients software. Note that our PRESS framework follows the more general principle of "cloud-based architecture" as presented in [3], by allowing to provide new data in a plug&play and extensible manner as long as client-side software components interact with server-side components while respecting its standards, for e.g., by exploiting one of the supported protocols;
- **Efficiency**. The adoption of messaging-based solutions, in particular MQTT [4] and WebSocket [5] standard protocols, increases decoupling and efficiency of data dispatching when considering both the PRESS cases of client-to server and server-Web-based GUI communications, as detailed later [6,7]. On the one hand, the MQTT protocol is generally recognized as the de facto standard to dispatch packets in IoT scenarios considering its limited memory footprint and transmission overhead; these characteristics also make it a suitable solution for resource-constrained clients. In addition, MQTT allows for the adoption of a publish/subscribe approach to seamlessly dispatch the same information to several interested destinations, for e.g., sending the same temperature setpoint to every thermostat of a given building. On the other hand, the WebSocket protocol can push new data from the server to one or more instances of the PRESS Web GUI, enhancing the real-time visualization of new information;
- **Scalability**. The adoption of publish/subscribe interaction models also greatly improves the scalability of message dispatching, not only because of their limited overhead, but also since

they can be easily scaled horizontally, for example, by gracefully adding computational resources at runtime in the case of workload growth. On the one hand, in case the number of messages generated by sensors overcomes the typical workload that a single MQTT broker can process, it is possible to horizontally scale by deploying multiple brokers, connected either in a hierarchy or in a flat topology [8], or even by exploiting enhanced versions of MQTT. For instance, [9] proposes a novel solution to further increase scalability and quality in MQTT by supporting both traditional and broker-less multicast communication and by allowing multiple MQTT brokers to cooperate. On the other hand, if the amount of Web GUI instances accessing the PRESS framework via WebSocket overcomes the capabilities of a Web server, it is possible to seamlessly add new Web server replicas without any service interruption;

- **Congestion management**. The adoption of a dynamic and automatic mechanism to detect and overcome traffic congestion in case of heavy load can prevent service interruptions. In fact, despite the adoption of scalable and efficient communication protocols, there is the need to also consider the possibility that traffic spikes may overcome the actual capabilities of the network and client/server nodes. To ensure service continuity, it is required to efficiently monitor traffic conditions and, in case of abrupt quality degradation, to conveniently shape the actual ingress traffic to the PRESS components. In particular, we aim at ensure the dispatching of the most important (and thus with highest priority) messages and some of the regular messages, by finely tuning their delivery in relation to current traffic conditions and bandwidth capabilities. It is worth noting that in content-based publish/subscribe solutions such as [10], it is possible to exploit additional information to provide more articulated content-aware congestion detection and recovery mechanisms. However, in contrast to [10], our solution does not impose any constraint on message payload while correctly detecting and managing congestion, as better detailed below.

3. Architecture and Selected Implementation Insights

PRESS architecture consists of three primary components (see Figure 1):

- the field-side Gateway module, in charge of interacting with the physical world to gather data and send them to the server-side;
- the server-side Web-app module, receiving data from the Gateway module and dispatching them to users via a Web application;
- the client-side User Interface module, presenting data to the final user via a Web browser.

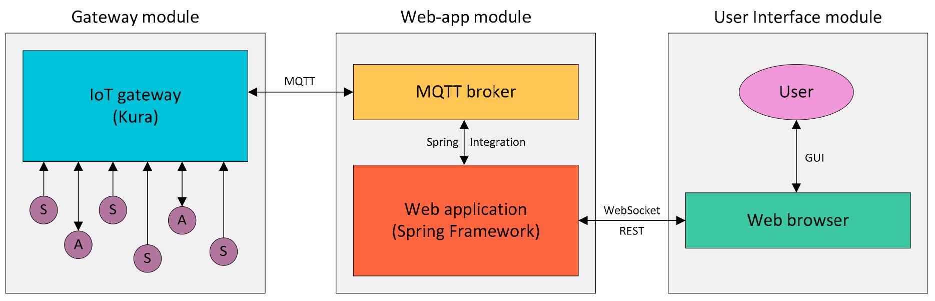

Figure 1. Overall architecture of the PeRvasive Environment Sensing and Sharing (PRESS) framework.

3.1. Gateway Module

The Gateway module is implemented based on the open-source project Kura [11], a Java-based Open Services Gateway initiative (OSGi) enabled framework specifically designed to be easily deployed on different kinds of devices, with different levels of available resources. We decided to exploit Kura as the basis of our PRESS Gateway module in order to take advantage of several features already supported by Kura, ranging from tools to easily manage software components from remote locations to well-defined Application Programming Interfaces (API) to dispatch/receive MQTT messages and setup networking interfaces. In particular, our Gateway module primarily exploits two components:

- DataPublisher, in charge of interacting with the Kura built-in Data Service component to delegate the sending of data and their filtering, if required;
- DataSubscriber, in charge of receiving packets and dispatching them to other components by registering itself to the Data Service as a DataServiceListener.

The current implementation of the PRESS Gateway module allows for the gathering of sensor data, such as air pollution and temperature, and is also designed to interact with actuators, for e.g., thermostats to change the setpoint temperature and mechanized windows to automatically open/close them (Figure 2). In this manner, it is possible to enable smart scenarios such as opening windows in the case of low O_2 levels in a room, while turning off heaters to limit energy waste. Once the air quality has become acceptable again, it is then possible to close mechanized windows and modify the setpoint temperature at the value that is considered more regular for that environment.

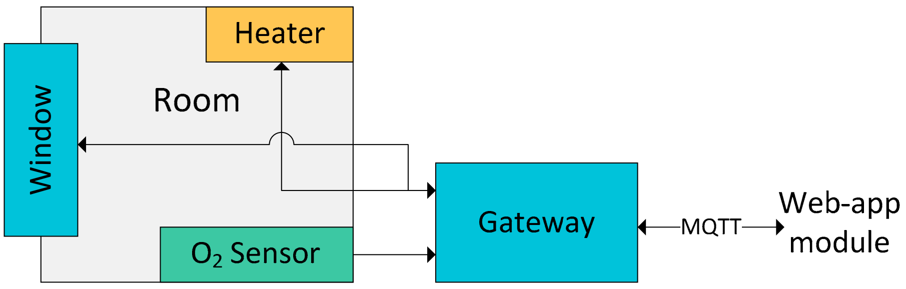

Figure 2. Typical deployment scenario of our PRESS Gateway module.

