Messaging Infrastructure for IoT at Scale

WRITTEN BY PAOLO PATIERNO PRINCIPAL SOFTWARE ENGINEER AT REDHAT

CONTENTS

The Internet of Things isn't just a buzzword anymore — it's real, it's around us, and we are already living in the IoT era. At same time, IoT isn't completely new. Before this term was coined, a lot of companies were already developing embedded devices able to connect to a network in order to communicate each other or with a related monitoring system. Thanks to the hardware evolution, a decrease in related costs, and new cloud technologies and infrastructures, developing powerful "connected" devices is now much simpler than before.

Today, IoT is the field for the revenge of a "hidden" technology that is not always considered properly: the messaging.

IoT is always about messaging. Devices communicate each other by exchanging messages. They send (telemetry) data to a cloud platform and receive commands through messages. IoT can be considered a specific messaging use case in which the clients are just devices in the field and services in the cloud.

The real big difference between a business platform using a messaging infrastructure and an IoT solution is scale. The number of connected devices can range from a few units to millions, which means that the messaging infrastructure has to support a huge number of connections and a massive message exchange while providing high throughput.

WHAT MAKES AN IOT PLATFORM? MESSAGING AS A "LEVER"

The core of an IoT platform is the messaging infrastructure, which has to provide all the features described in the previous section. Around such an infrastructure, there are other "core" services focused on IoT use cases, such as:

- A device registration component for handling devices information and the related status.
- An authentication/authorization component for providing access control.
- A device provisioning component for delivering software updates on devices.

All of these "core IoT" services will use the underlying messaging infrastructure to communicate with each other and with devices in the field.

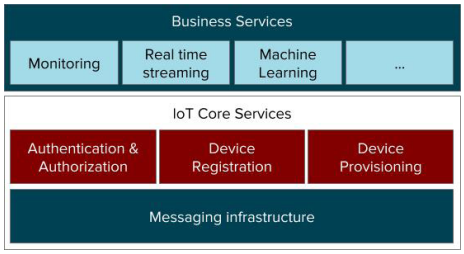

Figure 1: *What makes an IoT platform?*

Of course, all the ingested data needs to be processed in order to bring value to the entire IoT solution. For this reason, other "business" services can leverage the messaging infrastructure in order to receive such data for monitoring, real-time analytics, machine learning, and so on.

THE NEED FOR ELASTICITY... AND MORE

When it comes to supporting a huge number of connected devices that produce telemetry data with high throughput or receive commands from different backend services, the underlying messaging infrastructure in the overall IoT solution becomes the main point of success... or failure.

The main need is to use resources productively to handle spikes in terms of connected devices and messages exchanged from lower to higher throughput. The messaging infrastructure needs to be elastic.

Elasticity means that the user is able to "tune" the messaging resources according to the traffic load so that they are allocated only when it is necessary. For example, speaking of message brokers, it's useless to have a big cluster for data ingestion when there are only a few connected devices. The ability to scale resources up and down provides the elasticity we need, which could be available

automatically (based on some metrics like network I/O, message throughput, connected devices, and so on).

Other than elasticity, the messaging infrastructure needs to provide resiliency and high availability, which make such an infrastructure more complex. All the messaging components need to recover from failures, which always happen in a distributed system. We cannot avoid failures, but we have to handle them properly. At same time, downtime should be approximately zero in order to have the IoT solution always available and "reacting" to the devices and backend services inputs.

Last but not least, whoever wants to develop an IoT solution doesn't also want to take care of all the problems related to the messaging infrastructure deployment. It should be really simple to provide a higher abstraction layer so that the IoT developer can focus on using well-defined messaging addresses to allow devices to communicate with backend services and vice versa.

In conclusion, elasticity, scalability, resiliency, high availability, deployment, and simplicity are the main features we need from a messaging infrastructure in order to address the complexity of an IoT solution.

ENMASSE: AN OPEN-SOURCE MAAS

EnMasse is an open-source "messaging as a service" platform that provides a simple way to deploy a messaging infrastructure both on-premise and in the cloud.

Being open-source allows all the interested IoT developers to get involved in the community around it and influence its development, as well. At same time, it avoids the "vendor lock-in" problem: in the cloud, there are a lot of really good IoT platforms, but they have some disadvantages.

First of all, such platforms are available in the cloud only; sometimes, an IoT solution starts as a POC (proof of concept), handling a few devices and then growing if it needs to. Maybe using the cloud is not a good solution for smaller projects, but having *something* to deploy on-premise on a few internal servers could be even better.

