applied sciences

MDPI

Article

FLINT: Flows for the Internet of Things

Bart Moons [1,*,†], Michiel Aernouts [2], Vincent Bracke [1,†], Bruno Volckaert [1,†], Jeroen Hoebeke [1,†]

[1] IDLab—Department of Applied Engineering, University of Ghent—Imec, 9052 Ghent, Belgium; vincent.bracke@ugent.be (V.B.); bruno.volckaert@ugent.be (B.V.); jeroen.hoebeke@ugent.be (J.H.)
[2] IDLab—Faculty of Applied Engineering, University of Antwerp—Imec, 2020 Antwerp, Belgium; michiel.aernouts@uantwerpen.be
* Correspondence: bamoons.moons@ugent.be
† Current address: Technologiepark-Zwijnaarde 126, 9052 Ghent, Belgium.

Abstract: New protocols and technologies are continuously competing in the Internet of Things. This has resulted in a fragmented landscape that complicates the integration of different solutions. Standardization efforts try to avoid this problem, however within a certain ecosystem, multiple standards still require integration to enable trans-sector innovation. Moreover, existing devices require transformations to fit in an ecosystem. In this paper, we discuss several integration problems in the field of Low Power Wide Area Networks in the context of the Port of the Future and propose a new distributed platform architecture, called FLINT. FLINT is a framework to program flexible and configurable flows on a per device basis. A flow is constructed from fine-grained components, called adapters. Due to the modularity of an adapter, users can easily integrate existing software. We evaluated FLINT based on five levels of interoperability and show that FLINT can be used to interconnect non-interoperable systems and protocols on every level. We have also implemented FLINT in a container based environment and demonstrated that a basic configuration has a 99% forwarding rate of 17.500 513-byte packets per second, showing that the architecture can deliver good performance.

Keywords: Internet of Things; distributed component systems; interoperability; flow-based programming

Citation: Moons; Aernouts M; Bracke C; Volckaert B; Hoebeke, J. FLINT: Flows for the Internet of Things. *Appl. Sci.* **2021**, *11*, 9303. https://doi.org/10.3390/app11199303

Academic Editors: Dan García Carrillo, Laurent Toutain and Rafael Marín López

Received: 8 September 2021
Accepted: 5 October 2021
Published: 7 October 2021

1. Introduction

Due to the plethora of Internet of Things (IoT) technologies and standards available, an overarching platform is required to interconnect heterogeneous sources of data. The purpose of an IoT platform is to match non-interoperable data sources, ranging from something as small as a device to something as large as a platform spanning multiple application domains. This requires extensive flexibility. Unfortunately, many platforms, such as Amazon Web Service (AWS) IoT [1], Kaa IoT [2] and ThingSpeak [3], have closed, inflexible designs and do not enable low-level interaction between components in the system. Furthermore, extending an IoT system with existing applications and software libraries can often be complex due to the imposed programming paradigms and/or languages. Finally, the upcoming trend of mobile IoT devices that can switch between networks on the one hand and very constrained devices on the other hand, requires a platform aware of the device its connection state and the network it is connected to.

To address this, we present FLINT: a flexible, modular and scalable network architecture for interconnecting IoT devices, networks, middleware and platforms. A FLINT configuration can be built from fine-grained components on a per device basis. These components are data processing elements called *adapters*. An adapter consists of two sub-elements: an *agent* and a *sink*. The sink connects the adapter to the platform by means of a Message Broker. The Message Broker provides a message bus for efficient data delivery using the publish/subscribe paradigm. The sink and the agent communicate over a socket

interface so that the agent can be built from any programming language. This allows users to recycle their existing programs and implementations. A FLINT configuration can be built by chaining adapters. The user chooses a collection of adapters for a given device and connects them into a chain that represents the path the data will follow.

We have implemented FLINT in a container based environment and deployed it in Kubernets cluster to evaluate its scalability. A basic FLINT configuration has a 99% forwarding rate of 17.500 513-byte packets per second on a 2.4 GHz intel E5645. The 1% packet loss is caused by Central Processing Unit (CPU) spikes to 100% in the *sink* implementation. Our evaluation shows that, compared to other platforms, FLINT's architecture is still able to deliver good performance. We show that the forwarding rate drops significantly when adding more hops to the system. However, by demonstrating that the modular architecture can distribute the load over different machines we show that the platform is able to scale for more demanding applications.

Next, we evaluated FLINT based on five levels of interoperability: syntactic interoperability, device interoperability, network interoperability, semantic interoperability and platform interoperability. We thereby show that FLINT provides tools to cover all levels of interoperability as described in [4]. This is illustrated by our experiments in a Port of the Future context (Section 2), though can be extrapolated to any other use case requiring IoT connectivity integration. As a final contribution, FLINT is available as open source under the LGPL-3.0 license.

The remainder of this paper first analyzes other IoT platforms and their focus (Section 3) before describing FLINT's architecture (Section 4) which is evaluated in Section 5. Finally, Sections 6–8 present conclusions and avenue for future work.

2. Case Study—Port of the Future

The Port of the Future will be equipped with technology to cope with day-to-day challenges more efficiently [5]. Sensor measurements can be used to analyze and monitor ports in order to make better operational decisions. For example, transportation can be tracked between different port terminals to increase the coordination of traffic and better manage transport logistic chains. Furthermore, the environment, engineering structures, vehicles and vessels can be tracked in order to optimize their behaviour and contribute to sustainable ports. This either involves (1) existing data sources which have already adopted a specific data format or (2) (wireless) IoT devices.

In order to benefit from information that is already available (1), data from existing systems, legacy systems and systems that support different standards, should be converted to a common format [6].

Furthermore, depending on the available wireless infrastructure of the port, this data will originate from a multitude of data sources (2):

- Multiple wireless communication networks can coexist that serve devices equipped with a single radio;
- Devices can be equipped with multiple radios that connect to different networks over time;
- Devices can have a radio that supports multiple modulation schemes.

For example, devices equipped with a sub-gigahertz (sub-GHz) compatible radio and multiple modulation schemes, such as Long Range (LoRa), DASH-7 (Gaussian Frequency Shift Keying (GFSK)) and Sigfox (Ultra Narrowband (UNB)), can alternate between long range, low throughput and medium range, higher throughput technologies [7]. FLINT was initially developed to interconnect such networks and devices. However, multiple cases, such as the integration with Obelisk [8], made clear that its flexible design was also well suited for fast prototyping and use case support. Therefore, this section presents several modules that were developed and illustrates the evolution of FLINT from a simple tool to a scalable IoT platform.

2.1. Heterogeneous Low Power Wide Area Networks (LPWANs)

The typical size of a port requires wireless technologies that can transmit traffic over large distances. Low Power Wide Area Networks (LPWANs) can span very large areas, however are characterized by severe bandwidth constraints [9]. Traffic towards the wireless network is often restricted by duty cycle limitations and must be operated using different transmission schemes. Furthermore, incorporating multiple wireless networks into a single platform requires some sort of routing. This can be achieved with a central entity that stores the last active network and the properties of all available networks.

Due to their bandwidth constraints, LPWANs often use proprietary payload encoding formats. This limits the application portability and interoperability among systems. In order to solve these issues, the Internet Engineering Task Force (IETF) LPWAN Working Group (WG) developed the Static Context Header Compression (SCHC) technique. This standard uses a *static* context to represent the most common Internet Protocol Version 6 (IPv6), User Datagram Protocol (UDP) and Constrained Application Protocol (CoAP) patterns of the sensor node, known to both the end-device and the gateway. Every flow is distinguished by means of a unique identifier that precedes the payload of the message. This identifier is used by the translating gateway to perform the decompression of the message [10].

This way, very constrained devices can establish an end-to-end IPv6 link over a low bandwidth technology. However, LPWAN systems typically consist of a star topology in which an intermediary captures packets from receiving gateways to remove duplicates. The SCHC router [11] (i.e., the edge router that (de)compresses packets from the IPv6 network) must therefore implement an abstraction to capture packets from the intermediary. Typical examples of such abstractions include the Message Queuing Telemetry Transport (MQTT), the Hyper Text Transfer Protocol (HTTP) and the Advanced Message Queuing Protocol (AMQP). These observations indicate the need to incorporate abstractions for different wireless networks and devices that have multiple network interfaces and their mapping to standardized outputs.