3.2. Web-App Module

The Web-app module is the core component of the PRESS framework. On the one hand, it behaves as a bridge between sensors/actuators and users, by implementing the support logic to receive and dispatch messages from/to the Gateway module via the MQTT broker. On the other hand, it is designed to contain the PRESS business logic, for e.g., receiving commands from users via either REpresentational State Transfer (REST) or WebSocket to change the setpoint temperature, dispatching them to the correct device, and then opening windows in case of low quality air.

The Web-app module includes the Mosquitto MQTT broker supporting all three different quality levels specified by the MQTT standard [12]:

- QoS 0 (at-most-once), dispatching packets in a best-effort fashion without any guarantee that packets are actually received by subscribers;
- QoS 1 (at-least-once), with the guarantee that packets are dispatched to subscribers, possibly multiple times via packet replication;
- QoS 2 (exactly-once), ensuring that each packet is received by subscribers once and only once.

As better detailed later, increasing the enforced quality level can improve the reliability of packet dispatching; but, at the same time, it also increases the communication overhead in a non-negligible way, by negatively affecting the scalability of the overall system.

The implementation of the Web-app message dispatching system is based on the Spring Integration project [13]. It is in charge of interacting with the MQTT broker by exploiting five primary channels (see Figure 3):

1. mqtt-inbound: a DirectChannel (very similar to a queue, since it is a one-to-one topic according to the Spring Integration terminology) in charge of interacting with the MQTT broker to receive messages published to a given topic and created by defining a <int-mqtt:message-driven-channel-adapter> in the Spring Integration configuration file;
2. mqtt-outbound: a DirectChannel in charge of interacting with the MQTT broker to send outgoing messages and created by defining a <int-mqtt:outbound-channel-adapter> in the Spring Integration configuration file;
3. mqtt-channel: a PublishSubscribeChannel used to allow multiple consumers to access messages arriving on the mqtt-inbound channel, required since DirectChannel allows only one consumer to get incoming messages (mqtt-inbound and mqtt-channel can interact via a simple bridge not shown in Figure 3);
4. jpa-adapter: an EventDrivenConsumer exploited to interact with a persistent database via Java Persistence Architecture (JPA), thus allowing for the storage of data in a completely transparent manner from the point of view of the application;
5. mqtt-async: an ExecutorChannel used to dispatch persistence operations to a thread pool, thus, greatly enhancing scalability and performance. In fact, the current implementation of the Spring jpa-adapter component is based on a blocking strategy, thus, potentially representing a critical bottleneck.

The mqtt-channel always has one and only one fixed consumer, i.e., the mqtt-async channel, and zero or more temporary consumers, i.e., one for each user connected to the Web application through WebSocket. On the contrary, REST-based users access information in an asynchronous manner, thus, they do not employ and burden the mqtt-channel.

Figure 3 shows the overall logical flow of message dispatching in PRESS. Solid arrows represent the main message flow. Dashed lines represent the message flows exploited only to send data to the User Interface module by exploiting either the WebSocket or the REST protocols; in the former case, the User Interface module receives only packets currently traversing the communication infrastructure and received through the mqtt-channel, in the latter case, it receives only packets already persisted in the database.

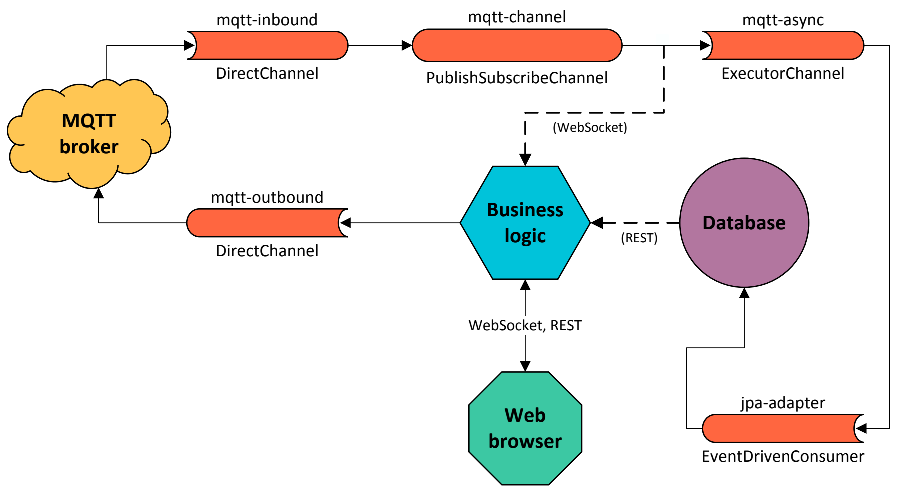

Figure 3. Overall communication infrastructure of the PRESS Web-app module.

To maximize the generality and usability of PRESS in several scenarios and in different application domains, the business logic of the Web-app module is easily and dynamically configurable through a JavaScript Object Notation (JSON) document (loaded and applied at PRESS boot). The PRESS JSON document contains three primary data elements:

- An MQTT topic that the Web-app module subscribes to in order to receive every message sent by every PRESS gateway deployed in-the-field. The topic has to be general enough to enable the successful read of every message sent by the Gateway module;
- An MQTT topic where the Web-app module publishes every message received by the User Interface module (typically representing commands sent by users to control in-the-field actuators). The Gateway module has to be registered to that topic to receive the messages sent by the Web-app module;
- A list of managed devices, each one characterized by a logical name, a brief description, and an MQTT topic where device-related messages are published. PRESS is based on the assumption that each device associates with a given topic; this requirement is non-limiting and does not negatively affect the achieved performance and overall scalability.

