Universitat Oberta de Catalunya

uoc.edu

Event Streaming Open Network

Emiliano Spinella
Master en Ingeniería Informática
Open Networks

Amadeu Albós Raya
Joan Manuel Marquès Puig

28th December 2021

Index

1. Introduction

1.1. Context and justification of this work

After the global crisis caused by COVID-19, companies have begun to recognize the important role real-time analytics play when going through intense and disruptive crises. These analytics enable a broad range of use cases that significantly enhance decision making when relevant business events happen (i.e., from keeping abreast of supply-chain issues all the way to ensuring timely deliveries to customers' homes) (Gartner, 2021).

Currently, counting with business events information in real time has been possible despite its high economic barrier of entry, although only for organizations with an adequate budget for it. Additionally, these types of solutions enable continuous artificial intelligence systems which, in turn, empower decision-making for different functions within the organizations.

However, the current scenario of available infrastructure for real-time data streaming is basically composed of isolated private offers. These non-homogeneous offers integrate different technological stacks to solve needs which organizations or individuals are willing to pay for. However, Frischmann (Frischmann, 2007) defends that this regime might stifle "*the generation of positive externalities through the downstream production of public goods and non-market goods*".

As Frischmann argues, "*the general value of commons as a resource management principle is that it maintains openness, does not discriminate among users or uses of the resource, and eliminates the need to obtain approval or a license to use the resource*". A great example of IT common resources that satisfy these requirements are the World-Wide-Web as well as the Email Service.

We can mention email as an enabler of an innumerable quantity of downstream producers of public and non-market goods. Public goods include all the direct benefits individuals obtain by being able to connect with people and institutions as well as how companies and governments obtain greater efficiency by using it. Also, individuals are indirectly benefited by this increase in efficiency in the public and private sector. On the other hand, non-market goods are also generated by the downstream producers of email (i.e., a government increased efficiency to reduce air pollution).

In this context, the need of adopting a common infrastructure that efficiently solves the management of data in real time arises. Moreover, it is also necessary to provide flexibility and dynamism as well as allowing creative innovations to be built on top of this common infrastructure. Basically, a common resource infrastructure would enable the generation of a wide range of productive processes. The outputs of these processes would be public and nonmarket goods that generate positive externalities that benefit society as a whole.

Thus, this work will focus on the establishment of an open access network or commons network for real time event data streaming. The overall objective is to

solve the problems at hand as well as to enable downstream innovation opportunities in entrepreneurship and many other activities.

1.2. Objectives of this work

First, the main objective of this work is to apply the main principles of open access infrastructures to define a real-time event data streaming open network. According to Navarro (Navarro, 2001) these principles are:

- Non-discriminatory and open access: Access is open because everybody has the right to join and use the infrastructure according to the access rules.
- Open participation: Everybody has the right to join the community to participate in the construction, operation, provision, and governance of the infrastructure.

Secondly, it will also be required to research and select the most optimal existing software for this open event-streaming network. In this sense, all considered software shall be open source to remove all barriers of entry regarding licensing costs.

Thirdly, explore the need to design protocols and software components to achieve the common resource management principle. All products generated, whether documentation or software code, shall also be licensed as open source.

Finally, this work aims to establish the foundation for a standard for event data streaming in the same way email protocols conform a standard for mail interchange. In the future, the foundation established by this document could take the form of an Internet Society RFC, in the same way as email protocols like SMTP, IMAP, etc.

1.3. Approach and methodology

The approach to follow on this work is composed of the following phases:

1. **Research** Open Access Networks principles and Event Streaming technologies.
2. **Analyze** the output of the research with the focus of establishing an open access network for event streaming.
3. **Design a system architecture** that enables open participation for Event Streaming and provides extensibility capabilities to participants.
4. **Describe the components of the architecture** that should be developed highlighting both software and communications requirements.
5. **Identify existing software and protocols that can be leveraged** for the architecture components.
6. **Describe the high-level implementation requirements** that should be considered when building the architecture components.
7. **Propose an initial implementation strategy** that covers the requirements and maximizes its leverage on existing technologies.
8. **Describe a strategy to prove the concept** of an Event Streaming Open Network before developing a complete implementation.

1.4. Work Breakdown Structure

We provide the work planning in Figure 1.

Task Name	Duration	Start	Finish	Predecessors
Master's Final Work	**78 days?**	**Wed 9/29/21**	**Fri 1/14/22**	
Reseach Phase	**6 days**	**Wed 9/29/21**	**Wed 10/6/21**	
Research on Event Stream technologies and market	3 days	Wed 9/29/21	Fri 10/1/21	
Research on Open Networks principles	3 days	Mon 10/4/21	Wed 10/6/21	3
Analysis Phase	**6 days**	**Thu 10/7/21**	**Thu 10/14/21**	
Analyze the requirements for an Event Streaming Open Network	1 day	Thu 10/7/21	Thu 10/7/21	4
Specify requirements	3 days	Fri 10/8/21	Tue 10/12/21	6
Prioritize requirements	2 days	Wed 10/13/21	Thu 10/14/21	7
Initial Design Phase	**4 days**	**Fri 10/15/21**	**Wed 10/20/21**	
Architectural design for the Open Network	2 days	Fri 10/15/21	Mon 10/18/21	8
Select open-source tecnologies to leverage	1 day	Tue 10/19/21	Tue 10/19/21	10
Define and design required components and/or network protocols	1 day	Wed 10/20/21	Wed 10/20/21	11
Progress Report	**5 days**	**Wed 10/20/21**	**Tue 10/26/21**	
Write Chapter 1: Event data streaming uprising	1 day	Wed 10/20/21	Wed 10/20/21	11
Write Chapter 2: Event streaming adoption and the need of an open network	1 day	Thu 10/21/21	Thu 10/21/21	14
Write Chapter 3: Event Streaming Open Network Architectural Design	1 day	Fri 10/22/21	Fri 10/22/21	15
Write Chapter 4: Required software components to be built	1 day	Mon 10/25/21	Mon 10/25/21	16
Write Chapter 5: Required network protocols to be designed	1 day	Tue 10/26/21	Tue 10/26/21	17
PEC2 Submission	1 day	Tue 10/26/21	Tue 10/26/21	
Final Design Phase	**15 days**	**Wed 10/27/21**	**Tue 11/16/21**	
Define and design required components and/or network protocols	5 days	Wed 10/27/21	Tue 11/2/21	19
Deploy initial MVP in the cloud for demonstration purposes	10 days	Wed 11/3/21	Tue 11/16/21	21
Progress Report	**5 days**	**Wed 11/17/21**	**Tue 11/23/21**	
Write Chapter 6: How organizations can participate in the network	1 day	Wed 11/17/21	Wed 11/17/21	22
Write Chapter 7: How individual users can participate in the network	1 day	Thu 11/18/21	Thu 11/18/21	24
Write Chapter 8: A glimpse on the potential positive externalities for society	2 days	Fri 11/19/21	Mon 11/22/21	25
Write Conclusion	1 day	Tue 11/23/21	Tue 11/23/21	26
PEC3 Submission	1 day	Tue 11/23/21	Tue 11/23/21	
Corrections and fine-tuning	25 days	Wed 11/24/21	Tue 12/28/21	28
PEC4 Submission	1 day	Tue 12/28/21	Tue 12/28/21	
Presentation Video	**5 days**	**Wed 12/29/21**	**Tue 1/4/22**	
Write script	3 days	Wed 12/29/21	Fri 12/31/21	30
Record and edit video	2 days	Mon 1/3/22	Tue 1/4/22	32
Presentation Video Submission	1 day	Tue 1/4/22	Tue 1/4/22	
Final Defense	5 days	Mon 1/10/22	Fri 1/14/22	
Master's Final Work Done	1 day	Fri 1/14/22	Fri 1/14/22	

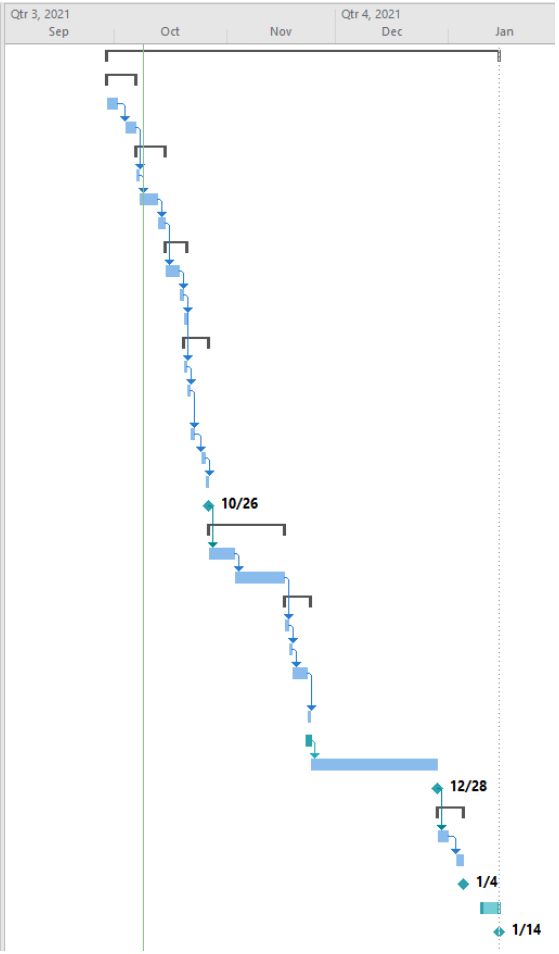

Figure 1. Work planning and Gantt chart.

1.5. Summary of products obtained

The main product of this work will be the design of the open network for event data streams. This overall design includes:

1. Overall architecture of the network, including computing infrastructure and networking requirements.
2. Selection of open-source tools leveraged for different functions of the network.
3. Definition of specialized protocols for network management data interchange.
4. Functional description of required software components whose behavior cannot be leveraged by existing tools.

Additionally, a demonstration environment shall be deployed in the cloud as a proof of concept.

1.6. Brief description of the other chapters of the report

Chapter 1: The Emergence of Event Streaming
Describe the current landscape of event streaming technologies focusing on the paradigm shift that event-driven architectures impose on software development.

Chapter 2: The need for an Open Network
Delineate the current state of event streaming adoption and justify the need of an open network.

Chapter 3: Event Streaming Open Network Architecture
Proposal of an initial architectural design for an event streaming open network. This proposed design includes the participant nodes software components, computing infrastructure needs and networking resources requirements.

Chapter 4: Event Streaming Open Network Protocol
Design description of the protocols that the network requires for administrative data interchange. The design includes the recommended networking protocol stack to be used in the network.

Chapter 5: Implementation approach
A thorough description of an initial implementation is provided, which includes the rationale for leveraging existing open components. Also, a description is provided for the software components that need to be built from scratch.

Chapter 6: Proof of Concept
Implementation of the main components with minimum functionalities that prove the feasibility of the overall Event Streaming Open Network.

Chapter 7: Summary and Conclusions
Summary of the results of this work together with some insights for improvement and potential impact on society.

2. The Emergence of Event Streaming

2.1. Introduction to Event Streaming

Event Streaming is a concept that requires some explanation. Currently, there is no clear consensus on the definition of Event Streaming, but we can find it in technology articles as well as in market analysis reports. While there are scientific articles about the implementation of this technology, these studies were made by the academic sector for exceptional use cases, like the Event Streaming Service for ATLAS in LHC[1]. Thus, we will consider more relevant to this work the market definitions for event streaming.

In this sense, analysts at Gartner mentions in a report named "*How to Identify Your Event-Driven Architecture Use Cases to Select the Best-Fit Event Broker*" (Guttridge, 2021) that:

"Event streaming aims to provide reliable event ingestion and distribution using numerous data sources, including web browsers, desktop clients and Internet of Things (IoT) devices and providing that data to subscribers for processing."

One important difference to bear in mind is that all event streaming is data streaming but not vice versa. This is basically because of the broad generality of the concept of data. To the contrary, an event is a very well-defined structure of data. For data to be an event means that the information conveyed includes context details. In this line, James Urquhart's book "*Flow Architectures*" provides an excellent explanation for this difference between data and events.

In Figure 2 we can see a diagram both for a data stream and an event stream. In the former data is sent as it arrives through the network without adding any context. The raw data stream imposes on the consumer the need to add context information when the data arrives. This context details addition can be achieved by different techniques: (i) knowing the source of the stream (i.e., the MAC address of the device), (ii) looking for hints in the data stream itself (i.e., GPS latitude and longitude), (iii) capturing a timestamp from the consumer's own clock to identify when a state change occurred; etc.

