Unified IoT Platform Architecture

Platforms as Major IoT Building Blocks

Dejan Mijić, Member IEEE
Faculty of Technical Sciences, University of Novi Sad
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
e-mail: dmijic@acm.org

Ervin Varga, Senior Member IEEE
Faculty of Technical Sciences, University of Novi Sad
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
e-mails: evarga@uns.ac.rs, e.varga@ieee.org

***Abstract*—Ad-hoc approaches to building Internet of Things (IoT) solutions in the past had mostly failed. The sheer complexity of managing a mesh of distributed interconnected devices was unbearable. The appearance of IoT platforms has enabled the proliferation of successful products in this area. A full-scale, comprehensive IoT platform embraces the following three major capabilities: application enablement, data storage, and connectivity management. This paper presents an overview of technologies and techniques to realize such an IoT platform. The focus is on the platform that unifies edge and cloud computing. The paper refers to Mainflux as such reference example. Mainflux serves as a software infrastructure, that is built around the micro-services architectural style, for development of IoT solutions, and deployment of intelligent products. It is conceived as an open-source project, and is already used in production settings. Therefore, the paper describes a real product, whose chosen set of methods, technologies and techniques are feasible and effective in crafting IoT systems. The Mainflux project contains an extensive set of automated tests, that could serve as an additional documentation regarding its usage.**

Keywords—Cloud Computing; Edge Computing; Internet of Things; Micro-services; Platform; Software Architecture

I. Introduction

Data-driven business solutions are gaining attention in the industry. Internet of Things, as an embodiment of the data-driven business paradigm [1-3], is becoming very popular in various domains (for example, smart grid [4], telecommunications, robot networks [5], etc.). Smart devices need to efficiently communicate, exchange data and synchronize actions to implement high quality services demanded by today's standards. The core idea is to leverage structured and ad-hoc mesh device networks to deliver functions that are greater by scope than the sum of its constituent parts (when each device's capability is taken in isolation). To achieve this vision devices should be managed by an IoT platform. The platform serves the role of a middle-ware for custom applications. Applications may be focused on the essential properties of higher order services, rather than to repeat mundane device handling logic. This boosts the overall dependability of a software solution, as device handling is quite a complex realm, and requires a lot of effort to implement correctly. Finally, applications may be easily integrated by exchanging business instead of raw device data.

As more and more appliances are driven by software, software engineers will need to know how to support them. Part of the support is to work on or use sophisticated IoT platforms to provide higher level services in various application domains. This paper will equip the reader with necessary insight how such an IoT platform is implemented. To make the discussion as much pragmatic as possible we will deep dive into Mainflux, as a concrete production ready implementation of the data-driven business idea. The authors are part of the team that has developed Mainflux, so we have first hands experience with this product. Mainflux is freely available at www.mainflux.com.

Mainflux embodies the following features inside the same product (only 14% of existent IoT platforms possess these characteristics [6]), that delivers agility, interoperability and sustainability on device as well as platform level:

- Application enablement
- Connectivity management
- Data aggregation and storage

Mainflux's novelty is the focus on the concept of a unified IoT platform, i.e., the platform that supports regular and constrained devices using the same code base. Moreover, it has a remarkably small footprint, and can be run on a low capacity computer system. This paper describes an IoT platform from this innovative viewpoint. This aspect is further boosted by the set of advanced technologies leveraged by the system.

II. Methodology

Bass et al. define the software architecture as the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both [7]. Clemens and Kazman states that the design of the software architecture should be performed systematically, considering all the relevant aspects of the system required to satisfy the architectural drivers in a cost-effective and repeatable way [8]. The same authors emphasize the attribute-driven design (ADD) approach to software architecture design.

In general, ADD is an iterative method for designing software architecture, having the following lifecycle [7]:

1. Choose a part of the system to design.
2. Identify the architecturally-significant requirements for that part.
3. Produce a design for the chosen element and verify it.
4. Choose an input for the next iteration.
5. Repeat steps 1-4 until all the requirements have been addressed.

978-1-5386-5928-1/18/$31.00 ©2018 IEEE

As the result of applying the ADD, a set of various architectural views will be produced. These artefacts will be used throughout the implementation.

The following subsections discuss the architectural drivers identified through the stakeholder sessions that will serve as an input to the design algorithm.

