Journal of Grid Computing (2021) 19: 32
https://doi.org/10.1007/s10723-021-09577-9

The Impact of Encoding and Transport for Massive Real-time IoT Data on Edge Resource Consumption

Francesco Tusa · Stuart Clayman

Received: 13 November 2020 / Accepted: 1 July 2021 / Published online: 13 July 2021

Abstract Edge microservice applications are becoming a viable solution for the execution of real-time IoT analytics, due to their rapid response and reduced latency. With Edge Computing, unlike the central Cloud, the amount of available resource is constrained and the computation that can be undertaken is also limited. Microservices are not standalone, they are devised as a set of cooperating tasks that are fed data over the network through specific APIs. The cost of processing these feeds of data in real-time, especially for massive IoT configurations, is however generally overlooked. In this work we evaluate the cost of dealing with thousands of sensors sending data to the edge with the commonly used encoding of JSON over REST interfaces, and compare this to other mechanisms that use binary encodings as well as streaming interfaces. The choice has a big impact on the microservice implementation, as a wrong selection can lead to excessive resource consumption, because using a less efficient encoding and transport mechanism results in much higher resource requirements, even to do an identical job.

Keywords Real-time · Iot · Edge analytics · Data encoding · Network transport

F. Tusa (✉) · S. Clayman
Department of Electronic and Electrical Engineering, University College London, London, England
e-mail: francesco.tusa@ucl.ac.uk

S. Clayman
e-mail: s.clayman@ucl.ac.uk

1 Introduction

The Internet of Things (IoT) is a well-known paradigm that envisions the interconnection and the exchange of data between many of the physical objects that surround us [13]. Billions of machines, devices, and "things" are responsible for the generation of massive amounts of data that need to be stored, processed and presented in an efficient, homogeneous and easily understandable form. If this data can be gathered and processed in real-time, for instance, a factory production line might be controlled while it operates, to identify potential anomalies and products' defects. Similarly, real-time data information on a person's heartbeat might be used to save lives and even predict ailments in advance [41].

Real-time IoT analytics aims to provide optimised IoT services by analysing the collected IoT data – using distributed network, computation, and storage resources – within a fixed time. Effectively leveraging the right IoT data at the right moment, to create proactive and predictive business models, is not straightforward. Moreover, due to the massive volume of generated data and the stringent time constraints, traditional cloud computing is not effective in this type of scenario. As such, *real-time IoT analytics* is receiving a great deal of attention, e.g., from research initiatives on In-Network Control [18] and Edge Computing [28].

The edge has been gaining a consensus as one of the most viable approaches for the execution of *real-time IoT analytics* tasks [38]. It can help reduce the

processing latency as data can be processed closer to the source instead of being sent to the central cloud. However, the edge also introduces limitations in the amount of available resources and, as consequence, the way the "things" located in the IoT Domain interact with the edge *processing applications* becomes a very important research aspect that should not be overlooked [2].

Emerging application design patterns use fine-grained and independent blocks in order to build software systems. Rather than developing an application as a single monolithic piece, *microservices* and *Function as a Service (FaaS)* architectures rely on autonomously-running components, each dealing with a smaller specific task, usually deployed in a lightweight containerised environment [21]. These models certainly offer intrinsic advantages, such as more agile software development and maintenance workflows, continuous integration / deployment, and even transparent support for dynamic scaling. The individual components of microservices applications, built with most of the currently available frameworks, either expose APIs based on REST endpoints [43], or support data streaming via lightweight messaging protocols like MQTT [26]. HTTP-based REST with JSON has become the de-facto standard interface for synchronous interaction with those applications [14]. Although it provides loose coupling with a high degree of interoperability, allowing it to be used by applications or browsers distributed over the Internet, we ask the question: *can it also be considered as the right option for edge applications that perform real-time IoT analytics tasks?*

Through an extensive evaluation of a number of alternative technologies used for the design of the interfaces exposed by IoT edge analytics applications, this paper answers the above question. Its main contribution is to provide an in-depth analysis and comparison of various data encoding, messaging protocols, and transmission mechanisms that can be utilised specifically for the communication between the IoT domain and the Edge. The evaluation of the costs of processing and transmitting data has been considered by different research communities [8, 10, 14, 19, 25, 36]. However, we argue that a comprehensive study that encompasses performance evaluation of IoT data encoding and transmission protocols, as well as edge computing and (interface) applications design is missing for large-scale scenarios; moreover, some of the available results are currently being overlooked or ignored due to silo effect.

The evaluation of these mechanisms is supported by an actual distributed IoT system that was built, and deployed on a real testbed, in order to *orchestrate* and *monitor* the *generation*, *collection* and *processing* of IoT data in the *Cloud-to-Edge continuum*. Tens of thousands of *IoT devices* were utilised to transmit data to IoT edge analytics applications via different types of communication mechanisms: the most commonly used synchronous REST/JSON interface was compared with alternative approaches based on binary encodings as well as streaming mechanisms.

The development of a *Programmable Massive IoT Platform*, which was utilised in order to build the large-scale IoT testbed and generate the huge amount of data required for our evaluation, is the second main contribution of this work. Even though physical IoT devices could have been used for building this system, we chose *software sensors* in order to minimise the cost of the hardware required by our large-scale IoT deployments while maximising the flexibility of the experimental environment. This fully distributed platform allowed us to manage and orchestrate the system entities required for the various experiments (such as the IoT elements, the processing functions, etc.) in a programmable way and under fully automated software control.

Nonetheless, the validity of our experimental results was not impacted by the usage of the above *software sensors* as the IoT data generation and collection process is completely transparent to the edge nodes. The details of how the data required for the execution of large-scale IoT experiments is generated and gathered is not relevant for the edge, as long as actual network packets are delivered to it via different types of data encoding, transport mechanisms, and interfaces. Finally, our experimental analysis did not require any simplifications to the software stack and allowed to uncover any problems that might be induced by the timing of messages, as opposed to the behaviour of discrete event generators utilised in simulations [22].

Our results show that, when doing real-time edge processing, all of the above aspects related to data encoding and transmission have a big impact on the implementation of microservices, and that a wrong choice can lead to excessive resource utilisation. The higher the processing cost and resource usage when dealing with the incoming data, the less resource is

available for the analytics tasks. As a consequence, more edge node resources are consumed and higher bills are generated. We demonstrate in this paper that these mechanisms greatly affect both the number of bytes transmitted over the network, and the overall consumption of CPU and memory resources required to perform real-time analytics at the edge. This is particularly important when the relatively small amount of edge resources is considered together with the potential massive scale of the IoT data to be processed.

2 Background and Related Works

Real-time IoT analytics targets the enhancement of IoT business applications and services by performing the analysis of huge sets of IoT data within a strict *fixed time*. The concepts of edge and fog can dramatically reduce the latency by bringing the computing nodes closer to the data sources [41]. Whilst this approach perfectly matches the demands for rapid processing of large amounts of data of some emerging use cases, such as smart health, co-operative intelligent transport systems and Industry 4.0 [38], it also introduces a new set of challenges related to the end-to-end orchestration of resources, i.e., from the "things" to the cloud.

Existing works in this area include the automated distribution of data, between the edge and the cloud, according to both the dynamic conditions of the IoT infrastructure and the applications requirements [28]; the evaluation of networking and computing capabilities of edge nodes, along with their reliability, as part of a score-based edge service scheduling algorithm [3]; the combined usage of discrete optimisation algorithms and task prioritization methods to find the best task execution order for scientific workflows [15]; algorithms for workload allocation that trade-off energy consumption and delay [1]; unified management of edge and cloud resources for IoT with fault-tolerance capabilities [17]. We noticed that the analysis of the impact of the transmission mechanisms between the IoT and the edge domains is not taken into account in the design of the above solutions.

Edge computing is supported by the growth of emerging technologies, such as container-based virtualisation and microservices [2]. When microservice architectures are considered, it is fundamental ensuring adequate quality aspects for their APIs in terms of functionality, as well as reliability, performance (e.g., network latency), security, and scalability [43, 44]. Existing works in this context provide a performance evaluation of some API implementations. The usage of RabbitMQ and REST APIs has been analysed for microservice web applications [14], and experiments show that, when a large number of users send parallel requests to the same web application, the RabbitMQ approach offers better performance compared to REST.

Additional studies evaluate the usage of various messaging protocols in the context of Vehicular Cloud Computing [19] and IoT [8, 25, 36]. Established technologies include MQTT and CoAP [27]. MQTT [26] supports the publish/subscribe model, and is designed for lightweight machine-to-machine communications in constrained networks via an intermediate broker. It uses a binary data encoding and either TCP or WebSocket as the transport protocol. MQTT is the most widely adopted protocol in IoT [36]; it is more suitable for usage on devices with no power constraints and for multi-cast applications. However, the presence of a single intermediate broker poses both fault-tolerance and scalability issues [25]. CoAP [32] is another lightweight protocol that supports both request/response and a variant of the publish/subscribe architecture. It was designed to inter-operate with HTTP and RESTful Web Applications. Like MQTT, CoAP is based on a binary data encoding but uses UDP as the transport protocol. As such, CoAP offers reliable data transfer with higher delays [25]; it also shows higher throughput compared to both MQTT and HTTP in the context of Vehicular Cloud Computing [19].

