Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1157–1166 DOI: 10.15439/2017F381 ISSN 2300-5963 ACSIS, Vol. 11

An efficient real-time architecture for collecting IoT data

Mark Phillip Loria, Marco Toja
See Your Box Ltd
2 Common Road
London. England SE5 9RA UK
Email: {mloria, mtoja}@seeyourbox.com

Vincenza Carchiolo, Michele Malgeri
Universitá di Catania,
Dip. Ingegneria Elettrica Elettronica e Informatica,
Viale Andrea Doria 6, 95126 Catania, Italy
Email: {vincenza.carchiolo, michele.malgeri}@dieei.unict.it

***Abstract*—IoT applications has some characteristics that set it apart from other fields mainly due to the multitude of different types of sensors producing data. In monitoring applications, data processing requires real-time or soft real-time responses in order to aid systems to make important decisions but also predictive analysis to leverage the potential of IoT by data mining vast datasets. This paper presents an architecture developed to efficiently process and store data coming from an huge number of distributed IoT sensors. The back-end of SeeYourBox services is currently based on the proposed architecture that has proven to be stable and meet all the requirements.**

I. Introduction

The ubiquitous presence of interconnected devices with sensing and communication abilities has brought us to the era of the Internet of Things (IoT). RFID tags, sensors and actuators are only a few examples of components that enable embedded devices and mobile phones to create a network of things that collect and transmit data from the environment they are placed in [3]. Data in IoT applications has some characteristics that set it apart from other fields. We shall go through them as presented by Li et al. [16]. With information flowing from a multitude of different types of sensors data is heterogeneous, with a direct impact on how it's organized within the data storage solution. Reduction in sensor cost, miniaturization and advances in wireless technologies and techniques contributed to the pervasive distribution of connected objects [13]. This leads the design of the application to take into consideration the factor of scale in the early stages of development. Added value is given to recorded data by defining location or context-awareness [18].

Data collected from distributed sensors can be enriched by a multitude of sources. Common examples are city traffic or emotional status of people [20]. Devices can have more than one sensor, effectively requiring multidimensional data management [8]. The most evident consequence of these characteristics is that data itself shapes and influences the design of an IoT system architecture. Organization of data storage and processing technologies and techniques are the two main aspects to consider [24]. In monitoring applications data processing requires real-time or near real-time responses in order to aid systems to make important decisions [25]. On the other hand predictive analysis can leverage the potential of IoT by data mining vast datasets [15]. Management of IoT data can be seen and considered from two view points i) Processing and ii) Storing the IoT data.

In the typical IoT scenario millions of devices constantly feed a data ingestion system with data sourced by integrated sensors. The system must be able to handle and process incoming data with as low as possible response times to avoid building system bottlenecks. The challenge is to design a system capable of processing requests that can have real-time requirements [19]. On the other hand, the volume of data produced by the average IoT system over time quickly translates into ever growing datasets. These can be used for historical reasons only or for creating predictive analysis systems that use data mining for extracting valuable information. The speed with which these datasets grow requires a system that is capable of handling big data from it's early stages.

It is common to consider these two main aspects of IoT system as separate and a way to refer to them is *hot-path* or online processing and *cold-path* processing. The former term is used to refer to data that is processed before being stored. The latter, or offline processing, is used to refer to analysis that is performed after data is stored. Typical examples are statistical analysis, reporting and data mining. In the following sections we will dive into common strategies that revolve around data stores to support these requirements in an efficient way. Both aspects require the support of a data storage solution capable of managing the challenges of IoT data. Both will need to face common challenges but with different goals.

The challenges that big data generate in IoT systems are [6]:

- Variety, with things equipped with multiple sensors and things that serve different purposes, data in an IoT system is typically unstructured and varies rapidly over time to adapt to dynamic environments and tasks.
- Volume Lower costs, smaller dimensions and better battery life achieved by embedded systems in recent years has enabled the IoT to become more popular and pervasive [13]. The widespread availability of connected things constantly transmitting data generates a demanding amount of data to process.
- Velocity, while processing speed and performance is usually sought as a positive quality of any data processing system, in the IoT it has direct implications on how things can react to the environment and events. It is not

uncommon for IoT systems to be required to meet real-time constraints.
- Veracity, the power of unleashing computation and sensing capabilities to devices that can blend with their environment without direct supervision raises issues regarding the trustworthiness of data. Dealing with trustworthiness and repution, also the problem to define a strategy that minimize the attachment cost is a relevant problem [7]
- Value, the ability of sourcing data that would be otherwise unavailable and the simplification of creating telemetry systems offers tremendous opportunity to generate value by analyzing data both online and offline.

The characteristics of big data are the most influencing aspects in designing a system that handles it. However, solely focusing on database management systems is not the only aspect that needs to be addressed when designing a system that needs to manage big data and this paper deals with them.

A key component of an IoT system (or a big data system in general) is the *middleware software layer* that interfaces things and *application back-end* [3]. By abstracting backend implementation and details it allows application developers to focus on delivering value based solutions faster and more efficiently without having to be concerned with technology details of the infrastructure they rely on.

The massive volume of data processed, performance requirements and robustness that are demanded from industrial applications translate often into a system that is able to be distributed and replicated across multiple machines and locations. A carefully engineered middleware allows a system to scale horizontally effortlessly. In this paper we present an architecture designed to collect, elaborate and store information of IoT system. In section II we discuss the different solutions presented in literature. In section III we reason about the motivation to design a new and personalized solution. Section IV discuss the proposed architecture; in particular, in this section we present the solution adopted to collect and store the vast amount of data produced by an IoT system. Finally, Section V presents some conclusive remarks.