2.2. Localization

Location-based Services (LBSs) are an essential aspect of IoT applications, as they provide valuable context information. For instance, sensor measurements from IoT devices in a port can only be interpreted properly if they are correlated with the correct measurement location. Typically, Global Navigation Satellite Systems (GNSSs) such as Global Positioning System (GPS), Global Navigation Satellite System (GLONASS), BeiDou Navigation Satellite System (BDS) and Galileo are used for outdoor localization. Cellular networks or LPWANs are used to transmit location data from GNSS receivers on IoT devices to an IoT platform. To reduce the overall cost of IoT devices or to enable outdoor localization on devices with an extremely low energy budget, it is possible to omit GNSS receivers and apply localization methods such as Time Difference of Arrival (TDoA), Angle of Arrival (AoA) or Received Signal Strength (RSS)-based methods to the LPWANs instead [12–14]. These methods are especially popular for indoor localization scenarios. Technologies such as Wi-Fi, Bluetooth Low Energy (BLE) or Ultra Wideband (UWB), can cope with the fact that GNSS coverage is not available in indoor environments [15].

Hence, an LBS can obtain location information from a wide range of data sources, depending on the current environment and active technology of an IoT device. Since the performance of a wireless localization method strongly depends on estimation errors related to the environment, the network deployment and the IoT device itself [16], LBSs must be able to intelligently switch between data sources while estimating their reliability [17]. Therefore, there is a need for flexible, heterogeneous platforms such as FLINT that can incorporate an additional module for localization purposes. Specifically for a port use case, this module can switch between outdoor localization with GNSS if a valid fix is available, TDoA localization with LoRaWAN if a transmission is received by at least four gateways, or accurate indoor fingerprinting localization when FLINT detects an active Wi-Fi network.

2.3. Data Transformation

In the past, ports, port services and general services already have developed systems to improve their day-to-day operations. By interconnecting (existing) infrastructure, the efficiency of the entire port can improve even more. Connecting non-interoperable platforms is often addressed as *platform interoperability* and requires data transformation. In order to do so, traffic must be transformed and interconnected between several platforms.

A common way of interconnecting data is by means of the Linked (Open) Data schema (https://5stardata.info/en/) (accessed on 14 July 2021). Linked Data (LD) can be the result of applying the collection of Semantic Web technologies, such as (Resource Description Framework (RDF), Web Ontology Language (OWL), etc.) on non-interoperable data. These tools are standardized by the World Wide Web Consortium (W3C) to provide an environment where applications can query (using SPARQL) and interconnect data from several domains [18]. Converting real world data to interlinked data can be challenging, due to their heterogeneity. This can be optimized using the RDF Mapping Language (RML) [19]. RML uses a Turtle mapfile that defines customized mapping rules, that can be applied to heterogeneous data sources. However, managing multiple data sources requires a specific mapfile on a per device basis.

Another common problem exists in the management of wireless equipment and IoT devices. In order to limit coding and integration effort it can be useful to have a unified mechanism to manage such wireless tools. For this, several management protocols are available today [20]. However, integration of already deployed infrastructure and sensors with standardized solutions can be hard. In this paper, we focus on the Open Mobile Alliance (OMA) Light Weight Machine to Machine (LwM2M) protocol [21]. FLINT tries to minimize the required effort so that their proprietary format can be transformed into LwM2M compliant equipment.

An abstract overview of the required components is given in Figure 1.

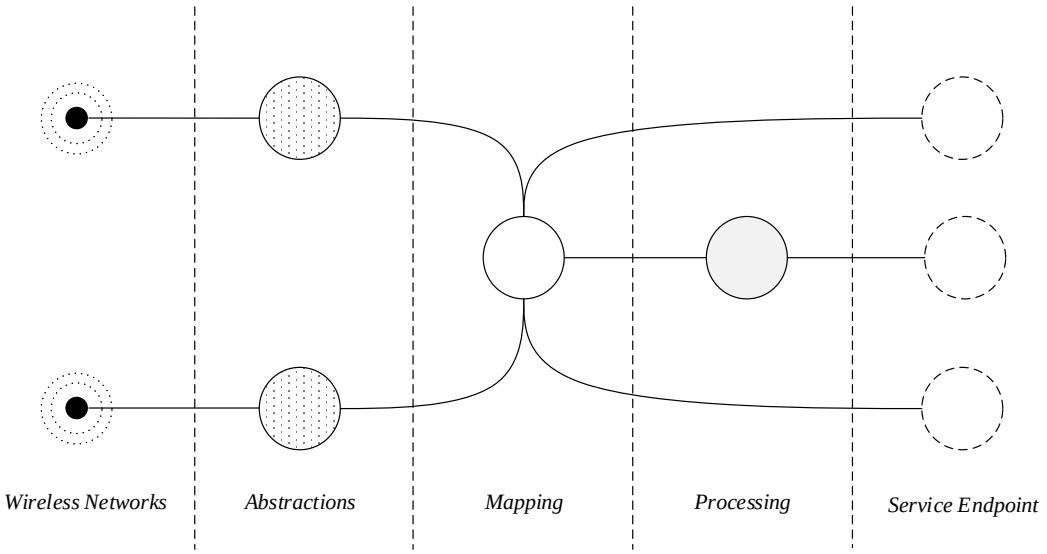


Figure 1. An abstract representation of the system. Wireless sensor networks on the left are integrated using Application Programming Interfaces (APIs). Their data are collected in a central point that provides an abstraction for devices with multiple network interfaces. From this central point, data are distributed to processing nodes and other platforms.

3. Related Work

A plethora of both open-source and commercial IoT platforms already exist. An overview of the most common systems is given in Table 1. The architecture of every platform is presented using a simplified notation representing the number of inputs, the number of intermediary processing elements and the number of outputs:

- N*-N*-1: there can be N inputs, N processing elements and only 1 output. This architecture often stores the data inside the platform.
- N*-N*-N*/N*N*/X*: there can be N inputs, N processing elements and N, $N * N$, or X outputs.

With

- **N*** representing a custom, reusable element, either input, processing or output that can be implemented by the user.
- **X*** representing an element, either input, processing or output offered as is by the platform.

Some platforms allow communication between elements inside the deployed system. This is visualized using N*N*. Flows that end with a single N* or X* provide elements that have a single output. Contrarily, flows that employ an N*N* approach provide reusable components for every node that can be integrated in the platform, i.e., every element has multiple inputs and outputs. An N*N* approach is illustrated in Figure 1, where the message can be passed back and forth between Abstractions, Processing elements and elements that provide platform interoperability at the Service Endpoint.

Next, the supported directionality of a platform is indicated using the Input/Output (I/O) column. Furthermore, the complexity of the platform is presented in column 'Complexity'. A distinction is made between low, medium and high complexity. A low complexity platform indicates that flows can be created by making use of a GUI. A platform that requires some technological background about the platform and technologies is labeled with medium complexity. A platform that requires in depth knowledge about the platform and the underlying technologies is labeled with high complexity.

Table 1. Non-exhaustive list of commercial and open source IoT platforms.

Project	Solution	License	Protocols	Usability	Interoperability	Architecture	Language	Direction	Complexity
AWS IoT [1]	Message broker, digital twin, AVS	Commercial license, open source libraries	MQTT, HTTP	Draw flows in a GUI	No	N*-N*-X*	GUI	I/O	Low
OpenRemote [22]	Integrate assets, data visualization	AGPLv3	HTTP, WS, MQTT, custom	Draw flows in a GUI	No	N*-N*-1	GUI	I	Low
Kaa [2]	Flows	Commercial license, free plan up to 5 devices	MQTT, HTTP	Draw flows in a GUI	No	N*-N*-1	GUI	I/O	Low
ThingSpeak [3]	Data visualization	Commercial license	WS, MQTT, HTTP	Integrate things in Matlab	No	N*-1 - 1	Matlab	I/O	Low
INTER-IoT [23]	Multi-layered approach	Apache-2.0	HTTP, CoAP, MQTT	Distributed platform	Yes	N*-N*-N*	Java	I/O	High
ThingsBoard [24]	Flows	Apache-2.0/Commercial	HTTP, MQTT, UDP, TCP	Draw flows in a GUI	No	N*-N*-X	GUI	I/O	Low
Eclipse node-wot [25]	Web of Things platform	EPL-2.0	HTTP, CoAP, WS, MQTT, custom	Access things using web technologies	Yes	N*-N*-1	JavaScript	I/O	High
Eclipse Hono [26]	Container based IoT-platform	EPL-2.0	HTTP, MQTT, custom	HTTP endpoint	No	N*-N*-N*	Any [1]	I/O	High
Node-RED [27]	Visually integrate data flows	Apache-2.0	HTTP, MQTT, Websockets, custom	Draw flows in a GUI	No	N*-N*-N*N*	JavaScript	I/O	Low
FLINT	Flows	LGPL-3.0	HTTP, MQTT, custom	Program flows	Yes	N*- N*-N*N*	Any [2]	I/O	Medium

[1] Any programming language that supports AMQP. [2] Any programming language that supports sockets.