A. Functional Requirements

Functional requirements define what the system must do, and how it must react given runtime stimuli [7]. Fig. 1 shows all the actors present in the platform and use cases associated to them.

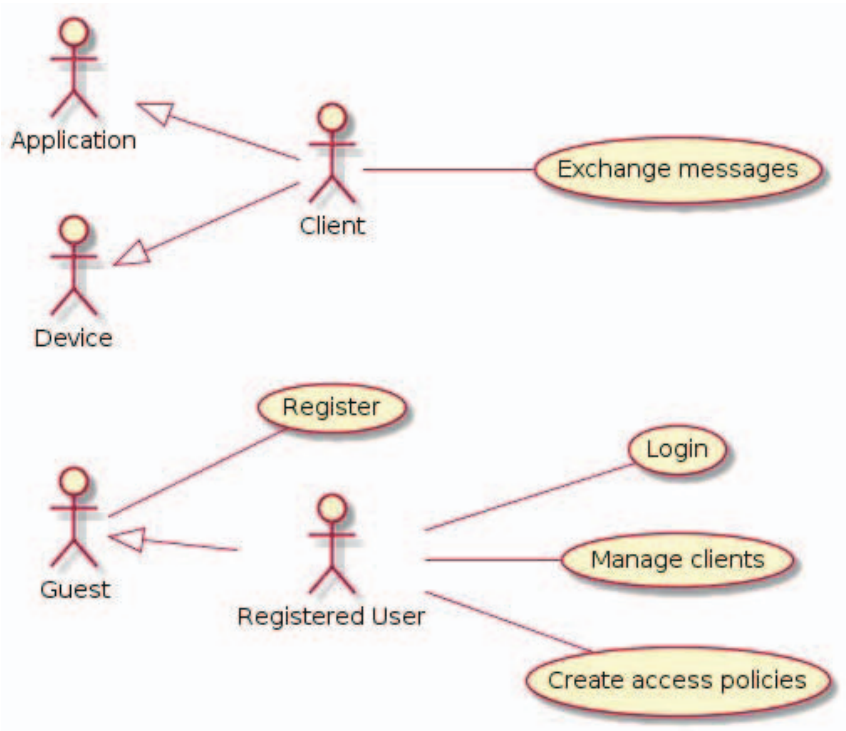

Fig. 1. Use-case diagram enrolling the main actors and their goals.

Guests are all persons and organizations without a registered account. The only thing they can do is register an account. During registration, guests are required to provide their unique platform identifier (i.e. an e-mail address) and an arbitrary password. Other meta-data (i.e. name, address etc.) is optional at this point. Once their accounts are activated, guests become users.

Users are all persons and organizations that have an account registered at the platform. Prior to accessing their resources, users must authenticate to the platform by providing the credentials set during registration. Once logged in, users can manage the clients they own. Client management assumes the ability to create, provision and retrieve a client, update its meta-data or delete it.

Besides managing the clients, users can specify the access policies. While the platform must ensure complete isolation of resources between users, the users must be able to create isolation layers between the resources they own. Note that the term resource relates both to the clients and the data they produce.

Clients are all applications (both 3rd party and internal) and devices that were created or provisioned (i.e. assigned the access key) by some user. Each client can participate in message exchange, assuming it possess a sufficient permission level.

B. Quality Attributes

Quality attributes are qualifications of the functional requirements, and the best way to represent them is via quality attribute scenarios – a testable, falsifiable hypotheses about the quality attribute behavior of the system under consideration [7]. The following attributes were identified as mandatory: performance, scalability, availability, security, connectivity, and deployability.

Performance is concerned with the capability of the system to meet any imposed temporal requirements. The following scenarios were identified:

- During normal regime, the platform must be able to process and store up to 40000 messages per second.
- During peak loads, the platform must be able to process each message and deliver it to the “recipient” in under 1 second.

Scalability is closely related to performance. It is defined as the ability of the system to handle an increased amount of work, or its potential to be enlarged to accommodate that growth [9]. It is up to the platform to ensure the following:

- Raw message data will be persisted for two days.
- Aggregated message data will be persisted for six months.
- There must be no data-loss in user's and client's meta-data.

To achieve the requirements, the platform must be horizontally scalable, i.e. it must be built in a way that guarantees that adding more processing instances (e.g. service instances, database nodes etc.) will help it meet the demands.