Figure 4 presents an example of a configuration file for four sensors/devices that expose monitoring data about temperature, light intensity, humidity, and presence of people. To integrate a new sensor/device, the PRESS configuration file is modified, and then the new sensed information becomes immediately available and accessible without any modification to the PRESS framework implementation itself. In this manner, it is possible to greatly improve the reusability of the whole PRESS solution, by taking advantage of its extendibility and dynamicity, and by easily including new and heterogeneous monitoring sources and associated monitoring data types even if not considered at system design and implementation time.

```
{
    "inboundBaseTopic": "iot/webapp/#",
    "outboundTopic": "iot/webapp/commands",
    "devices": [
        { "name": "temperature",
        "description": "A temperature sensor",
        "topic": "iot/webapp/temperature" },
        { "name": "light",
        "description": "A light sensor",
        "topic": "iot/webapp/light" },
        { "name": "humidity",
        "description": "A humidity sensor",
        "topic": "iot/webapp/humidity" },
        { "name": "proximity",
                    "description": "A proximity sensor",
        "topic": "iot/webapp/proximity" }
    ]
}
```

Figure 4. Example of the PRESS configuration file.

Finally, the Web-app module provides information to the User Interface module by exploiting either REST or WebSocket. The REST approach requires that the User Interface module explicitly retrieves information according to a pull scheme of interaction, based on the HyperText Transfer Protocol (HTTP) request/response protocol. In this case the User Interface module periodically sends a request to the Web-app module that is in charge of interacting with the database that maintains previously received data via MQTT, retrieves required information, and replys with an HTTP response. The freshness of information provided by the final users and the communication overhead depends on the frequency with which the User Interface module interacts with the Web-app module. By increasing the frequency it is possible to provide more up-to-date information; this comes at the cost of imposing additional load to the Web-app module and eventually sending requests even in cases where there are no new data to retrieve. On the contrary, the WebSocket approach proactively pushes data to the User Interface module as soon as (if and only if) new MQTT messages are dispatched to the Web-app module. In this case, final users always get up-to-date information, at the cost of imposing an always-on Transmission Control Protocol (TCP) connection between the Web-app and User Interface modules on which the WebSocket protocol is based.

Table 1 provides a brief comparison of required operations to receive data in case of usage of either REST or WebSocket. It is worth noting that REST-based communication is completely stateless (thus, not exploiting any session mechanism) because it follows the "stateless constraint" of the REST interaction paradigm.

Table 1. Comparison between REpresentational State Transfer (REST) and WebSocket operations.

Operation	REST (Pull)	WebSocket (Push)
Initial configuration	Not required	New WebSocket connection
List of available devices	HyperText Transfer Protocol (HTTP) GET request	Automatically sent at new WebSocket connection establishment
Expression of interest in a given device	Not required	Sending of a message
Get of new data	HTTP GET request with the list of interested devices	Automatically sent whenever new data is available
Send of user's commands	HTTP POST request	Sending of a message

3.3. User Interface Module

The entry point of the whole PRESS framework is a generic Web browser that retrieves the GUI from the Web-app module in terms of HyperText Markup Language (HTML) and JavaScript files. The adoption of a standard Web browser as the PRESS entry point greatly improves its usability and widens its applicability, since it allows any Internet-enabled device to parse/process/render HTML5 tags and JavaScript code (commonly supported in any modern desktop and smartphone) and behave as the User Interface module, without any deployment of any special purpose PRESS code. Figure 5 shows the main components of the User Interface module: the Java-based backend exploits the Spring Framework while the frontend is implemented in JavaScript. By delving into finer details, WebSocket communication exploits the SockJS library [14], while the REST interaction exploits native components of the Web browser, in particular XMLHttpRequest objects to execute Ajax requests. In both cases, messages are based on the JSON syntax.

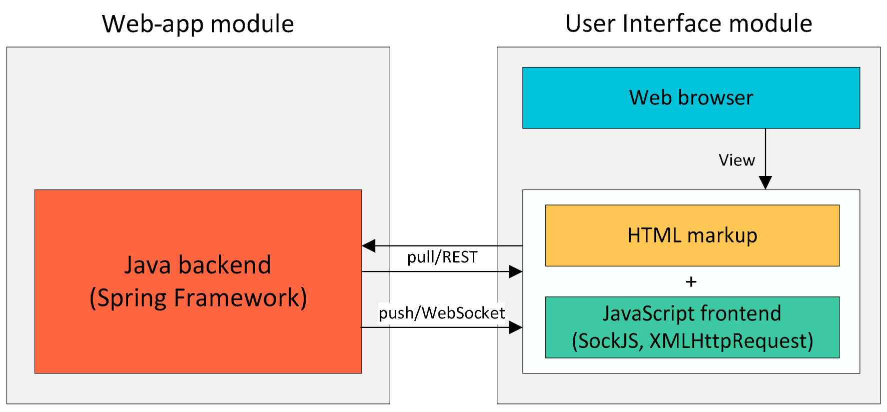

Figure 5. The PRESS User Interface module and its interaction with the PRESS Web-app module.

As briefly anticipated in the previous section, the User Interface module can receive information via either push or pull interaction models; in the former case, the Web-app module is in charge of sending messages to the User Interface module based on the WebSocket protocol whenever and as soon as new messages are received from the Gateway module. In the latter case, the User Interface module is in charge of actively requesting new information to the Web-app module by exploiting the REST paradigm based on HTTP. The business logic provided by the Web-app module ensures that only messages that the user is interested in will be actually provided. In the case of WebSocket, the User Interface module specifies the devices it is interested in by sending specific messages through the persistent connection towards the Web-app module. In the case of REST, the User Interface module has to specify the set of devices it is interested in each time it performs a request of new data. Let us notice that PRESS supports data sharing among different users by providing clients with the capability of indicating the type and subset of data in which they are interested. For instance, if a client is interested in data about devices in the "Engineering" building in the city of Bologna, Italy, in our PRESS-supported REST case the clients can specify the HTTP query string "?state=IT&city=Bologna&building=Engineering", while in the WebSocket case they can register to the /IT/Bologna/Engineering object.

4. Congestion Detection and Recovery Mechanism

To guarantee the effectiveness of the PRESS solution, it is mandatory to ensure the successful delivery of messages, either MQTT messages between Gateway and Web-app modules or WebSocket messages between Web-app and User Interface modules. In the following section we will focus on the former since in real-world scenarios the most challenging technical aspects usually relate to the dispatching of messages sent from/to the Gateway module to/from the Web-app module. In fact, it is very likely that the number of monitored devices (and thus the amount of MQTT messages) is much greater than the number of active users that monitor devices by themselves (and thus the number of WebSocket Messages).