II. Related work

In this section we will go through the state of the art of current technology and research in the field of system architecture. In respect to system architecture we will analyze the solutions adopted by two of the major cloud computing providers, Amazon AWS and Microsoft Azure, for creating a specific IoT platform. The recent IoT revolution has raised attention for scalable applications and storage solutions. Mileage might vary a lot between different yet similar applications and there is no silver bullet for solving all classes of problems.

The impact of big data in IoT raises questions on how to manage and store data efficiently. A commonly accepted solution still hasn't emerged as a de facto standard and it is left to system architects and developers to come up with a solution that can provide adequate performance [18]. The challenges to face are numerous and often each is best dealt with different database management systems. NoSQL databases, with their dynamic schema and support for horizontal scaling aid developers in handling scale and heterogeneity of IoT data. However, lack of strong ACID compliance and often lack of support for complex queries, results in reasons to not exclude traditional SQL databases from the possible candidates. Ultimately it's difficult to set guidelines that can define a common solution for dealing with IoT data in an effective and successful way. Small differences in data structure and processing can lead to very different results and approaches.

Since the public announcement of Amazon EC2 cloud computing platform in 2006 [4] an array of companies started to offer on-demand computing solutions. Examples are Google App Engine and Microsoft Azure. These platforms allow flexible pay-to-go solutions to implement an all-in cloud computing infrastructure that is able to scale according to the applications they run. The rich suite they often offer allows developers and designers to customize the architecture to fit the requirements of their applications. However since 2015 two of the major cloud providers, Amazon and Microsoft, launched specific IoT oriented cloud platforms. Other than commercial or marketing reasons, that will not be addressed in this work, targeting a specific field such as IoT is a reasonable move from a purely technological prospective since many of the requirements and problems are recurrent and standardized. IoT telemetry is usually characterized by hot-path and cold-path data analysis. While the first is bound with real time concerns and constraints, requiring event handling and device control, in the cold-path response time is set aside in favor of scale and big-data management. A common solution is to separate these two flows in order to optimize each one accordingly without having to surrender to compromises. Powering virtual cloud environments with optimized messaging paradigms and protocols eases composability of highly optimized modules. The IoT targeted solutions focus greatly on providing data ingestion, routing and processing capabilities.

1) IoT Amazon AWS: Amazon AWS is a suite of cloud computing services offered by Amazon since 2006. Two of the most popular services are Amazon Elastic Compute Cloud (EC2) and Amazon S3 (Simple Storage Service). Both active since 2006, they offer respectively an on demand solution for deploying virtual servers and storage in the cloud. Launched in October 2015 AWS IoT is Amazons' answer to the growing IoT industry that requires secure, bi-directional communication between Internet-connected things and the cloud [5]. The core of Amazon AWS IoT is a publisher-subscriber pattern. To enable the developer to control communication, the platform offers multiple communication protocols that include MQTT, HTTP and MQTT over Websockets. Since internet is not always available for IoT devices (ZigBee or Bluetooth) it's possible to interface these devices with physical gateways. From a device view point AWS offers a dedicated SDK implemented in a variety of languages, such as C, Javascript or Arduino. Particular attention in AWS IoT is dedicated to connectivity and its characteristics in the IoT field. Aside from using protocols optimized for publisher-subscriber communication it offers solutions for managing the high latency or unstable connectivity that often characterizes WSNs and IoT in general. The message broker offers three different solutions: MQTT AWS IoT offers a customized MQTT (Message Queue Telemetry Transport) message broker implementation, HTTP The message broker also supports a pure publishing protocol as a REST API over standard HTTP and MQTT over Websockets. By implementing MQTT over Websockets AWS IoT enables browser based and remote application to interact with the connected devices using AWS credentials.

Message handling and delivery is augmented by a Rules Engine that enables the use of business logic rules for event handling and message routing. Rules can be applied to specific devices or groups of them. The engine is a key component in the Amazon AWS ecosystem as it allows integration with the comprehensive tool set offered by Amazon and allows devices to directly interact with all components of the application. Rules can be defined by using SQL syntax to filter messages received by the broker and examples of associated actions triggered by a rule could be writing to a database or invoking lambda functions.

The Thing Registry, supports the need of a representation of a device or logical component in the cloud. Information regarding a thing is memorized in JSON files that contain a device identifier and attributes. These could be a serial number or manufacturer code. While not mandatory, a registry entry eases management and search of things. Using the AWS IoT console or CLI it's possible to create, update and search things within the registry. Non reliable networking and intermittent connection result in a not always connected device. Such behavior could be enforced also by power saving strategies. To simplify interaction with things Amazon AWS offers a component called Thing Shadow. This feature enables the developer to manage the state of a thing and applications to read messages and interact with it at all times. The underlying system takes care of publishing data when possible. A thing shadow is implemented with a JSON document and acts as an intermediary between actual devices and applications.

Finally, while good security policies are never a bad feature to claim in an information system, in the IoT where things can directly interact with the physical world, they acquire particular importance. Connected devices are required to have credentials to access the messaging broker and the traffic must be encrypted using Transport Layer Security (TLS). Authentication is provided with the use of AWS method (called SigV4) or by using X.509 certificates.

Pricing In Amazon AWS IoT is based on a pay per use structure and priced on the number of messages published from and to the platform. To encourage new customers and developers there is a free tier that allows 250,000 messages per month for 12 months. In this pricing model a message represents a block of data counted in increments of 512 bytes.

2) Azure IoT platform: Microsoft's counterpart to Amazon's AWS IoT is the Azure IoT platform. It allows an organization to connect, store, and analyze device data in both large scale or hobbyist IoT environments. The architecture follows four guideline principles: heterogeneity, security, hyperscale and flexibility [10]. Only outbound connections are allowed and security protocols are implemented at transport and application level. The system allows for direct and indirect connectivity, the latter used for non IP capable devices and can be built on top of AMQP, MQTT or HTTP communication protocols. Devices and gateways can implement edge intelligence and analysis to provide reduction of transmitted data and local decision making. Incoming connections and transmission protocols are managed by the cloud gateway. It enables remote communication between field devices and the cloud and can make use of multiple application level messaging protocols. High-volume telemetry ingestion and device control is supported by message brokering systems. This allows decoupling the edge from the cloud for performance, composability and scalability. Additionally, the platform offers a dedicated solution for high-volume ingestion only scenarios called Azure Event Hub. Devices can connect by direct connectivity, agents or by using client components provided in the form of libraries or SDKs.