The CAP theorem states that in a distributed environment, one cannot sacrifice partition tolerance. Instead, the choice must be made between consistency and availability [10]. Availability is the property of the system that guarantees each request will receive a non-error response. Having said that, it is required of the platform to be available 99.99% of the time.

Security is one of the most important quality attributes. The following security-related scenarios were identified:

- Only requests containing valid authentication credentials should be accepted.
- The platform must ensure that the request initiator is authorized to perform the underlying action.
- Users can access only the resources they own.
- Users must be able to define access rules between the resources they own (i.e. application A can send message to device B).

Connectivity is the ability of the system to handle requests sent via different messaging protocols. The following protocol adapters must be provided: HTTP, Websockets, MQTT, and CoAP.

Deployability is the ability of the system to be deployed into various runtime environment without losing its functionalities. The platform must be able to function properly both in cloud (e.g. Amazon Web Services), as well as in constrained environments (e.g. Raspberry Pi).

Mainflux's unique value proposition is the concept of a *unified IoT platform*, that occasions further architecturally significant requirements. The idea is to have a single configurable code base, that can run both on constrained gateways as well as in the cloud on powerful machines. This is a pragmatic direction proven in practice. Setting up a cloud environment isn't that easy (especially from the viewpoint of networking). It is many times more convenient to jus ship a preinstalled gateway, which may detect devices in a plug-and-play fashion. Such a gateway would run a full blown IoT platform as Mainflux.

The desire to support both edge and cloud computing entails balancing opposing forces. A gateway expects the following characteristics from an IoT platform: low memory footprint, modest performance, and low latency. For the cloud, you demand the next attributes: huge number of simultaneously connected devices, high throughput (number of transactions/writes per second), and scalability. There is a limited set of technologies, which comply with the above list. Most existent IoT platforms are tuned to support either the edge or the cloud.

C. Constraints

The only architectural constraint is that the platform will be implemented using only the available open source technologies due to the licensing and cost reasons.

D. Evaluation Criteria for IoT Platforms

Besides the previously enumerated quality attributes, there should be well-defined IoT platform evaluation criteria comprised from a set of properties. This can serve as a good foundation to compare various IoT platforms ([11] contains a detailed comparative analysis of AWS IoT, Microsoft Azure IoT and Mainflux written by the authors of this paper). The evaluation properties are enrolled below:

Device bindings specifies the supported communication protocols of an IoT platform regarding devices (for example, HTTP, MQTT, etc.).

Analytics elaborates about the type of technology used to perform analytical calculations. Analytics is the mechanism to synthetize higher-level knowledge from raw data. Rules may be associated with such derived values.

Visualization defines the offered technologies for data visualization (for example, an HTML5-based UI dashboard).

Rules engine and alarming elaborates how rules and alarms are defined on top of accumulated data. The ability to easily customize conditions to trigger alarms, and rules to associate actions with events, is of utmost significance. Hardcoding these (or using similarly rigid constructs) isn't an option. Domain-specific languages are an attractive choice here.

Security is the set of supported security technologies by an IoT platform. This cover securing device–platform, platform–platform, and platform–application communication channels, controlling access to data via authentication/authorization, and so on.

License defines the type of license attached to the IoT platform (like Apache 2.0), and whether an IoT platform is freely available or not.

Deployment technology describes the applied deployment technology (for example, Web archive, Docker image, operating system dependent package manager file, etc.).

Auto-scaling enrolls the set of technologies and techniques that allows scaling of an IoT platform. This aspect is tightly related to availability. If an internal service or component becomes unresponsive, then such a condition must be auto-detected and acted upon (typically by summoning a new instance to preserve the desired capacity). Obviously, keeping the determined number of services up also improves availability.

Device data persistence defines what database technologies (or other data storage mechanisms) are used by an IoT platform to store raw device data.

Management database defines what database technologies (or other data storage mechanisms) are used by an IoT platform to store management-related data, including derived values.

Implementation language specifies the main implementation programming language for an IoT platform (for example, Go, Java, etc.).

Data model describes the event (message) format for communicating with devices (like Sensor Markup Language).

III. Solution

Mainflux is comprised from multiple micro-services – small, autonomous services, focused on one domain aspect [12]. The following subsections describe each of the services, as well as the technologies used to implement them.

A. Technology Stack

All of the services were developed using the Go programming language. Go is a fast, statically typed, compiled language with an expressive, concise syntax, built with concurrency in mind [13]. Combined with great community, and small learning curve, these characteristics make Go an ideal language for developing scalable, high-performing micro-services.