Many of these projects aim to provide a user friendly interface to interconnect systems and applications. The open source flow-based editor, *Node-RED*, provides a GUI where modules can be connected to develop a flow [27]. Extra modules are also shared by the *Node-RED* community. *Node-RED* runs as a single Node.js instance and focuses on usability and fast prototyping, rather than on scalability and interoperability.

Another open source project, *node-wot*, implements the Web of Things (WoT) specification and aims to provide an abstraction between physical things and the current World Wide Web (WWW) [25]. This abstraction makes it possible to access things using web technologies. The W3C Thing Descriptions (TDs) provide semantic interoperability. Devices can be accessed and interconnected using LD concepts. *node-wot* runs inside a Node.js server and focuses on interoperability and usability, rather than on scalability.

INTER-IoT grew out of the objective to design and implement a cross-layer framework to provide interoperability among heterogeneous IoT platforms. [28]. The INTER-IoT Framework provides a Representational State Transfer (REST) API that exposes interactions to any IoT platform. Developers can then access the underlying IoT platforms through a single interface, thereby mainly focusing on interoperability.

Finally, *Eclipse Hono* provides a container based IoT platform. Different networks can be added by means of protocol adapters. Setting up an instance of *Eclipse Hono* can be done rather quickly using Kubernetes deployment tools. Devices can be added using an HTTP interface.

FLINT tries to combine these properties and proposes an open source IoT platform with support for devices with multiple network interfaces, intended for fast prototyping, while maintaining scalability. FLINT can be used to interconnect platforms, while most platforms keep data locked in the platform or do not directly support platform interoperability.

4. Flint Architecture

A FLINT configuration can be built from fine-grained components on a per device basis. These components are data processing elements called *adapters*. An *adapter* represents a unit of processing. This can be seen as the transformation between two data sources to match the format from heterogeneous data sources, to enrich data by performing complex computations, to add semantics, etc. A FLINT configuration is a graph with *agents* at the vertices. An edge, or *sink*, between two agents represents a path for data transformation and/or data output. The user determines the configuration for a device or a fleet of devices by choosing a set of adapters between them. A running system consists of one or more Kubernetes deployments, made up of at least two containers that contain the sink and the agent. Several *sinks* can communicate via a Message Broker (currently MQTT), that provides a (distributed) message bus. The most important properties of an adapter are the following:

- *Adapter identifier*. Each adapter has a unique identifier. This specifies the topic that is used by the Message Broker to interface between adapters.
- *Configuration files*. An agent and a sink have a configuration file. These files are passed to the containers at initialization time. Adapters use these configuration files to set a per-adapter state and provide flexibility during deployment.
- *Socket connection*. An agent and a sink interface over a socket connection. Incoming data from the platform are forwarded to the agent. Modified data or data coming from other data sources is returned to the sink over the socket connection.
- *Sink port*. The sink port is an interface from the sink to the Message Broker. Data flows from the output sink port of an adapter to the input sink port of another adapter.
- *Agent port*. The agent port provides an optional connection to implementation specific interfaces. An input agent port, for example, can be used to feed data from a network source to the platform. An output agent port, on the other hand, can be used to forward data to another platform.

Figure 2 shows the layout of an adapter.

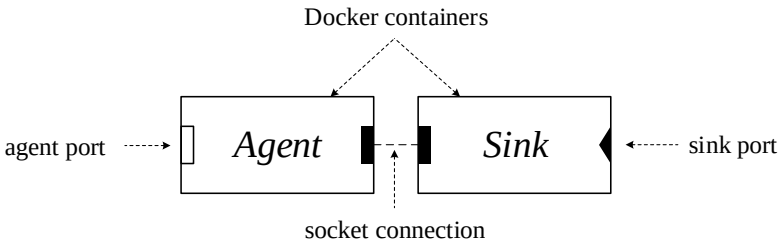


Figure 2. The basic components of an adapter. Triangular ports connect to the platform, filled rectangular ports connect the agent and the sink. Open rectangular ports connect the agent to non-FLINT sources.

4.1. Adapter Types

FLINT supports three types of adapters—*I/O* adapters, *processing* adapters and *direct* adapters. I/O adapters use their agent port to fetch data from a data source or push data to a destination. *Processing* adapters, on the contrary, do not use their agent ports and return the data to the sink after processing. Figure 3 shows how the different types of adapters work in a simple configuration. Data coming from a data source is captured using the input port of the *Data in* agent. The corresponding element forwards the data to the *Mapper* adapter. This is a mandatory processing adapter responsible for data delivery to the next adapter in the chain. Incoming data are passed to the agent and distributed over one or more outputs. The *Data out* I/O adapter can perform extra processing to match the format of the data source it is connected to or immediately forward the data to the destination.

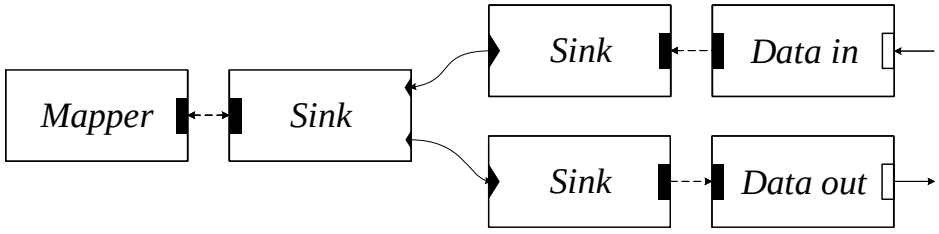


Figure 3. The different types of adapters in a sample configuration. The central adapter is the Mapper. This is a processing adapter that forwards data from input adapters to processing adapters and to output adapters.

Some configurations require bidirectional communication. I/O adapters therefore support data flowing in both directions. Their corresponding sink element is subscribed to a unique topic for communication happening between adapters. Communication coming from an external data source is published to the central *Mapper* adapter acting as a router. This allows for fine-grained configuration in both the upward and downward direction.

Finally, FLINT also supports the use of *Direct* adapters. These are a special type of adapter that omit the *Mapper* adapter. This can be useful to connect a fleet of devices directly to an adapter chain.

4.2. Device Based Context

FLINT uses the *Mapper* adapter as a hub for data distribution. The Mapper is a simple processing adapter with multiple sink inputs and outputs. Incoming data are processed on a per-device basis. The agent matches a device's Medium Access Control (MAC) address with an adapter based on the input topic. A routing scheme is constructed from the configured information for a particular device. The sink then forwards the original payload and the routing scheme to the next adapter in the chain. From there, based on the routing scheme, the processed packet is distributed to the next hop.

Adapters have a unique identifier. These identifiers are used to construct chains of adapters. Every adapter, except for the Mapper adapter, subscribes to their input topic, i.e.,

urn:uuid:adapter-uuid/in. The Mapper, on the contrary, is subscribed to the output topic of every adapter, i.e., *+/out*.

Figure 4 shows a directed graph using the different types of adapters. The *io-1* and *io-2* adapters are I/O adapters. Adapter *p-1* is a processing adapter and *d-1* is a direct adapter. In this example, adapter *io-1* publishes its input data to the output topic *urn:uuid:io-1/out* to request a routing scheme from the mapper. An exact match for a device's MAC address and the requesting adapter will return the configured routing scheme. The packet from adapter *io-1* is sent to the next hop in the chain, i.e., *p-1*. From there, *p-1* forwards the modified data to the next adapter in the chain—*io-2*. At this point, data are output using the adapter's agent. Similarly, communication flowing in the other direction is published to the adapter's output topic to request routing information from the mapper. The scheme in the other direction excludes *p-1* from the chain and is directly forwarded to *io-1*. Direct adapters, such as *d-1*, forward data directly to their destination. This can be useful to serve a fleet of devices.