While the exploitation of the MQTT broker provides scalability to the message exchange system, it is also important to consider the role of the logically centralized, possibly replicated, Web-app module, which has to be able to correctly receive and promptly manage the messages sent from several gateways. In particular, it is crucial to promptly detect congestions between the Gateway module and the Web-app one and adopt proper countermeasures. A congestion may occur whenever the Gateway module generates messages at a rate greater than the Web-app module is able to manage. In such a scenario, the Web-app module may represent the bottleneck of the whole framework, with the risk of significantly lowering the overall performance and of either delaying the delivery of messages to the User Interface module or, much worse, even starting to lose messages. To minimize the negative effects of congestion, we have decided to adopt and implement two different mechanisms:

- a congestion detection mechanism, in charge of analyzing traffic to sense whether and when the message rate is greater than the processing capabilities of the Web-app module;
- a congestion recovery mechanism, in charge of lowering the message rate in case of congestion and of re-establishing the regular message rate whenever the congestion condition is terminated.

To this purpose, the Web-app and Gateway module must cooperate to exchange information about congestions and take proper countermeasures, for e.g., by lowering the amount of published MQTT messages.

The **congestion detection** mechanism is inspired by DFlow [15], i.e., an algorithm proposed to improve the TCP congestion mechanism. It is based on the amount of transmitted messages not received by the receiver. DFlow is based on the monitoring of end-to-end delay of each message, i.e., between its transmission and reception, by periodically verifying if the average delay is either constant or considerably increased (in the latter case a congestion detection is triggered). To this purpose, the sender (typically the Gateway module) adds a timestamp in each message it sends. Then, the receiver (typically the Web-app module) exploits the timestamp to evaluate if the delay between message delivery and reception is regular. It is worth noting that the congestion detection mechanism does not require that clocks in the sender and the receiver nodes are synchronized, but it only requires that the clock drift of the two nodes is limited (a reasonable hypothesis in many real-world scenarios).

The proposed congestion detection mechanism exploits the lowest delay during long periods of [1.0, 2.0] s and short periods of [0.25, 0.50] s, the latter at the end and within the former. The congestion detection mechanism is applied only at the end of long periods (see Figure 6). The mechanism takes into consideration the Lowest Delay of Long periods (LDL) and the Lowest Delay of Short periods (LDS); in addition, it also considers the Previous LDS (PLDS), the lowest delay of the previous short period.

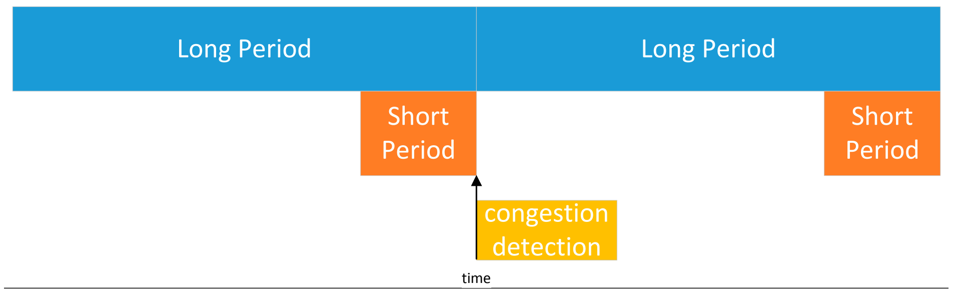

Figure 6. Phases of the PRESS congestion detection mechanism.

The congestion detection mechanism exploits two different conditions:

$$\text{LDS} - \text{LDL} > \text{Threshold} \tag{1}$$

$$\text{LDS} - \text{PLDS} > 0 \tag{2}$$

where Threshold is set in the [0.025, 0.100] s interval.

Condition (1) allows for detection if the delay among messages considerably increases, while Condition (2) ensures that the delay is actually increasing. Then, the congestion detection mechanism exploits the two equations to fire reinforcement actions:

- Negative reinforcement: if both Equations (1) and (2) apply for three consecutive long periods, the mechanism considers the system as congested and triggers the congestion recovery mechanism. Then, the negative reinforcement is inhibited for three long periods to provide enough time to correctly apply the congestion recovery mechanism by lowering the network load;
- Positive reinforcement: if both Equations (1) and (2) do not apply for 15 consecutive long periods, the mechanism assumes that there is no congestion and notifies the congestion recovery mechanism (if active) that there is no need to reduce the traffic load. Then, the positive reinforcement is inhibited for 15 consecutive long periods, thus, ensuring a gradual arise of traffic load.

The firing of negative/positive reinforcements based on multiple consecutive long periods ensures that the mechanism is able eventually to filter out brief traffic spikes due to occasional message delays not related to actual traffic congestion, thus, increasing the stability of the system and the generated network traffic. In addition, the exploitation of a greater number of long periods for negative reinforcement than for positive ones ensures prompt countermeasures in case of traffic congestion and then proper traffic reestablishment fitting actual network capabilities (similar to what happens in the slow-start TCP traffic congestion mechanism).

The **congestion recovery** solution is based on a probabilistic message filtering solution that discards regular packets with probability p_i while always ensuring the transmission of priority packets. As better detailed in the following, discarding part of the regular packets in case of congestion reduces the traffic among Gateway and Web-app modules, ensuring service continuity and the capability of dispatching priority packets and at least part of the regular packets. This solution fits many real-world IoT scenarios, producing a great amount of low priority data and some high priority messages. For instance, while monitoring a building, to overcome a possible congestion, it could be worthwhile to temporarily reduce the period of humidity sensing of each room (and thus the generated traffic) to always ensure the successful dispatching of commands, for e.g., to modify the setpoint temperature.