[1] ATLAS is one of two general-purpose detectors at the Large Hadron Collider (LHC). It investigates a wide range of physics, from the search for the Higgs boson to extra dimensions and particles that could make up dark matter. The ATLAS experiment at the LHC is gradually transitioning from the traditional file-based processing model to dynamic workflow management at the event level with the ATLAS Event Service (AES). The AES assigns fine-grained processing jobs to workers and streams out the data in quasi-real time, ensuring fully efficient utilization of all resources, including the most volatile.

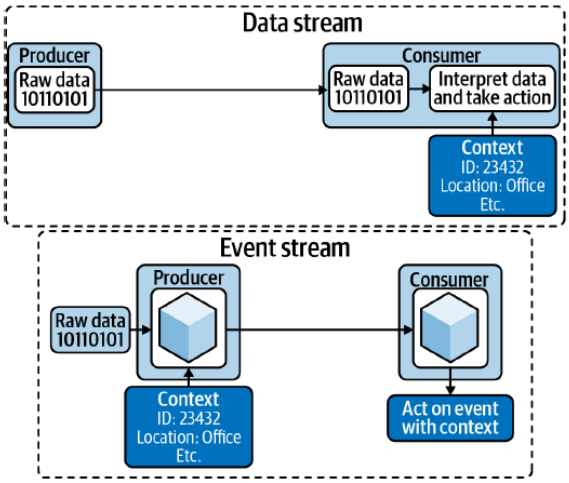

Figure 2. Difference between a data stream and an events stream.

In the case of event streaming, the producer must assure that there are context details included within the payload itself. In Urquhart's opinion, this fact is what turns a data stream into an event stream. The context added to the transmitted data allows the consumer to better understand the nature of the data. Also, this greatly simplifies the behavior to be programmed in the consumer, avoiding the need of additional functionalities to understand when and where the event occurred. Moreover, according to the author, event streaming will dominate streams used to integrate systems across organization boundaries.

Nevertheless, the consumer must be able to interact with the producer and interpret the received data. The producer must be able to produce events and to transmit them in a format known by the consumer. Thus, there are two main needs for this to be accomplished: (i) an interface by which the consumer can contact and initiate a connection with the producer and (ii) a protocol by which the producer and consumer agree to format, package, and transport the data.

2.2. State of the art of Event Streaming

When a concrete use case is detected that can benefit from Event Streaming, it is needed to determine the different components that are involved. We can consider that one application has access to a resource for which it notifies changes as events to other actors. This application will be named as the Producer. On the other hand, we will have another application (or set of applications) interested in these events, which we will name Consumers.

Currently, the state of the art for Event Streaming includes a middle-man system, called Event Broker, which is in charge of receiving events from producers and making them available for consumers. We can see this architecture in Figure 3.

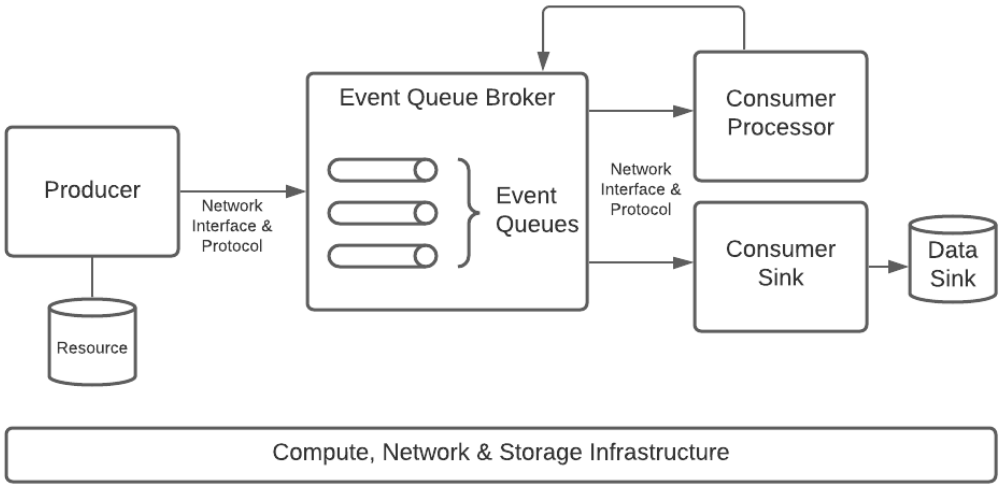

Figure 3. Simple architecture of an Event Streaming integration

In the rest of this section, we will use Figure 3 to describe all the components required for an Event Streaming implementation

2.2.1. Infrastructure

In Figure 3, we can see a common layer of infrastructure for all components. This layer includes the capabilities of compute, network and storage, which are needed to implement an overall Event Streaming solution. Moreover, this layer involves hardware, software, servers, storage and network devices, network protocols, operating systems, databases as well as all required operational tools for them to work cohesively.

The main promise of the infrastructure layer is to deliver core network, storage and compute resources on which event-driven computing is built upon. Additionally, we can consider these capabilities to be in the commodity phase, meaning that there are widely adopted standards for virtualization, container management, databases querying languages, etc.

Nevertheless, it's relevant to consider how public and private cloud computing provide this layer of capabilities given the connected nature of event streams. Some examples of public cloud computing providers and private cloud ISVs (Independent Software Vendors) are:

Public Cloud providers:

1. **Amazon Web Services, Microsoft Azure & Google Cloud Compute**
   These vendors provide computing infrastructure as a utility by means of a broad portfolio of Platform as a Service components, ranging from server hosting to data analytics and services for Internet of Things.

Private Cloud ISVs:

1. **VMware ESX, Microsoft Hyper-V, XEN, KVM**
   Virtualization is the base infrastructure allocation technologies for on-premises data centers, meaning it enables private clouds.
2. **Hewlett-Packard, Dell, Cisco, Juniper, Netapp, Hitachi**
   For large Data Centers or on-premises installations, these companies provide networking and storage components.

For both Public and Private Cloud, we must also mention **Kubernetes**, an open-source project of the CNCF (Cloud Native Computing Foundation). The CNCF is a non-profit organization that supports and curates the development of many distributed technologies. Kubernetes provides distributed deployment and an operation platform for container-based workloads. Over the last years, Kubernetes has become the de facto standard for container orchestration and management, and it has been implemented both by Public Cloud providers as well as Private Cloud ISVs:

- Amazon Web Services provides EKS (Elastic Kubernetes Service)
- Google Cloud Compute provides GKE (Google Kubernetes Engine)
- Azure provides AKS (Azure Kubernetes Service)
- VMWare provides support for Kubernetes in its hypervisor product called vSphere.

2.2.2. Integration Components

To integrate a stream of events between producers and consumers, a network-based integration mechanism is needed. This mechanism will allow both the producers and consumers to be disparate, meaning that they can be instances of programs developed in different programming languages, with separate code bases. In the same way we can develop a SMTP client using Java or Python, we can have producers built in Java and consumers built in Python, or any other mix.

However, it's important to notice that a network-based integration like this requires (i) an **interface** that enables defining, creating and controlling stream connections; and (ii) a **protocol** that defines how and when data will be transmitted between the two actors (producer and consumer), including formatting and flow control.

Regarding the **interface**, there is not common consensus on a specification yet. Nevertheless, there are standards such as HTTP, WebSocket and MQTT that define how to establish and maintain streaming connections. On the other hand, there's also the need for an API that enables publish-and-subscribe relationships. Therefore, there are four popular protocols that serve this purpose: MQTT, AMQP, Apache Kafka and RabbitMQ. None of these are yet a commodity standard for the goal of publish-and-subscribe needs. Urquhart argues that the adoption of Event Streaming largely depends on counting with a technology for this purpose in the commodity phase.

Finally, Discovery APIs are largely nonexistent, except for those cases of proprietary solutions. In this sense, Discovery as an Event Streaming component is its Genesis phase.

2.2.2.1. Interfaces

Network interfaces include low-level mechanisms that define the data structure for the communication among producers and consumers. These interfaces are APIs (Advanced Programming Interfaces) that enable event streaming. Software uses streaming APIs to initiate connections with other software at a level of abstraction that both parties understand.

Urquhart defines two interfaces needed to enable consumers to identify usable streams and connect to them:

1. **Logical connection Interface:**
   This interface establishes a contract between the producer and the consumer about how data is to be structured and delivered. This will allow consumers to send authentication information to establish subscriptions, manage subscriptions while valid, and close subscriptions when no longer needed.
2. **Discovery Interface**
   A Discovery Interface would be helpful to understand what event streams are available for consumption from a given producer. This interface would also be employed to determine the qualities of a stream, such as required technology, metrics about stream volume, payload schemas, financial fees, etc.

Some examples of current interfaces follow:

1. **Apache Kafka Consumer API, Apache Kafka Connect**
   Apache Kafka is an open-source project of the Apache Foundation. It is currently widely used in commercial streaming solutions and its API is a critical piece to show what is possible with Event Streaming today. The Consumer API is the one used by external entities that connect with an Apache Kafka instance to subscribe to event streams. Apache Connect is a framework built upon the Consumer API and it's meant to simplify the interaction with Apache Kafka in common high-throughput scenarios.
2. **EdgeX API**
   EdgeX Foundry is an open-source platform specification for industrial IoT applications. It defines a common set of services as well as an API for devices and supporting services to communicate with one another.
3. **CNCF CloudEvents Subscription API**
   CloudEvents is a specification for describing events that shows great promise for an Event Streaming. It's defined as a common metadata model that can be mapped to any number of connections or protocols. CloudEvents is simple and capable of carrying a wide variety of payload types.

2.2.2.2. Protocols

The protocol component refers to the agreed-to criteria by which a producer and a consumer will exchange and interpret event streams. There are two subcomponents worthy of mentioning out separately:

1. **Metadata format**
   Protocol format for describing metadata that can be used to understand payload formatting, encrypt/decrypt payloads, understand origin, etc.
2. **Payload format**
   Protocol format for understanding the specific data payloads sent by a producer. These formats will vary significantly from use case to use case, but standard payload format may be defined for common streams in each industry or market.

Currently, we can find the following technologies providing the Protocol component:

1. **CNCF CloudEvents**
   As mentioned previously, CloudEvents is a project of the CNCF. Regarding protocols, it's aiming to allow the event processing across different vendors. For instance, processing events produced by AWS SQS in a private Apache Kafka deployment. It is currently binding HTTP, MQTT, Apache Kafka and NATS. Also, it defines a set of metadata, and the bindings define specific structures that combine metadata and payload data for the target protocol or interface.
2. **NATS Client Protocol**
   NATS (Neural Autonomic Transport System) is a CNCF project that delivers simple, secure messaging for several high-performance uses. However, NATS counts with a simple protocol for publishing and subscribing to a topic of interest or queue. It is worth mentioning that while NATS is an open-source project, the protocol is proprietary.
3. **MQTT (Message Queueing Telemetry Transport)**
   MQTT is a lightweight publish-and-subscribe protocol that handles many of the same functions as the NATS Client protocol does. Additionally, it incorporates features to manage functionalities like expiration, rate limiting, length limiting as well as other control functionalities. MQTT is widely adopted mainly in IoT environments, and it is vendor neutral.
4. **AMQP (Advanced Message Queueing Protocol)**
   AMQP has been defined by OASIS (Organization for the Advancement of Structured Information Standards), a standards body focused on promoting product-independent data standards like HTML and XML. AMQP is focused on enabling common message exchange between any message-aware software, regardless of use case, networking environment, etc. AMQP expressly declares it's goal to be the integration of organizations.
5. **HTTP (Hypertext Transfer Protocol)**
   HTTP is the de facto standard for the World Wide Web. Also, it is used by applications to interact with HTTP servers through REST APIs as well as for machine-to-machine communication, device operations, microservices, etc. The main inhibitor for HTTP to become the standard protocol for an Event Streaming is that it does not currently support a publish-subscribe method.
6. **WebSocket**
   WebSocket enables bidirectional communication between actors over HTTP. The fact that it is built on top of HTTP provides great compatibility

with most of the internet security and network services. WebSocket could be used to carry formats like CloudEvents.