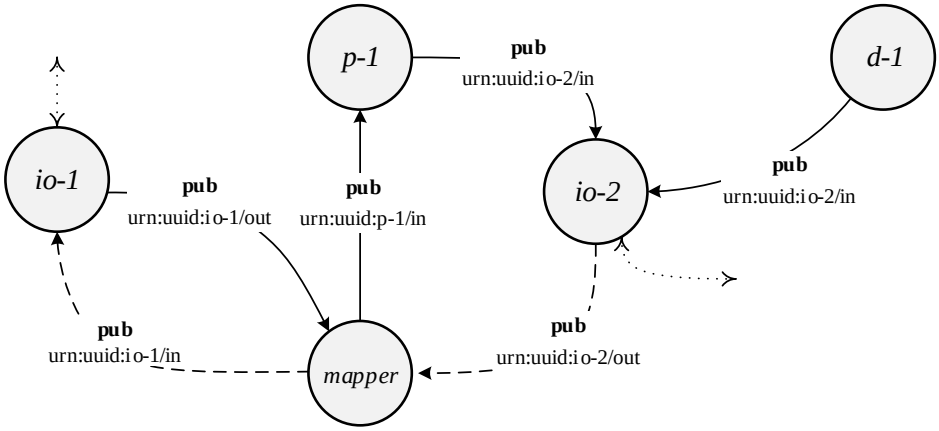


Figure 4. A directed graph constructed using the different types of adapters. Adapter *io-1* is an I/O adapter. Adapter *p-1* is a processing adapter and adapter *io-2* is another I/O adapter. The mapper adapter passes the messages along the vertices. Adapter *d-1* communicates directly with *io-2*.

4.3. Packet Storage

FLINT is built from the observation that IoT devices can have multiple network interfaces that must be routed to multiple outputs. These devices can also have restricted communication opportunities due to their energy savings mechanisms. Therefore, the *Mapper* adapter keeps track of the last active network for every device and their corresponding downlink scheme. Traffic flowing in the downward direction will be routed towards the active network interface or will be queued when no interface is active. Consequently, other FLINT adapters do not require the implementation of queues between adapters or any other routing logic. The Mapper adapter forwards a routing scheme upon a request on its input, which can be interpreted by the *sink* from any other adapter. Hence, the *Mapper* adapter is a mandatory component in more advanced configurations.

4.4. Device Configuration

FLINT device configurations are written in JavaScript Object Notation (JSON), based on the W3C Thing Description (TD) ontology. A standard TD is extended using three keys: `IPv6`, `ioAdapterDefinitions` and `adapterScheme`. `ioAdapterDefinitions` contain values required for bidirectional communication. The `adapterScheme` indicates how the different adapters are connected. An example is given in Listing 1.

Listing 1: A Thing Description contains a list of adapter definitions. These adapters can be used in a scheme to construct a directed graph.

```
{
"id": "urn:5f3cc650-c91a-477f-9726-f5f27097b6c7",
"@context": "https://www.w3.org/2019/wot/td/v1",
"title": "SampleDevice",
"ipv6": ["2001:0db8:85a3:0000:0000:8a2e:0370:7334"],
"ioAdapterDefinitions":
[{
"uuid": "urn:uuid:cc0c1e3b-ce09-4c44-b9da-57ee9871497a",
"deviceDefinitions": {
"interfaceType": "continuous"
"mac": "0004a30b0024e96c"
}
},
{
"uuid": "urn:uuid:489edd56-4b1c-4332-a5ba-cad14994668d",
"deviceDefinitions": {
"interfaceType": "uplink_triggered",
"mac": "92BC10"
}
},
{
"uuid": "urn:uuid:1323b31a-1af7-4548-aad7-89f45f7b5713",
"deviceDefinitions": {
"id": "2"
}
}],
"adapterScheme": [
[
["urn:cc0c1e3b-ce09-4c44-b9da-57ee9871497a",
"urn:uuid:489edd56-4b1c-4332-a5ba-cad14994668d"]
["urn:uuid:8b6092c2-09a4-4aca-aca2-2948195c10de"],
["urn:uuid:1323b31a-1af7-4548-aad7-89f45f7b5713"]
],
[
["urn:uuid:1323b31a-1af7-4548-aad7-89f45f7b5713"],
["urn:cc0c1e3b-ce09-4c44-b9da-57ee9871497a",
"urn:uuid:489edd56-4b1c-4332-a5ba-cad14994668d"]
]
],
...
}
```

An `ioAdapterDefinition` contains routing information. Every definition can be linked to an adapter using its uuid. The *Mapper* adapter also requires the MAC address of every I/O interface to uniquely target a device and manage the queue based on the `interfaceType`. These types can be divided in three groups. A `continuous` interface can be reached at any given moment. A `beacon` interface can be interfaced with during predefined intervals and an `uplink_triggered` interface can be reached only after an uplink transmission.

The `adapterScheme` is a JSON array constructed of *chains* that contain *shackles*, which on their turn contain a list of adapters. A chain starts with an array of I/O adapters (a shackle). Every adapter in this array can be a possible source of data. The *Mapper* adapter

will therefore keep track of the last active adapter in this shackle. The chain ends with a shackle of I/O adapters. Processing adapters can be added at any level in between them. Every shackle in the chain indicates a hop. An adapter that receives a packet on its input will forward the processed packet to *every* adapter in the next hop.

The example `adapterScheme` in Listing 1 contains two chains. The first chain is used for traffic flowing in the upward direction and contains three shackles. The first shackle contains two adapters that are used to interface with the device directly. Data from any of these adapters will flow to the adapter in the next shackle that contains a processing adapter (urn:uuid:8b6092c2-09a4-4aca-aca2-2948195c10de). The last shackle in the chain will output the processed data to a given destination. The second chain contains the inverted configuration, used for communication in the **downward** direction. In this example, the processing adapter is not required for communication in this direction and is removed from the chain. The last shackle again contains the two adapters that can interface directly with the device. The *Mapper* adapter keeps track of the last active adapter to forward data to.

Communication between the adapters also happens in JSON. Every adapter must therefore adhere to a JSON scheme to provide syntactical interoperability.

5. Evaluation

This section evaluates a real FLINT configuration: two Low Power Wide Area Networks (LoRaWAN and Sigfox) serve as a communication technology for several low power devices. Some devices use the novel SCHC standard to compress IPv6 packets. Others require enriching their data with LPWAN localization before forwarding it to an IoT platform (e.g., Linked Data Platform (LDP), Obelisk). An overview of the complete system configuration is given in Figure 5.

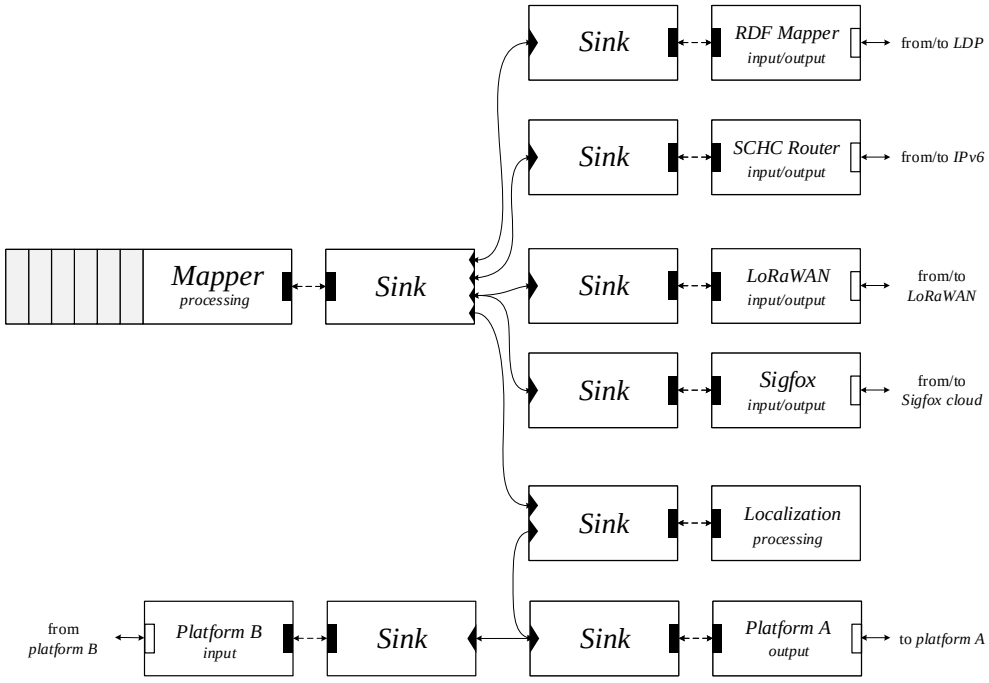


Figure 5. More complex system configuration. Every *sink* is connected to others via the message bus. The *Mapper* adapter implements a queue.