The other problem is that after starting with a specific cloud-based IoT platform, it's quite difficult to move the solution to a different one. Even if these platforms support standard protocols like AMQP 1.0, MQTT, and HTTP, they may handle devices and connectivity in a different way — moving from one platform to another could mean massive changes on both the backend service and the application running on the devices.

With EnMasse, the great advantage is the possibility to start "playing" with a solution locally, then moving to a real installation on-premise and eventually to the cloud if it grows in terms of devices, connections, throughput, and so on. Through all these phases, the user experience around the platform usage is always the same.

Another point is that when EnMasse runs on a cloud provider (i.e. Amazon AWS, Microsoft Azure, Google GCE), it's possible to move it from one provider to another with little impact on the overall IoT solution. Maybe just changing connection information for devices and services could be enough.

EnMasse supports all the well-known messaging patterns (request/reply, publish/subscribe, and competing consumers), which are really strictly related to the available IoT communication patterns (telemetry, notification, command/control, and inquiry).

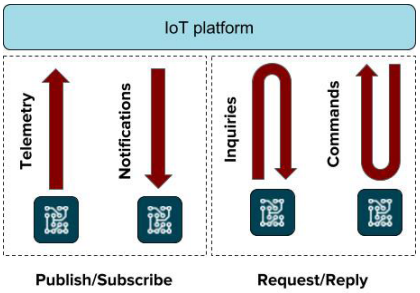

Figure 2: *IoT and messaging patterns.*

From a protocol perspective, EnMasse supports both AMQP 1.0 and MQTT (but adding HTTP support is on the roadmap). It's important to mention that they are open standard protocols, and EnMasse is focused on interoperability.

Furthermore, it allows developers to build a multi-tenancy IoT solution in which different tenants share the same infrastructure but are isolated from each other. This kind of feature is really useful when a company adopting EnMasse to develop an IoT platform can provide a "free" plan for end users (maybe with some limitations in terms of connected devices and throughput). Such a plan could be set up with a single EnMasse instance, which would use a single messaging infrastructure but use multi-tenancy to separate communications and data flows between different users.

Finally, it provides both internal and external security: all the internal EnMasse components are connected to each other using encrypted connections with TLS protocol and, at same time, the "external" devices and clients can connect to the messaging infrastructure in the same way with the same degree of security. Of course, all the parties are authenticated and authorized on exchanging messages, and this feature is provided using the Keycloak project as the identity management system.

THE INTERNAL COMPONENTS AND THEIR ORCHESTRATION

EnMasse is made by different open-source components connected to each other in order to provide the overall infrastructure.

The "entry" point of the entire system is an AMQP 1.0 router network based on the Apache Qpid Dispatch Router project. It's really about "routing" messages, but at the application level (with AMQP) instead of the usual network level (with TCP/IP). As an "entry" point, all the

clients connect to such a network for exchanging messages in different ways, as we'll see in the next section. Only MQTT clients represent an exception because they connect to the messaging infrastructure through a corresponding gateway bridging MQTT to AMQP.

Going through these routers, the messages are directly delivered from producers to consumers while, in order to store them, EnMasse leverages the Apache ActiveMQ Artemis broker project. Behind the routers, one or more brokers can be deployed in order to provide storage. Of course, it's not only about storing but also about forwarding messages from the brokers to the clients with the usual messaging "entities" like queues and topics.

An admin console provides many different features, from deploying the needed components (routers and brokers) based on the addresses and messaging patterns that the user wants to a web UI for monitoring the messaging infrastructure (i.e. connected clients, messages, and throughput).

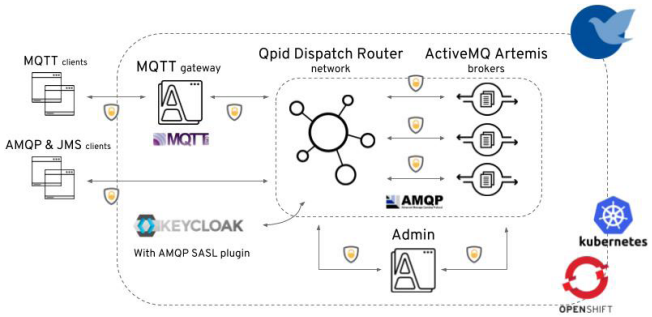

Figure 3: *EnMasse architecture.*

What has not been mentioned yet is that the EnMasse platform runs on Kubernetes and OpenShift in order to be elastic, scalable, and resilient and to effectively handle failures.