Among multiple competitors, Apcera's NATS was chosen as platform's messaging system. Even though it is not as mature as its competition, its small footprint, high-performance, and great client library made it perfectly compliant to Mainflux requirements.

According to the research results by Gartner and Cisco, between 20 and 50 billion connected devices are expected by 2020. [14, 15]. With that in mind, the choice of database becomes one of the most important aspects in design of IoT platforms. Apache Cassandra is distributed, decentralized, elastically scalable, highly available, fault-tolerant database with tuneable consistency [16]. Due to the aforementioned characteristics, as well as its highly-optimized writes and "proven track" in data intensive environments, Apache Cassandra has been chosen as the platform's primary data storage solution, i.e. its source of truth. Mainflux also comes in a flavor with ScyllaDB, which is implemented in C++ and may be compiled on an ARM. It has an optimized write mechanism supporting high throughput, as well.

Apache Spark is a general-purpose cluster computing platform [17]. Combination of Spark's streaming library for handling stream of messages generated by various clients, with machine learning algorithms present in its ML library, Mainflux provides a "smart" alerting capabilities to its users.

B. Services

The following subsections deals with the scope of individual services as presented in Fig. 2.

The Manager service exposes a public REST API for registering user accounts, creating, retrieving, updating and deleting its clients, as well as for managing inter-client access policies.

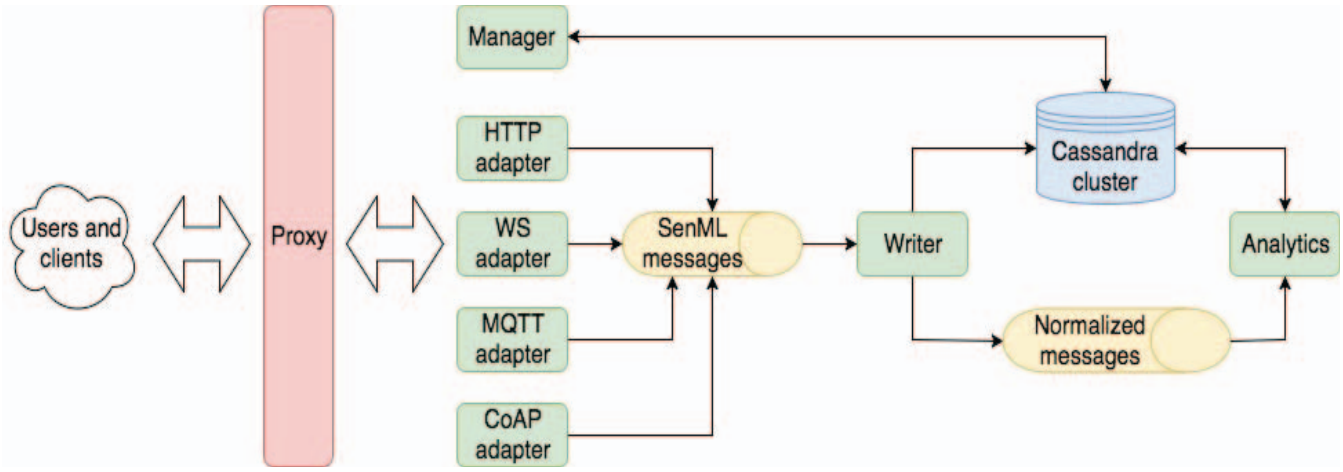

Fig. 2. Data flow diagram depicting the movement, processing, and storage of data. Data is read out by a Message Reader component, which is not separately depicted here. Fig. 4 also highlights some aspects of data communication.

Fig. 3 shows the relationships between user, client and channel. While the former two entities represent the actors specified in functional requirements, the latter is introduced to enable the access policy management. What this means is that any client connected to some channel can exchange messages with all other clients connected to the same channel.

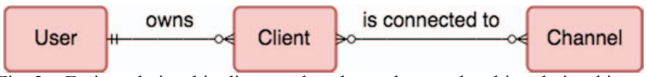

Fig. 3. Entity-relationship diagram that shows the membership relationships.

Since all the exposed resources are protected, a proper authorization level must be obtained prior to any of the available operations offered by the service.