Once data has reached the cloud gateway its flow is directed by data pumps and analytic tasks. Microsoft Azure offers the possibility to use a Stream Analytics service or custom event processing solutions. Common tasks that can be performed at this stage are data aggregation or enrichment. Another feature that can be implemented is a rules engine to dynamically execute data driven rules that can be activated or deactivated accordingly. Output produced at this stage can be forwarded to a storage solution or an event handling hub, called Event Hub.

In Azure IoT platform the cloud gateway is the entry point to the cloud infrastructure and enables communication between devices and the application. It's responsible for connection management, authentication and authorization. It usually implements brokered communication model to support event handling and decoupling of components. Multiple application level messaging protocols are available for data routing and management. Azure IoT offers two alternatives in respect to the cloud gateway technology: Azure IoT Hub and Azure Event Hub. The former offers high-performance bidirectional traffic support by combining telemetry ingestion with command and control traffic. The latter is an ingestion only gateway capable of handling heavy concurrent sources at high data rates.

A Device Identify Store offers a direct lookup means for device identity and cryptographic secrets used during authentication procedures. Identity and device registry are kept separate also for performance and security concerns. The identity store can be internal to Azure Hub or implemented as an external component with an array of options such as Azure DocumentDB, Azure Tables, Azure SQL database or third-party solutions. A Device Registry Store keeps information for discovery and reference data related to the device. Metadata associated to devices is contained in this resource and the main difference between this and operational data is that the former is slow changing. The device registry can be implemented in different ways:

- DocumentDB: each device is described by a document and the id corresponds to the device id. This solution is suited for registry function because it accepts arbitrary data structures.
- SQL database: this solution uses a hybrid approach by storing properties as columns or as JSON or XML objects if they represent complex data.
- Third-party solutions: third-party solutions are allowed (e.g. MongoDB or Cassandra), however the actual schema will rapresent a variation of the previous two options.

Azure Iot provides a Device State Store. It contains operational data relative to the device and is separate from the registry. While in the Amazon AWS the device shadow is a core component in Azure the device state store is optional. Data can be pushed directly to storage. An array of implementation options are available for the device state storage: Azure Data Lake used as distributed data store for relational and non relational data, Azure Blob storage that allows to store raw data and Azure Tables to manage device records and values.

The brokered nature of the communication architecture allows for flexible data flow management. Data entering the cloud through the gateway may flow across different data pumps or analytics tasks. This feature allows for efficient parallel data processing. Examples are raw telemetry for registering data from a sensor, hot-path analytics for pattern recognition or alert triggering. The implementation can make use of Azure's stream processing services or custom third-party solutions to also create complex rules engines and event processors.

Pricing Microsoft Azure IoT uses a completely different pricing model compared to Amazon AWS IoT. In place of a flexible pay-per-use, Microsoft offers four tiers that set a ceiling to the maximum number of messages that can be processed per day and their size.

The two IoT cloud solutions presented share some similarities, such as a brokered message management but are quite different in the way they implement it. This is mainly due to the underling protocols they use, AMQP for Microsoft and MQTT for Amazon. However they both support HTTP, a protocol that is commonly used withing cloud based systems. They also take two different approaches on the interactions between things and the cloud platform. In Amazon AWS IoT interaction revolves around the concept of state with the device shadow, a feature that is supported but not mandatory in Azure IoT.

Features supported by security protocols and SDKs are comparable for both solutions. On one side Amazon AWS IoT offers a highly focused platform that defines clearly the architecture of the system. Combined with the rich feature set of the popular Amazon AWS suite it is easy to deploy and integrate the IoT platform within large scale existing systems. On the other hand Microsoft Azure IoT offers a much higher level of customization and will attract interest of designers that are in need of a higher degree of control. This is also reflected by the richer feature set that the AMQP protocol exhibits [22].

Ultimately the pricing models differ a lot. The Amazon model is based on million messages exchanged while Microsoft's on the concept of a Hub and the maximum number of messages it is able to handle. It's difficult to compare these different approaches in a general way since final pricing depends a lot on customer needs, volume and payload size of messages exchanged (AWS's messages are priced in 512B increments).

III. Motivation

In the previous section we analyzed the potential of cloud based IoT platforms for data ingestion and processing. The offerings from Amazon and Microsoft are specially tempting for small to medium scale projects or ones that have to be integrated within an existing system. We can imagine for instance an IT company developing an IoT branch of development to find these solutions particularity attractive as they reduce the R&D costs by offering a reliable turn-key solution that is scalable. The biggest concern remains however focused on two of the major arguments on opting for a cloud based solution or an in house system: costs and control over data

Regarding costs, for a company that founds on IoT its core business and that expects to scale to millions of active devices transmitting constantly every day, the yearly fees can quickly translate into six figure invoices. This is sufficient to require deep investigation on developing an in house solution.

Regarding control over data, when working with high-value and mission critical information it is not infrequent for a customer to require that data is not sent or stored on cloud systems that are not under direct control of the company offering a service. Furthermore, government laws of different countries can apply and require that data is stored in a certain matter.