In particular, each in-the-field Gateway module sends the outgoing message m_i with probability

$$\begin{array}{ll} p_i = 1 & \text{if } m_i \text{ is a priority packet} \\ p_i = 0 & \text{if } N_{max} = 0 \\ p_i = 1 - N_{cur}/N_{max} & \text{otherwise} \end{array}$$

where

$$\begin{array}{l} N_{max} \geq 0 \\ 0 \leq N_{cur} \leq N_{max} \\ N_{max} \text{ and } N_{cur} \text{ integers} \end{array}$$

At system startup, the value of N_{cur} is set to 0, and then is modified by negative/positive reinforcements of the congestion detection mechanism by decrementing/incrementing whenever a reinforcement is fired. When N_{cur} = 0, the Gateway module dispatches every message since p_i = 1, while when $N_{cur} = N_{max}$ it delivers only priority messages since p_i = 0.

The value of N_{max} can be modified to tune the behavior of the traffic shaping solution between the Gateway module and the Web-app module. It is worth noting that the greater is the N_{max} the finer is the congestion recovery mechanism. As notable examples, let us consider the following:

- if N_{max} = 0, p_i is set to 0 and, thus, the Gateway module only sends priority messages;
- if N_{max} = 1, the congestion recovery mechanism has two steps, i.e., it adopts an "all-or-nothing" behavior, since N_{cur} is either 1 (only priority messages are delivered) or 0 (every message is delivered);
- if N_{max} = 2, the congestion recovery mechanism has three steps, since N_{cur} may assume the 0, 1, or 2 values; p_i may be equal to 1, 0.5, or 0;
- if N_{max} = 3, the congestion recovery mechanism has four steps, since N_{cur} may assume the 0, 1, 2, and 3 values; p_i may be equal to 1, 0.66, 0.33, or 0.

5. Performance Results and Discussion

We have thoroughly tested our Java-based PRESS framework prototype to quantitatively evaluate its scalability and to assess its efficiency along two primary guidelines:

(1) Evaluation of the overall system scalability (without congestion detection and recovery mechanisms) in an end-to-end perspective, i.e., by considering the whole chain from the PRESS Gateway module to the PRESS User Interface module, while varying:

    a. MQTT broker quality levels between Gateway and Web-app,
    b. protocols between Web-app and User Interface, either REST or WebSocket.

    In this case, performance is quantitatively evaluated in terms of lost messages, receiver throughput, and end-to-end latency;

(2) Evaluation of our congestion detection and recovery mechanisms when working for the PRESS Gateway and Web-app modules. In this case, performance is quantitatively evaluated in terms of receiver throughput and end-to-end latency.

The measured performance indicators reported below (average values over 50 runs) are based on experiments over physical nodes with Intel Core i7 3770 processor, 12 GB of RAM, Windows 7 operating system, and maximum inter-node bandwidth of 10 Mbit/s. Packets are dispatched by an executor component written in Java that exploits the functions offered by our PRESS Gateway, sending messages at variable packet frequencies and different MQTT quality levels Packets are received by a gauge component written in JavaScript and exploit features provided by our PRESS User Interface; it receives packets from the Web-app module via either REST or WebSocket. Let us notice that we have concentrated the reported results on packet frequency rather than packet size because, given the targeted scenario, we consider the case of many devices sending relatively small packets as more plausible. However, for other application domains and deployment scenarios, we are already investigating and measuring how packets with huge payloads may impact the achieved performance; this is out of the scope of this paper.

In addition, we have developed a proof-of-concept PRESS box composed of a battery-enabled UDOO Dual single-board PC, equipped with GPS, Wi-Fi, and some environmental sensors that we have selected as notable examples, such as the MQ-2 gas sensor, DS18B20 temperature sensor, and DHT11 humidity sensor. In the default configuration, one small message is sent every 15 min. It includes two floats (latitude and longitude) and three integers (temperature, humidity, and gas concentration). In the case of messages sent from fixed devices that are deployed indoors (typically without usable GPS positioning), latitude and longitude are replaced by a string describing the logical location the device, for e.g., IT/Bologna/Engineering. Note that, even if not supported in the current version of the prototype, the proposed framework is designed to allow the easy integration of additional virtual sensors, i.e., non-GPS-based indoor localization techniques, by, for example, using our previous research activities on integrated and synergic indoor positioning techniques [16].

5.1. Overall System Scalability

Figure 7 reports the percentage of lost messages while increasing the sender throughput in case of different MQTT quality levels, i.e., best effort, at-least once, and exactly once, and different protocols between the Web-app and User Interface modules, i.e., REST and WebSocket.

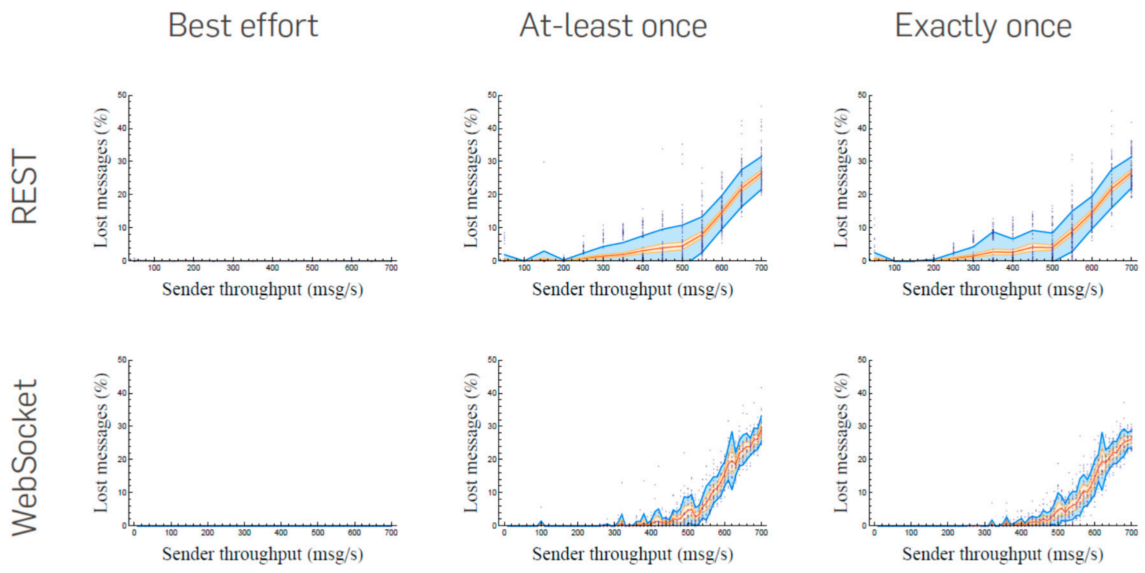

Figure 7. PRESS lost messages while increasing sender throughput for different MQTT broker quality levels (top) and for different protocols between Web-app and User Interface modules (left). Average values, 95% confidence, and standard deviation lines are reported.