The usage of platform-neutral IoT data interchange formats is a key aspect to enable portability, interoperability, and efficient transportation [9]. Self-describing formats, such as XML and JSON, and binary (schema-based) formats, like XDR [34] and Protocol Buffers [12], have their advantages and drawbacks: the self-describing ones are human readable and easy to understand, but, from the transmission perspective, they contain redundant components that affects the size of data transfers; the binary formats are not human readable but are more efficient for transmission [10].

The performance evaluation of data encoding and transmission technologies should ideally consider the scalability aspects [43]. However, this analysis is not always straightforward, especially for large-scale distributed systems, such as massive IoT scenarios,

where thousand of physical objects may be involved. A cost-efficient approach, which allows testing beforehand the suitability of those systems for the execution of specific applications, is based on the usage of either simulators or emulators. MAMMotH is a large-scale IoT emulator that can be used for the definition of experiment scenarios, their deployment on a testbed, and the collection of the associated results [22]. An alternative solution utilises container-based virtualisation in order to build a large-scale testbed for wireless / IoT networks [24]. A simulation methodology can also be suitable for testing IoT systems with a large number of interconnected devices from an application-layer perspective [4].

In this section, we see how recent research works target different aspects of the end-to-end orchestration of edge and cloud resources, however, the details of the impact of both data encoding and transmission in those systems is usually not discussed. Also, microservices represent a viable design pattern for edge applications, provided that the API whereby they can be accessed is carefully designed to support the expected functional and non-functional requirements [43, 44]. Existing performance analysis has shown how the API design is impacted by the choice of the network transport protocol [14]. Similar performance analyses consider the communication of entities in the IoT context and show the usage of different data formats [9, 10], messaging protocols and network transport mechanisms [19, 25, 27, 36].

Although different studies have been conducted in different areas, we argue that a systematic unified evaluation that considers the impact of data encodings and transport mechanisms end-to-end, from the IoT devices to the edge processing functions is missing. Moreover, existing studies do not consider large-scale scenarios with tens of thousand of communicating entities. Evaluating the scalability of those technologies is indeed very important, but not always straightforward. Either emulated [22, 24] or simulated [4] approaches may support the design of large-scale systems before their actual implementation. Unfortunately, some of the existing frameworks either consider only the wireless communication layer [22] or support small IoT topologies with limited programmability features [24]; others, could provide results that lack of completeness due to the usage of simulations [4]. This paper overcomes those limitations with the introduction of a distributed and fully Programmable Massive IoT Platform that deals with the automated management of thousands of sensors sending data to the edge, allowing the evaluation of the cost of various encoding and transport mechanisms.

3 IoT Analytics Scenario

This section provides an overview of a distributed IoT computing scenario, namely a system able to perform various functions on a very large set of data generated by a massive number of IoT devices. It encompasses ten to hundreds of thousands entities generating streams of data that need to be analysed in order to extract valuable information. Common IoT analytics scenarios include *Smart Home / City*, *Disaster Management*, *Healthcare*, *Transportation* and *Industrial IoT / Smart Factory* use cases [28, 41].

In a *Smart Factory*, for instance, the system would collect and process data in order to control a production line, identifying potential anomalies and product's defect in real-time, or even supporting "smarter" production processes via the execution of particular predictive data analytics. Similarly, in a *Smart City* scenario, CCTV cameras and analytics functions would be utilised to process video frames in order to detect and prevent specific situations in real-time, e.g. accidents, crimes, potential threats, or recognise specific features (face recognition, demographics, etc.).

Figure 1 shows a design of such a *Cloud-to-Thing* system: in the IoT Domain, the data generated by the IoT devices are received by the IoT Gateways and are gathered, aggregated and adapted by the *IoT Adaptation Layer*, which acts as a local data pre-processing (preparation) function. We assume that further data processing will not happen locally (e.g., in the Smart Factory) and will be delegated to a remote processing site located at the edge. Although some IoT may offer embedded computation capabilities, in this work we mainly consider lower power consumption devices, that do not provide such a feature. As such, the *IoT Adaptation Layer* is responsible for sending those pre-processed data to the Edge Domain, where they will be received by the *Real-time Edge Processing* function.

In this scenario, this function performs real-time analytics on the received data, in order to extract knowledge that will be utilised to provide feedback to the IoT Domain through the highlighted red (inner) control loop shown in Fig. 1. Additional long-term

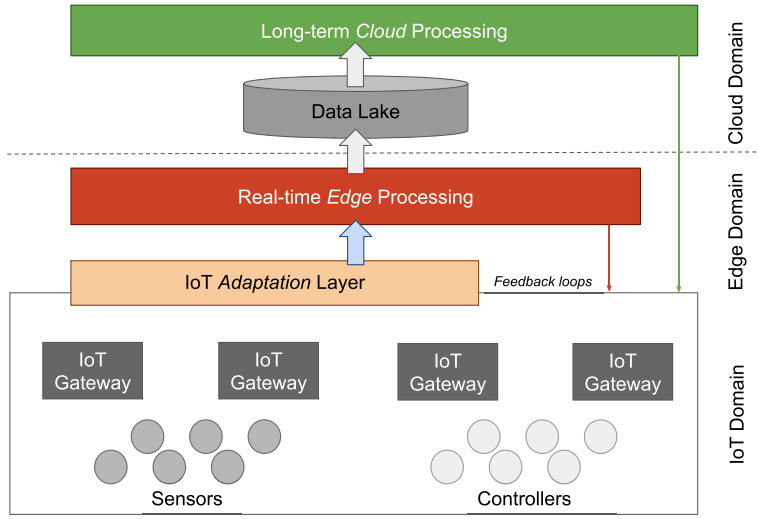

Fig. 1 IoT analytics scenario

processing operations, which do not need to be performed with strict low-latency requirements, will instead be performed in the central cloud infrastructure. In this case, data might still be first processed at the edge and the results stored within a data lake, which will decouple edge processing from long-term processing operations. The result of the latter operation will again be provided as feedback to the IoT Domain through the green (outer) control loop represented in the figure.

This paper is specifically focussed on the IoT data collection / adaptation, and on the real-time edge analytics. It investigates how the components involved in both these tasks have to be designed in order to guarantee efficient real-time processing functionality on large-scale streams of data.

The particular design chosen for the interfaces (e.g., APIs) of the *Real-time Edge Processing* has the biggest impact on the edge resource utilisation. As such, the **data encoding** and the **network transport** technologies utilised when data are sent from the sensors to the edge should both carefully be evaluated in order to avoid excessive processing cost and resource utilisation.

We now describe what are the features that the functional blocks should include, across the various *Domains* of Fig. 1.

3.1 IoT Adaptation Layer

The IoT Adaptation Layer is the system functional block that takes care of gathering data from possibly different types of IoT devices, regardless of their specific implementation. It works as an adaptation layer by providing the mechanisms to pre-process (e.g., re-encode, re-aggregate) the collected data before they are sent to the Edge Domain for the first stage of (real-time) processing. Considering the heterogeneity of different IoT devices, several types of data encoding might be utilised within an IoT Domain. The IoT Adaptation Layer is responsible for hiding these different technologies and providing a homogeneous set of data to the edge, based on a specific encoding / serialisation approach. Such adaptation layers are often used in orchestrated distributed cloud environments for normalization of data [40].

This part of the system also takes care of transmitting the set of measurements towards the Edge Domain, where the first stage of processing will take place. The data is transmitted over the network to the *Real-time Edge Processing* using a specific network transport technology. The type of data encoding and the mechanism utilised to transmit the pre-processed data over the edge network depend on the particular design and implementation of the *Real-time Edge Processing* and of the *IoT Adaptation Layer* functions.

3.2 Real-time Edge Processing

The Real-time Edge Processing is the function of the IoT analytics system responsible to perform real-time processing on the set of data coming from the *IoT Adaptation Layer*. We assume this function to be designed to include a dynamic number of processing

elements that can be created and destroyed on-demand, under orchestrator control, according to the characteristics of the received data. This processing is expected to happen without the use of intermediate long-term storage in order to minimise the end-to-end processing latency [6].

Choosing the proper design and implementation for the *Real-time Edge Processing* function is therefore of utmost importance. A wrongly made design decision for the type of data encoding and / or the type of network transport mechanism of the associated API, may overload the constrained resources available at edge, and potentially jeopardise the main point of this functional block, i.e., to provide support for real-time analytics.

As an example, if the Edge Domain hosted a FaaS framework, such as OpenFaas [21], the running applications would be based on loosely coupled software containers accessible via a REST API. Data could be sent to the ingress point of that application via POSTing (JSON) encoded content to a specific REST endpoint, and a reply would be sent back for each invocation. Later on, we will see how this approach has a different impact on the usage of the edge network and computational resources when compared, for instance, to another application that receives streams of XDR encoded data via WebSocket.