These two reasons forced us to investigate and build an in-house cloud infrastructure and develop from the ground up a cutting edge architecture that could handle the volume of data generated by a ultra-large-scale IoT project. Investigation of the state of the art in database management systems led to a deep understanding of how data shapes the architecture of a system and what are the true guidelines to take into consideration when designing an IoT processing system. The most valuable outcome was that a high performance large scale system could not rely on a centralized data storage solution for the whole system, and furthermore, on a single DBMS engine for the different components of the system. Research pointed into this direction and preliminary prototyping confirmed that by combining different DBMSs it was possible to achieve performance levels otherwise unreachable with a single shared engine. Additionally, research and empirical evidence demonstrated that performance of a DBMS is heavily related to data structure and the way that it is manipulated. With this in mind and with an openness to reshape dataflow within the system it is possible to expand the array of possible candidates that can match the requirements for data storage. Ultimately the freedom that results from this allows a company to consider the choice of a DBMS not only under the concern of raw performance but also from the points of view of licensing, learning curve, expandability and availability of development tools.

When designing the architecture of a system that needs to process IoT data, one of the first challenges that a designer has to face is how to manage scaling to possibly millions of devices transmitting data simultaneously. Founding an application onto a scalable cloud based infrastructure can represent a viable strategy as it allows designers to focus on core technology without the burden of managing in-house legacy IT systems [1]. Cloud technology has also a very important impact on the financial lifecycle of a startup as it enables companies to capital infrastructure expenses into variable costs [4]. There are also some important points to consider when evaluating a cloud computing platform in place of an in-house infrastructure. Evaluating extensively advantages and disadvantages of cloud computing systems is beyond the purpose of this work. However in respect to the specific class of systems we are considering, it has to be said that immense flexibility that platforms like Amazon AWS IoT or Azure IoT offer comes at a price.

Where developers pay this price is in the limited control over the whole process and the inability to fine tune the system to their specific needs. Where the companies pay the price is in the potentially ever growing running costs that reflect the horizontal scaling of the system. As an alternative a bespoke system where all components are carefully designed and integrated, can potentially offer much better performance. The ability to tailor fine details and control over data are just a few of the reasons that *See Your Box* took into consideration when deciding to develop an IoT server architecture from the ground up in spite of the tempting aspects of Paas and Iaas services. Most of the research and work was focused on four aspects:

- Define an architecture for hot-path and cold-path data analysis
- Design a scalable private cloud infrastructure
- Distribute computational load
- Managing big-data.

See Your Box is a real-time monitoring service where the telemetry pattern sustains dynamic business logic applied to incoming data. Hot-path is used for detecting specific events that clients want to monitor while cold-path data feeds a predictive analysis machine learning system. With a goal to scale up to millions of devices transmitting data simultaneously the system must be able to scale quickly and easily. The five key characteristics of the system that influenced the architecture design are:

- Flexibility, in the *See Your Box* system two devices can be sourcing different types of data and require to encode it differently. Once received by the servers it must be processed and handled according to business logic rules customized for each client.
- Edge computing - See Your Box devices are not only sensors with a transmission module but an active re-programmable OTA smart sensing devices capable of data processing and event detecting. The system must provide a bidirectional communication means to control the devices.
- Scalability - While not subject to extremely variable and bursty traffic spikes, common for websites and social networks, the system must be able to replicate, distribute and scale over the private cloud network.
- Integration See Your Box provides APIs to allow customers to integrate their systems with its monitoring platform.
- Security Privacy laws and regulations require the company to have full control over data, especially where it is stored.

Since early stages of development it was evident that the scale of data involved and the level of flexibility required would make the datastorage the most critical part of the system. If not well engineered it would soon become the bottle neck of the system. By analyzing data flow it was also evident that different parts of the application had different requirements when accessing databases. This pointed to a strategy of combining multiple databases [14]. This aspect together with the desire of developing a scalable system brought See Your Box to design a totally modular system where each component could be fine tuned and optimized for its task.

IV. Architecture

The whole architecture is based on a fully scalable infrastructure based on virtual machines that are responsible of fulfilling specific tasks. A lot of research and effort was invested in creating an efficient self load balancing system that could use the full potential of the available hardware. This allows the system to take advantage of instant and dynamic vertical scaling driven by the actual load of the system. The resulting architecture is summarized in figure 1 The system is

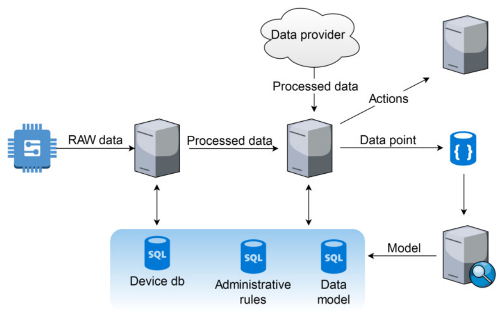

Fig. 1. See Your Box Architecture

subdivided in three main component:

- Gateway, accepts incoming requests from devices, authenticates and decrypts data, forwards data within the system and delivers messages to devices.
- Engine, devoted to analyze collected data

- Databases, to store data.

By separating responsibilities over multiple machines it is possible to scale them (horizontally or vertically) separately and selectively. Performance, however, is not the only concern that motivates such architecture. Spatial redundancy, for instance, can be achieved by replicating machines within the system for increased security and availability. Performance and behavior of a distributed system is only as good as the efficiency of the underling protocols that enable communication between its components. Traffic exchanged between the telecommunications infrastructure and the servers travels over the HTTP protocol. While many other more specialized alternatives exist (as seen with AWS IoT and Azure IoT), the simplicity of the HTTP protocol and the availability of tools that enable diagnostic and manipulation make it a valid candidate for cloud based solutions. Additionally, its popularity and widespread usage allow a simpler process of integration of the APIs developed and distributed by the company to its clients. In the following sections we will go through the different key components that define the architecture, highlighting findings and elements that led See Your Box in its design related decisions.