The Auth service acts as platform authentication provider. It exposes an API for generating and verifying the access credentials. Due to the simplicity and wide-spread language support, it was decided to use JSON Web Tokens to represent the access credentials. As of latest release, the service API is exposed through the manager service.

The Protocol adapters platform layer is comprised of multiple services providing the protocol-specific message publishing API. The following protocols are supported:

- HTTP is a stateless application level protocol for distributed, collaborative, hypertext information systems [18].
- WebSocket enables two-way communication between a client and server over a single TCP connection. The primary goal of this protocol is to enable browser-based clients to establish full-duplex connection to the server without opening multiple HTTP connections [19].
- Message Queuing Telemetry Transport is light weight, open, simple, easy to implement, client-server publish-subscribe messaging transport protocol [20]. These characteristics makes it ideal for use in constrained environments, as well as in the machine-to-machine and IoT communication.
- Constrained Application Protocol is a specialized web transfer protocol for use with constrained nodes and low-power, lossy networks [21]. It is designed for machine-to-machine applications.

Clients are obliged to publish messages in Sensor Markup Language [22] (SenML), represented either as JSON or CBOR. Because publishing require access to protected resources, the clients must ensure that each request contains valid access credentials, otherwise they will be rejected. Once a message is accepted, protocol adapters will append additional meta data to it (i.e. publisher's ID, protocol etc.), and publish it to the appropriate NATS topic for post-processing.

The Writer service consumes events originated at protocol adapters, performs SenML message normalization, stores them into the Cassandra cluster, and publishes them to the another NATS topic for data analytics.

This Analytics subsystem is comprised of multiple Apache Spark jobs for stream processing and smart alerting. It has two inputs: the stream generated by writer service, and historical data persisted in the database. Historical data is used to train the anomaly detection models, that are applied on data consumed from the stream. In case that any anomalies are detected, they will be reported to the owner of the entities that participate in erroneous message exchange.

C. *Architecture*

The unification approach is visible on the following diagrams (see Fig. 4 and 5), where a single point of control and data processing is realized for the following categories of equipment:

- Constrained devices are small devices with limited CPU, memory, and power resources, that can form a network, thus becoming "constrained nodes" in that network [23].
- Edge is a distributed, decentralized device interconnection and data processing facility. Through the edge computing paradigm, disparate IoT platforms comprise a unified system of systems.
- LoRaWAN is a Low-Power Wide-Area Network specification intended for wireless battery operated things in a regional, national or global network. LoRaWAN targets key requirements of IoT such as secure bi-directional communication, mobility and localization services [24].

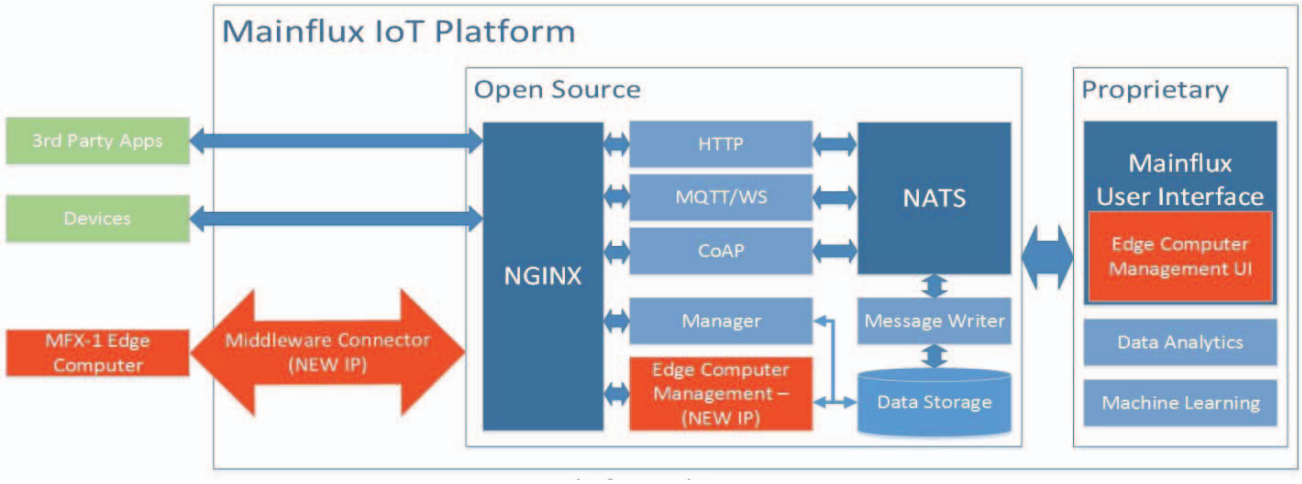

Fig. 4. Module decomposition view showing the structure of the platform.