All the described components are provided as Docker images that can run as containers orchestrated by Kubernetes/OpenShift on a related cluster. Deploying EnMasse is really simple, only leaving the developers and the operators with the need to handle higher-level concepts from the messaging world.

Running on Kubernetes/OpenShift gives developers the flexibility to move the messaging infrastructure and the related IoT platform built on top of it from an on-premise solution to the cloud. Kubernetes and OpenShift (using the Origin project or the OpenShift Container Platform) can be installed "locally" for development and testing purposes or even on a few servers on bare metal.

If needed, such a solution can be simply moved to the cloud on any provider such as Amazon AWS, Microsoft Azure, or Google GCE using one or more virtual machines in order to deploy a Kubernetes/OpenShift cluster. EnMasse will run in the same way as it did on-premise. Other than using VMs and managing your own cluster, these cloud providers offer Kubernetes clusters "as a service," which deploys a "managed" cluster for you without the need to install all the Kubernetes components on the VMs.

It's really simple to move an IoT solution based on EnMasse from on-premise to the cloud and from one provider to another in the cloud itself, avoiding the "vendor lock-in" problem.

DIRECT OR STORE-AND-FORWARD MESSAGING?

Thanks to its internal architecture based on routers and brokers, EnMasse provides two different messaging mechanisms: **direct messaging** and **store and forward**.

The direct messaging mechanism is not new. AMQP 1.0 is a peer-to-peer protocol, so you can have two clients directly connected each other. In this way, the producer is able to send a message only when the consumer is online (providing "credits"), and it receives an acknowledgement, which means that the consumer has received the message. Between the two parties, there is a single contract so that the producer knows that the message is received by the consumer when it gets feedback.

EnMasse provides this kind of mechanism in a more reliable and scalable way through the router network, where routers are connected to each other making a "mesh" — the clients aren't connected directly but instead through this network. Every router, unlike a broker, doesn't take ownership of the message but just forwards it to the next stop in the network in order to reach its destination. If a router goes offline, the network is automatically reconfigured in order to identify a new path for reaching the consumer; it means that high availability is provided in terms of path redundancy. Furthermore, as it happens in a real direct connection with AMQP 1.0, a producer doesn't get "credits" from a router for sending messages if the consumer isn't online or can't process more messages. Finally, direct messaging is synchronous by nature, so it's really useful for RPC communication.

The entity used for identifying how producers and consumers exchange messages is the address, as it's defined by the AMQP 1.0 specification as just a string.

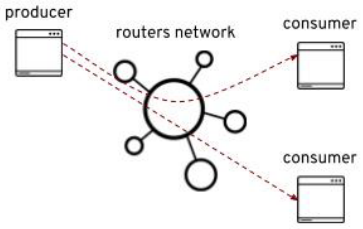

Figure 4: *Direct messaging.*

The store and forward mechanism is provided by the brokers behind the router's network in two different steps. First of all, a broker takes ownership of the received message, storing it internally (just in memory or in a persistent way), but it doesn't mean that such a

message is immediately forwarded to the final consumer, which could be offline in that moment. It allows asynchronous communication and time decoupling because the consumer can get the message later and at its own pace, which can be different from the producer. There is always a double contract between producer-broker and broker-consumer, so the producer knows that the message reached the broker, but not the consumer (a new message exchange on the opposite direction is needed for having acknowledgement from the consumer).

The entities used for storing messages are queues and topics, which allow point-to-point (or competing consumers) and publish-subscribe patterns. In any case, the name of a queue or a topic is just a string, like an AMQP 1.0 address.

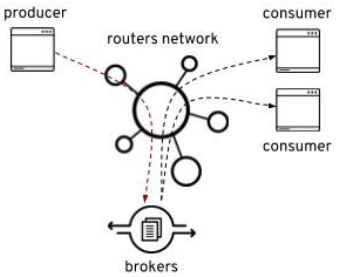

Figure 5: *Store and forward.*

From a client's perspective, when they connect to EnMasse through the unique entry point (which is the router's network), they just connect to an address for exchanging messages; they don't know that a broker could be behind such an address with a corresponding queue or topic.

The EnMasse operator has to define the addresses and their semantics in order to support the different messaging mechanisms as described above. Using the related console, it's really simple and it's a matter of defining what kind of pattern the clients need. The supported address types are the following:

- **Queue**: Backed by a broker for "store and forward" and providing point-to-point (competing consumer) patterns.
- **Topic**: Backed by a broker for "store and forward" and providing publish/subscribe patterns.
- **Anycast**: Similar to a queue in terms of direct messaging. A producer can send messages to such an address only when one or more consumers are listening on it and the router's network will deliver them in a competing consumer fashion (from a round-robin way to a more sophisticated one based on load balancing).
- **Multicast**: Similar to a topic in terms of direct messaging, it has a producer publish messages to more consumers listening on the same address so that all of them receive the same message.