The system is evaluated in terms of interoperability, scalability and performance.

5.1. Levels of Interoperability

Interoperability in the IoT can be seen from different perspectives such as syntactic interoperability, networking interoperability, device interoperability, semantic interoperability, and platform interoperability [4]. These subdivisions are further explained in the following sections while being applied to FLINT.

5.1.1. Syntactic Interoperability

Processing tasks that involve local information do not require any adaptation to the network. The SCHC router, for example, must store information about the fragmentation state of a device. If the device does not acknowledge a fragment in time, the SCHC router will re-transmit the fragment or discard the packet. These actions depend on the packet's content and do not involve any other adapters. Other processing tasks, however, do require information from another adapter in the chain.

Passing information between adapters requires a common messaging pattern, often referred to as syntactic interoperability. FLINT uses a message scheme that adapters must adhere to. The message scheme contains fields to carry information along and may include:

- *Device information.* Information such as the device MAC address and the original data packet are stored in this field. Any additional information can be added to the `device-custom-ctrl` field.
- *Adapter information.* The `adapter-ctrl` field contains information about the previous adapter in the chain. Information about the chain is also stored in this field. Every I/O adapter first consults the *Mapper* adapter to generate the routing scheme that is added to this field.
- *Input information.* The `input-ctrl` field contains information about the adapter that first received the packet. Custom information about the network the adapter is connected to, can be added to the `input-custom-ctrl` field.
- *Output information.* Additional information about the device is stored in the `output-ctrl` field. The TD of every device contains the uuid of each device and is added to the message by the *Mapper* adapter. Other adapters in the chain must be able to uniquely identify devices, regardless of their input adapter. The uuid provides an abstraction for devices that have multiple network interfaces.

Every *sink* serializes these messages using the above grammar in order to provide *syntactical interoperability*.

5.1.2. Device and Network Interoperability

The lowest level of interoperability that FLINT can guarantee, is device and network interoperability. Device interoperability ensures that *high-end IoT devices*, such as smartphones, can communicate with resource constrained, *low-end IoT devices* [4]. The *Mapper* interface provides a queue and supports bidirectional communication over multiple network interfaces. Both *low-end devices* and *high-end devices* can exchange information using different communication technologies with diverse downlink patterns. Figure 6 shows an excerpt of the complete FLINT configuration that provides device and network interoperability for a *low-end device* with a Sigfox and LoRaWAN interface. The Sigfox adapter forwards packet p together with the device's MAC address to the *Mapper* adapter. The routing scheme is calculated based on the information in the TD. Packets coming from these networks are SCHC compressed and forwarded to the SCHC router. This adapter handles requests to, and responses from, the IPv6 network. The response from the IPv6 network (packet f) is stored in the queue of the *Mapper* adapter, since downward traffic for these devices can not flow continuously. Finally, due to a network change, the device transmits packet s over its LoRaWAN interface. The *Mapper* adapter will immediately dequeue packet f and forward it to the corresponding LoRaWAN adapter. Packet s is also delivered to the SCHC router.

As illustrated, the *Mapper* adapter is required to provide *device and network interoperability*.

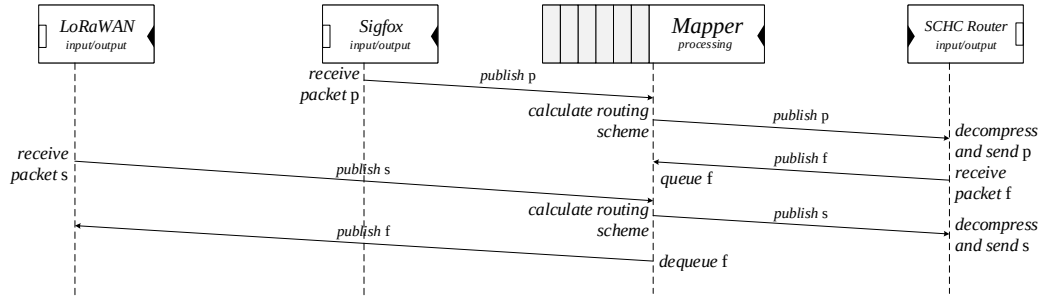


Figure 6. Mobility and queue management in a FLINT system. This diagram shows how packets travel through a FLINT system configuration. Time moves downwards. The central element is a *Mapper* adapter. Packets are distributed from transmitting devices to the destination. Packets in the downward direction are queued and dequeued according to the device's `interfaceType`.

5.1.3. Semantic and Platform Interoperability

FLINT is well suited to provide both semantic and platform interoperability. For example, imagine a LPWAN sensor network with the following requirements:

- *Low-end* sensors must be manageable through the LwM2M protocol. Preferably, their packets should be compressed using the SCHC compression and fragmentation standard.
- For proprietary sensors, an adapter must run as a digital twin in the FLINT platform. This can be done by converting their proprietary format using the Virtual Device Manager (VDM) [29].
- Data from the LwM2M server must be delivered to a LDP. Hence, the LwM2M ontology must be mapped to RDF. This can be done using the RML.

FLINT's modular and extensible architecture makes this easy; Figure 7 shows the configuration. Data coming from the LPWANs are delivered either to the SCHC router or the VDM. Both adapters send requests to the LwM2M server and deliver responses to the *Mapper* adapter. The LwM2M adapter subscribes to the event stream of the LwM2M server. The RML adapter on its turn maps the LwM2M ontology to the Semantic Sensor Networks (SSN) ontology. The result is published to a LDP.

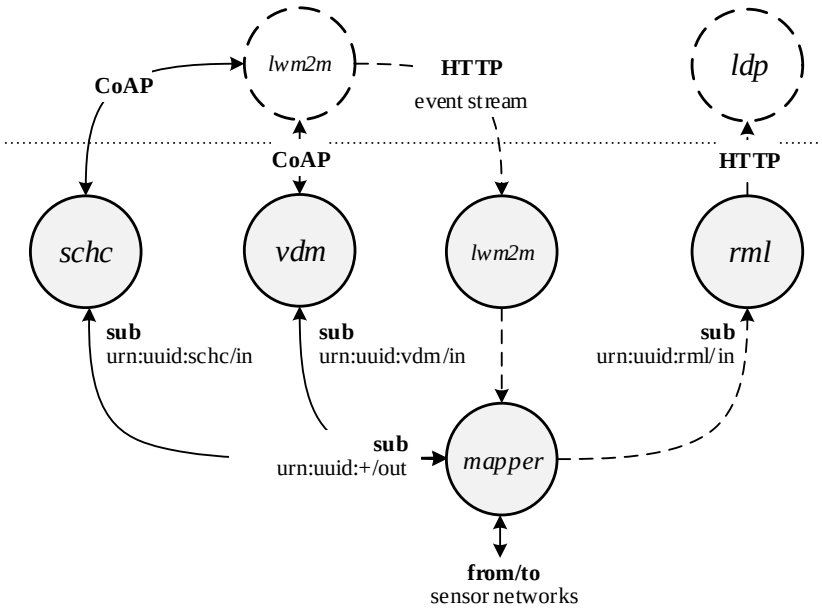


Figure 7. FLINT configuration that provides semantic and platform interoperability. Packets from the LPWANs are delivered to the LwM2M server. Data from the LwM2M server are mapped to an ontology that matches the semantics of the LDP.

The above configuration shows how two incompatible platforms can be interconnected through FLINT. An adapter is dedicated to match their semantics and thereby provides *semantic and cross-platform interoperability*. Once cross-platform interoperability is achieved, *cross-domain interoperability* can be enabled by adding adapters that integrate platforms from heterogeneous domains.

5.2. Scalability

This section evaluates FLINT in terms of scalability. Every FLINT adapter consists of at least two components—the agent and the sink. Consequently, every adapter consists of (at least) two containers. Components can be updated easily when pulling an image in a Kubernetes deployment. Kubernetes thereby provides a fast iteration cycle and scalability. Consider as an example the data flow from Section 5.1.3.