3.3 Long-term Cloud Processing

The Long-term Cloud Processing Layer is the system component that deals with processing tasks that are not expected to be performed with strict real-time guarantees and usually require additional resource capabilities compared to the ones available at the edge. It is considered as out of the scope of this paper.

4 IoT Analytics System Implementation

Here we present our implementation of an IoT analytics scenario. It is based on a software system that supports the generation and transmission of data streams – from the IoT Domain to the Edge Domain – through various data encodings and network transport mechanisms.

It has to reproduce the huge scale of the IoT Domain, which in real-life deployments is expected to include thousands of physical IoT devices. As highlighted, a simulation approach would have not supported the in-depth type of evaluation we needed, and these problems were solved with the introduction of the *Programmable Massive IoT Platform*, shown in grey, in Fig. 2.

The *Distributed Software IoT Layer* is the platform's component that implements a fully programmable IoT Domain, and supports the software generation of data without relying on actual IoT physical devices. The *Pluggable Networking / Adaptation Layer* provides the blocks that enable the actual collection / adaptation of the above data, and their transmission using various types of encoding and network transport mechanisms to the *Real-time Edge Processing* subsystem, which provides the important *Edge Data Collection* and *Edge Data Processing* functionalities. The interaction between these software subsystems is based on the *Network Transport* facet shown in the figure. These subsystems implement the bottom part of the Cloud-to-Thing scenario shown in Fig. 1 – which is highlighted in Fig. 2 by the dashed rectangle that surrounds them. The *Distributed Monitoring* and the *Orchestrator* subsystems provide the functionalities to support both the end-to-end automated deployment and the life-cycle management of the software components belonging to the IoT analytics system. They specifically take care of the dynamic provisioning of computing resources along the Cloud-to-Things path by implementing a closed-loop system capable of orchestrating (activating, deactivating, integrating, scaling, etc.) resources provided by heterogeneous computing infrastructures.

In our work, the above subsystems perform those functionalities by automating the activation (on our IoT testbed) of the resources and of the software components utilised for the execution of various IoT experiments. In particular, the *Distributed Monitoring* collects metrics (KPIs) via the dynamic deployment of both *Data Collection* and *Data Reporting* elements. The *Orchestrator* deals with the configuration and deployment (on the testbed's hardware resources) of all the IoT software components: it is able to create – on-the-fly – tailor-made instances of the other three subsystems, purposely customised to inter-work and support the setup of a particular IoT scenario to be evaluated.

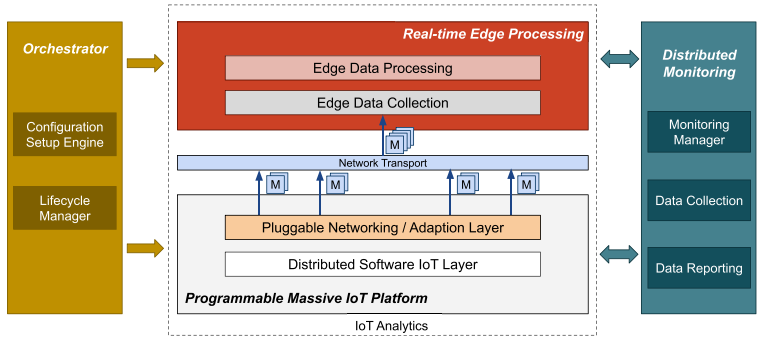

Fig. 2 IoT analytics system implementation overview

4.1 Programmable Massive IoT Platform

Our implementation of the *Programmable Massive IoT platform* utilises the dynamic features and composable software elements of Lattice monitoring framework [7, 39].

The software elements of the IoT Platform use Lattice as a foundation, and are used to implement the system elements represented in Fig. 3. It allows the platform to perform: i) the generation of data in the *Distributed Software IoT Layer*; ii) the dynamic data collection and pre-processing in the *Adaptation Layer*.

Distributed software IoT layer This supports the programmable definition of topologies of interconnected software IoT entities characterised by a set size (i.e., the number of IoT entities) and by the rate at which data is generated. A sensor-like IoT device can be programmed to produce custom defined measurements based on a particular model and encoding. A sensor's measurement consists of a set of N attributes, where each *Attribute* is defined as the combination of the following fields: *ID*, *Name*, *Type*, *Unit*, and *Value*.

Those programmable IoT software sensors are then attached to a *Software Gateway* that acts as a data multiplexer. Internally it queues the measurements generated by the sensors before they are distributed over a Data Plane, and then collected by the *Adaptation Layer*. A full IoT topology can be configured in such a way, as to distribute the elements across a cluster of nodes, in order to scale up the number of sensors.

Pluggable networking / adaptation layer This is implemented via a set of *Adaptor*s that receive measurements from one or more *Software Gateways* from a Data Plane. The *Adaptor*s are programmed to perform actions on specific types of measurements, and they support the on-the-fly adaptation of the format (e.g., the type of encoding) of a received measurement. Each *Adaptor* sends a measurement to the next stage of processing via a particular implementation of the *Network*

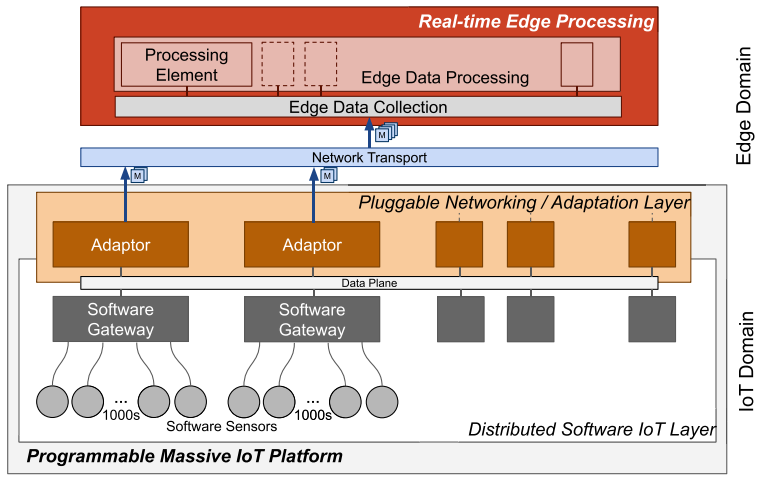

Fig. 3 Platform and edge subsystems implementation

Transport. The *Adaptation Layer* includes different types of *Adaptor*s supporting various types of data encoding and a number of *Network Transport*s. Each can be selected at run-time, according to the particular interface exposed by the *Edge Data Collection* in the *Real-time Edge Processing*.

4.2 Real-time Edge Processing

This subsystem is responsible for two main functionalities: *Edge Data Collection* and *Edge Data Processing*, and is shown in the top part of Fig. 3.

It is expected to be implemented via a microservice application where, for instance, one microservice could act as the *Edge Data Collection*, and one or more microservices could implement the *Edge Data Processing* backend. For this work, we followed this design pattern, but only developed the Edge Data Collection microservice, leaving the Real-time Edge Processing empty as there is no processing to do. The Edge Data Collection is accessible from the outside world via a particular interface, and is responsible for decoding the data received through the network.

Edge interfaces To support the evaluation of the various encodings and network transports, different versions of the microservice were developed, each with a different type of interface. There are 2 encodings used: ***JSON*** which is a text based representation, and ***XDR*** which is a binary encoding. JSON has become popular in recent times, as JavaScript's influence of software trends in the last decade caused JSON to receive more attention than any other data interchange format [35]. We chose XDR for the binary format, as it has been a standard for a long time [34] (since 1995), and is known to be performant in many network technologies (such as NFS). For the transports, we used: i) ***request/response*** using REST, ii) ***data streaming over UDP***, which gives a connectionless model, and iii) ***data streaming using Websocket***, which gives a UDP like socket interface over a TCP stream. The HTML5 specification includes Websockets, which represents a full-duplex, bidirectional communications channel that operates through a single socket over the Web [23], and provide a mechanism to build scalable, real-time web applications.

The first interface selected is the commonly used encoding of JSON over REST. This has become the preferred method for synchronous interaction for many services, as REST with JSON provides loose coupling with a high degree of interoperability. The initial implementation that was chosen was written in Python, as many edge processing frameworks are written in Python. A second implementation was written in Java, as this is both common and has many libraries.

We then chose to implement Websocket, carrying a JSON encoded payload, as the strong point about Websocket is that it is more interoperable and reliably works like a "socket" through firewalls. We next changed the payload format to XDR, for binary encoded data over Websocket. Finally we added a UDP transport for streaming both JSON and XDR.

Table 1 shows how each microservice interface implementation determines the choice of a particular data *encoding* type (second column) and of a network *transport* mechanism (third column). Furthermore, in order to have a more complete evaluation that considered several frameworks and libraries, the usage of different programming *language*s was also considered (i.e., Python and Java – fourth column). Note that a *Type* label (first column), has also been assigned to each microservice implementation in order to easily reference them in the experiments description. For each of these encodings, the transport, and associated implementation language, a performance evaluation is presented in the experiments described in Section 6.