In an IoT-specific use case, the main two communication patterns, like telemetry, command, or control, could be implemented in a few different ways.

In the telemetry scenario, a device could be enabled to send telemetry data only if a backend service is online and able to get such data; we don't care about data if no one is able to process it, and we are not interested in storing the related messages. In this case, using direct messaging is the right solution. On the other side, it's possible that we want to ingest data from devices even when no services are running (or maybe they are busy) for processing. In this case, store and forward is the way to go, putting messages inside queues or topics (depending on whether we want to distribute messages to one or more services in parallel).

In the command and control scenario, we could have the same two approaches. In one case, we want to send a command to a device only if it's online and we are sure that it can execute (at least receive) the command itself in that moment. Direct messaging can help with this use case. On the other side, it's possible that we want to handle situations in which the device isn't online, but we want it to execute the command when it comes back online. In this case, the command message needs to be stored for later delivery. In order to avoid sending "stale" commands to a device that comes back online too late (for the command), the message can have a related TTL (time to leave) so that it disappears from the queue on expiration.

GETTING STARTED

The official EnMasse website provides a great documentation section for digging into all the features and getting started with the platform, but it's worth showing here how easy it is to deploy and use it locally on OpenShift.

The first step is to get the OpenShift client tools needed to interact with the OpenShift cluster; you can download the client from the OpenShift Origin project.

If you don't have an OpenShift cluster already running, you can deploy one just using the `oc cluster up` command or using the Minishift project.

From the EnMasse releases page, you can download the latest release and unpack it locally. It provides a script for deploying EnMasse on top of the OpenShift cluster with just the following command:

```
./deploy-openshift.sh -m "https://localhost:8443" -n enmasse
```

(**Note**: The "localhost" refers to an OpenShift cluster running locally; if you are using Minishift, you have to use the related virtual machine IP address instead.)

The deployment will take some time to have all the components up; the EnMasse platform is running in a single tenant mode and without authentication.

After that, you can log into the OpenShift console, select the "enmasse" project where EnMasse was deployed, and open the web console to create a new address.

For a simple example, let's create a new address using "anycast" as type.

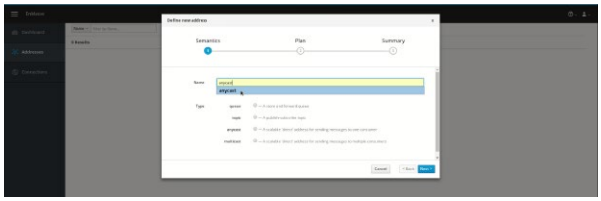

Figure 6: *Address "anycast" creation from the web UI console.*

The way that EnMasse is exposed outside of the cluster is through OpenShift routes.

Finally, the simpler way to have a couple of clients exchanging messages via AMQP through the created address is to use the following Python sender and receiver.

To start a receiver, just run:

```
./simple_recv.py -a "amqps://$(oc get route -o
jsonpath='{.spec.host}' messaging):443/anycast" -m 10
```

The receiver will block until it has received ten messages.

To start the sender, just run:

```
./simple_send.py -a "amqps://$(oc get route -o
jsonpath='{.spec.host}' messaging):443/anycast" -m 10
```

You should see all the messages ssent and received on the other side.

CONCLUSION

As described, EnMasse provides a really powerful messaging infrastructure for developing an end-to-end IoT solution with the needed scalability. Today, one of the main open-source projects focused on IoT devices connectivity is already using EnMasse: Eclipse Hono.

Hono provides a uniform interface as a set of well-defined APIs for telemetry, events, and command/control for connecting a large number of IoT devices. One of the available Hono deployments relies on EnMasse for the underlying messaging infrastructure. These projects together and their integration show how open source is driving the IoT world.

Written by Paolo Patierno

Paolo is a Principal Software Engineer working for Red Hat on the messaging and IoT team. He has been working on different integration projects using AMQP with Apache Kafka and Spark, and on the EnMasse messaging-as-a-service project about integration with MQTT. Currently, he is focusing on Apache Kafka and how to deploy and run it on Kubernetes and OpenShift. In the IoT space he takes part in the definition of the API for the Eclipse Hono project. He is also a committer for Eclipse Paho and a maintainer for different IoT-related components in Eclipse Vert.x.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300