In the following subsection we present only the Gateway and the Databases solution used in the system. The Engine is out of the scope of this paper. It is structured in two components, the Rules Engine, that applies business logic to incoming data, providing real-time analysis for event detection and data processing and the Actions engine, that implements the event handling logic by processing actions such as sending e-mails, connecting to external APIs or producing messages to send to other devices.

A. Gateway

HTTP protocol allows the gateway to expose its services and APIs with a single protocol simplifying development and maintainability of the system. Implemented with a lightweight Python framework it can take advantage of many best practices and policies that have emerged in the last years with the rapid widespread of web applications. The gateway was developed using Flask, a Python simple yet extensible micro framework serving APIs through an nginx web server. Data is returned to the client in the form of JSON files.

When using a web application to deliver content for HTTP requests it is common practice to enclose all code to manage the request inside the same module that processes the request and provides a response. While this is an intuitive way to handle HTTP requests it does have its drawbacks. There are times when the processing of a request and the corresponding output can be decoupled. In figure 2 a device is sending data to a server that has to be processed and subsequently stored in a database. In a fully sequential synchronous approach response time to the device depends on processing time of the tasks associated to incoming data introducing a delay in the response. In figure 3 instead, by decoupling server and workers it is possible to keep short response times to the device

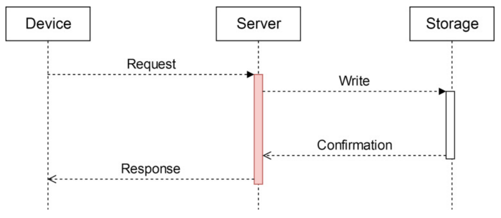

Fig. 2. Synchronized approach

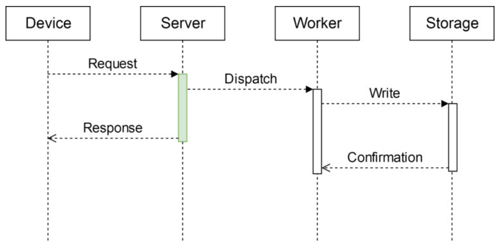

Fig. 3. Not Synchronized approach

requests while deferring heavy workloads to other actors of the system. Many are the reasons to investigate this choice:

- Scaling is possible to distribute over multiple nodes the computation related to incoming data and time consuming operations.
- Power management, usually IoT devices are battery powered and enforce heavy power management policies. A common strategy is to power on transmission related hardware when only strictly necessary. Short response times translate in smaller windows of time when transmission modules are powered on.
- Resource management The dynamic vertical scaling of the infrastructure allows to distribute resources instantly as needed by the single components, maximizing performance.

In this scenario the main question concerning this matter was what information really does the device need in the response sent by the server. If data processing in the system is viewed as single action the response usually indicate the processing status. If however, we breakdown processing into steps it becomes evident that by operating a separation of concerns the most important piece of information that must return to the device is the confirmation of successful reception of data by the gateway. What the server does to that piece of information is generally not a concern of the device. Since it is possible to let the API quickly return the outcome of gathering the incoming data. To rephrase the last concept, the main purpose of the return message is to inform the device if the transmission was successful, regardless of what happens when the system will process the data. However See Your Box is not only a pure telemetry system. Device flexibility and edge computing are only two features that clearly require bidirectional data exchange between things and the server. The design of the system calls for only outbound connections from devices, so, for instance, any data directed to a device will require an initiative of the device. To solve this issue, in asynchronous systems, we use a message box where data to be sent to the device is stored until emptied. Two options were evaluated in designing the system: internal or external message box. An internal message box is advisable in those

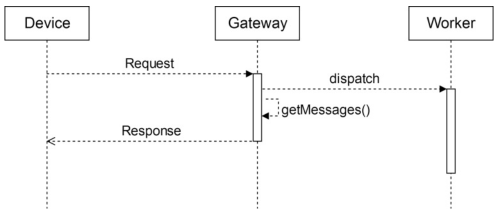

Fig. 4. Message box as a component of the gateway

situations were the content of the message box is produced from the gateway itself. Examples could be to store the status of the execution of the worker and request a re-transmission. Messages can be cached in volatile memory and reduce the overhead of having to initiate yet another transaction as show in figure 4 . Whilst using an external message box (figure 5) the gateway must foreword a request to the service that implements this function and the added latency clearly impacts the response time to the device. We opted for an external message box contained into the Device DB. By doing so the gateway only needs to query once an external service that returns both messages directed to the device and the metadata needed to authenticate and foreword the incoming data to the rules engine.

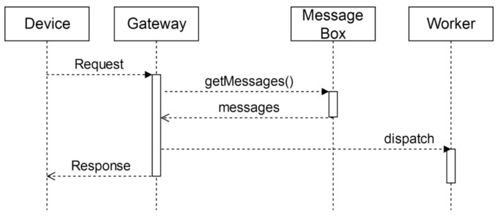

Fig. 5. Message box as a separate component

Ultimately the main operations performed by the gateway when it receives data form a device are:

1) Decode incoming data
2) Query the device DB for metadata and pending messages
3) Forward data and device metadata to the rules engine
4) Return messages to the device

The complete sequence diagram of a generic request to handle data from a device is highlighted in figure 6. Due to the limited size of data packets involved we will neglect the time necessary to perform the decode phase. This leaves most of the responsibility on the efficiency of the communication protocol (delegation to worker) and performance of the Device DB data storage.