The services encompassed by Mainflux are as follows:

- Messaging bridge (HTTP Sender, MQTT and WS, CoAP) relays messages between devices and applications.
- System manager (Manager).
- Device manager (Device) accepts device connections on southbound interface.
- Application manager (App) accepts application connections on northbound interface.

- User manager (Mainflux UI, Mainflux CLI) provides user management for the applications.
- Time-series storage engine (Message Reader, Message Writer, Cassandra) stores and queries measurements data points in the time-series format.
- Complex Event Processing (uses the system event bus NATS) consumes the incoming time-series streams and can automatize triggers and actions based on a configurable set of rules.

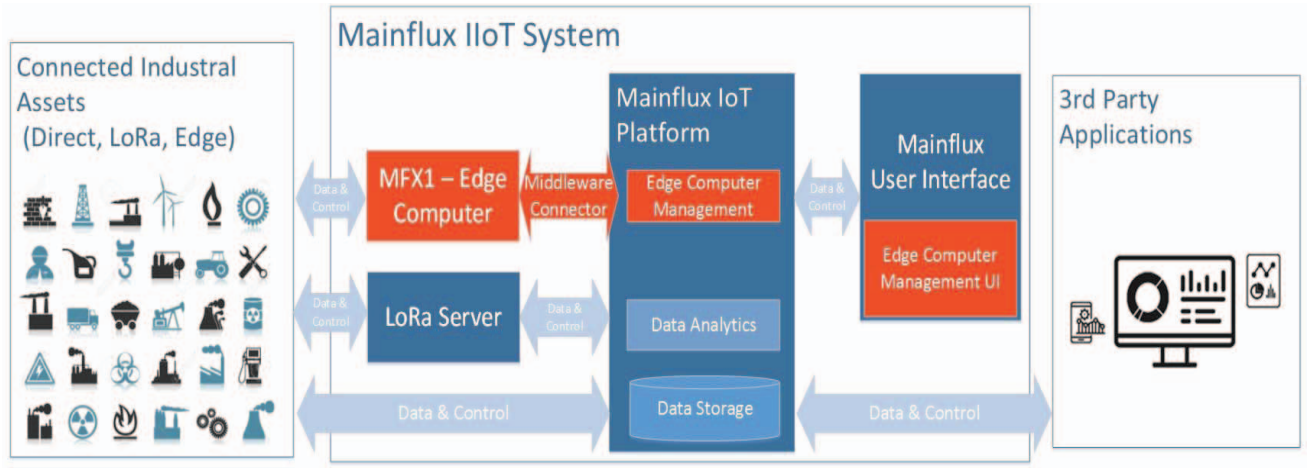

Fig. 5. Deployment view presenting the allocation of elements to processing nodes.

System Manager

In order to explain the role of the System Manager in more detail, it is necessary to present entities that this service creates and manages [11].

Users consist of the accounts of the people (administrator, applica- tion creators, and developers) who will use Mainflux platform. A user is an owner of other entities, so it must be created first. As an owner, a user gets sufficient permissions to manage devices, applications, and channels under its control.

Clients are a structure that is used to represent an actual client of the Mainflux server. There are two types of clients that connect to Mainflux: devices and applications.

Devices are usually constrained microcontrollers with a limited power budget. Because of constrained memory (several KB of RAM and ROM) they usually cannot hold heavyweight protocol stacks in their firmware. That's where MQTT and CoAP come into play, as they are lightweight protocols designed for M2M communications. Also, on the security side, TLS can be heavyweight. CoAP and LwM2M propose usage of the DTLS instead. Mainflux exposes MQTT and CoAP APIs to connect devices via these protocols.

If the devices are more powerful machines (like, for example, Rasp- berry Pi), they can be connected via standard HTTP or WebSocket protocols, as this is usually the case for web applications.

Applications are run on the servers, and Mainflux developers can create an application similar to the Twitter or Facebook application; this app will use Mainflux in the background.