2.2.3. Interaction Components

The Interaction Components are inspired in modern Event-Driven Architecture nomenclature involved in creating value on both ends of a connection. These components implement a function that can be generalized in:

- **Source components**: software that collects data which can be hardware devices with sensors, webhooks, API, databases, etc. Any component that gathers data to put into a stream can be considered a Source. There is a broad variety of possible sources, ranging from IoT devices to databases. Thus, any piece of software that receives data in a non-stream format and places that data in an event stream is a Source.
- **Processor components**: software whose main behavior is to listen to an event stream, process it and return output events. Processes may act both as consumer and producer.
- **Queue components**: mechanisms for collecting, storing, and forwarding streaming events on behalf of producers and consumers. Basically, queues are like buffers that enable producers to deliver their event stream payloads without direct interaction with the consumer. Also, queues enable consumers to receive event data only when they are ready and available to process it. The main advantage of queues is that they decouple the producer from the consumer -or consumers- of the event stream while enabling asynchronous communications and independent operations.
- **Sink components**: software whose purpose is to display or store the results of processor components. This can be an application that delivers event data to a dashboard or into a database. It also represents that the stream stops in a sink.

As a conclusion, in Figure 4 we can see all the components that have been described as the state of the art of Event Streaming. In this diagram the color determines the stage of commoditization of the different components. Green represents that the component is closer to be considered a commodity, while red means that there are currently proprietary solutions in place.

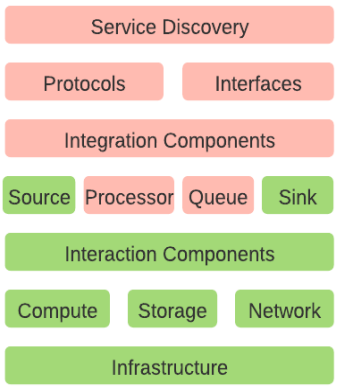

Figure 4. Current maturity stage for the Event Streaming components.

3. The need for an Event Streaming Open Network

According to Urquhart (Urquhart, 2021), Event Streaming plays a key role in how the economic system evolves. Society is rapidly digitalizing and automating the exchanges of value that constitute the economy. Also, considerable time and energy is spent to assure that key transactions can be executed with reduced human involvement with better, faster, and more accurate results.

However, most of the integrations executed today across organizational boundaries are not in real time and they currently require employing mostly proprietary formats and protocols. On the other hand, some industries have adopted data formats for exchanging information between organizations, such as Electronic Data Interchange (EDI). However, those integrations are limited to specific use cases and represent a small fraction of all needed organizational integrations.

Even when application programming interfaces exist for event streaming, these are largely proprietary. For instance, Twitter offers an API for consuming social media streams, but it is not implemented by other parties. Thus, there is no consistent and common consensus on a mechanism for the exchange of events across organizations. This results in a completely custom landscape for real-time cross-organization integration. In this scenario, development teams must invest plenty of time into understanding and defining a common interface for data exchange.

In this context, we can now introduce how this landscape would radically change with the adoption of an Event Streaming Open Network. When needing to integrate real-time information across organizations, developers would have a common basis for finding, publishing, and subscribing to event streams. Also, given a set of standard formats to encode and transmit events, developers could use the programming language of their choice.

Overall, this set of standards would drastically reduce the cost of real-time integration, which would also enable experimentation by users. This experimentation can create an innovation space for new uses of event streaming.

3.1. Necessities for broad Event Streaming adoption

In this section, we will describe the main needs for the broad adoption of Event Streaming. The focus will be made on detecting and describing the missing capabilities that could not only enable but also accelerate the event data integration among different organizations. The different necessities detailed in this section will serve as input for an architecture design.

3.1.1. Necessity 1: Availability of an Events Public Registry

A public registry of an organization's available event streams does not exist. We will argue in this section why this is the core component that an Event Streaming Open Network can provide.

Nowadays, when an organization needs to publish an event stream or event flow, they usually follow some form of the following steps:

1. Develop and deploy a producer application that writes events to a queue.
2. Create all necessary networking permissions for external public access to the queue.
3. Inform the remote user the access information (i.e., Hostname/IP, protocol, and port) together with the required client details and technology for accessing the stream (i.e., Apache Kafka Protocol, RabbitMQ API, etc.).
4. Create credentials for consumer authentication and authorization access to the queue.
5. Develop and deploy a consumer application that reads the queue.

Now, we can compare this process to a simple email interaction:

1. Sender opens a graphical Mail User Agent application and sends an email to an email address formatted as user@domain.
2. The message is sent to an SMTP server that routes it to the destination SMTP servers for the given domain. Once received, the message is put into the user mailbox.
3. When the recipient checks its mailbox by IMAP or POP3, the new email is transferred to the Mail User Agent.

In these two scenarios, we can see that the information needed to be exchanged offline by the actors is completely different in size and content.

First, in the case of email, there is a shared naming space given by the Domain Name Service (DNS). The email format has been standardized by the IETF in RFC 5321, section 2.3.11. Thus, there is a common naming space that is used for referencing mailboxes in the format user@domain. Thus, the offline details communicated by the peers is only the recipient email address. There is no analogous standard nor an open alternative for Event Streaming.

Therefore, in the case of Event Streaming, users need to perform plenty of offline communication to agree not only on the technology to use but also on the queue to use. For instance, two organizations may be currently using Apache Kafka and need to share an event stream among themselves. The organization having the source of the stream should provide the following details to the consumer organization:

- **Bootstrap servers:** Fully Qualified Domain Name list of the Apache Kafka brokers to start the connection to the Apache Kafka Brokers. Example: tcp://kf1.cluster.emiliano.ar:9092, tcp://kf2.cluster.emiliano.ar:9092, tcp://kf3.cluster.emiliano.ar:9092
- **Topic or Queue name**: name of the topic resource in the Apache Kafka Cluster
- **Authentication information:** User and password, TLS Certificate, etc.

In the case these organizations were not both using Apache Kafka, the use case cannot be simply solved without incurring in development or complex configurations as well as adopting proprietary components.

We can conclude that an Event Streaming Open Network should provide a global accessible URI for streams in a similar fashion than email, to reduce offline developers' interactions. This means being able to name event streams in a common naming space like DNS, as well as providing a mechanism for users to discover the location and connections requirements.

3.1.2. Necessity 2: Establishment of a User Space for Events

Another need for broad adoption is due to the inexistence of a common and agreed user convention. In the general literature, we cannot find reference to the types of users that would consume or produce events to and from an event stream.

In this sense, it is also appropriate to consider the email use case. Basically, an email user only needs to know the email address, the password, the URL of a web mail client or the details of IMAP/POP3 server connection. Once the user has this information, it's possible to access an email space or mailbox where the user can navigate the emails in it. Also, IMAP provides the possibility for the user to create folders and optionally share them with other users.

There is no analogous service currently available for Event Streaming analogous to the email case. This means that the user concept in Event Streaming is limited to authentication and authorization. Thus, the user does not have access to a "streambox". The result is the impossibility for a person or an application to possess a home directory containing all the streams owned by the user.

As a conclusion for this section, we can mention that it is necessary to embrace a user space resource for Event Streaming. This resource should not only solve the users' motivations and requirements but also reduce the offline verbal communications and custom development dependencies. In the next sections, we will refer to this component as the Event User Space Service.

3.1.3. Necessity 3: An Agnostic Subscription Protocol

A third need for wide adoption is an agnostic protocol to manage subscriptions to event streams. For this need to be solved, it would be necessary first to count with an Event User Space Service. Then, in case a user has created a stream and wants to enable public subscriptions by other users, there is no general protocol to inform other parties of this subscription intention nor its confirmation.

The result is the inability for the users to seamlessly subscribe to an event stream. They either must employ protocols like MQTT or, in the need of employing other application protocols like Apache Kafka, hardcode the subscription details in the different software implementations. This means that there is no general subscription protocol for Event Streaming that is agnostic of the application protocol employed. This protocol implements both the Metadata Payload Format and Payload Format.

A good example to illustrate the difference between a control protocol that implements a Metadata Payload Format from a payload protocol that implements a Payload Format is how SIP (Session Initiation Protocol) works with RTP (Real Time Protocol) to provide VoIP capabilities. The former is a control protocol that initiates and maintains a session or call while the latter is the one responsible for carrying the payloads, which in the case of VoIP it would be coded audio.

Consequently, a similar definition of protocols could potentially mitigate this limitation for Event Streaming. If a protocol can be used to establish and maintain the subscriptions relationships while another different protocol is used for the events payload, all the current application protocols implementations could be supported.

Additionally, by counting also with an Event Streaming Public Registry, it would be possible to provide URI for streams in a similar way as email works with the “mailto” URI. For instance, in web pages one can find that email addresses are linked to mailto URIs which, when clicked, open the default email user application (i.e., Microsoft Outlook) to send an email to the referenced email address.

If a user counts with a user space or streambox, then a user application like an email client could provide access to it. Then, if the user clicks on a link of a stream URI (i.e. “stream:myeventflow”), the streambox application would open and subscribe to the given stream.

Currently, the Metadata Payload Format as well as the Payload Format are both provided by the queue or log application protocol. In the case of Apache Kafka, both formats are implemented within the Apache Kafka Protocol. This introduces a barrier for interoperability among different technologies, meaning that flows of event data cannot be seamlessly connected, without relying on custom development or proprietary software licensing.

We can conclude that there is an actual need for an open specification of an Event Subscription Service for event streams, which implements what Urquhart calls Metadata Payload Format. This specification could be materialized in a

network protocol that introduces an abstraction for the event queue or log technologies implemented by different organizations.

3.1.4. Necessity 4: An Open Cross-sector Payload Format

Currently, the different implementations of Event Streaming combine both the Payload Format with the Metadata Format. This means that the same protocol utilized for payload transport is used for subscription management.

In Figure 5, we can see how these two formats are combined in the case of Apache Kafka.

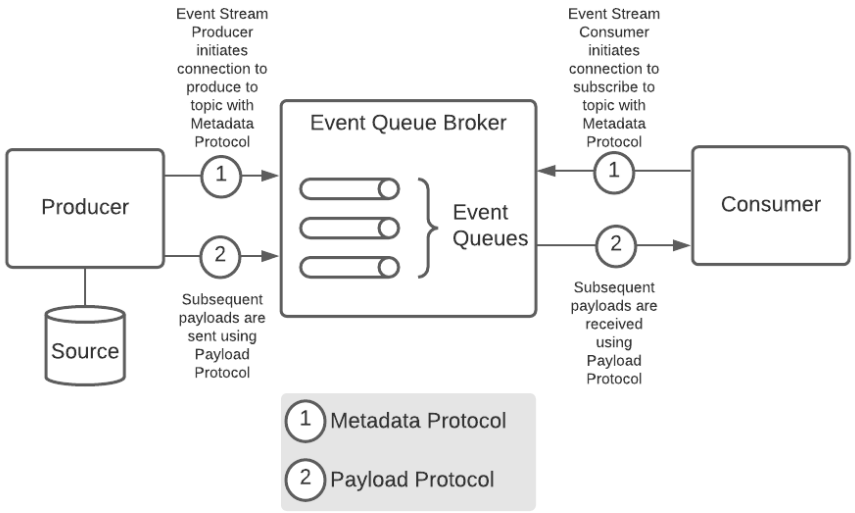

Figure 5. Apache Kafka Protocol combines the metadata and the payload formats.

When a producer intends to publish events to a queue or, using Apache Kafka terminology, when a producer intends to write records to a topic, first it needs to initiate a connection to at least one of the Apache Kafka Brokers. In that initial exchange of TCP packages, the producer is authenticated, authorized, and informed with topic details. This set of transactions would belong to a protocol that implements a Metadata Payload Format. Afterwards, when the Producer starts writing the events to the topic, it encapsulates the event payload in a Kafka Protocol message. This latter behavior makes use of a Payload Format. Thus, we can observe how both theoretical formats are coupled in a single protocol. Similar behavior of a coupled Metadata and Payload Format in one single protocol happens also in AMQP, MQTT and RabbitMQ.

As for the consumer, the behavior is the same with the difference that the initial intention is to subscribe to a queue or, in Apache Kafka terminology, to consume records of a topic. Then, a set of TCP packages encapsulating the Apache Kafka protocol authenticates, authorizes, and informs the Consumer with topic details for consumption. Afterwards, the consumer can start polling for new records in the different partitions of the topic. It is worth mentioning that the consumer needs

to implement more queue management logic than the Producer, especially when multiple replicas of a consumer type are deployed.