5 Testbed Setup

This section highlights the testbed setup and describes the implementation of both the *Orchestrator* and *Distributed Monitoring* subsystems. It also presents the testbed from the hardware perspective, presenting how

Table 1 Variants of the edge data collection microservice

Type	Encoding	Transport	Language
json_rest_python	JSON	REST	Python
json_rest	JSON	REST	Java
json_ws_python	JSON	WebSocket	Python
json_ws	JSON	WebSocket	Java
xdr_ws_python	XDR	WebSocket	Python
xdr_ws	XDR	WebSocket	Java
json_udp	JSON	UDP	Java
xdr_udp	XDR	UDP	Java

the available resources were organised to support the execution of various IoT experiments.

The *Orchestrator* plays a main role in the automated setup of the testbed's components. This is represented in Fig. 4 via the brown-dashed arrows that interconnect the *Orchestrator* to the *Real-time Edge Processing*, the *Programmable Massive IoT Platform* and the *Distributed Monitoring* subsystems. The latter deals with the gathering and storage of relevant metrics associated with the execution of those components.

The same figure also shows an example of on-the-fly deployment of those software subsystems on the available testbed's hardware resources. These, consist of three logically separate clusters of servers hosted in a Data Center at University College London (UCL), and each offering a different type of resource capabilities. The three logical clusters (see Table 2) are inter-connected via a 1 Gbps Ethernet. The host operating system is Linux CentOS 7 with kernel version 3.10.

The instantiation of the above software subsystems is dynamic and customised for each experiment, but for simplicity their deployment is pre-mapped on a specific logical partition of the testbed, based on the type of available resources. Figure 4 shows that due to the featured type of resources, the *Clay Cluster* was a good candidate to host the Edge Domain's components, the *Edu Cluster* had enough resource availability to act as a large-scale IoT Domain, and the *Gas Cluster* was selected to host the components of the *Orchestrator* and (some of) the *Distributed Monitoring* subsystems. The implementation of these subsystems are now described, highlighting how they are involved in the automated setup and execution of different IoT experiments.

Table 2 Testbed's Hardware features

Cluster Name	CPU	RAM
Clay	2x *AMD Quad-Core Opteron 2347HE* @1.9GHz	32GB
Gas	4x *Intel Quad-Core Xeon E5520* @2.27GHz	32GB
Edu	4x *Intel 12-Core Xeon E5-2650 v4* @2.20GHz	192GB

5.1 Orchestrator

The Orchestrator subsystem here translates the high-level definition of an IoT scenario into the set of software components that need to be activated (on the other subsystems). These components are configured and deployed on-the-fly on the available testbed resources, as it is represented in Fig. 4 by the brown-dashed arrows. A prototype version of this Orchestrator, developed in Python, is available at [16].

Configuration setup engine This receives a high-level description of an IoT scenario and generates an *Experiment Deployment Template* that specifies what entities, of the other testbed software subsystems, are required for the setup. It then forwards the *Experiment Deployment Template* to the *Lifecycle Manager*,

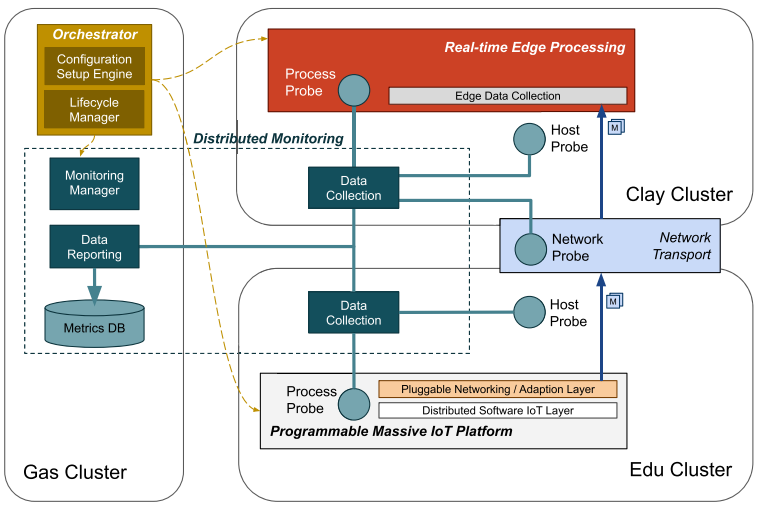

Fig. 4 Testbed deployment

which will initiate the activation of the requested components.

For instance, if the IoT scenario to be setup is data encoded as XDR and streaming over UDP, the *Configuration Setup Engine* will generate a template as follows: the *Real-time Edge Processing*'s will be configured to include the *xdr_udp* microservice reported in Table 1. The *Programmable Massive IoT Platform* will be configured with an *Adaptation Layer* capable of performing XDR re-encoding on the measurements received by the *Distributed Software IoT Layer*, and to stream them towards the Edge Domain over UDP.

Lifecycle manager This takes care of deploying instances of the required software components on the testbed's resources, and receives as input an *Experiment Deployment Template* generated by the *Configuration Setup Engine*, as represented in Fig. 4. The current implementation performs this task via the Python Fabric library [11], interacting with the participating testbed remote hosts over SSH.

The experiment deployment phase terminates when all the required entities have successfully been activated. As soon as the execution of an experiment is completed (e.g., a timeout has expired), the *Lifecycle Manager* begins the decommission phase. The previously allocated software entities will be dismissed (using an approach similar to the deployment phase), and the hardware resources will be freed up and made available for the execution of the next experiment.

5.2 Distributed Monitoring

This subsystem allows the collection of metrics from an IoT experimentation, and is performed by the *Monitoring Controller* via the dynamic configuration and deployment of instances of the *Data Collection*, *Data Reporting*, and *Probe* entities on the relevant parts of the testbed. The *Monitoring Controller* acts as an interface toward the *Orchestrator* and provides the functionalities to start / stop other monitoring components on-demand. The *Orchestrator* utilises that interface to trigger the dynamic deployment of a given monitoring topology, based on the *Experiment Deployment Template* generated by its *Configuration Setup Engine*.

The current implementation of this subsystem reuses some of software entities provided by Lattice [20]. Figure 4 shows an example of monitoring topology that consists of two *Data Collection* entities running respectively on the *Edu Cluster* and on the *Clay Cluster*. These entities receive measurements from one or more *Probes*, each gathering a relevant type of metric: the amount of bytes received and the resources utilised by the *Edge Data Collection* component, the CPU and Memory usage of the edge hosts, etc.

The figure also shows that the *Data Collection* entities send the above metrics values, over the network, to a remote *Data Reporting* component running on the *Gas* management cluster. This interaction is highlighted by the solid teal coloured lines that interconnect all of those entities. The *Data Collection* acts as an aggregation point and takes care of logging the metric values, by writing them to a file or database for later usage (such as cross-correlation, filtering, visualisation, etc.).

6 Experimental Results

This section provides the results of several experiments performed to evaluate the cost of various encoding and transport mechanisms, utilised by edge applications and APIs, while dealing with the massive data generated by thousands of IoT sensors. The results are presented and analysed to show the impact design choices can have on the utilisation of the available edge resources.

For each experiment, an instance of the *Edge Data Collection* microservice is selected from the API Types presented in Table 1, and deployed onto a single edge host. A bespoke instance of the *Programmable Massive IoT Platform*, customised to generate measurements at a given rate and a specified topology of *software sensors*, is also activated on the testbed.

Each experiment has a lifespan, and throughout, the sensor data is continuously generated by the IoT Domain at a specified rate. The data stream is transmitted over the network to the edge host, where the sensors measurements are received and decoded by the *Edge Data Collection* microservice. The data encoding format and the transmission mechanism utilised during an experiment depend on the API type required for the evaluation of the *Edge Data Collection*. In these experiments a lifespan of 3600 seconds, namely 60 minutes, was selected.

For a given experiment, the setup is characterised by the following set of parameters:

- **Data Encoding**: the type of data encoding format exchanged between the IoT and the Edge Domains;
- **Network Transport**: the mechanism whereby the IoT and the Edge Domains exchange data;
- **Data Rate**: the cumulative number of *sensor measurements* sent every second;
- **Software Gateways**: the number of *Software Gateways* within the IoT platform;
- **Software Sensors**: the total number of *software sensors* attached to the *Software Gateways*;
- **Attributes**: the number of *Attributes* included within each *sensors' measurement*;
- **Lifespan**: the length of the experiment, in seconds;
- **Edge Data Collection**: the Type of the API

The evaluations described here compare four metrics collected during each experiment by the *Distributed Monitoring Subsystem*, where:

- The first and second metrics are the *number of bytes received* and the *number of bytes sent* by the microservice application running on the edge;
- The third and fourth metrics are respectively the *percentage of CPU* and the *amount of Memory resources* utilised on the edge host for the execution of the same applications.

The results of the execution of the multiple experiments have been grouped and presented in separate subsections based on different combinations of the parameters listed above. In each subsection, a graph for each of the 4 metrics compares the results gathered during the experiments, highlighting the performance of the associated *Edge Data Collection* applications.