Applications connect often via HTTP REST or WebSockets, as this is a fast and bidirectional socket, so updates can be pushed from Mainflux (server) to the application (client) in the RT. Nevertheless, applications can choose any of the other protocols for connections (MQTT or CoAP).

Channels are used to model a communication channel. A channel structure is a generic bidirectional message stream representation. Channels are like MQTT topics; several devices or applications can subscribe or publish on a channel. All the values that flow through channels are persisted in the database.

Device Manager holds the internal representation (data structure model) of every device provisioned in the system. This model represents the current state of the device.

Each device that connects to Mainflux thus has its own model (record, structure) in the internal Mainflux database (device registry) that is regularly updated by the device. So, when a device changes internal (physical) state it sends an UPDATE request to Mainflux, and then Mainflux changes its model (record, structure) in its internal database. As a consequence, when the application (or some other device) now queries the database, it finds a new (updated) device model, and this way information (message) is relayed from device to application (and vice versa).

Bearing in mind that Mainflux has all these internal device models and it keeps devices connected to it, we can use Mainflux to manage devices, i.e., to obtain various types of device management information, like:

- How many devices are connected?
- Where are they located?
- What is the firmware version?
- What is the battery status?
- What is the serial number of each device?

We can also use Mainflux to send various commands to devices as a part of the device management process:

• Enable/disable device

• Push firmware updates

• Group devices or change access privileges

• And more

User and application manager support the realizaton of a multiuser and multitenant application management platform. As such, Mainflux developers can create new multiuser applications on top of Mainflux without the need to handle their end users themselves, i.e., Mainflux handles user management for these applications.

In this sense, Mainflux is similar to Twitter or Facebook; creating a Mainflux application should be a process similar to creating a Twit- ter application. For example:

- User creates developer account on Mainflux.
- User creates new application using Mainflux public API.
- User develops his application that calls Mainflux public API for user creation, messaging, management, etc.

The main point here is that this "Mainflux application" can itself be multiuser. So, Mainflux is a multiuser platform (it can have many user accounts, some of which are of type developers), but it can host multiuser applications themselves.

PACKAGING AND DEPLOYMENT

The primary way of platform delivery is by shipping the services as Docker images. At this moment, the cumulative size of service images has been reduced to slightly below 30MB. Combined with infrastructure images (e.g. Apache Cassandra, NATS), the whole platform requires at most 400 MB of disk space, making it usable at any computer system capable of running the Docker engine.

However, the resource demands can be even less by replacing images with platform-specific executables. Building such executables is directly supported by Go's standard toolchain. Besides dropping the container engine, the platform's footprint can be further reduced by decreasing the number of nodes in the database and analytic cluster, their hardware requirements, or even dropping the analytic cluster completely, which should be performed with caution. Without analytics, the platform's power diminishes, as it turns into a bare messaging hub.

IV. CONCLUSION

This paper has presented the way to design and implement a scalable, performant and minimalistic unified IoT platform, using Mainflux as an example. The following features of the system have to be underlined:

• Protocol bridging (i.e., HTTP, MQTT, WebSocket, CoAP)

• Device management and provisioning

• Linearly scalable data storage

• Fine-grained access control

• Platform logging and instrumentation support

• Container-based deployment using Docker

These features are indispensable in building "vertical solutions," end-to-end applications for smart city, smart agriculture, health, transportation, and many other fields where industrial context is important and demand for quality, testing, and system robustness is of primary importance (see also Table 1).

TABLE I. EVALUATION CHARACTERISTICS OF MAINFLUX.

Trait Name	Trait Value
Device bindings	HTTP, MQTT, WebSockets CoAP
Analytics	Apache Spark
Visualization	HTML5
Rules engine and alarming	N/A (work in progress)
Security	JSON Web Tokens, ACLs, TLS
License	Apache 2.0
Deployment technology	Kubernetes, Docker
Auto-scaling	Microservices architectural style
Device data persistence	Apache Cassandra
Management database	Apache Cassandra
Implementation language	Go
Data model	SenML

With the possibility to be deployed both in the cloud and on premise, scaling from gateway to full-blown clusters, Mainflux has a lot to offer when it comes to building modern industrial IoT systems.

IoT platforms may be successfully leveraged in smart grids, which is a primary territory for applying smart devices to better monitor and control the network. An ensemble of interconnected IoT platforms may be wrapped with a higher-level API to serve as an entry point for applications [25]. This is a good example why/how IoT platforms may be used as principal building blocks in IoT.