The two primary lessons learnt from the collected results are that (i) WebSocket slightly reduces the percentage of lost messages and (ii) the adoption of best effort MQTT message delivery decreases the percentage of lost messages. The former observation is in line with expectations since REST generally imposes greater overhead when compared with WebSocket. The latter observation, at first sight, seems to be counterintuitive, since a higher quality level forced at the MQTT broker has the effect of decreasing the overall efficiency. However, it is worth noting that higher levels of MQTT quality impose much higher overhead, because with higher MQTT quality there is the need to acknowledge messages published to the broker; moreover, in case a message dispatch to a subscriber fails, it is mandatory to maintain messages and try to deliver them again. In other words, while the adoption of higher quality levels ensures better message delivery at a per-packet point of view, it increases non-negligibly the overall overhead at the same time, by possibly reducing overall scalability, as also identified by [17].

Figure 8 reports receiver throughput in terms of msg/s while increasing sender throughput. In each case, the maximum receiver throughput is about 500 msg/s, with slight variations in case of different MQTT quality levels (note that the relatively low percentage of lost packets does not considerably decrease the measured receiver throughput). In terms of the scalability of the proposed solution, if we consider deployment scenarios in which a 15 min period is reasonable, as in the previously presented one, our solution has demonstrated its ability to correctly manage more than 400 K devices. Of course, to further increase the overall scalability of the Web-app module, it is possible and relatively straightforward (not particularly innovative and thus out of the scope of the paper) to replicate it, for e.g., on a cluster of Web servers with a load balancer façade. Moreover, the usage of REST and WebSocket in PRESS achieves almost the same results, and, thus, the two different protocols do not considerably alter the framework scalability in the tested environment. Further tests with message persistence disabled have achieved much higher receiver throughput, up to 8700 msg/s; at the same time, this shows on the one hand the good behavior of the framework in high-traffic conditions and on the other hand the fact that the primary bottleneck of the current PRESS framework implementation relates to persistence management and database access; the last observation is directing our ongoing research work towards the integration of PRESS with non-relational NoSQL persistence storage solutions, such as MongoDB.

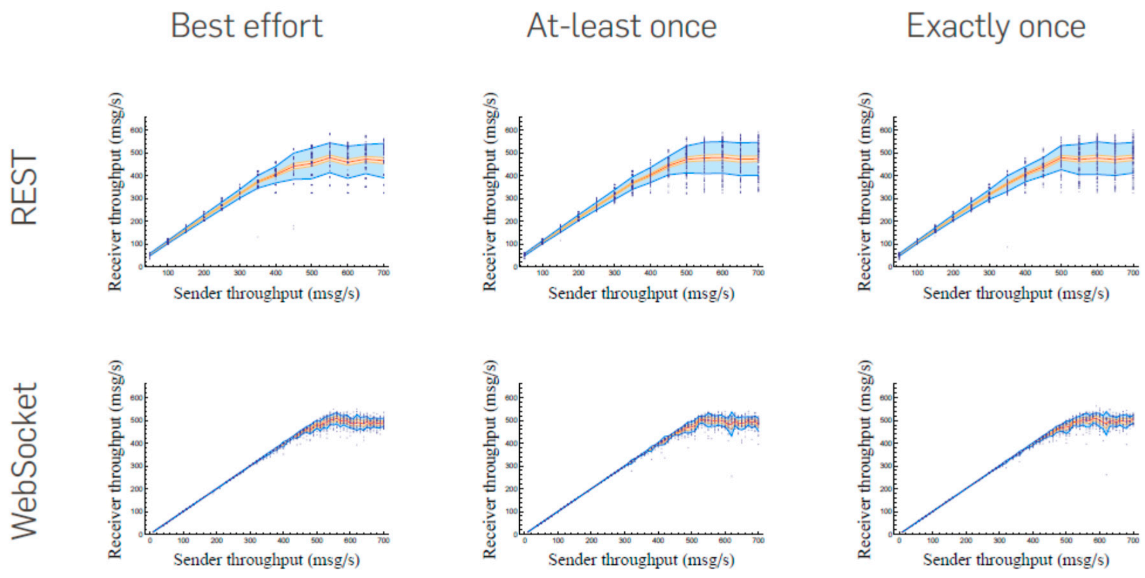

Figure 8. Receiver throughput while increasing sender throughput in case of different MQTT quality levels set at brokers (top), and different protocols between Web-app and User Interface (left). Average values, 95% confidence, and standard deviation lines are reported.

In addition, Figure 9 shows end-to-end message latency while increasing sender throughput. In the WebSocket case, latency is always very close to 0 until in the proximity of the receiver throughput limitation (pointed out in Figure 8), confirming that PRESS delivery delay increases mainly due to the limitation in receiver throughput. In case of best-effort MQTT quality, latency starts increasing at about 400/500 msg/s, since the system has to process a higher amount of packets (in fact, in case of at-least once and exactly once quality levels, beyond this threshold the framework starts to experience first sporadic packet losses). In addition, the overall latency of WebSocket is considerably lower than the REST one because the latter introduces delays mainly connected with the polling approach of the User Interface module to retrieve data; on the contrary, the former dispatches messages as soon as they are available in the Web-app module. Finally, note that the reported results message latency only considers correctly received packets. For at-least once and exactly once quality levels, packet loss starts to emerge from about 500 msg/s; above that threshold, the average latency stabilizes because the PRESS framework autonomously starts dropping packets, and this permits overall stability and respect of the preferred quality levels.