If we focus on the Payload Format, there is the need for an implementation-agnostic payload format suitable for Event Streaming. In this sense, CloudEvents project of the CNCF proposes a specification and a set of libraries for this purpose. The goal is to use CloudEvents specification as a Payload Format regardless of the Payload Protocol being used. For instance, we could transmit events in the CloudEvents format using the Kafka or AMQP Protocol.

In Figure 6 we can observe an UML diagram of the CloudEvents object that could be used to serialize the data before the producer sends the event, as well as to deserialize it once a consumer receives it.

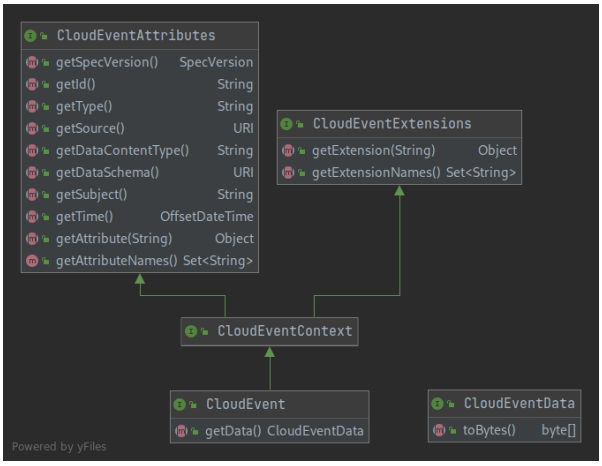

Figure 6. CloudEvents Payload Format UML diagram.

The general structure of the CloudEvents Payload Format includes a standardized methodology to include event data in an event message. For instance, instead of defining a customized JSON structure for sending the events of temperature changes measured by a device, a CloudEvent object could be used. Temperature could be included as an attribute in the CloudEvent object.

We can then conclude that while there is no current protocol candidate that implements the Metadata Format, CloudEvents is a good candidate for the Payload Format needed in an Event Streaming Open Network. In this way, the different CloudEvents libraries made available in several programming could be leveraged.

3.2. Solving Event Streaming necessities with an Open Network

In the previous sections, we described the main different necessities currently inhibiting the broad adoption of Event Streaming. In this section we will argue how Internet standards are developed and why this is the case for an Event Streaming Open Network.

There are two main ways in which standards can be developed adopted:

1. First, there are organizations that produce standards for different sectors. For instance, there is the ISO (International Standards Organization) that develops and publishes worldwide technical, industrial, and commercial standards; the W3C (World Wide Web Consortium) which defines standards such as HTML or CSS; and the IETF (Internet Engineering Task Force), which produces documents called RFC (Request For Comments) that contain specifications for important Internet protocols such as DNS, FTP, SMTP, IMAP, POP3, IMAP, SIP, RTP, etc.

2. Secondly, we have the private sector that continuously innovates to efficiently and rapidly solve new problems. The goal of the private sector is not necessarily to provide open standards but to solve problems in more convenient way than their competitors. This does not eliminate the possibility that standards can be achieved, which can happen because of a broad adoption and the Network Effect.

Therefore, standards are produced both by standards organizations and by the market competition. While it is evident that most of the market products do not become standard, also not all standards defined by standards organizations become widely adopted.

An interesting example of this phenomenon is the case of ISDN (Integrated Services Digital Network), a set of communications standards for the transmission of voice, video, and data over the PSTN (Public Switched Telephone Network) developed by the ITU-T (Telecommunication Standardization Sector) in 1988. ISDN pretended to use the existing public telephone network to transmit digital data in a time when the Internet connectivity access was not as broadly available as it is today. The main competitor of this standard was the incipient Internet itself, which could be used to transmit the same data.

The Internet alternative needed a protocol to support the same services offered by ISDN, which was initially developed by the conjoint effort of the academic and private sector. Consequently, in 1992 the Mbone (Multicast Bone) was created. This project was an experimental network backbone built over the Internet for carrying multicast IP traffic, which could be used for multimedia content. After some important milestones of this project, the SIP (Session Initiation Protocol) was defined in 1996 and was published as a standard protocol in IETF's RFC-3261. The reality today is that SIP has completely won the standards battle for

multimedia transmission over the Internet, and ISDN usage has been on continuous decline.

This lesson teaches us that it is not enough to define standards if these are not implemented and broadly adopted. Also, it shows the need for open standards instead of proprietary specifications. However, having open standards is not enough to guarantee adoption and this will greatly depend on market factors. If a given problem can be quickly solved using existing open specifications and implementations, it may have chances of becoming a standard. Then, it does not really matter if initially there is no standards organization behind the specification as long as it is openly accessible. Afterwards, it could be officially standardized with the support of a standards organization.

As for Event Streaming, we see a similar scenario set-up in the market today. There are currently several open specifications and implementations for Event Streaming, like AMQP (Advanced Messaging Queueing Protocol), supported by RabbitMQ. However, while AMQP can be used for several purposes, Kafka Protocol specializes on Event Streaming Processing and its specialized features make it more convenient than RabbitMQ.

An Event Streaming Open Network would imply at least the definition of an open specification and an open-source implementation that solves the currently necessary components mentioned previously in this chapter.

However, there is a relevant difference between an Event Streaming Network with other open networks, like guifi.net. The reason is that guifi.net possesses governance over the network and there is a community behind for management and operation. In the case of Event Streaming, if we guide ourselves by the history of the most widely adopted protocols on the Internet, the governance should be similar to that of the World Wide Web or Email.

Both the World Wide Web and Email have open specifications as well as open-source implementations. We can mention the Apache Web Server as an open-source implementation of the HTTP protocol; Postfix for SMTP; and Bind for DNS. Nevertheless, the governance for these protocols' specifications relies on the IETF.

In order to define the characteristics of an Event Streaming Open Network, we will first focus on the definition of shared and openly accessible infrastructure. First, we will show how DNS complies with the criteria to be considered an infrastructure resource. Then, we will demonstrate how this is also true for Event Streaming. Finally, we will review the principles of Free, Open & Neutral Networks and why they should be followed for an Event Streaming Open Network.

3.2.1. Open Access Infrastructure Resources

The literature about Commons Infrastructure (Frischmann, 2007) defines a set of criteria to evaluate if a resource can be considered an infrastructure resource. This analysis is relevant since it can provide some arguments to prove the need of an infrastructure of commons for Event Streaming, which could then be materialized in an Open Network for Event Streaming. The demand-side criteria for evaluating if a given resource can be considered as an infrastructure resource are:

1. The resource can be consumed nonrivalrously.
2. Social demand for the resource is driven primarily by downstream productive activity that requires the resource as an input.
3. The resource is used as an input into a wide range of goods and services, including private goods, public goods and/or non-market goods.

First, a nonrival good describes the “shareable” nature of a given good. Infrastructures are shareable in the sense that the resources can be accessed and used by multiple users at the same time. However, infrastructure resources vary in their capacity to accommodate multiple users, and this variance in the capacity differentiates nonrival resources from partially rival resources. A nonrival resource represents those resources with infinite capacity, while a partially rival resource has finite but renewable capacity. As an example, Broadcast Television is a nonrival resource since additional users do not affect the capacity of the resource. On the other hand, natural oil resources are completely rival since its availability is limited and it is not renewable. In the middle, we have partially rival resources like a highway, which may be congested. This last characteristic is also true for the Internet since it supports additional users without degrading the service to existing users to a certain extent.

Secondly, infrastructure resources consumption is primarily driven by downstream activities that require this resource as an input. This means that the broad audience consumes infrastructure resources indirectly. For instance, highway infrastructure is used to transport every kind of physical good which people and organizations purchase. This facilitates the generation of positive externalities for society through the downstream production of public goods and non-market goods. These positive externalities might be suppressed under a regime where resource availability is driven solely based on individuals’ willingness to pay.

Regarding willingness to pay, it is relevant to analyze this factor more exhaustively. Frischmann states that if infrastructure access is allocated based on individuals’ willingness to pay the potential positive externalities of that infrastructure might be stifled. Thus, infrastructure resources behave differently than end-user products: if the former are made available solely based on the end-user demands and willingness to pay, those needed infrastructure resources might never be made available. As an example, we can mention that if airports were built based on individuals’ willingness to pay for them, they might not even be built. However, individuals are willing to pay for the airport's downstream

activities, such as purchasing a flight or consuming air-transported goods. Then, a whole set of positive externalities are generated by the existence of an airport in a city.

In the third place, infrastructure resources are used as input for a wide range of outputs. This criterion emphasizes both the variance of the downstream outputs and their nature. Thus, the infrastructure resources possess a high level of *genericness* which enable productive activities that produce different goods with high variance. If we consider how an airport complies with this criterion, we can mention that not only airports serve individuals that need to travel by air but are also used to transport many kinds of physical goods. These goods then enable other activities throughout the downstream value chain. Then, the output variance of the activities that take airport infrastructure as input is significantly high.

3.2.1.1 Open Access DNS Resource Example

Now, we will provide as an example how DNS complies with these criteria and why it can be considered an infrastructure resource.

1. DNS infrastructure is a partially rival resource because individuals and organizations can register domains in the Domain Name addressing space. It is partially rival because not every actor can acquire the same domain name. However, the access to registering domain names is open and non-discriminatory. Moreover, DNS is also prone to congestion, which emphasizes its partially rival nature.
2. DNS infrastructure demand is driven principally by downstream products and services. An average Internet user is not paying directly for this infrastructure, but all the Internet services the user consumes pay for DNS infrastructure. This is true for all the Internet services due to the ubiquitous nature of DNS infrastructure.
3. All Internet services take as input DNS infrastructure and produce a broad variety of outputs, which then generate positive externalities to society as a whole by means of private goods, public goods and/or non-market goods.

We can conclude that DNS complies with Frischmann criteria for being considered as an infrastructure resource. The resource is represented both by the domain name that can be and by the querying capacity of DNS servers.

3.2.1.2 Flow: Open Event Streaming Resource

Now, we can evaluate how an Event Streaming Open Network can comply with the infrastructure resource criteria together with the FONN principles..

To begin with, we need to define what elements could be considered as infrastructure resources in an Event Streaming Open Network. First, the resource must be capable of delivering streams of events to consumers. Secondly, it must also permit producers to write events to the stream. Thirdly, each stream must be identifiable (i.e., URI) and able to be located (i.e., URL). **From now on, we will use “Flow” to refer to the infrastructure resource of an Event Streaming Open Network.**

The first Frischmann criterion requires the resource to be consumed nonrivalrously. Complete nonrivalrously for any Internet Service cannot be achieved due to the possibility of congestion and potential unavailability of different elements of the network. The same would be true for a Flow resource. Moreover, the public naming addressing space for Flows would be limited to the same level as that of domain names.

We will continue now with the third criterion. To illustrate the potential of Flow being used as inputs for downstream activities, we will refer to Urquhart's vision for Event Streaming. He lists two areas in which significant changes can happen:

1. The use of time-critical data for customer experience and efficiency. This is driven because today's consumers are increasingly expecting great experiences, and organizations are almost always motivated to improve the efficiency of their operations.

2. The emergence of new businesses and business models. Businesses and institutions will quickly discover use cases where data processed in a timely manner will change the economics of a process or transaction. They may even experiment with new processes, made possible by this timely data flow. Thus, flow resources will also enable innovation. These innovations are responsible for generating positive externalities.

Then, we have demonstrated why Flow resources can be considered as infrastructure resources using Frischmann's Demand-side Theory of Infrastructure. These resources can be managed in an open manner to maximize positive externalities, which basically means maintaining its open access, not discriminating, and eliminating the need to obtain licenses to use the resources. Consequently, managing infrastructure resources in this manner eliminates the need to rely on either market actors or governments.

Lastly, the adoption of an Event Streaming Open Network implies taking Flow resources as inputs for productive activities. These inputs would then be used downstream to generate private goods, public goods and/or non-market goods. Additionally, we can assure that most of the consumers of Flow would not directly consume Flow resources. They would consume the outputs of downstream activities that use Flow as input. Again, the consumers may not be willing to pay for Flow resources directly.

We can conclude this section mentioning that an Event Streaming Open Network would enable one infrastructure resource called Flow. The access to this resource can be managed in an openly manner: maintaining open access, not discriminating users or different uses of the resource, and eliminating the need to obtain approval or a license to use the resource.