There, a FLINT configuration uses RML to match different ontologies. The RML adapter uses a Turtle mapfile that describes the mapping. The Kubernetes configuration file is used to input the mapfile to the RML adapter that runs in a Node.js server instance. In a running system, this adapter can be replicated so multiple adapters can match the LwM2M vocabulary to various ontologies. This can be done by simply replacing the mapfile. Figure 8 shows two Kubernetes deployments that are managed by the Kubernetes master. A Kubernetes deployment represents an adapter and can be configured by mounting configuration files. In this example, the LwM2M data model can be translated to SSN by one adapter and to Next Generation Service Interfaces (NGSI)-LD (NGSI-LD) by another one. By simply replicating the adapter and mounting a different configuration file, the FLINT configuration can be expanded easily.

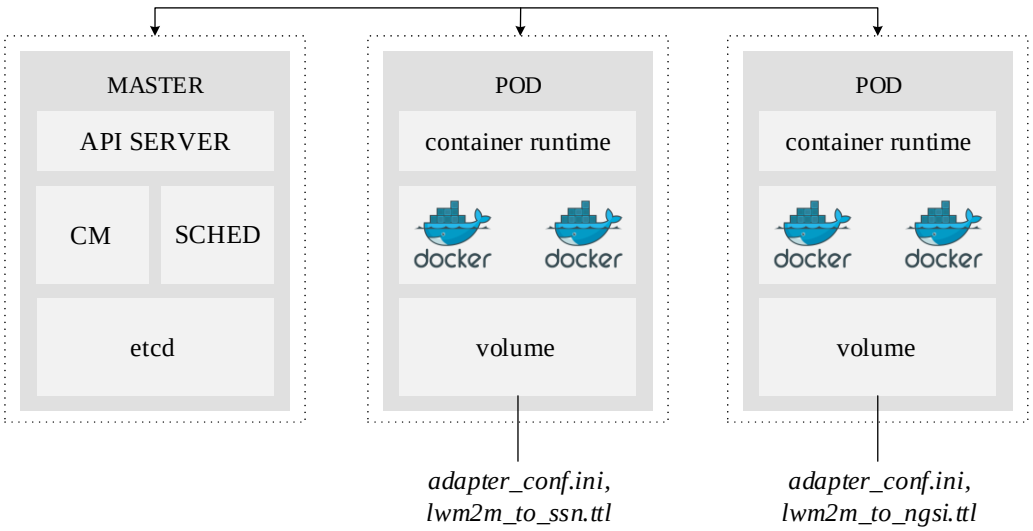


Figure 8. An excerpt of a FLINT system in a Kubernetes environment. Every Kubernetes *deployment* consists of (at least) two Docker containers—the sink and the agent. Both containers can be configured by mounting configuration files. Every deployment is managed by the Kubernetes master.

5.3. Performance Evaluation

This section first compares FLINT with other State of the Art (SoTA) platforms that provide similar functionality (i.e., open source N*-N*-N* platforms). The evaluation uses six MQTT clients that publish data to a broker. For every platform, a platform adapter processes the incoming data in order to feed it to the IoT platform. The platform delivers the data to a UDP client that forwards the data to a UDP server. This is illustrated in Figure 9.

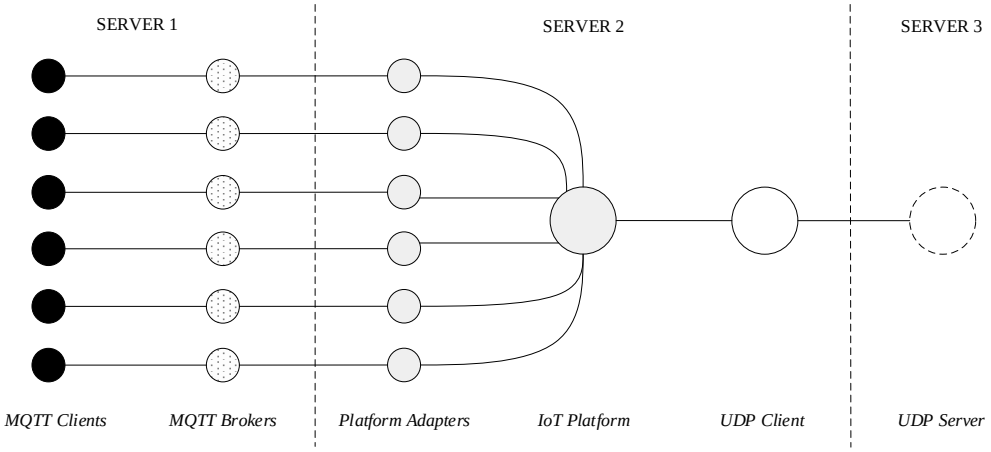


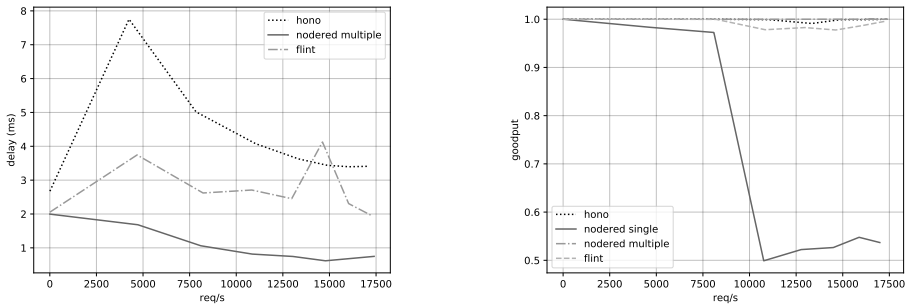
Figure 9. The test setup. Every test consists of six MQTT clients, subscribed to six MQTT brokers. The clients forward their data to the IoT platform that is being tested. A UDP client fetches data from the platform and forwards the packet to a UDP server.

5.3.1. Experimental Setup

The experimental setup consists of three servers running Ubuntu 18.04 and are synced using the Network Time Protocol daemon (ntpd). All three servers have a gigabit Network Interface Card (NIC), 2 Hexacore Intel E5645 (2.4Gigahertz (GHz)) CPU and 24 Gigabyte (GB) Random Access Memory (RAM). One server is used to host the MQTT clients and brokers. Another to host the platforms and one to host the UDP server. Each IoT platform is configured to run in a Kubernetes cluster.

5.3.2. Analysis of Platform Performance

This section analyzes the different selected platforms under different loads. The responsiveness of every platform is measured using the goodput. This has been defined as the total number of successfully received packets divided by the total number of sent packets. Figure 10 shows the average time and goodput for Node-RED (v.1.3.5), Eclipse Hono (v 1.9) and FLINT.



(**a**) Average delay in seconds (**b**) Goodput

Figure 10. The latency (in seconds) and goodput measured for six MQTT adapters in different platforms. Multiple instances of Node-RED perform best compared to Hono and FLINT. Node-RED single had too large delays for the Figure, but can be retrieved from Table 2.

Excluded from the Figure are the very large delays for a single Node-RED instance. For completeness, these values are provided in Table 2. These high delays are due to the single-threaded design of the underlying Node.js instance [30]. Furthermore, the goodput for a Node-RED (single instance) application dropped significantly for data rates above 7500 messages per second, which can be seen in Figure 10b. To take advantage of multi-core systems, a cluster of Node.js processes can be deployed to handle the load. Therefore, six Node-RED instances were deployed and is referenced as *Node-RED multiple*. Every instance

subscribes to a MQTT broker, processes the data and forwards it to the UDP server. This resulted in near-zero latency and a 100% goodput.

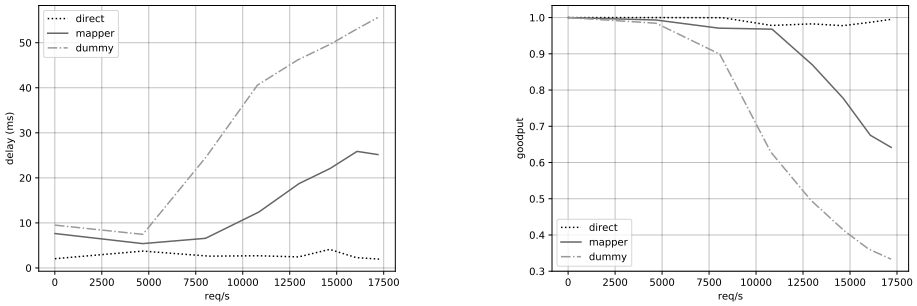
For this experiment, Hono used the default configuration. However, as Hono seemed to struggle with data rates above 5000 messages/second (which was also reported in [31]), we deployed a Vert.x MQTT adapter for every MQTT client. This resulted in slightly worse results in terms of latency and goodput, compared to Node-RED.