In order to determine any data loss or delay within the experiments, the number of generated IoT measurements is compared with the number of messages received at the edge, during the 3600 second lifespan. In general the experiments showed a high level of delivery reliability, and data loss and delay was not detected during any of the experiments, expect for Experiment 1, which had a considerably lower data collection.

6.1 Experiment 1 – JSON REST Python

This first experiment targeted the setup of an IoT analytics scenario where the *Edge Data Collection* had features similar to the applications developed via the most widely utilised microservices and FaaS frameworks. Hence, this experiment was based on a Python edge application that received JSON encoded data via REST [29].

Experiment Setup:

Data Encoding	JSON
Network Transport	REST
Data Rate	10,000 messages/second
Software Gateways	10
Software Sensors	10,000
Attributes	1
Lifespan	3600 seconds
Edge Data Collection	**python_json_rest**

python_json_rest was developed in Python 3 with Flask 1.1, and using the Waitress 1.4 WSGI [42] with 4 worker threads

Experiment Results In this experiment, the transport mechanism utilised by *Edge Data Collection* was based on REST, which eliminates data loss during the communication between the IoT Domain and the Edge Domain as it uses HTTP, which is based on TCP. The results highlighted something unexpected. Each sensor generated 1 message per second, and although the total number of measurements generated by the sensors during the experiment was $= 10000 * 3600 = 36\ million$, the total number of measurements received and decoded by the *Edge Data Collection* was considerably lower than that, at $\approx 1\ million$.

When using REST, the interaction between the IoT Domain and the Edge Domain is synchronous. An IoT platform *Adaptor* takes a measurement off its queue and performs a HTTP POST, by invoking the designated endpoint of the *Edge Data Collection*, and providing a JSON encoded measurement as the request body. This synchronous invocation terminates when a HTTP Accepted (*Status Code 202*) is returned as reply.

However, if the rate at which the *Adaptor* sends the above data is higher than the rate at which the *Edge Data Collection* is able to decode them, then the measurements will accumulate within the internal queue of the Adaptor, even causing possible memory overflows. Our investigations confirmed this as the cause of the mismatch between the generated measurements and the ones actually received / decoded at the edge.

This issue was caused by the *python_json_rest* implementation, which was not able to decode all the received measurements in real-time, and therefore too slow.

During the experiment's time frame, each *Adaptor* was expected to receive 3.6 $million$ measurements from the *Software Gateway* connected to it, which is the total amount of generated measurements divided by the number of deployed *Software Gateway*s, i.e., $36\ million/10 = 3.6\ million$. At the end of the experiment, the number of measurements received by each *Adaptor* was 3.6 $million$, but the number sent to the edge was only 100,000. This gives a total transmission of $100,000 * 10 = 1\ million$ measurements. Therefore, when the experiment was completed after 3600 seconds, most of the generated measurements were still queued in the *Adaptation Layer* of the IoT Platform. They could not be sent to the edge due to: i) the usage of a synchronous communication pattern between the *Adaptation Layer* and the *Edge Data Collection*; ii) the intrinsic low decoding throughput of the particular implementation of the *Edge Data Collection* used for this experiment.

Some preliminary conclusions of this experiment show that the real-time decoding can only happen with significantly increased computing power, by adding to the number of instances of the *python_json_rest* microservice running at the edge, or by dramatically reducing the rate at which the measurements are generated by the IoT platform and transmitted to the edge. It is this last approach that is selected for Experiment 2.

6.2 Experiment Set 2

In this set of experiments, the platform was configured to generate measurements at a lower rate compared to *Experiment 1*, to overcome the low throughput issues.

The rate was reduced by a factor $1/36$ in order to allow real-time data decoding. The performance of the Python JSON/REST application was compared with i) a Java-based implementation of JSON/REST and ii) alternative Python implementations, based on data streaming over WebSocket, using either XDR or JSON.

Experiment Setup:

Data Encoding	JSON and XDR
Network Transport	REST and WebSocket
Data Rate	300 messages/second
Software Gateways	10
Software Sensors	10,000
Attributes	1
Lifespan	3600 seconds
Edge Data Collection	**json_rest_python**, **json_ws_python**, **xdr_ws_python** and **json_rest**

json_rest_python, **json_ws_python**, **xdr_ws_python** were developed in Python 3 and use the PyPI WebSockets library [30] and standard Python libraries for JSON and XDR. **json_rest** was developed in Java 1.8 and is based on the Simple HTTP framework [33] and the Resty JSON library [31]

Experiment results The performance of the different types of *Edge Data Collection* microservices is shown in Fig. 5. There are 4 graphs, one for each of the 4 metrics. The graph *Received Bytes* shows that $\approx$ 600 MB was transmitted from the IoT Domain to the Edge Domain by both the *json_rest_python* and the *json_rest* implementations. The graph *Reply Bytes* shows that over 180 MB was transmitted as reply bytes for both those implementations. This is due to the overhead associated with the use of the REST messaging protocol.

The number of bytes received on the edge host when either the *json_ws_python* or the *xdr_ws_python* implementations were running, is smaller than the bytes reported for the previous two cases which used REST. It is $\approx$ 400 MB for *json_ws_python* and $\approx$ 200 MB for *xdr_ws_python*. Furthermore, for both of them the number of reply bytes is negligible. This is due to WebSocket relying on HTTP but using a data streaming approach, as opposed to the request / reply model of REST. XDR is a more efficient way of encoding the measurements, with a total lower count of exchanged bytes.

The graph *CPU Usage* shows that the highest consumption of resources (throughout the whole experiment) is associated to *json_rest_python* ($\approx$ 100%). Both *json_ws_python* and *xdr_ws_python* consumed less CPU ($\approx$ 25%), regardless of the data encoding type. Although *json_rest* was similar to *json_rest_python*, it reported the lowest consumption of CPU resources (less than 20%) among all the implementations. This result is likely due to the usage of the Java runtime, which outperformed all the Python based implementations.

As the *Memory Usage* graph shows, the *json_rest* used more Memory than all the Python implementations. The difference is quite significant, with $\approx$

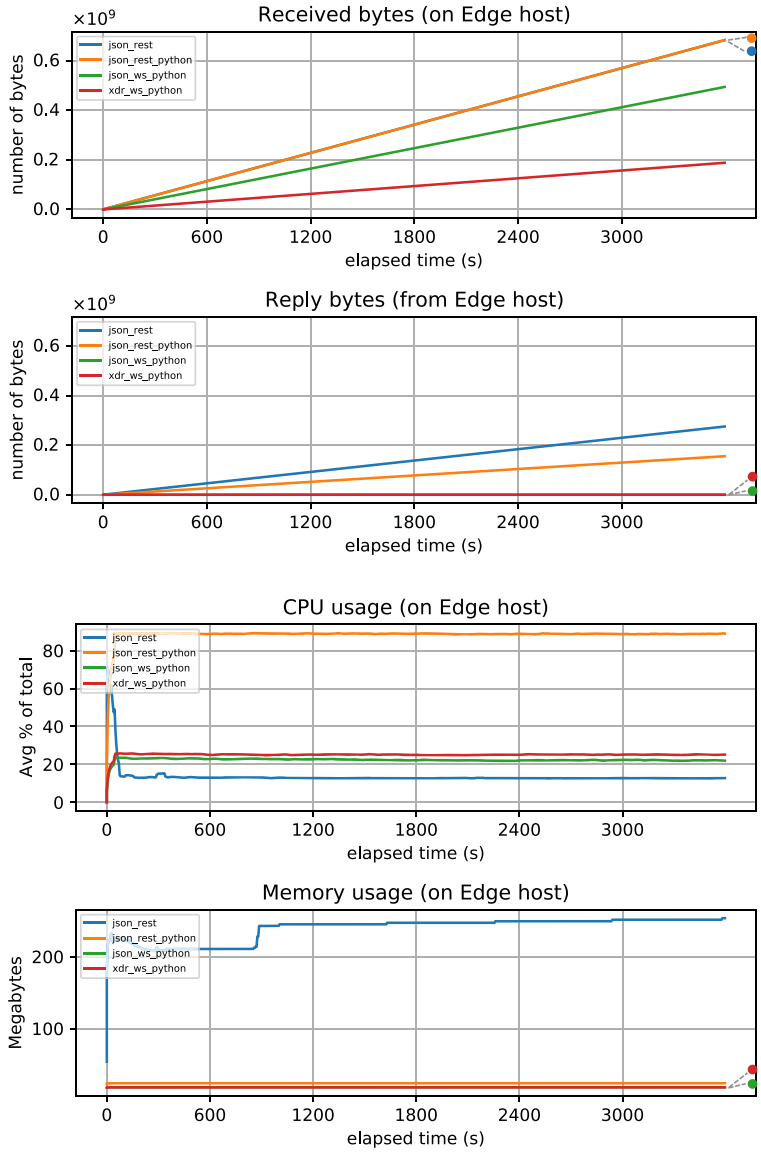

Fig. 5 Results of experiment set 2

250 MB memory usage for the *json_rest* implementation, and less than 20 MB the amount of memory utilised by the other ones. This result is aligned with the default Java runtime memory allocation policies.