3.2.2. Free, Open & Neutral Networks (FONN)

The main principles of a Free, Open & Neutral Network are:

- It is open because it is universally open to the participation of everybody without any kind of exclusion nor discrimination, and because it is always described how it works and its components, enabling everyone to improve it.

- It is free because everybody can use it for whatever purpose and enjoy it independently of his network participation degree.

- it is neutral because the network is independent of the contents, it does not influence them and they can freely circulate; the users can access and produce contents independently to their financial capacity or their social condition. The new contents produced are orientated to stimulate new ones, or for the network administration itself, or simply in exercise of the freedom of adding new contents, but not to replace or to to block other ones.

- It is also neutral with regard to the technology, the network can be built with whatever technology chosen by the participants with the only limitations resulting of the technology itself.

3.2.2.1. Non-discriminatory and open access

Services such as DNS, the World Wide Web and Email do not discriminate and are open-accessible. Basically, people and organizations can access these networks as long as they can register an Internet Domain and host the required server components. Nowadays, there are alternatives to avoid having to register a domain name to have a web page or an email, such as Cloud WordPress Hosting or Gmail. However, we will focus on the network participants that provide services to end-users.

In the case of Guifi.net, we can highlight how this principle has been adopted in the fact that everybody can take part in the project without discrimination. Moreover, an emphasis is made in easing the participation of the disadvantaged collectives, with less resources or less opportunities to access information technologies, telecommunications, and the Internet.

An Event Streaming Open Network should provide resources in a similar way than the most widely adopted Internet Services. Thus, individuals and organizations must be able to register Flow address spaces for which the existing DNS infrastructure could be leveraged. Moreover, the specification of the protocols that implement the Metadata and Payload formats must also be openly accessible.

3.2.2.2. Open participation

Internet Services like DNS, WWW and Email provide individuals and organizations with different ways of participation. First, anybody can obtain the protocols' specification and build a custom implementation, which would result in a new product compatible with the protocols. Secondly, anybody can register a domain name and set up servers using compatible products. Thirdly, anybody can join and participate in the IETF, the institution that governs the specifications for these protocols.

As for Guifi.net, not only anybody can extend the network with new nodes but also can also participate in existing projects of network extension. Also, the participants can add services on top of the network such as VoIP, FTP servers, broadcast radios, etc.

Regarding active participation on an Event Streaming Open Network, we can highlight the possibility for individuals and organizations to expand the services provided by the open network. This extensibility could be made possible by different uses of the event payloads and will vary significantly depending on the sector. Since we have already proved how Flow is an infrastructure resource, innovation would play its part and its results would be materialized in services expansion.

We can conclude that the same kind of openness of DNS, WWW and Email is necessary for an Event Streaming Open Network. Anybody should be able to obtain the specifications to build an implementation of the service. Also, since it should leverage the DNS infrastructure, anybody would be able to register Flow address spaces. Lastly, the specification could be governed by an institution such as the IETF, due the dependency of Flow with other Internet Services governed by this institution.

4. Event Streaming Open Network Architecture

In this chapter, we will describe the overall architectural proposal for an Event Streaming Open Network. This description will include the different actors in play, the software components required, as well as the network protocols that should be specificized.

4.1 Architecture overview

In **Figure 7** we illustrate a high-level overview of an architecture proposal for the Open Network.

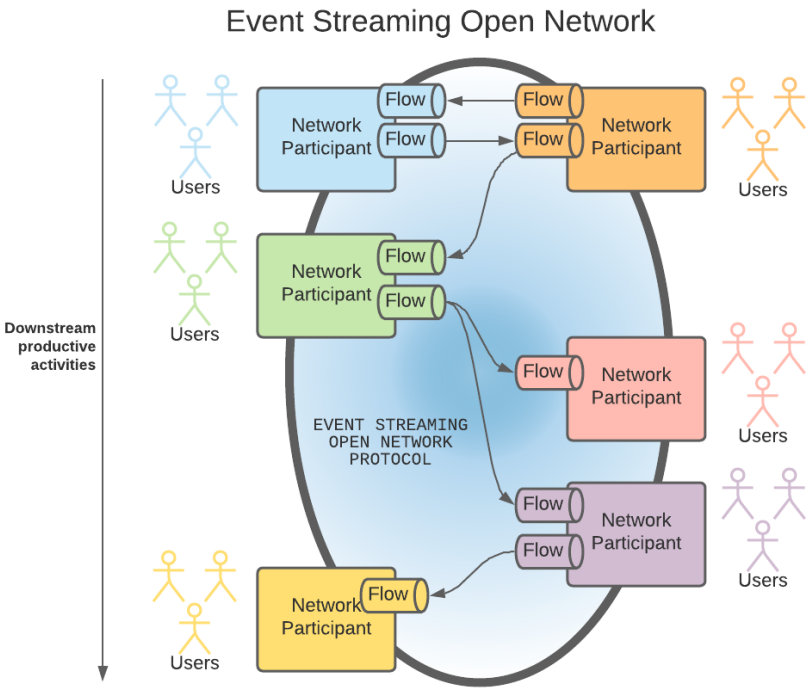

Figure 7. High-level overview of the Event Streaming Open Network

We can identify different **Network Participant (NP)** in **Figure 7** represented by different colors. **The different NPs act as equals when consuming or producing events as part of the Flows they own. All of NPs implement the Event Streaming Open Network Protocol, which Is described in the next chapter.**

In the diagram, an initial flow starts on the orange NP to which a user in the blue NP is subscribed. After processing the events received in the first flow, the results are published to a new flow in NP blue, to which the orange NP is subscribed as well. Now, the green participant is subscribed to the same flow, enabling downstream activities across the rest of the network participants.

It is possible to observe how the high-level architecture allows sharing the streaming of events across different network participants and their users. Also, there is also the need for security, in order to allow or deny the access to write to and read from flows.

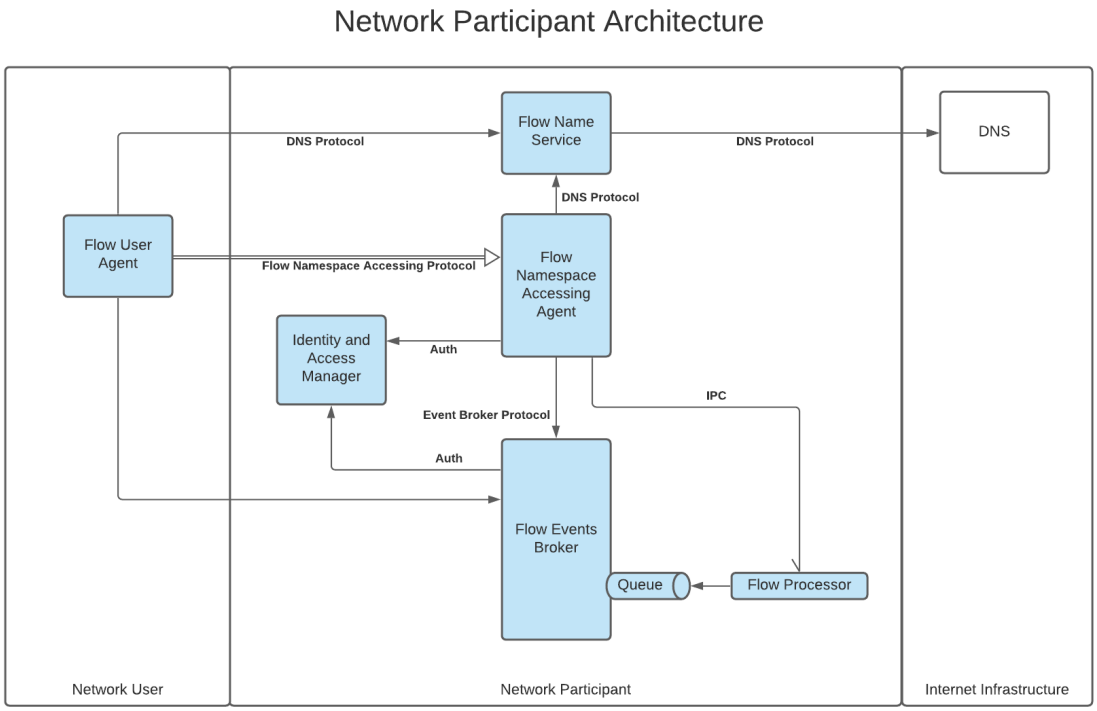

Figure 8. Event Streaming Open Network Architecture components

Regarding security**, the architecture considers the integration with an Identity & Access Management service**, which could implement popular protocols such as OAuth, SAML or SASL. However, the network should also enable anonymous access in the same way FTP does. This means that a given NP could publicly publish flow and allow any party to subscribe to it.

For example, nowadays the Network Time Protocol (NTP) is used to synchronize the day and time on servers. There are many NTP servers available that allow anonymous access, meaning that the service is openly available. The same must be considered for the Event Streaming Open Network.

Additionally, the NP must be able to expand the capacity to support any number of flows, as well as extending the network with new services. Not only NP must be able to include any given set of data within events but also, they must be able to build applications and services on top of the network by employing the architecture primitives.

Now, we provide a brief description of all the components that appear in the diagram of Figure 8. In the next sections further details of the components are provided.

- **Flow Events Broker (FEB):** a high-available and fault-tolerant service that provide queues to be consumed by network services, by users, and their applications. An example of an Event Queue Broker can be Apache Kafka, AWS SQS or Google Cloud PubSub. **The payload format implemented by these tools are what in 3.1.4 we called Event Streaming Payload Format.**

- **Flow Name Service (FNS):** a DNS-based registry that acts as an authoritative server for a set of domain names, which are used to represent flow addresses in a flow namespace. These domains contain all the necessary information to resolve flow names into flow network locations. **This component refers to what in 3.1.1 we named Event Streaming Registry.**

- **Flow Namespace User Agent (FNUA):** an application similar to User Mail Agents like Microsoft Outlook or Gmail. This application provides access to flow namespaces to users of the network.
  The definition of this component implies the specification of a dedicated protocol. We will refer to this protocol as **FNAP (Flow Namespace Accessing Protocol).**

- **Flow Namespace Accessing Agent (FNAA):** the server-side of the Flow Namespace User Agent. This component is the one that must provide convenient integration methods for GUI. **This component refers to what in 3.1.2 we named Event User Space Service.**
  This component must implement the same protocol selected for the Flow Namespace User Agent: **FNAP (Flow Namespace Accessing Protocol).**

- **Flow Processor (FP):** a flow processing instance used to set up subscriptions that connect local or remote flows on demand. **This component implements the processing part of what in 3.1.3 we called Event Subscription Service.** This component will be created and managed by a FNAA instance, and the communication is held through an Inter-process Communications (IPC) interface. Also, this service must implement an Event Payload Format, for which we will mainly consider CNCF's CloudEvents and Protobuf.

- **Flow Namespace Accessing Protocol (FNAP):** the protocol implemented in the Flow Namespace Accessing Agent as well as in the Flow Namespace User Agent. The former will act both as a server and a client while the latter only as a client. This protocol is described in the next chapter.

4.1.1. Flow Events Broker (FEB)

The FEB implementation that we will mostly consider is Apache Kafka. This open-source project is quickly becoming a commodity platform, and major cloud providers are building utilities for it. However, as a design decision, **it should be possible to use the same protocols to support other applications**, such as

RabbitMQ, Apache Pulsar or the cloud-based options like AWS SQS or Azure Events Hub.

Apache Kafka is the ecosystem leader in the Event Streaming space, considering mainly adoption. There is a growing set of tools and vendors supporting its installation, operation, and consumption. This fact makes Apache Kafka much more appealing to enterprise developers. However, the broker should provide a common set of functionalities which can be seen in the diagram of Figure 9.

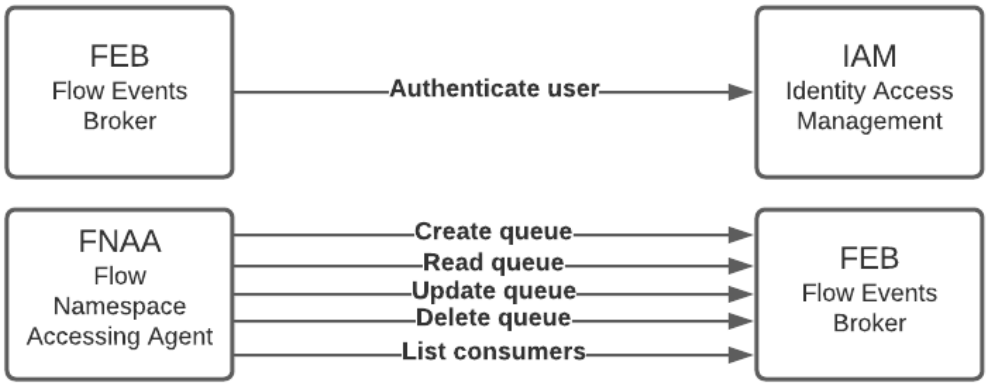

Figure 9. High-level overview of the Event Queue Broker component interactions.