1) Communication protocol: Splitting the execution of a task and distributing its load over multiple threads requires a form of coordination and interprocess communication. A common way to do this is by using messages queues. They offer an

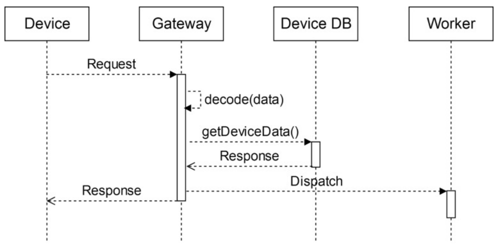

Fig. 6. Sequence diagram of the Gateway

asynchronous communication protocol, allowing senders and receivers to exchange messages without directly interacting with each other. Tasks are submitted as messages to an inbox and are eventually read and executed by a worker when ready. See Your Box Gateway uses Celery, an open source task queue based on message passing. Its architecture abstracts from the underlying communication protocol and it can be implemented with a number of different options. The main components of a task queue are [12]:

1) Messages Tasks are submitted to the queue in the form of messages. These could be binary objects, strings or JSON files.
2) Broker is the component that actually stores the messages. Acts as a middle man between producers and consumers. Examples of message brokers are Redis or RabbitMQ.
3) Producer is the portion of application generating the tasks. This could be an API endpoint that requires the execution blocking or time consuming operations.
4) Consumer Commonly referred to as a worker, it is the component that will actually execute the operations associated with the task.

There are a number of things to consider when implementing a task queue system for asynchronous processing. The first is regarding persistence of messages on disk or in memory. This is an important decision that influences directly the performance of the system. When evaluating what strategy to adopt we considered that the main reason that could motivate the adoption of a permanent storage solution is to avoid loosing messages due to an unexpected power down or crash of the system. Upon reboot the system could ideally continue executing the tasks associated to messages delivered before the event. What was discovered was that in case of unexpected crashes or hardware failures the risk of corrupting the disk under the heavy write and read load was very high. The benefit of being able to possibly recuperate messages stored in queue upon a crash was minimal compared to the potential gain in performance when implemented as a in memory message broker. Efficiency of the system depends on how fast the workers are able to process the incoming messages. To take full advantage of the scalable infrastructure it is also necessary for the application to monitor system load and performance and automatically deploy new workers within the system.

Finally, when configuring a message queue, and specifically a task queue, it is important to take into consideration ordering of task execution and completion. The broker will generally work as a FIFO (First In First Out) queue. Tasks, or messages, are delivered to the broker in temporal order and executed by any of the available workers. An alternative is to configure

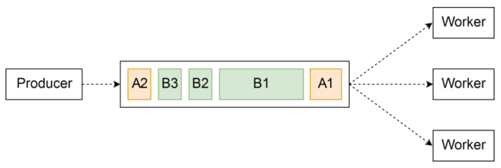

Fig. 7. Message handling order with single queue

the system with a FIFO queue for each worker. Messages are delivered in order to the broker and the corresponding worker will execute the tasks associated with the messages in the same order. There are a few aspects to consider when choosing one configuration over the other.

1) Load distribution. A single queue implementation results in a simple load balancing mechanism, where tasks are distributed to free workers as they become available. In a multi-queue configuration load balancing depends a lot on how the messages have been distributed to the broker. The producer has the added responsibility to distribute messages accordingly onto the queues. Uneven delivery would result in some workers being overloaded while others remaining in an idle state.
2) Queue distribution A single queue configuration results in a single point of failure. If the node that contains the message queue becomes unavailable due to a system crash, for instance, the whole task queue comes to a halt until a recovery strategy kicks in. Distributing the queue on more machines allows the system to contain the effects of a node failing.
3) Completion order In a queue messages, and the respective tasks, are executed in order. However no guarantee can be given on the required execution time of a single task. This could result in a queue populated by short and long tasks as shown in figure. Assuming a three worker scenario, tasks TA1, TB1 and TB2 will start executing shortly one after the other. However the smaller task TB2 could complete before TB1, effectively braking the order.

In a multiple queue scenario, where one worker is dedicated to a single queue the execution of a task necessarily follows the completion of the preceding task. In other words, when a single worker is dedicated to a single queue, delivery, execution and completion order coincide. Configuration of the message broker and how queues and workers are deployed within the system has to take into consideration load balancing, distribution and message ordering. It is important to consider carefully the specific problem to solve. In the See Your Box system message ordering was a fundamental requirement as it reduces greatly the need of storing additional data for hot path analysis. A lot of the conditions that are typically monitored in IoT systems have direct correlation to temporal evolution of variables and occurrence of events. When data is provided to the system out of order it has to be stored provisionally and subsequently retrieved and reordered for analysis. A very simple way to make queries faster is not to run them at all. Enforcing temporal order of processed messages allows the system to use incremental and cumulative analysis that is able to support almost the majority of monitoring scenarios. This resulted in multiple queues, one for each worker, and a simple yet very efficient algorithm for distributing tasks generated by a device always to the same worker.

B. Databases

As seen in the architecture overview, it was not convenient nor feasible to use a single database management system to serve data processing in the whole system. The strategy adopted was to focus on the single interactions step by step and define what were the most important requirements for each one of them. This architecture results in a data storage solution that has to cover three key components:

- Device DB. This must be a high speed and robust data store optimized for reads and updates. It contains metadata associated to the device producing incoming data and an inbox for messages to deliver in return. The faster the system can read from this source the quicker it can serve a request.
- Business DB The system must be able to serve thousands of clients, handle accounts, ACLs and financial transactions.
- Data points DB This database collects data points of processed data. To minimize processing time this database must be optimized for inserting data, however the main concern over this database is horizontal scalability and ability to manage big data.

In processing incoming data we have two goals: in the first place ensure the fastest possible response to the device and in the second place optimize processing time as a whole along both the hot path and the cold path. The approach followed was in reality very simple. The idea is to define clearly hot and cold paths and break them up into stages. For each stage define what the critical component was and optimize it. As a rule of thumb the quicker both hot and cold paths are traversed by incoming data, the smaller the fraction of shared resource for time unit is necessary to process a request. Smaller fractions will result in a smaller infrastructure that allows the company to optimize costs. Quick analysis on prototype architectures revealed that the bottlenecks were database interactions. It was clear that it was necessary to optimize reads and updates in the hot path and writes in the cold path. Sporadic writes in the hot path and offline reads in the cold path were not to be taken into account in the optimization phase and choice of solutions.