Finally, FLINT consisted of six *direct* MQTT adapters that published their data directly over the message bus to the UDP I/O adapter. Even though FLINT requires at least twice the payload size of the other platforms, the experiment showed slightly better results compared to Hono and slightly worse results than Node-RED running multiple instances. We noticed CPU spikes to 100% for some threads running the *sink* implementation, which likely causes the 1% packet loss.

5.4. Mapper Forwarding Rate

FLINT provides flexibility in connecting non-interoperable IoT networks. Due to its modularity, it is possible to interconnect various IoT components on top of which the *Mapper* adapter can queue packets based on the interface type of the device. These advantages, however, come at a cost. This section analyzes the drawbacks introduced by the platform.

The above section used six *direct* adapters. This type of adapter forwards the data directly to another endpoint in the platform. However, the *Mapper* adapter can be used to forward data between several components. As shown in Figure 11, the average latency increases and the goodput decreases, since the message bus must process twice as many messages.



(**a**) Average delay in seconds (**b**) Goodput

Figure 11. The latency (in seconds) and goodput for six MQTT adapters measured for different configurations in FLINT.

In order to measure the impact of another hop, a *dummy* adapter was added to the system. This is a processing adapter that forwards the packet after a *No Operation* (NOP). Messages will now flow from the MQTT adapter to the *Mapper* adapter, to the *dummy* adapter and finally to the UDP adapter. Using this configuration, the message bus must process three times the amount of messages compared to the second configuration.

An overview of the measurements is given in Table 2. It can be seen that the message bus and the *Mapper* adapter form a bottleneck for the system. In the future, this can be solved in several ways. A first solution could use MQTT v5 Shared Subscriptions. This feature distributes the load across all subscribers on the same topic and is sometimes referred to as client load balancing. This allows setting up multiple Mapper adapter instances without having duplicates. To avoid saturation of the MQTT broker, a cluster of MQTT brokers can be deployed. Managing a cluster of MQTT brokers, however, requires non-standard broker discovery, subscriber takeover and routing management. Therefore, other messages busses, such as Apache Kafka or Pulsar [32] can be used. This would require a few changes to the system. First, the sink must be replaced in order to communicate with the improved message bus. In Apache Kafka, multiple partitions can then be used to form consumer

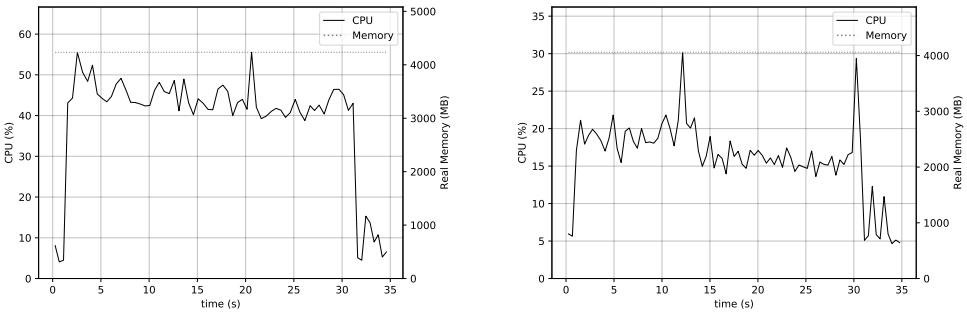
groups. These groups avoid duplication of data over multiple members subscribed to the same topic. As such, multiple mapper instances can be deployed, without the need to redesign the Mapper adapter completely. However, all Mapper adapters should access the same Thing Descriptions, and therefore require access to a shared Thing Directory.

Table 2. The Latency (L) in seconds, the Goodput (G) and the Payload size in bytes for the evaluated platforms.

req/s	4500		10,500		14,500		17,500		All
	L (s)	*G*	*L (s)*	*G*	*L (s)*	*G*	*L (s)*	*G*	*Payload size (B)*
Node-RED (1)	0.007	0.98	11.633	0.5	20.843	0.53	26.278	0.54	259
Node-RED (6)	0.0016	1	0.0008	0.99	0.0006	1	0.0007	0.99	259
Eclipse Hono	0.007	1	0.004	0.99	0.003	0.99	0.003	0.99	259
FLINT (direct)	0.003	1	0.002	0.98	0.004	0.98	0.002	0.99	513
FLINT (mapper)	0.005	0.99	0.012	0.97	0.022	0.78	0.025	0.64	1102
FLINT (dummy)	0.007	0.98	0.04	0.63	0.05	0.41	0.054	0.33	1202

5.5. Resource Consumption

Finally, in Figure 12, two system configurations are presented in order to explore the impact of a very basic workload offloading approach. In Figure 12, a, every component runs on the same machine. This results in an average of around 45% CPU load and 4.5 GB memory consumption. In the second configuration, the MQTT broker was moved to a different machine. This resulted in an average of only 17% CPU usage and 4 GB memory consumption.



(**a**) FLINT Direct adapter configuration without load balancing

(**b**) FLINT Direct adapter configuration with external broker

Figure 12. Comparison of a very simple load balancing method. The first configuration runs every component on the same machine, while in the second configuration the broker is moved to a different machine.

This confirms the previous statement that the message bus forms a bottleneck for the system, since every message between components passes through the broker. However, due to FLINT's modularity, the message bus can be transferred easily to a different system in order to divide the load, at the cost of a higher network delay.

6. Discussion

Other platforms, such as INTER-IoT and Eclipse node-wot have been around to provide interoperability in the IoT. However, INTER-IoT has become a rather complex platform that provides multi-layer interoperability for large scale projects. Eclipse node-wot, implemented in Node.js, merely focuses on interoperability, rather than scalability.

These observations led to the initial design of FLINT with the main objective to interconnect non-interoperable IoT networks that are characterised by intermittent connectivity. FLINT was designed to support five levels of interoperability—syntactic interoperability,

semantic interoperability, device interoperability, network interoperability and platform interoperability. The modularity of the design, consisting of a *sink* and *agent*, allows the recycling of existing programs through the use of a socket connection, while still being able to scale.

We implemented FLINT to support our design choices and to show how the implementation behaves against different loads on comparable platforms. Our evaluation showed that Node-RED performs best in terms of latency and goodput. However, this requires to break apart the application in order to take advantage of the available CPU cores of the system. Eclipse Hono and FLINT show similar results. However, due to its flexible design using the *sink*, developers require only little understanding of the underlying protocols and can use any language that supports UDP sockets. Integrating business logic in Eclipse Hono, on the contrary, requires both an AMQP library and understanding of the protocol.

The evaluation showed that FLINT is able to deliver good performance in a scalable, container based system. However, both the performance evaluation of the *Mapper* adapter and the resource consumption of the basic load balancing approach show the impact of a single, central message bus. Since every message has to pass through the broker and, if configured, the *Mapper* adapter, this forms a bottleneck for the system. Therefore, other distributed streaming platforms, such as Apache Kafka or Pulsar can be used in the future to provide support for use cases that require higher data rates.

7. Conclusions

FLINT is an open and extensible distributed platform architecture. Chains of adapters can be built in order to serve non-interoperable IoT devices. FLINT was developed and tested in the scope of the Port of the Future where several adapters interconnect networks and platforms. Due to the modularity of an adapter, developers can easily integrate existing software while maintaining a flexible design. This will allow further evolution of separate networks and platforms and may contribute to inter-port interaction, port-city interaction and trans-sector innovation in general. Our performance analysis shows that the modularity is comparable to other existing platforms, while offering the ability to develop software using rapid prototyping approaches. In order to maintain a scalable platform, components can be replicated in a distributed network. However, scalability should still be looked at in more detail. Duplicates must be avoided when deploying multiple adapter instances and the abilities of different messages busses should be analyzed. FLINT is free software; it is available for download at https://github.com/imec-idlab/flint (accessed on 6 October 2021).

8. Future Work

Currently, every adapter is given a Unique Resource Name (URN) which must be known to the *Mapper* adapter in order to route packets. However, adapters should be able to configure themselves, request a URN and update existing routing schemes. Depending on the message bus, other *sink* implementations must be provided. Furthermore, since Kubernetes is deprecating Docker and Docker actually sits on top of containerd, FLINT will evolve in the future to use containerd as a Container Runtime Interface (CRI). Due to the fact that containerd is more lightweight, it will contribute to the rapid forwarding which FLINT is targeting. In order to improve the responsiveness and scalability of the system even more, the proposed methods to cope with the determined bottlenecks should be evaluated. Finally, it should be investigated whether injecting environment variables is more lightweight instead of using configuration files.