These results clearly show that there is a significant difference between the performance of *json_rest_python* and the performance of both *json_ws_python* and *xdr_ws_python*. This is mostly related to the type of network transport and the chosen data encoding. While there is a much lower consumption of CPU resources for the Java *json_rest* implementation, compared to any of the Python implementations, the memory consumption shows an opposite trend.

To evaluate higher measurement throughput, the next set of experiments considers additional implementations in Java.

6.3 Experiment Set 3

In this set of experiments, the behaviour of the Java *json_rest* microservice was compared with additional Java implementations. These were based on APIs receiving either XDR or JSON encoded data streamed

over WebSocket or over UDP. The platform was configured to generate measurements at the higher rate utilised in *Experiment 1*, in order to test whether the Java implementations were able to achieve a higher data decoding throughput, compared to the previous Python ones. During these experiments, the option of sending the data with *no names* for the Attributes within each measurement, was also evaluated.

Experiment Setup:

Data Encoding	JSON and XDR
Network Transport	REST, WebSocket and UDP
Data Rate	10,000 messages/second
Software Gateways	10
Software Sensors	10,000
Attributes	10
Lifespan	3600 seconds
Edge Data Collection	**json_rest**, **json_ws**, **xdr_ws**, **json_udp** and **xdr_udp**

JSON data decoding is based on the Resty [31] library, and XDR data decoding is performed via a custom implementation from [20]. The WebSocket network transport was built via the event-driven TooTallNate library [37]. The UDP network transport uses the standard *java.net.DatagramSocket* class

Experiment Results The results gathered show the decoding throughput of the tested Java microservices was higher than the throughput achieved by the Python ones, and allowed performing real-time data decoding at the higher selected rate, and also confirmed a match between the number of generated measurements and the number of measurements decoded at the edge.

The 4 graphs of Fig. 6 show the collected results, which are consistent with the outcome of *Experiment Set 2*, however, the total amount of data generated for this experiment is larger than the previous one due to the higher sensor data rate. The graph *Received Bytes* again highlights that the largest amount of data was transmitted when using *json_rest*, and is slightly more than 20 GB. That is followed by the other JSON based implementations: *json_ws* and *json_udp*, which transmit $\approx$ 17 GB. The XDR versions *xdr_ws* and *xdr_udp* showed a considerably lower amount of received data, due to the usage of a more efficient binary encoding, with a value of $\approx$ 6 GB for these implementations.

For this group of experiments, additional tests that involved configuring the sensors to generate measurements that did not include the names of the Attributes, were setup for both *xdr_ws* and *xdr_udp*. The related results are labelled in all the graphs with the *_no_names* suffix. As all the experiments only included a single Attribute, the graph *Received Bytes* shows that a lower amount of data was received by not sending the names, $\approx$ 5 GB versus $\approx$ 6 GB with names.

On the *Reply bytes* graph, bytes were sent from the edge host towards the IoT Domain by the *json_rest* implementation ($\approx$ 10 GB). This is due to the REST request / reply messaging protocol on which it is based. The streaming APIs replies were minimal with WebSocket and zero with UDP.

The *CPU usage* graph shows that the highest CPU consumption was associated to the *json_rest* implementation ($\approx$ 300%), consistent with *Experiment Set 2*. This is followed by *json_ws* ($\approx$ 100%), and *json_udp* ($\approx$ 60%). Both the *xdr_ws* and the *xdr_udp* implementations performed similarly, and much better than the others, with ($\approx$ 30%), regardless of the measurements including the Attribute names (*_no_names*) or not. The *Memory usage* graph shows a much higher utilisation associated to the *json_rest* implementation ($\approx$ 600 MB). All the other microservices reported a lower memory consumption in the range of $200 - 300$ MB.

The number of measurements was increased by 36 times over *Experiment Set 2*, and we observe that Java is reliable and performant with this data rate.

6.4 Experiment Set 4

This set of experiments is an extension of *Experiment Set 3*, with a higher number of Attributes being sent within the generated measurements.

Experiment Setup:

Data Encoding	JSON and XDR
Network Transport	REST, WebSocket and UDP
Data Rate	10,000 messages/second
Software Gateways	10
Software Sensors	10,000
Attributes	10
Lifespan	3600 seconds
Edge Data Collection	**json_rest**, **json_ws**, **xdr_ws**, **json_udp** and **xdr_udp**

Experiment Results The observations made for the *Experiment Set 3* can also be applied here (Fig. 7).

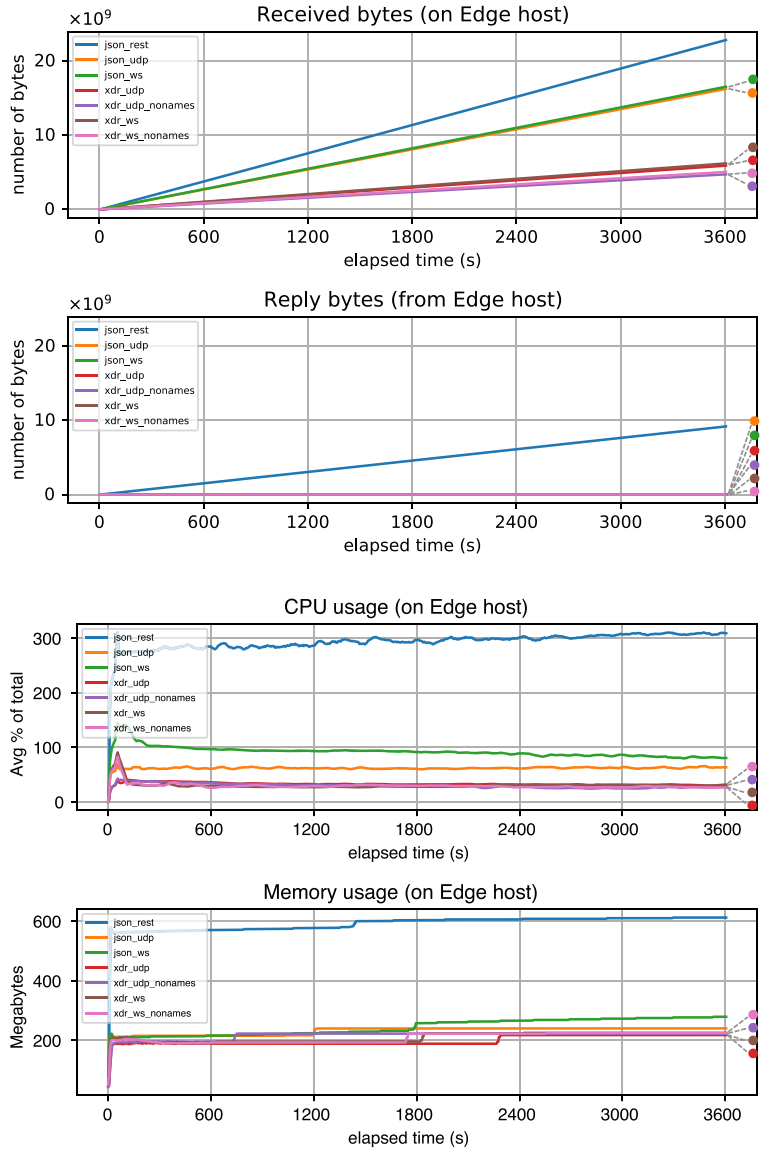

Fig. 6 Results of experiment set 3

This time the larger number of Attributes sent within each measurement had a bigger impact of the total amount of bytes received on the edge host. More importantly, the final amount of *Received Bytes* reported during the XDR *_no_names* tests ($\approx$ 10 GB), was about half of the amount reported for the other XDR tests with the Attributes names ($\approx$ 18 GB). This shows how much of the full data stream can be consumed by these names.

The graph *CPU Usage* shows an increase in the required resources for all the implementations, in comparison with the results of *Experiment Set 3*. Furthermore, there is a lower consumption of CPU resources associated to the *xdr_ws* and the *xdr_udp* implementations during the execution of the *_no_names* tests. This result is again due to the higher number of Attributes (10 versus 1) encoded within each measurement.

The graph *Memory Usage* highlights a much higher usage of memory for both the *json_rest* and *json_ws* implementations. On the other hand, the amount of memory required for the execution of the XDR implementations does not seem to be particularly affected by the larger number of Attributes within the measurements.