Business DB is used for storing crucial data such as accounts, customer's options and financial data. This database is generally not directly involved in data processing during either hot-path or cold-path. During the lifetime of an active device generally the system will need to interact with this database only during power-on, poweroff and special maintenance procedures. For this reason its impact on the overall performance is limited. The main requirement for this data storage are support for transactions and consistency. For this reason, and because data contained exhibits strong relations it was decided to use a relational database management system. Research on related work pointed to two possible candidates, PostgreSQL and MySQL. While the former has an extensive and powerful feature set and PostgreSQL was already used inside the location service of the company, its complexity limited the obtainable performance. MySQL on the other hand was a proven database with which the team had significant experience. A concern was raised regarding the costs of licenses that can have a high impact on yearly operational costs of the system.

Despite the support and service provided as benefit of the annual subscriptions, once again similarly to what happened when evaluating cloud providers it was necessary to consider alternative solutions. In 2009 before the acquisition of MySQL, an open source fork of the original project was released under the name MariaDB.

The deviceDB contains metadata related to the device and messages that need to be sent to the device. This storage must be optimized for read and update speed. The DeviceDB has a crucial role in the system and its performance influences mostly processing time of incoming data. Following the general architecture of the system, this is the component that required most attention from the R&D department of the company. The DeviceDB contains two important components that are necessary for handling incoming data. These are the message box for data to be returned to the device and the metadata associated to it. The latter is used by the rules engine to know how to interpret data, actions to be performed on it and state of the device. The requirements of this storage:

- Flexibility - Metadata related to a device can vary a lot. Smart sensing devices can monitor a large number of parameters with different encoding schemes. It must be possible to embrace this difference and not be limited by a fixed scheme.
- Performance As previously noted, read operations must be extremely fast in order to obtain low response times. Write performance is less crucial since this would happen with a low frequency.

The first database management system that went under examination for this task was MongoDB. The main reason was the required flexibility of the data structure used to describe metadata. Repeated tests demonstrated however that it's performance wasn't up on par with the high speed key-value NoSQL database or MariaDB tables powered by a TokuDB engine. Redis was taken into consideration due to the fact that it minimizes disk access by keeping the database in memory. This is a problem for scalability since the size of the database is limited by the quantity of available RAM. Furthermore some form of persistance on disk needed to be provided, since the stored data isn't short lived. Additionally it was found that read performance wasn't very different from an optimized MySQL/MariaDB database for comparable queries. Similar results were confirmed in literature [18]. These findings quickly made us discard Redis as a possible candidate. Ultimately one of the company policies was to keep the set of adopted technologies as narrow as possible in order to favor interoperability of expertise of the team. Ultimately this led us to explore what we defined as a hybrid solution that was to use a SQL database as a key-value storage and use a text field to store a JSON files representing metadata and message inbox. Each row would represent a device and would be indexed upon the device id. This unorthodox approach to data management proved over time one of the most valuable decisions in the design of data storage support to the system. Performance wise we were achieving read and update speeds comparable to the top class key-value memory based data storages and flexibility was on the same level of the NoSQL databases thanks to the adoption of JSON objects. However, the most important benefit was that, while braking some of the ACID properties for the data contained inside the JSON fields, these were guaranteed for the other fields. This allowed us to integrate the DeviceDB tables with the BusinessDB, enforcing all consistency benefits of a traditional SQL RDBMS. Not being able to query single fields of JSON files such as in MongoDB was not a problem since each incoming packet would require all the data contained inside the row and never a part of it.

Once incoming data has traversed all the processing path in the system and has been augmented by external data and real time manipulation it must be stored in a database management system for offline analysis and visualization. Write operations at this stage are very well defined if not unique. Conceptually the only storing procedure that is necessary is saving a data point. This piece of information is essentially a collection of sensor readings and location photographed at a given moment in time. However the structure of a data point is extremely dynamic and heterogeneous.

Evidence in literature coupled with advice provided by IT consulting companies pointed in the direction of a document based NoSQL database, particularly MongoDB. The widely recognized features of this database were soon confirmed in the prototyping stage of the architecture. The document based nature of MongoDB allowed for a simplification of data representation across the system. It uses BSON, a binary representation of JSON files. The latter was the format under which data was managed across the system and particularly fed through the customer accessed APIs. While it might appear as a trivial detail, it allowed for a more compact code base that would reduce the abstraction and translation layers across systems. For a developer a data point is created, manipulated, stored and finally returned to the client API in the same format: a simple semi structured JSON file. This allows for much faster integration, debugging and analysis of the system, particularly data flow. Ultimately, but most importantly, it was the support for massive scale dataset that confirmed MongoDB as the key solution. Support for auto sharding and distribution reduced the need of designing a complex mechanism for horizontal scaling of the system. The only true challenge that was encountered when developing this component was the interference of the read and write operations. Sudden slowdowns and reduction in performance was experienced when these two operations would happen at the same time. The solution was to implement a semaphore system that would lock writes when a read operation was performed.

V. Conclusions

The IoT industry has experienced an exponential growth in the last years. It has been pushing the boundaries of conventional architectures by challenging developers with massive quantities of data to be processed with near real time requirements. Scale, heterogeneity and velocity of data have an immense impact on the system design. We analyzed how two of the major cloud computing providers tackled the challenges of the IoT in their comprehensive service suites. With a strong focus on modularity, composability and horizontal scaling they both offer valuable solutions for an array of scenarios. The commodity of a turn-key cloud based platform comes at a cost that could potentially grow out of control, impacting the finances of a company quite heavily. Costs don't always grow linearly with the scale of the system due to the nature of some computational operations that are performed on data or on pricing model.