Author Contributions: This work has been part of the PortForward project, where all the above authors have contributed to the outcome. J.H. and B.M. started the investigation and conceptualization. B.M. performed the formal analysis, wrote the software, performed the validation and wrote the original draft. M.A. and V.B. contributed to the software and edited and reviewed the original draft. J.H. supervised the implementation and validation of the software. J.H. and B.V. revised the draft of the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research has received funding from the European Union's Horizon 2020 research and innovation program under grant number 769267 (PortForward project).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in https://gitlab.ilabt.imec.be/bamoons/flint_evaluation_data.

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

References

1. Amazon. AWS IoT Core for LoRaWAN. Available online: https://docs.aws.amazon.com/iot/latest/developerguide/connect-iot-lorawan.html. (accessed on 13 July 2021).
2. Technologies, K. Kaa Enterprise IoT Platform. Available online: https://www.kaaiot.com. (accessed on 13 July 2021).
3. Mathworks Inc. ThingSpeak Internet of Things. Available online: https://www.thingspeak.com. (accessed on 13 July 2021).
4. Noura, M.; Atiquzzaman, M.; Gaedke, M. Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mob. Netw. Appl.* **2019**, *24*, 796–809. [CrossRef]
5. Yang, Y.; Zhong, M.; Yao, H.; Yu, F.; Fu, X.; Postolache, O. Internet of things for smart ports: Technologies and challenges. *IEEE Instrum. Meas. Mag.* **2018**, *21*, 34–43. [CrossRef]
6. Inkinen, T.; Helminen, R.; Saarikoski, J. Port Digitalization with Open Data: Challenges, Opportunities, and Integrations. *J. Open Innov. Technol. Mark. Complex.* **2019**, *5*, 30. [CrossRef]
7. Famaey, J.; Berkvens, R.; Ergeerts, G.; De Poorter, E.; Van den Abeele, F.; Bolckmans, T.; Hoebeke, J.; Weyn, M. Flexible Multimodal Sub-Gigahertz Communication for Heterogeneous Internet of Things Applications. *IEEE Commun. Mag.* **2018**, *56*, 146–153. [CrossRef]
8. Bracke, V.; Sebrechts, M.; Moons, B.; Hoebeke, J.; De Turck, F.; Volckaert, B. Design and evaluation of a scalable Internet of Things backend for smart ports. *Softw. Pract. Exp.* **2021**, *51*, 1557–1579. [CrossRef]
9. Buurman, B.; Kamruzzaman, J.; Karmakar, G.; Islam, S. Low-Power Wide-Area Networks: Design Goals, Architecture, Suitability to Use Cases and Research Challenges. *IEEE Access* **2020**, *8*, 17179–17220. [CrossRef]
10. Minaburo, A.; Toutain, L.; Gomez, C.; Barthel, D.; Zúñiga, J.C. *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*; Technical Report RFC8724; RFC: Fremont, CA, USA, 2020. [CrossRef]
11. Moons, B.; Karaagac, A.; Haxhibeqiri, J.; De Poorter, E.; Hoebeke, J. Using SCHC for an optimized protocol stack in multimodal LPWAN solutions. In Proceedings of the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), Limerick, Ireland, 15–18 April 2019; pp. 430–435. [CrossRef]
12. Aernouts, M.; BniLam, N.; Podevijn, N.; Plets, D.; Joseph, W.; Berkvens, R.; Weyn, M. Combining TDoA and AoA with a particle filter in an outdoor LoRaWAN network. In Proceedings of the 2020 IEEE/ION Position, Location and Navigation Symposium (PLANS), Portland, OR, USA, 20–23 April 2020; pp. 1060–1069. [CrossRef]
13. Janssen, T.; Weyn, M.; Berkvens, R. A Primer on Real-world RSS-based Outdoor NB-IoT Localization. In Proceedings of the 2020 International Conference on Localization and GNSS (ICL-GNSS), Tampere, Finland, 2–4 June 2020; pp. 1–6. [CrossRef]
14. Anagnostopoulos, G.G; Kalousis, A. A Reproducible Analysis of RSSI Fingerprinting for Outdoor Localization Using Sigfox: Preprocessing and Hyperparameter Tuning. In Proceedings of the 2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN), Pisa, Italy, 30 September–3 October 2019; pp. 1–8. [CrossRef]
15. Zafari, F.; Gkelias, A.; Leung, K.K. A Survey of Indoor Localization Systems and Technologies. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 2568–2599. [CrossRef]
16. Li, Y.; Zhuang, Y.; Hu, X.; Gao, Z.; Hu, J.; Chen, L.; He, Z.; Pei, L.; Chen, K; Wang, M.; et al. Toward Location-Enabled IoT (LE-IoT): IoT Positioning Techniques, Error Sources, and Error Mitigation. *IEEE Internet Things J.* **2020**, *8*, 4035–4062. [CrossRef]
17. Aernouts, M.; Lemic, F.; Moons, B.; Famaey, J.; Hoebeke, J.; Weyn, M.; Berkvens, R. A Multimodal Localization Framework Design for IoT Applications. *Sensors* **2020**, *20*, 4622. [CrossRef] [PubMed]
18. Data—W3C. Available online: https://www.w3.org/standards/semanticweb/data (accessed on 14 July 2021).
19. Dimou, A.; Vander Sande, M.; Colpaert, P.; Verborgh, R.; Mannens, E.; Van de Walle, R. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In Proceedings of the 7th Workshop on Linked Data on the Web, Seoul, Korea, 8 April 2014; p. 5.

20. Sinche, S.; Raposo, D.; Armando, N.; Rodrigues, A.; Boavida, F.; Pereira, V.; Silva, J.S. A Survey of IoT Management Protocols and Frameworks. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 1168–1190. doi: 10.1109/COMST.2019.2943087. [CrossRef]
21. OMA LwM2M. Lightweight Machine to Machine Technical Specification: Core. 2020. Available online: http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf (accessed on 6 June 2021).
22. OpenRemote Inc. OpenRemote: The 100% Open Source IoT Platform. Available online: https://openremote.io (accessed on 13 July 2021).
23. Fortino, G.; Savaglio, C.; Palau, C.E.; de Puga, J.S.; Ganzha, M.; Paprzycki, M.; Montesinos, M.; Liotta, A.; Llop, M. Towards Multi-layer Interoperability of Heterogeneous IoT Platforms: The INTER-IoT Approach. In *Integration, Interconnection, and Interoperability of IoT Systems*; Series Title: Internet of Things; Gravina, R., Palau, C.E., Manso, M., Liotta, A., Fortino, G., Eds; Springer International Publishing: Cham, Switzerland, 2018; pp. 199–232. [CrossRef]
24. Authors, T.T. ThingsBoard: Open-source IoT Platform. Available online: https://www.thingsboard.io. (accessed on 13 July 2021).
25. Thingweb. ThingWeb: A Web of Things Implementation. Available online: https://www.thingweb.io. (accessed on 13 July 2021).
26. Project, T.E.H. Eclipse Hono: Connect, Command & Control IoT Devices. Available online: https://www.eclipse.org/hono. (accessed on 13 July 2021).
27. Foundation, O. Node-RED: Low-Code Programming for Event-Driven Applications. Available online: https://www.nodered.org. (accessed on 13 July 2021).
28. Giménez, P.; Llop, M.; Olivares, E.; Palau; Montesinos; Llorente, M.A. Interoperability of IoT Platforms in the Port Sector. In Proceedings of the Transport Research Arena (TRA 2020), Helsinki, Finland, 27–30 April 2020.
29. Karaagac, A.; VanEeghem, M.; Rossey, J.; Moons, B.; DePoorter, E.; Hoebeke, J. Extensions to LwM2M for Intermittent Connectivity and Improved Efficiency. In Proceedings of the 2018 IEEE Conference on Standards for Communications and Networking (CSCN), Paris, France, 29–31 October 2018; pp. 1–6. [CrossRef]
30. Cluster | Node.js v16.5.0 Documentation. Available online: https://nodejs.org/api/cluster.html (accessed on 15 July 2021).
31. Reimann, J. We Scaled IoT—Eclipse Hono in the Lab. 2018. Available online: https://dentrassi.de/2018/07/25/scaling-iot-eclipse-hono (accessed on 14 July 2021).
32. John, V.; Liu, X. A Survey of Distributed Message Broker Queues. *arXiv* **2017**, arXiv:1704.00411.