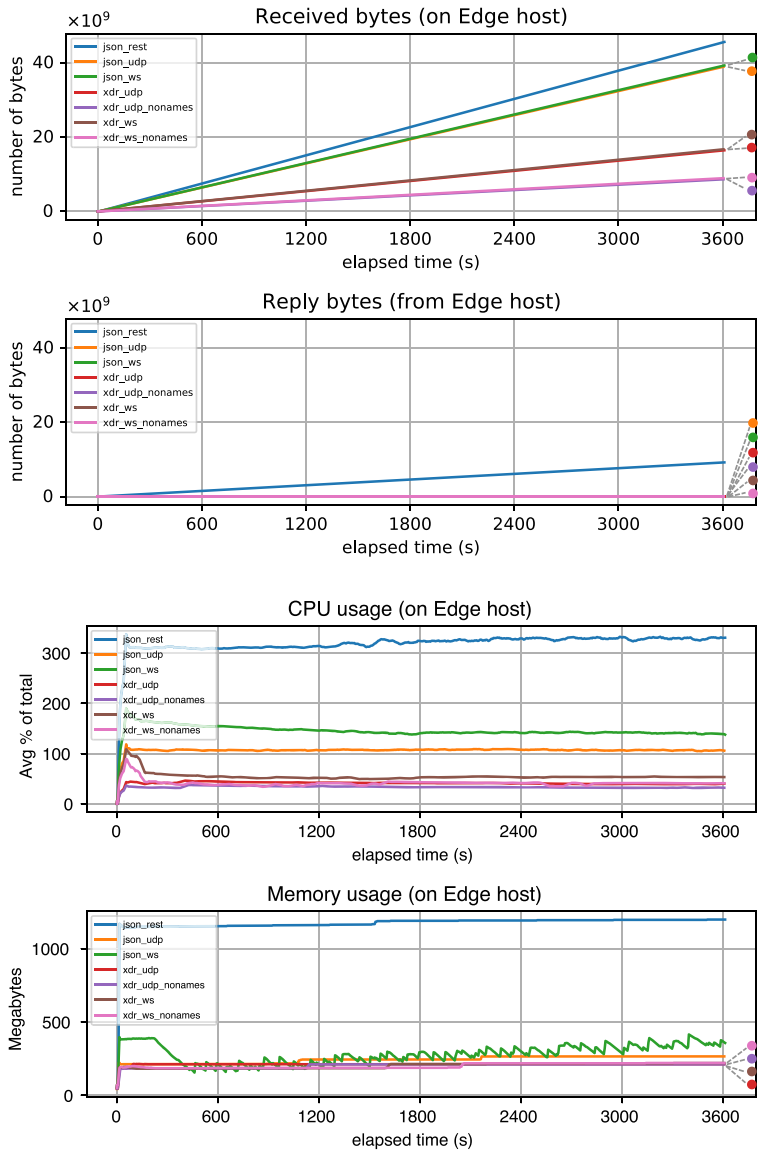

Fig. 7 Results of experiment set 4

6.5 Discussion on the Experimental Results

Looking at the results from the previous four sets of experiments, a common pattern in terms of behaviour and performance of all the tested *Edge Data Collection* microservices can be observed.

Firstly, the measured *decoding throughput* was consistently higher for all the Java implementations when compared to the Python ones. Moreover, the Java microservices always required lower amounts of CPU resources but higher amounts of memory. The results also showed that the number of bytes exchanged by the IoT Domain and the edge host was similar for a given type of API, regardless of the programming language, and frameworks / libraries, utilised for their development.

During the experiments, the average value of each metric is collected and calculated over 10 runs, and shown in the graphs. Due to the large scale of the y-axis, and the small variance of the statistical results, the confidence intervals are tiny, and so they would not be intelligible when shown in the graphs. The average *coefficient of variation* for these metrics has been calculated and show the following deviation: 3%

for the *amount of CPU*, 5% for the *amount of Memory resources* and 0.1% for both the *number of bytes received* and the *number of bytes sent*.

In general, the *number of bytes* exchanged by the IoT Domain and the Edge Domain was impacted by the type of network transport. Those based on REST showed a higher volume of *Received bytes*, when compared to the ones using either WebSocket or UDP. For this metric, the difference between WebSocket and UDP was negligible. The different behaviours are due to the higher overhead of managing multiple HTTP connections associated to REST, as opposed to the usage of either a persistent WebSocket connection or the connection-less "fire and forget" (unreliable) UDP approach. An additional difference between REST and the data streaming protocols is in the number of generated *Reply bytes*. As REST uses HTTP, data is always sent as reply; conversely, when streaming data either over WebSocket or UDP, the number of *Reply bytes* is negligible (or zero). This contributes to the overall amount of bytes exchanged by the IoT Domain and the Edge Domain.

The particular type of data encoding associated to an API significantly affected the amount of *Received bytes* metric. Self-describing data interchange formats such as JSON, although having the benefit of being human readable and widely supported by many applications over the Internet, resulted in a higher number of generated bytes when compared to more efficient binary formats like XDR. Moreover, optimising the measurement data model by not including the attribute names within the XDR encoded data, effectively reduced the amount of *Received bytes*, especially when the measurements contained 10 Attributes.

The values related to the *CPU utilisation* metric followed a similar pattern. In particular, the REST implementations required more CPU resources than the WebSocket or UDP ones. This happens because the REST microservices have to implement the HTTP handling functionalities and the generation of the HTTP replies. *json_rest* is based on the Java *simple-framework* [33] and it also allocates multiple threads.

Also note that WebSocket uses slightly more CPU than UDP, when considering the same type of data encoding, as it is built on HTTP.

The Java implementations use less CPU than the Python ones, at a given measurement rate, as Python is an interpreted language and is known for its flexibility rather than performance. For each measurement decoding operation, the overhead due to the interpretation of the instructions needs to be factored in. This gives a less efficient CPU utilisation and a lower decoding throughput.

Predictably, and consistent with the results for the amount of *Received bytes*, the usage of the JSON data encoding format required more CPU resources than XDR, regardless of the implementation chosen for the network transport mechanism. The option of not encoding the attribute names within the XDR data, provided a significant lower CPU utilisation, particularly when each measurement included 10 Attributes.

Finally, the *Memory usage* metric showed comparable values for all the considered Java microservices, with the only exception being the REST implementation (due to the usage of the multi-threaded *simple-framework*). Moreover, these values were consistently higher than those reported by the Python microservices regardless of the implemented type of API. It is commonly known that Java programs can, in general, use more memory resources than Python ones. This is due to the memory management and garbage collection mechanisms, as well as to the allocation of multiple threads.

From these results, Python may be more suitable for memory-constrained scenarios combined with a considerably lower decoding throughput; however, it may not be a good choice when a massive number of IoT data streams need to be processed in real-time. Java will fit well in scenarios that require high decoding throughput, allow the allocation of multiple threads for concurrency, yet still require limited CPU resources.

7 Conclusions

This paper presents a real-time IoT analytics scenario where processing functions are placed at the edge in order to minimise the end-to-end latency. When microservice architectures are considered for the design of those processing functions, it is paramount to ensure adequate resourcing for their APIs in terms of functionality and performance, especially when there are a massive number of IoT devices.

It was shown how the choice of a given API implementation determines both the type of *data encoding* and the type of *network transport* mechanisms for the communication between the "things" and the edge;

this, ultimately has a huge impact also on the edge resource utilisation in terms of required computing and networking capabilities.

The main research question targeted was an understanding of what of the above mechanisms have the lowest traffic and least resource cost and impact on the edge.

Various large-scale experiments, with thousands of IoT software sensors, were performed on a testbed hosting our *Programmable Massive IoT Platform*. Several types of transmission mechanisms were tested and evaluated against four main metrics: namely, the *number of bytes received* and the *number of bytes sent* by the edge microservices; the *CPU usage* and *Memory usage* of the associated edge resources.

The results show how the traffic and the resource consumption were clearly affected by the messaging protocol, network technology, and data format, and to some extent by the choice of the programming language and frameworks utilised for the implementation.

In general, the use of self-describing formats such as JSON has a significant impact on both the amount of exchanged data and on the amount of resources required to perform the data decoding operations. For the network transport, the REST request / response protocol showed a higher amount of exchanged data compared to data streaming, as well as a much higher usage of computational resources. A consequence of this lower performance is that many more resources are needed to do the same job.

The use of the binary encoded XDR consistently returned the lowest amount of exchanged data, as well as the lowest consumption of edge resources.

The microservices based on either WebSocket or UDP transport protocols showed comparable performance, and both always exchanged a lower amount of data than the REST ones. The resource utilisation was also lower than the REST microservices, with a slightly higher consumption of CPU associated to the WebSocket implementation.

A binary format like XDR represents the best choice for the data interchange format between the IoT Domain and the Edge Domain, as it allows both the transmission of a smaller amount of data and a lower resource utilisation than JSON. Either the WebSocket or the UDP network transport protocols offer comparable performance but, as WebSocket provides reliable data transmission and is widely supported by many Web Applications, it may be the preferred network transport solution for the design of the Edge Data Collection.

All the Python implementations showed lower memory consumption but higher CPU utilisation compared to the Java ones. They also provided a lower decoding throughput due to the intrinsic interpreted nature of Python and to the global locking mechanisms that prevented the concurrent execution of multiple CPU-bound threads. As such, Python is a good choice for memory-constrained scenarios when smaller amounts of data do not need to be processed in real-time. On the other hand, Java, demonstrating high-throughput and real-time data handling, plus multi-threaded capabilities, fits well with the massive real-time IoT data processing scenario presented in this paper.