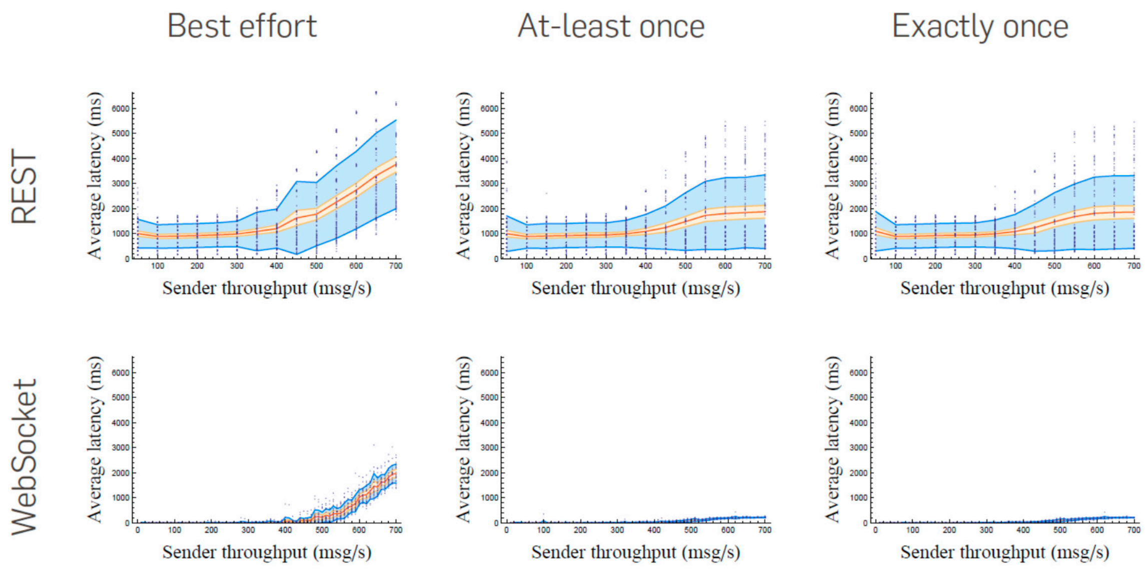

Figure 9. End-to-end message latency while increasing sender throughput in case of different MQTT broker quality levels (top) and different protocols between Web-app and User Interface modules (left). Average values, 95% confidence, and standard deviation lines are reported.

5.2. Congestion Detection and Recovery Mechanisms Performance

To evaluate the effectiveness and efficiency of the PRESS congestion detection and recovery mechanisms, we have primarily tested our framework prototype while varying sender throughput. The initial throughput is set to 100 msg/s and then it increases by an additional 100 msg/s every 120 s until 600 msg/s, then decreasing it by 100 msg/s every 120 s until 100 msg/s, for a total time of 22 min. The enforced MQTT quality level is set to 0, thus, avoiding message losses due to reduced system scalability; moreover, no priority messages are included, to better assess the effectiveness of congestion detection and recovery mechanisms in the most challenging technical conditions. Long/short periods are set to 1 s/0.25 s (so that our congestion detection mechanism takes 3 s/15 s for each negative/positive reinforcement). Threshold is set to 0.25 s, and N_{max} to 10 (with 11 possible steps for N_{cur}, with each associated transition incrementing/decrementing packet dropping 10%). The Web-app module is in charge of applying the congestion detection mechanism and sending proper messages to the Gateway module to properly trigger negative/positive reinforcements.

Figure 10 presents receiver throughput (top) and end-to-end latency (bottom) when disabling (left) and enabling (right) congestion detection and recovery mechanisms. Without the congestion detection and recovery mechanisms, target (ideal) and actual (shaped) sender throughputs are always the same, since the Gateway module sends every packet. When the sender throughput reaches the maximum supported by the framework in the used deployment environment (about 500 msg/s, as detailed in Section 5.1), messages start to be delayed and require more than 10 s to be correctly delivered. Then, when the sender throughput lowers again, end-to-end delays stay high for a non-negligible time window, since the system still has to dispatch a relevant quota of queued messages. Only at about time 1080 s do delays decrease to regular values. On the contrary, when congestion detection and recovery mechanisms are enabled, the actual sender throughput is slightly lower than the target one, by ensuring that the end-to end delay remains lower than 0.1 s most of the time (with few spikes slightly greater than 1 s). Then, when the sender throughput goes below the 500 msg/s threshold (at about 940 s), congestion detection and recovery mechanisms restore the target actual sender throughput equal to the ideal (target) one, at the same time without increasing the end-to-end delay.

In short, the proposed congestion detection and recovery mechanisms are able to actually avoid saturating the message delivery capabilities of the PRESS framework. On the one hand, by avoiding message congestion it is possible to always promptly dispatch priority messages, for e.g., commands sent by the User Interface to the Gateway or important alerts sent in the opposite direction. On the other hand, lowering the amount of messages sent by the Gateway module in case of congestion simplifies the provisioning of service continuity, for e.g., by reducing the frequency of sensed temperature and air quality values only when needed but ensuring their actual and prompt delivery.