The selection of the Events Broker will impact on the implementation of the Flow Namespace Accessing Agent. This last component will be responsible for knowing how to set up and manage flows on top of different Events Brokers.

4.1.2. Flow Name Service (FNS)

FNS is a core component for the overall proposed architecture. This component provides all needed functionalities for obtaining Flow connection details based on a Flow URI (Uniform Resource Identifier). Thus, it is required to define a URI format for Flow resources and to specify mechanisms for resource location resolution.

In this section, we will focus on describing both the URI for Flow as well as the DNS mechanism for obtaining Flow network location details.

4.1.2.1. Leveraging DNS infrastructure

As mentioned previously, this component must maximize its leverage on the existing Internet DNS infrastructure. The reason for this requirement is to avoid defining new protocols and services that prevent broad adoption. Currently, DNS is the *de facto* name resolution protocol for the Internet, and there exist libraries for its usage on every programming language.

Whereas DNS is mainly used to resolve FQDN (Fully Qualified Domain Names) into IP addresses, there are many other functionalities provided by the global DNS infrastructure. Theoretically, DNS is an open network of a distributed

database. Individuals and organizations that want to participate in the network need to register a domain name and set up Authoritative DNS servers for domains.

It is not in the scope of this work to detail the different available usages of DNS functionalities, but we can mention that it provides special Resource Records (i.e., types of information for a FQDN) that are solely used by special protocols. For instance, the MX Resource Records are used by SMTP servers to exchange email messages.

For the Flow Open Network, it will be required to define a URI format for flows as well as the mechanism to resolve an URI into all the needed information to connect to a flow. In the case of email, a URI is the email address while the connection details will be the SMTP server responsible for receiving emails for that account. For instance, an email URI could be user@domain.com while its connection details could be smtp://mail.domain.com. The way in which the connection details are obtained is by resolving the MX DNS Resource Records of domain.com, which in this example is mail.domain.com.

4.1.2.2. Flow URI

As we mentioned previously, the first needed element is a URI definition for flow resources. These resources identification must capture the following details:

- **Domain**, a registered domain in which create flow resources references. For example, **airport.com.**
- **Flow Namespace,** a subdomain which is solely used by users to host flow names. This subdomain must be delegated to the Flow Name Server component and desirable should not be used for any other purpose other than flow.
- **Flow Name**, a name for each flow that must be unique within its domain. The combination of flow name and flow domain results in an FQDN. For instance, we could have a flow named **arrivals** of the domain **flow.airport.com**. Thus, the FQDN of the flow would be **arrivals.flow.airport.com.** Also, the name can contain dots so that the following FQDN could be also used: **airline.arrivals.flow.airport.com.**

Thus, the general syntax of a flow URI would be:

flow://<flow_name>.<flow_namespace>.<domain>

This URI has the advantage that is similar to “mailto” URI and could be implemented in HTML to refer to flow resources. Some examples:

- flow://entrances.building.company.com
- flow://exits.building.company.com
- flow://temperature.house.mydomain.com
- flow://pressure.room1.office.mydomain.com

The flow URI must unequivocally identify a flow resource and provide, by means of DNS resolution mechanisms, all the information required to use the flow. Among these parameters, at least the following should be resolvable:

- **Event Queue Broker protocol** utilized by the flow. For instance, if Apache Kafka is used, the protocol would be “kafka”; In case RabbitMQ is used by the flow, “amqp”. Also, it must be informed if the protocol is protected by TLS.
- **Event Queue Broker FQDN** or list of FQDNs that resolve to the IP address of one or a set of the Event Queue Brokers. For instance, kafka-1.mycompany.com, kafka-2.mycompany.com.
- **Event Queue Broker Port** used by the Event Queue Brokers. For instance, in the case of Kafka: 9092, 9093.
- **Event Queue Broker Transport Security Layer** can be implemented. Thus, it is needed to know if the connection uses TLS before establishing it.
- **Queue Name** hosted in the Event Queue Broker, which must be equal to that of the corresponding flow name.

Refer to Annex C to see some examples of the URI definition.

4.1.2.3. Flow name resolution

In Figure 10, we can see how a Flow FQDN can be resolved by means of the Flow Name Service.

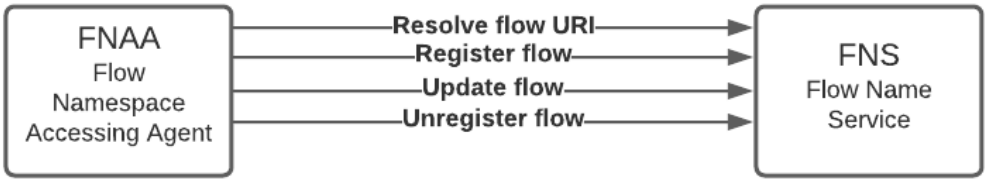

Figure 10. High-level overview of the interactions with the Flow Name Service component.

In order to illustrate the Flow Name resolution procedure by the FNAA (Flow Namespace Accessing Agent), we can consider the following flow URI:

flow://notifications.calendar.people.syndeno.com

First, the FNAA will perform a query to the DNS resolvers. These will perform a recursive DNS query to obtain the authoritative name servers for the Flow Namespace: people.syndeno.com. Thus, the authoritative name servers for syndeno.com will reply with one or more NS Resource Record containing the FQDN for the authoritative name servers of people.syndeno.com.

Secondly, once these name servers are obtained, the FNUA will perform a PTR query on the Flow FQDN adding a service discovery prefix. The response of the PTR query will return another FQDN compliant with SRV DNS Resource Records (RFC-2782) and DNS Service Discovery (RFC-6763).

Refer to Annex B to review the resolution process.

4.1.3. Flow Namespace Accessing Agent (FNAA)

The Flow Namespace Accessing Agent is the core component of a Network Participant. This server application implements the Flow Namespace Accessing Protocol that allows client connections.

In the diagram of Figure 11 we can see the different methods that the FNAA must support.

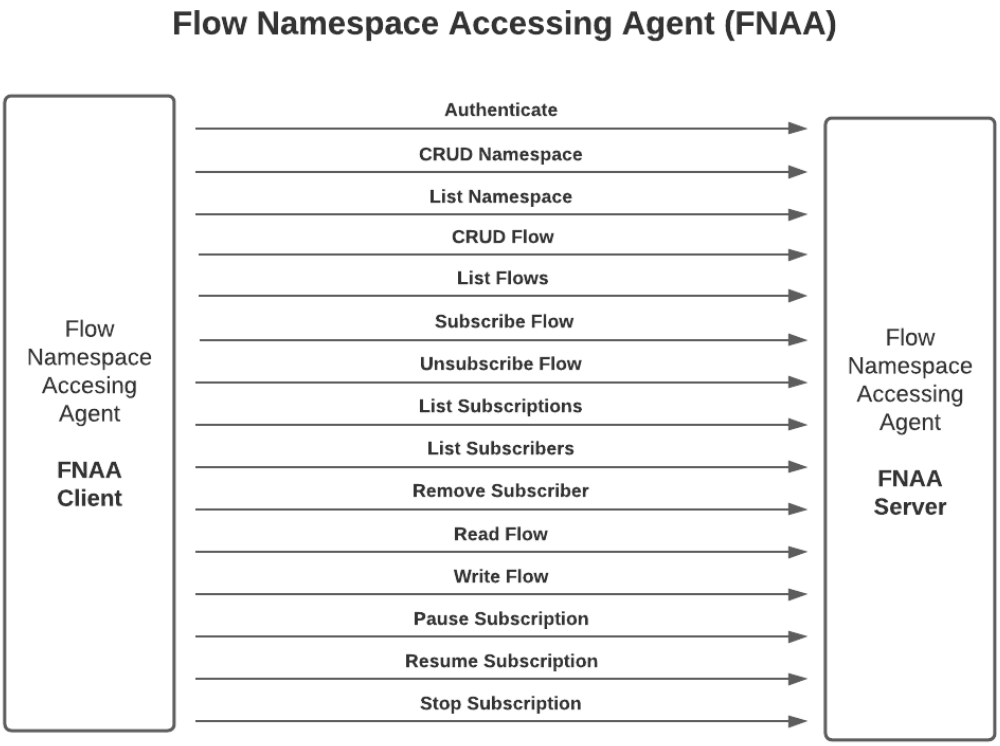

Figure 11. High-level overview of the interactions among FNAA servers.

The clients connecting to a FNAA server can be remote FNAA servers as well as FNUA. The rationale is that users of a NP connect to the FNAA by means of a FNUA. On the other hand, when a user triggers a new subscription creation, the FNAA of his NP must connect as client to a remote FNAA server.

4.1.4. Flow Processor (FP)

Whenever a new subscription creation is triggered and all remote flow connection details are obtained, the FNAA needs to set up a Processor for it. The communications of the FNAA to and from the FP is by means of an IPC interface. This means that there can be different implementations of Processors, one of which will be the Subscription Processor.

In the diagram of Figure 12, we can see the initial interface methods that should be implemented in a Flow Processor.

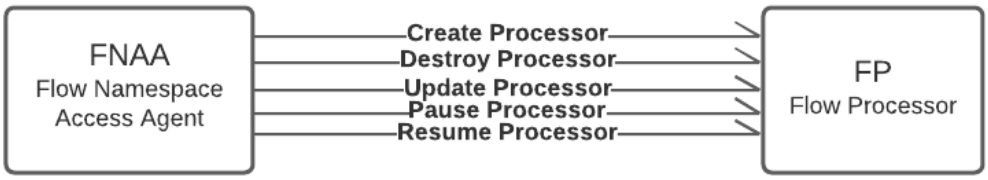

Figure 12. High-level overview of the IPC interface for the FNAA server and Flow Processors communications.

Depending on the use of the processor, different data structures should be added to the different methods. In the case of a Subscription Processor, the minimum information will be the remote and local Flow connection details. Moreover, the interface also should include methods to update the Processor configuration and to destroy it, once a subscription is revoked. Finally, due to the nature of the stream communication, there could also be methods available to pause and to resume a Processor.

There can be different types of Processors, which we can see in Figure 13.

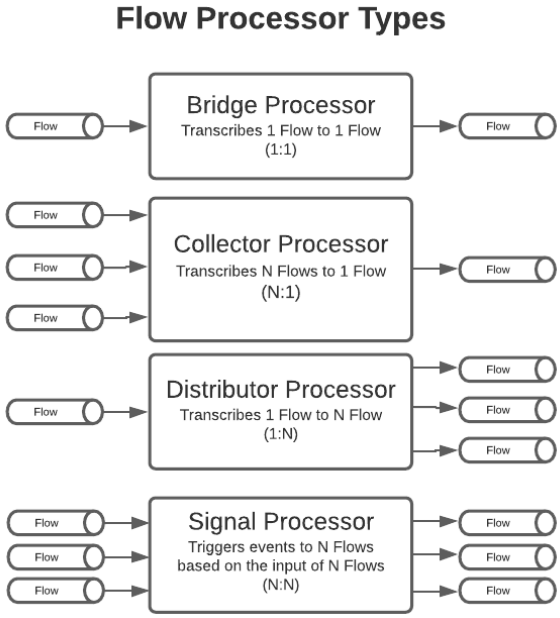

Figure 13. Different types of Flow Processors that could be supported by the FNAA server

In Figure 13, we can see that there are different types of Flow Processors:

- **Bridge Processor**: Consumes events from a Flow located in an Event Broker (i.e., Apache Kafka) and transcribes them to a single Flow (local or remote).
- **Collector Processor**: Consumes events from N Flows located in an Event Broker and transcribes the aggregate to a single Flow (local or remote).
- **Distributor Processor**: Consumes events from a single Flow and transcribes or broadcast to N Flows (local or remote).
- **Signal Processor**: Consumes events from N Flows and produces new events to N Flows (local or remote)

To implement the previously described Subscription Processor, we can utilize some form of the Bridge Processor. Although we are initially considering the basic use case of subscription, it must be possible for the network to extend the processor types supported. In any case, the different FNAA servers involved must be aware the supported processor types, with the goal of informing the users the capabilities available in the FNAA server. For instance, the fact that a FNAA supports the Bridge Processor should enable the subscription commands in the FNAA, for users to create subscriptions using the Bridge Processor.