As further work, we plan to evaluate the throughput and cost of using MQTT, which uses a publish/subscribe messaging protocol and a broker for queueing and distribution. Many pub/sub brokers have difficulty sustaining a high throughput of messages when there are many *topics* [5], so this may have a big impact on the total message delivery capability. We also plan to extend the current implementation of the Real-time Edge Processing by developing the data processing backend.

We target a distributed system where data processing functions can dynamically be orchestrated, and the data generated by the IoT devices can be off-loaded towards the edge or the cloud according to application requirements. An in-depth analysis based on the execution of multiple parallel instances of both the Edge Data Collection and the Edge Data Processing applications will also be undertaken, providing an assessment of how horizontal scaling can increase the overall processing throughput and performance, and how this affects the utilisation of the distributed edge resources. More orchestration algorithms that take those aspects into account might be developed as we further scale up and optimise the *Programmable Massive IoT Platform*.

Acknowledgements Dr F. Tusa and Dr S. Clayman are partially supported by Huawei Technologies Co., Ltd.

References

1. Abbasi, M., Mohammadi Pasand, E., Khosravi, M.R.: Workload Allocation in IoT-Fog-cloud architecture using a multi-objective genetic algorithm. Journal of Grid Computing **18**(1), 43–56 (2020)
2. Alam, M., Rufino, J., Ferreira, J., Ahmed, S.H., Shah, N., Chen, Y.: Orchestration of Microservices for IoT using docker and edge computing. IEEE Commun. Mag. **56**(9), 118–123 (2018)
3. Aral, A., Brandic, I., Uriarte, R.B., De Nicola, R., Scoca, V.: Addressing application latency requirements through edge scheduling. Journal of Grid Computing **17**(4), 677–698 (2019)
4. Brambilla, G., Picone, M., Cirani, S., Amoretti, M., Zanichelli, F.: A simulation platform for Large-Scale internet of things scenarios in urban environments. In: Proceedings of the First International Conference on IoT in Urban Space, p. 50–55 (2014)
5. Carzaniga, A., Hall, C., Toffetti, G.C., Wolf, A.L.: Practical High-Throughput Content-Based Routing Using Unicast State and Probabilistic Encodings (2009)
6. Chen, C.Y., Fu, J.H., Sung, T., Wang, P., Jou, E., Feng, M.: Complex event processing for the internet of things and its applications. In: 2014 IEEE International Conference on Automation Science and Engineering (CASE), pp. 1144–1149 (2014)
7. Clayman, S., Galis, A., Mamatas, L.: Monitoring virtual networks with lattice. In: 2010 IEEE/IFIP Network Operations and Management Symposium Workshops, pp. 239–246 (2010)
8. Dizdarević, J., Carpio, F., Jukan, A., Masip-Bruin, X.: A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. ACM comput. Surv 51(6) (2019)
9. Emeakaroha, V., Fatema, K., Healy, P., Morrison, J.: Towards a generic cloud-based sensor data management platform: a survey and conceptual architecture. In: SENSORCOMM 2014 - 8th International Conference on Sensor Technologies and Applications, pp. 88–95 (2014)
10. Emeakaroha, V.C., Healy, P., Fatema, K., Morrison, J.P.: Analysis of Data Interchange Formats for Interoperable and Efficient Data Communication in Clouds. In: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing, pp. 393–398 (2013)
11. Fabric: Pythonic remote execution. http://www.fabfile.org
12. Google: Protocol Buffers. https://developers.google.com/protocol-buffers
13. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. Futur. Gener. Comput. Syst. **29**(7), 1645–1660 (2013)
14. Hong, X.J., Sik Yang, H., Kim, Y.H.: Performance analysis of RESTful API and RabbitMQ for microservice web application. In: 2018 International Conference on Information and Communication Technology Convergence (ICTC), pp. 257–259 (2018)
15. Hosseinzadeh, M., Masdari, M., Rahmani, A.M., Mohammadi, M., Aldalwie, A.H.M., Majeed, M.K., Karim, S.H.T.: Improved butterfly optimization algorithm for data placement and scheduling in edge computing environments. Journal of Grid Computing **19**(2), 14 (2021)
16. IoT Experiment Orchestrator. https://github.com/francesco-tusa/iot-orchestrator
17. Javed, A., Robert, J., Heljanko, K., Främling, K.: IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications. Journal of Grid Computing **18**(1), 57–80 (2020)
18. Kunze, I., Wehrle, K.: Industrial Use Cases for In-Network Computing. Internet-Draft draft-kunze-coin-industrial-use-cases-03, Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-kunze-coin-industrial-use-cases-03. Work in Progress (2020)
19. Laaroussi, Z., Morabito, R., Taleb, T.: Service Provisioning in Vehicular Networks Through Edge and Cloud: An Empirical Analysis (2018)
20. Lattice: Dynamic and Programmable Monitoring Framework. https://github.com/UCL/lattice-monitoring-framework
21. Li, J., Kulkarni, S.G., Ramakrishnan, K.K., Li, D.: Understanding open source serverless platforms: Design considerations and performance. In: Proceedings of the 5th International Workshop on Serverless Computing, WOSC '19, Association for Computing Machinery, New York, NY, USA, pp. 37–42 (2019)
22. Looga, V., Ou, Z., Deng, Y., Ylä-Jääski, A.: MAMMOTH: A Massive-Scale Emulation Platform for Internet of Things. In: 2012 IEEE 2Nd Intl. Conf. on Cloud Computing and Intelligence Systems, vol. 03, pp. 1235–1239 (2012)
23. Lubbers, P., Greco, F.: HTML5 WebSocket: A Quantum Leap in Scalability for the Web. http://websocket.org/quantum.html (2015)
24. LY-Trong, N., Dang-Le-Bao, C., Le-Trung, Q.: Towards a Large-Scale IoT Emulation Testbed Based on Container Technology. In: 2018 IEEE Seventh International Conference on Communications and Electronics (ICCE), pp. 63–68 (2018)
25. Mishra, B., Kertesz, A.: The Use of MQTT in M2M and IoT Systems: A Survey. IEEE Access **8**, 201071–201086 (2020)
26. MQTT: The Standard for IoT Messaging. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf

27. Naik, N.: Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. In: 2017 IEEE International Systems Engineering Symposium (ISSE), pp. 1–7 (2017)
28. Patel, P., Intizar Ali, M., Sheth, A.: On Using the Intelligent Edge for IoT Analytics. IEEE Intell. Syst. **32**(5), 64–69 (2017)
29. Programming Languages You Should Learn in 2020. https://www.computer.org/publications/tech-news/trends/programming-languages-you-should-learn-in-2020
30. PyPI websockets: An implementation of the WebSocket Protocol. https://pypi.org/project/websockets/
31. Resty: A simple HTTP REST client for Java. https://beders.github.io/Resty/Resty/Overview.html
32. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252. https://rfc-editor.org/rfc/rfc7252.txt (2014)
33. Simple: embeddable Java based HTTP engine. http://www.simpleframework.org/index.php
34. Srinivasan, R.: XDR:External Data Representation Standard (1995)
35. Target, S.: The Rise and Rise of JSON. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html (2017)
36. Thota, P., Kim, Y.: Implementation and Comparison of M2M Protocols for Internet of Things. In: 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1St Intl Conf on Big Data, Cloud Computing, Data Science Engineering (ACIT-CSII-BCD), pp. 43–48 (2016)
37. TooTallNate: Java WebSocket implementation. https://github.com/TooTallNate/Java-WebSocket
38. Trinks, S., Felden, C.: Edge Computing Architecture to Support Real Time Analytic Applications : A State-of-the-art within the Application Area of Smart Factory and Industry 4.0. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 2930–2939 (2018)
39. Tusa, F., Clayman, S., Galis, A.: Real-time management and control of monitoring elements in dynamic cloud network systems. In: 2018 IEEE 7th International Conference on Cloud Networking (Cloudnet), pp. 1–7 (2018)
40. Tusa, F., Clayman, S., Valocchi, D., Galis, A.: Multi-domain orchestration for the deployment and management of services on a slice enabled NFVI. In: IEEE Mobility Support in Slice-Based Network Control for Heterogeneous Environments Co-Hosted with Conference on Network Function Virtualization and Software Defined Networks. Verona (2018)
41. Verma, S., Kawamoto, Y., Fadlullah, Z.M., Nishiyama, H., Kato, N.: A survey on network methodologies for real-time analytics of massive iot data and open research issues. IEEE Communications Surveys Tutorials **19**(3), 1457–1477 (2017)
42. Waitress pure-Python WSGI server. https://docs.pylonsproject.org/projects/waitress/
43. Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C., Lübke, D.: Guiding architectural decision making on quality aspects in microservice APIs. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) Computing, Service-Oriented, pp. 73–89, Springer Intl. Publ (2018)
44. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: Interface representation patterns: crafting and consuming message-based remote APIs. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs, EuroPLoP '17 (2017)