Further development will focus on two aspects: improving the anomaly detection algorithms, and replacing JSON Web Tokens as primary authentication scheme with solution(s) more suitable for constrained devices. We plan to further enhance the platform by bringing in solutions from the realm of Web of Things [26]. The focus will be on device/application integration/interoperation issues as well as conveying semantic information between parties.

We also plan to perform extensive benchmarks to showcase the viability of Mainflux as a unified IoT platform. Moreover, we will develop an extensive rules handling library (some work has already been done).

REFERENCES

[1] daCosta, F., Rethinking the Internet of Things: A Scalable Approach to Connecting Everything, Apress, 2013

[2] Georgakopoulos, D., Jayaraman, P. P., "Internet of things: from internet scale sensing to smart services," Computing, 2016, 98, (10), pp. 1041-1058

[3] Lin, J., Yuy, W., Zhangz, N., Yang, X., Zhangx, H., Zhao, W., "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications," IEEE Internet of Things Journal, 2017, PP, (99)

[4] Bayindir, R., Hossain, E., Kabalci, E., Perez, R., "A Comprehensive Study on Microgrid Technology," International Journal of Renewable Energy Research, 2014, 4, (4), pp. 1094-1107

[5] Pinciroli, C., Beltrame, G., "Swarm-Oriented Programming of Distributed Robot Networks," IEEE Computer, 2016, 49, (12), 32-41

[6] "Who Will Win the IoT Platform Wars?," https://www.bcg.com/fr-fr/publications/2017/technology-industries-technology-digital-who-will-win-the-iot-platform-wars.aspx, Accessed 29 September 2017

[7] Bass, L., Clemens, P., and Kazman, R., Software Architecture in Practice, Addison-Wesley Professional, 3rd edn. 2012

[8] Cervantes, H., Kazman, R., Designing Software Architectures: A Practical Approach, Addison-Wesley Professional, 2016

[9] "Scalability – Wikipedia," https://en.wikipedia.org/wiki/Scalability, Accessed 27 September 2017

[10] Brewer, E., "CAP Twelve Years Later: How the 'Rules Have Changed'," IEEE Computer, 2012, 45, (2), pp. 23-29

[11] Varga, E., Drašković, D., Mijić, D., Scalable Architecture for the Internet of Things - An Introduction to Data-Driven Computing Platforms, O'Reilly Media, Inc. 2018

[12] Newman, S., Building Microservices, O'Reilly Media, Inc. 2015

[13] "Documentation – The Go Programming Language," https://golang.org/doc, Accessed 27 September 2017

[14] Evans, D., The Internet of Things, Cisco, 2011

[15] "Gartner Says 6.4 Billion Connected 'Things' Will Be in Use in 2016, Up 30 Percent From 2015," https://www.gartner.com/newsroom/id/3165317, Accessed 29 September 2017

[16] Carpenter, J., Hewitt, E., Cassandra: The Definitive Guide, O'Reilly Media, Inc. 2nd edn. 2016

[17] Karau, H., Konwinski, A., Wendell, P., Zaharia, M., Learning Spark: Lightning-Fast Big Data Analytics, O'Reilly Media, Inc. 2015

[18] RFC 7230: "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," 2014

[19] RFC 6455: "The WebSocket Protocol," 2011

[20] ISO/IEC 20922:2016: "Message Queuing Telemetry Transport (MQTT) v3.1.1," 2016

[21] RFC 7252: "The Constrained Application Protocol (CoAP)," 2014

[22] "Media Types for Sensor Markup Language (SENML) draft-jennings-senml-10," https://tools.ietf.org/html/draft-jennings-senml-10, Accessed 29 September 2017

[23] "Terminology for Constrained Node Networks draft-ietf-lwig-terminology-07," https://tools.ietf.org/html/draft-ietf-lwig-terminology-07, Accessed 29 September 2017

[24] "Lora AllianceTM Technology," https://www.lora-alliance.org/technology, Accessed 29 September 2017

[25] Varga, E., Blagojević, B., Mijić, D., "Composing Internet of Things Platforms in Smart Grid," 2nd International Conference on Computer, Software and Modeling, Nice, France, 2018 (to be published)

[26] Guinard, D., Trifa, V., Building the Web of Things: With Examples in Node.Js and Raspberry Pi, Manning Publications Co., 2016