Cloud based platforms enable startups to quickly penetrate the market. However, for young companies it is not only a matter of balancing NRE and operating costs. Turn-key solutions like Paas and Iaas allow startups to focus on building a team with skills closer to the business core technology and penetrate the market faster and more effectively. Ultimately deciding for a cloud based solution or an in-house one requires balancing interests from different points of view that are not strictly IT related. According to the specific application scenario a bespoke system with a custom architecture, despite a significantly higher NRE can represent a better solution.

The proposed architecture is a brokered task queue system distributed over a private cloud infrastructure. Incoming messages from devices are paired with metadata stored in a hybrid SQL data storage that combines the flexibility of NoSQL key-value or document based DBMSs and reliability and ACID compliance of an SQL traditional relational database management system.

Ultimately the designed system, presented in this work, has been implemented and released onto the market. After 12 months and over 1 million data points collected, the system has proven to be stable and meet the preset requirements, enabling the company to expand its business and acquire new clients.

References

[1] A115. How cloud-powered FinTech start-ups are disrupting the banks. 2016. http://a115.co.uk/publications/awsfintech-startups.html.

[2] Inc. Aerospike. What is a Key-Value Store? 2016. http://www.aerospike.com/what-is-a-key-value-store/

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: Computer Networks 54.15 (2010), pp. 2787 -2805. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2010.05.010. http://www.sciencedirect.com/science/article/pii/S1389128610001568

[4] Amazon AWS. About Us. 2016. https://aws.amazon.com/about-aws/

[5] Amazon AWS. What Is AWS IoT? 2016. http://docs.aws.amazon.com/iot/latest/developerguide/whatis-aws-iot.html

[6] Galip Aydin, Ibrahim Riza Hallac, and Betul Karakus. "Architecture and Implementation of a Scalable Sensor Data Storage and Analysis System Using Cloud Computing and Big Data Technologies". In: Journal of Sensors 2015 (2015), p. 11. URL: 10.1155/2015/834217.

[7] V. Carchiolo at Al. "Users' attachment in trust networks: reputation vs. effort". In International Journal of Bio-Inspired Computation, 2013, pp. 199–209, ISSN: 1758-0366. DOI: 10.1504/IJBIC.2013.055450

[8] A. Chianese, F. Piccialli, and G. Riccio. "SMuNe: A Smart Multi-sensor Network Based on Embedded Systems in IoT Environment". In: 2015 11th International Conference on Signal-Image Technology Internet-Based Systems (SITIS). 2015, pp. 841-848. DOI: 10.1109/SITIS.2015.51.

[9] CompareBusinessProducts.com. Top 10 Largest Databases in the World. http://www.comparebusinessproducts.com/fyi/10-largest-databases-in-the-world

[10] Microsoft Corporation. Microsoft Azure IoT Reference Architecture. 2016.

[11] DB-Engines. DB-Engines Ranking. 2016. http://db-engines.com/en/ranking

[12] Bryan Helmig. Why Task Queues - ComoRichWeb. 2012. http://www.slideshare.net/bryanhelmig/task-queuescomorichweb-12962619.

[13] Marc Jadoul. How Big is the Internet of Things? 2016. http://www.business2community.com/business-innovation/big-internet-things-01593563

[14] L. Jiang et al. "An IoT-Oriented Data Storage Framework in Cloud Computing Platform". In: IEEE Transactions on Industrial Informatics 10.2 (2014), pp. 1443-1451. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2306384.

[15] J. Jin Kang et al. "Predictive data mining for Converged Internet of Things: A Mobile Health perspective". In: Telecommunication Networks and Applications Conference (ITNAC), 2015 International. 2015, pp. 5-10. DOI: 10.1109/ATNAC.2015. 7366781.

[16] T. Li et al. "A Storage Solution for Massive IoT Data Based on NoSQL". In: Green Computing and Communications (GreenCom), 2012 IEEE International Conference on. 2012, pp. 50-57. DOI: 10. 1109/Green-Com.2012.18.

[17] DigitalOceanTM Inc. O.S. Tezer. SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. 2014.

[18] C. Perera et al. "Context Aware Computing for The Internet of Things: A Survey". In: IEEE Communications Surveys Tutorials 16.1 (2014), pp. 414-454. ISSN: 1553-877X. DOI: 10.1109/ SURV.2013.042313.00197.

[19] T. A. M. Phan, J. K. Nurminen, and M. Di Francesco. "Cloud Databases for Internet-of-Things Data". In: Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical BIBLIOGRAPHY 53 and Social Computing(CPSCom), IEEE. 2014, pp. 117-124. DOI: 10.1109/iThings.2014.26.

[20] Evangelos Psomakelis et al. "Big IoT and social networking data for smart cities: Algorithmic improvements on Big Data Analysis in the context of RADICAL city applications". In: CoRR abs/1607.00509 (2016). http://arxiv.org/abs/1607.00509.

[21] Redis. Redis Documentation. 2016. http://redis.io/

[22] C. Rommel at al.. Amazon AWS & Microsoft Azure IoT Deep Dive. 2016.

[23] Bryce Merkl Sasaki. Graph Databases for Beginners: ACID vs. BASE Explained. 2015. https://neo4j.com/blog/acidvs-base-consistency-models-explained/

[24] W. Shi and M. Liu. "Tactics of handling data in Internet of things". In: 2011 IEEE International Conference on Cloud Computing and Intelligence Systems. 2011, pp. 515-517. DOI: 10.1109/ CCIS.2011.6045121.

[25] F. Xhafa et al. "A Software Chain Approach to Big Data Stream Processing and Analytics". In: Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on. 2015, pp. 179-186. DOI: 10.1109/CISIS.2015.24