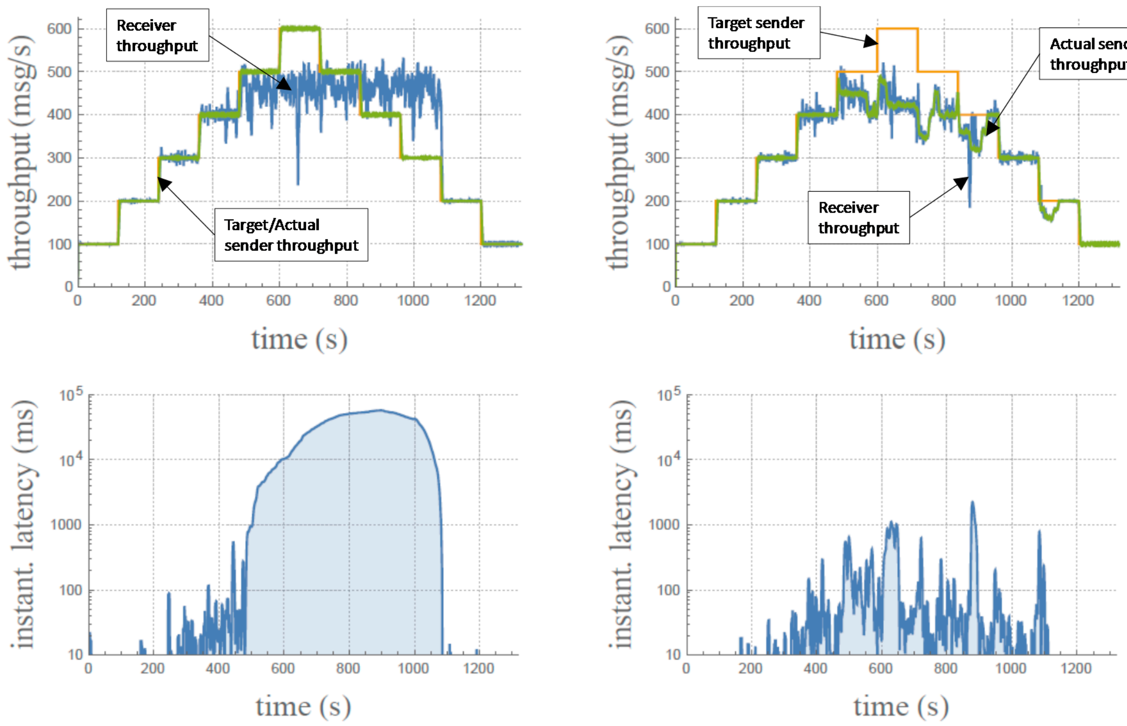

Figure 10. Receiver throughput (top) and end-to-end latency (bottom) without (left) and with (right) our PRESS congestion detection and recovery mechanisms.

6. Conclusions

This paper originally presents the design, implementation, and evaluation of our PRESS framework to solicit users' behaviors with better economic and environmental sustainability via improved ambient/situation awareness. Key original technical aspects of our proposed framework are (i) the exploitation of a flexible and dynamically extensible architecture, (ii) the ability to manage high and irregular volumes of heterogeneous data with good scalability at limited costs, (iii) the efficient dispatching of packets from/to in-the-field devices/actuators to/from the final users thorugh "smart" integration with different widespread communication protocols, i.e., MQTT, REST, and WebSocket, and (iv) the efficient integration with basic mechanisms and tools belonging to widespread and industry-mature open-source projects such as Kura.

The reported performance results demonstrate the feasibility of our approach in IoT scenarios, by showing that the synergic exploitation of MQTT and WebSocket can improve the scalability of the overall system. In particular, in-the-field deployment and evaluation activities have shown the relevance of differentiated exploitation of MQTT quality levels (for instance, high MQTT quality is adequate only for maximum priority and infrequent messages, while regular messages can use best-effort MQTT quality, with significant overhead reduction). Moreover, the WebSocket protocol has demonstrated to be preferable to the more commonly adopted REST. Finally, the reported results also clearly show the efficacy of our congestion management solution, with significant advantages in terms of service continuity via prompt reaction over correctly detected congestions.

Author Contributions: All the authors have equivalently contributed to the manuscript, with a strong focus of Paolo Bellavista and Carlo Giannelli on architectural and design choices, and a prevalent contribution of Riccardo Zamagna in terms of experimental evaluation.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lin, J.; Sedigh, S.; Hurson, A.R. An Agent-Based Approach to Reconciling Data Heterogeneity in Cyber-Physical Systems. In Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), Anchorage, AK, USA, 16–20 May 2011.
2. Jirkovsky, V.; Obitko, M.; Marik, V. Understanding Data Heterogeneity in the Context of Cyber-Physical Systems Integration. *IEEE Trans. Ind. Inf.* **2016**. [CrossRef]
3. Ngu, A.H.H.; Gutierrez, M.; Metsis, V.; Nepal, S.; Sheng, M.Z. IoT Middleware: A Survey on Issues and Enabling Technologies. *IEEE IOT J.* **2016**, *4*, 1–20. [CrossRef]
4. MQTT. Available online: http://mqtt.org/ (accessed on 9 Febrary 2017).
5. Fette, I.; Melnikov, A. The WebSocket Protocol. Available online: https://tools.ietf.org/html/rfc6455 (accessed on 9 Febrary 2017).
6. Babovic, Z.B.; Protic, J.; Milutinovic, V. Web Performance Evaluation for Internet of Things Applications. *IEEE Access.* **2016**, *4*, 6974–6992. [CrossRef]
7. Mun, D.-H.; Le Dinh, M.; Kwon, Y.-W. An Assessment of Internet of Things Protocols for Resource-Constrained Applications. In Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, 10–14 June 2016.
8. Baldoni, R.; Querzoni, L.; Virgillito, A. Distributed Event Routing in Publish/Subscribe Communication Systems: A Survey. Technical Report TR-1/06. Available online: http://midlab.diag.uniroma1.it/articoli/BV.pdf (accessed on 9 Febrary 2017).
9. Al-Fuqaha, A.; Khreishah, A.; Guizani, M.; Rayes, A.; Mohammadi, M. Toward better horizontal integration among IoT services. *IEEE Commun. Mag.* **2015**, *53*, 72–79. [CrossRef]
10. Malekpour, A.; Carzaniga, A.; Pedone, F. End-to-End Congestion Control for Content-Based Networks. In Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), Nara, Japan, 6–9 October 2014.
11. Kura. Available online: http://www.eclipse.org/kura/ (accessed on 9 Febrary 2017).
12. Mosquitto MQTT broker. Available online: https://mosquitto.org/ (accessed on 9 Febrary 2017).
13. Spring Integration. Available online: https://projects.spring.io/spring-integration/ (accessed on 9 Febrary 2017).
14. SockJS. Available online: http://sockjs.org (accessed on 9 Febrary 2017).
15. O'Hanlon, P.; Carlberg, K. DFLOW: Low latency congestion control. In Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP), Goettingen, Germany, 7–10 October 2013.

16. Bellavista, P.; Corradi, A.; Giannelli, C. Efficiently Managing Location Information with Privacy Requirements in Wi-Fi Networks: A Middleware Approach. In Proceedings of the 2nd International Symposium on Wireless Communication Systems (ISWCS), Siena, Italy, 5–7 September 2005.
17. Mijovic, S.; Shehu, E.; Buratti, C. Comparing Application Layer Protocols for the Internet of Things via Experimentation. In Proceedings of the 2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), Bologny, Italy, 7–9 September 2016.