In summary, the IPC interface should support all the possible processors that the network may need although we are initially considering the subscription use case.

4.1.5. Flow Namespace User Agent (FNUA)

The FNUA is an application analogous to email clients such as Microsoft Office or Gmail. These applications implement either different network protocols to access mailboxes by means of IMAP and/or POP3. In the case of FNUA, the protocol implemented is the FNAP (Flow Namespace Accessing Protocol).

The FNUA is an application that acts as a client for the FNAA server. Only users that possess accounts in a Network Participant should be able to login to FNAA to manage Flow Namespaces. The FNUA could be any kind of user application: web application, desktop application, mobile application or even a cli tool.

In the Diagram of Figure 14 we can see the actions that the user can request to the FNUA.

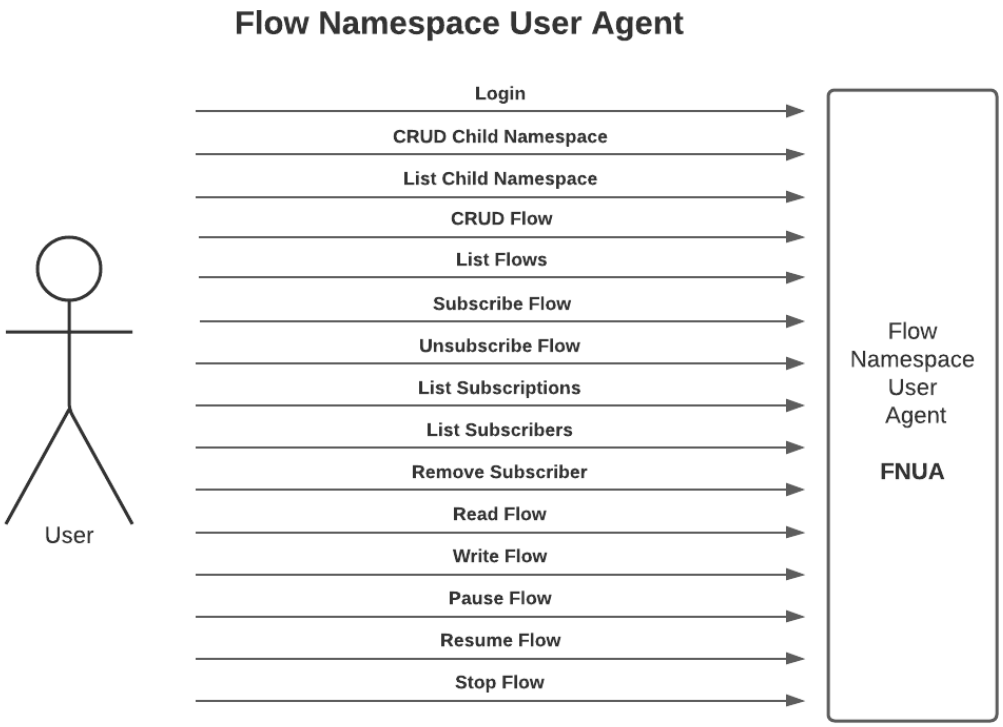

Figure 14. High-level overview of the interactions between a user and the Flow Namespace User Agent component.

The main goal of the FNUA is to provide the user with access to Flow Namespaces and the flows hosted in them. A user may have many Flow Namespace and many Flows in each of them. By means of the FNUA, the user can manage the Flow Namespaces and the Flows in them. Also, the FNUA will provide the capabilities required to subscribe to external Flows, whether local to the FNAA, local to the NP or remote (in a different NP FNAA server).

4.2. Communications Examples

In this section, two usage examples of Network Participants communications are provided. The first one, we call *unidirectional,* since one NP subscribes to a remote Flow of a different NP. The second one, we call it *bidirectional*, since now these NP have mutual subscriptions.

4.2.1 Unidirectional Subscription

In the diagram of Figure 15, we can see an integration between two NP. In this case, there is a FlowA hosted in the Orange NP to which the FlowB in the Blue NP is subscribed. Both FlowA and FlowB count with a queue hosted in the Flow Events Broker, which could be an Apache Kafka instance for example. However, it must be possible to employ any Flow Events Broker of the NP's choice.

The steps followed to set up a subscription to a remote flow are:

1. A user of the Blue NP creates a new subscription to remote FlowA by means of the Flow Namespace User Agent (FNUA).
2. The FNUA connects to the Flow Namespace Accessing Agent (FNAA) of the Blue NP to inform the user request.
3. The FNAA in the Blue NP discovers the remote FNAA to which it must connect to obtain the flow connection parameters. First, it needs to authenticate and, if allowed, the connection parameters will be returned.
4. Once the FNAA in the Blue NP has all the necessary information, it will set up a new Processor that connects the flow in the Orange NP to a flow in the Blue NP.
5. Once the subscription is brought up, every time a Producer in the Orange NP writes an event to FlowA, the Flow Processor will receive it, since it is subscribed to it. Then, the Flow Processor will write that event to FlowB in the Blue NP.
6. From now on, every Consumer connected to FlowB will receive the events published on FlowA.

In case the user owner of FlowA in the Orange NP wishes to revoke the access, it must be able to do so by means of security credentials revoking against the Identity & Access Manager of the Orange NP.

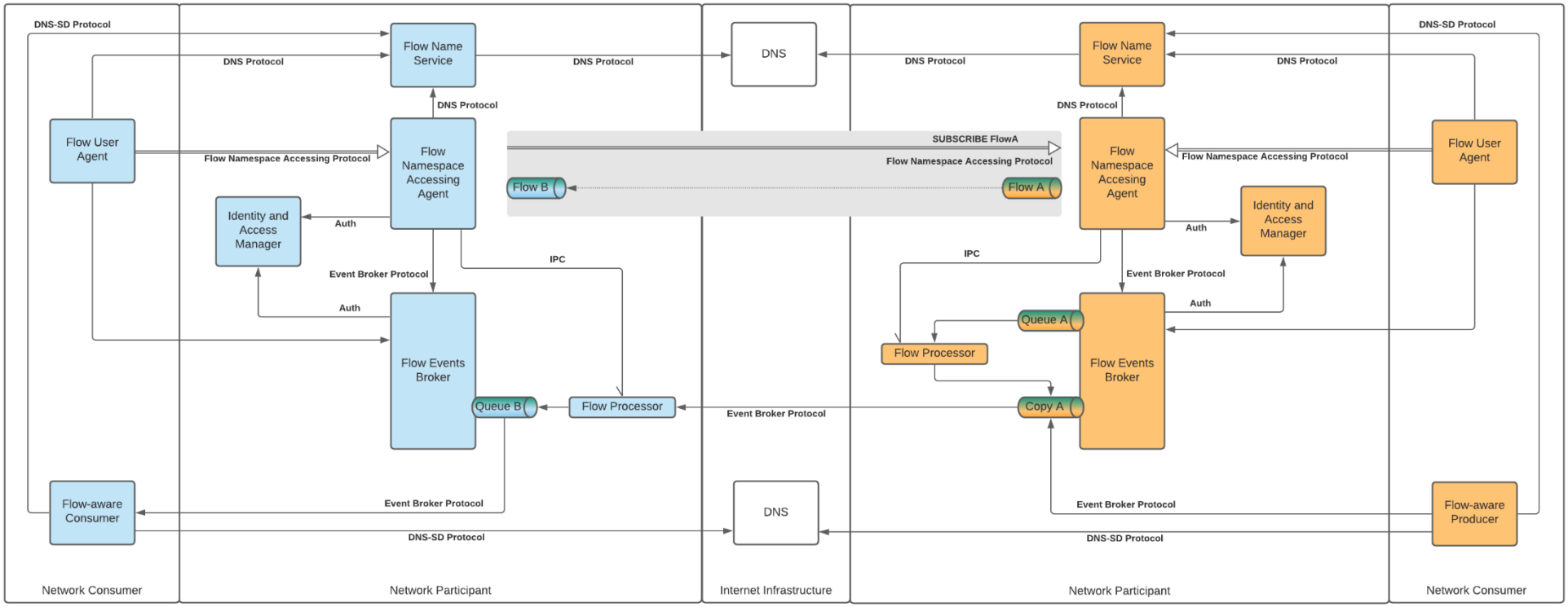

Figure 15. Example of a unidirectional subscription among two Network Participants.

4.2.2 Bidirectional Subscription

In Figure 16 we can see an example of all the components needed to set up a flow integration between two different NP. In this case, there are two flows being connected:

- FlowA of the Orange NP with FlowB of the Blue NP
- FlowC of the Blue NP with FlowD of the Orange NP

Each Flow has its corresponding Queue hosted in the NP Flow Events Broker. Also, there is one Flow Processor for each connection, meaning that these components are in charge of reading new events on source flows to write them to the destination flows as soon as received.

Also, we can see that in order to connect FlowB to FlowA, a connection from the Blue NP's FNAA has been initiated against the Orange NP's FNAA. This connection uses the FNAP to interchange the flow connection details. Analogously, the FNAA connection to set up the integration of FlowC with FlowD has been initiated by the Orange NP's FNAA.

After the flow connection details are obtained, the different Flow Processors are set up to consume and produce events from and to the corresponding Queue in the different NPs.

Once the two processors are initialized, all the events produced to FlowA in the Orange NP will be forwarded to FlowB in the Blue NP; and all the events produced to FlowC in the Blue NP will be forwarder to FlowD in the Orange NP.

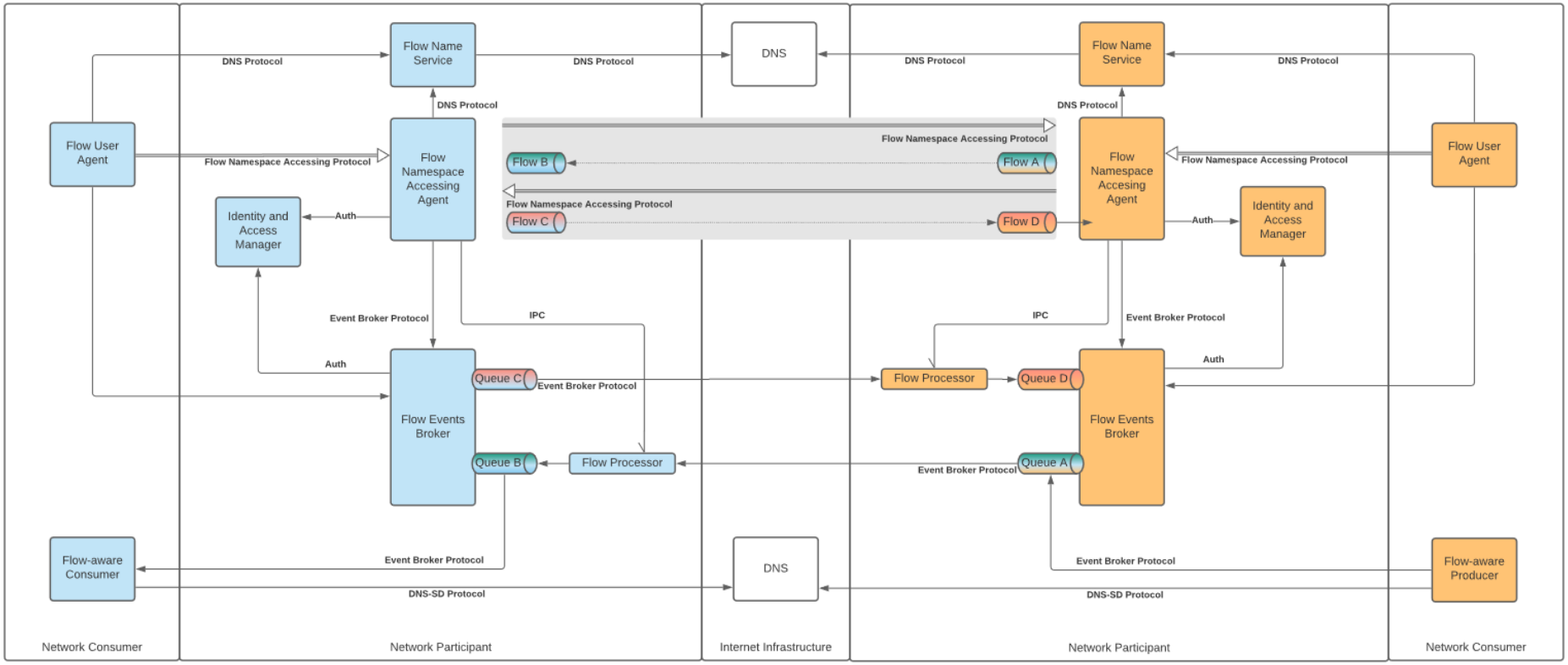

Figure 16. Example of a bidirectional subscription among two Network Participants.

5. Event Streaming Open Network Protocol

The protocol to be used in an Event Streaming Open Network is a key component of the overall architecture and design. This chapter is dedicated to thoroughly describe this protocol.

5.1. Protocol definition methodology

It is now necessary to specify the protocol needed for the Flow Namespace Accessing Agent or FNAA, which we have named the Flow Namespace Accessing Protocol or FNAP. In the diagram of Figure 17 we can see how an FNAA client connects with a FNAA server by means of the FNAP.

Figure 17. FNAA client and server communicate using FNAP.

In order to define a finite state machine for the protocol and the different stimuli that cause a change of state, the model presented by M.Wild (Wild, 2013) in her paper “Guided Merging of Sequence Diagrams” will be employed. This model is beneficial since it provides an integrated method both for client and server maintaining the stimuli relationship that trigger a change of state in each component.

In Figure 18 we have the method proposed by Wild for SMTP, in which there are boxes representing states and arrows representing transitions. Each transition has a label composed of the originating stimulus that triggers the transition and a subsequent stimulus effect triggered by the transition itself. For instance, when a client connects to an SMTP Server, the client goes from “idle” state to “conPend” state. The label of this transition includes “uCon” as the stimulus triggering the transition, which triggers the effect “sCon”. Then, on the diagram for the server we can see that the “sCon” triggers the transition from “waiting” state to “accepting” state in the server.

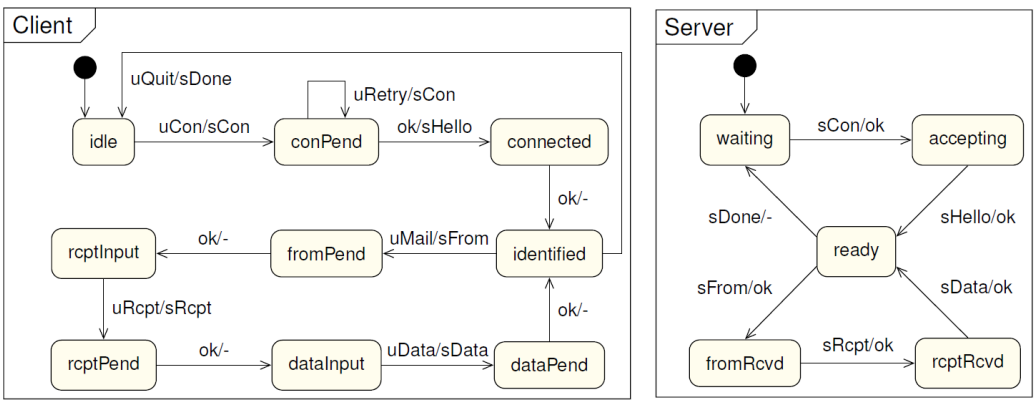

Figure 18. Merged Sequence Diagram for SMTP proposed by Wild, 2013.

This method will be used to define the states and transitions for the Flow Namespace Accessing Protocol both for client and server.

5.2. Flow Namespace Accessing Protocol (FNAP)

Using the model proposed by Wild described previously, we define the finite-state machine for the FNAA Server, which we can see in Figure 19.

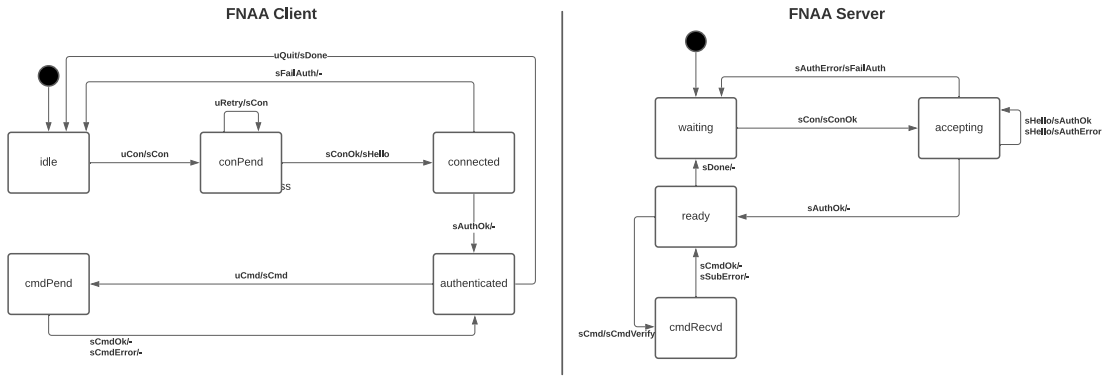

Figure 19. Finite-state machine for the Flow Namespace Accessing Protocol.

The model in right side of Figure 19 shows that the FNAA server starts in a **“waiting” state**, which basically means that the server has successfully set up the networking requirements to accept client connections. Then, when a client connects, a transition is made to **“accepting” state**, in which internally the authentication procedure is made. If the authentication is successful, a transition is made to **“ready” state**, meaning that the client can now execute commands on the FNAA server.

The commands that the client can execute are specified in Figure 11. For each command that the client executes, a transition is made to **“cmdRecvd” state**. Then, a response is returned to the client, transitioning again to **“waiting” state.** When the client executes the “Quit” command, a transition is made to the **“waiting” state** and the server must free all used networking resources for the now closed connection.

On the left side of Figure 19, we also have the client state machine with its corresponding transitions. The client triggers a connection to the server and once established, an authentication is needed. Once the authentication is correctly done, the client can start requesting commands to the server. For each command executed by the client, a transition is made to **“cmdPend” state**, until a response is returned by the server.

Eventually, a “Quit” command will be executed by the client and the connection will be closed.

6. Implementation

In this section, we provide an approach for the overall implementation of the proposed Event Streaming Open Network. Considering the components defined previously for the architecture, we will define which existing tools can be leveraged and those that require development.

6.1. Objectives

The objective of this implementation is to provide specifications for an initial implementation of the overall architecture for the Event Streaming Open Network. Whenever it is possible, existing tools should be leveraged. For those components that need development, a thorough specification is to be provided.

Finally, a subset of core functionalities for a Proof of Concept must be defined.

6.2. Implementation overview

In Figure 20, we have a diagram of the overall implementation proposal. The components that have the Kubernetes Deployment icon are the ones to be managed by the FNAA server instance. Then, we have a Kafka Cluster that provides a Topic instance for each flow. Finally, the DNS Infrastructure is leveraged.

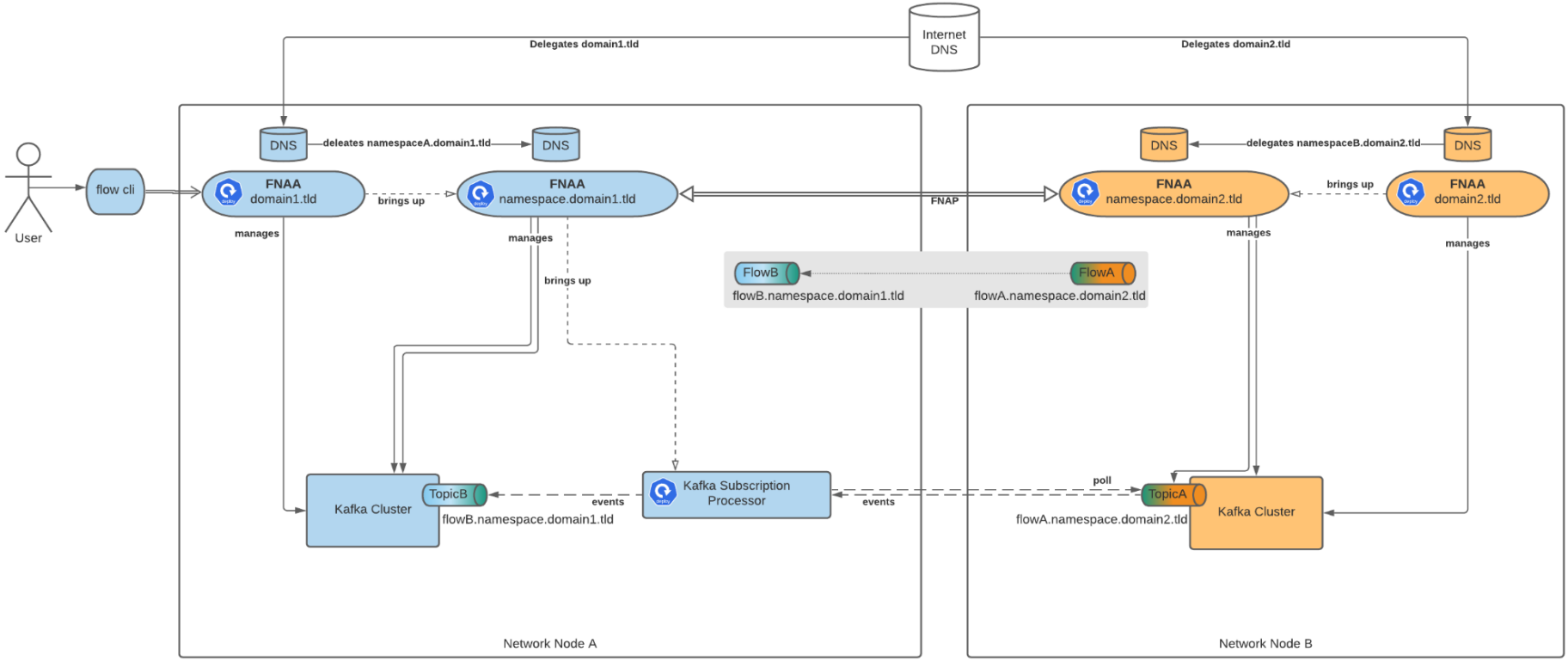

Figure 20. Implementation overview using Kubernetes, Apache Kafka, DNS Bind9 and the Flow CLI tool.

6.2. Existing components

In this section, we describe the existing software components that can be leveraged for implementation.

6.2.1. Flow Events Broker

Since there are currently many implementations for this component, it is necessary to develop the needed integrations of other components of the architecture to the main market leaders. Thus, we will consider the following Flow Events Broker for the implementation: Apache Kafka, AWS SQS and Google Compute PubSub.

In summary, this component does not need to be developed from scratch. However, the FNAA will need to be able to communicate with the different Flow Events Broker, meaning that it must implement their APIs as a client.

6.2.2. Flow Name Service

This component can be completely implemented by leveraging on the ISC Bind9 software component, which is the *de facto* leader for DNS servers. A given NP will need to deploy a Bind9 Nameserver and enable both DNSSEC and DNS Dynamic Update.

The impact of adopting Bind9 for the implementation means that the FNAA component needs to be able to use a remote DNS Server to manage the Flow URI registration, deregistration and execute recursive DNS resolution.

6.3. Components to be developed

In this section, we describe a set of tools that require development. These components, especially the FNAA, are the core components of every Network Participant. Moreover, these are the components that implement the network protocol FNAP.

Since these are the core components of the network, they are the natural candidates for validation. **In the next chapter, we will show the feasibility of the core network components in the form of a Proof of Concept.**

6.3.1. Flow Namespace Accessing Agent

The Flow Namespace Accessing Agent is a server component that triggers the creation of child processes that implement the different Flow Processors. This means that the instance running the FNAA will bring up new processes for each processor. One way of implementing this functionality can be a parent process that creates new child processes for each processor. However, this would imply the need of creating and managing different threads in a single FNAA instance.

The problem with the approach of a parent process and child processes for the FNAA is on the infrastructure level. The more processor a FNAA needs to

manage, the more compute resources the FNAA would need. In the current cloud infrastructure context, this is problem because it means that additional compute resources should be assigned to the FNAA, depending on the quantity of processors and the required resources for each of them. In summary, this approach would be vertically scalable but not horizontally scalable.

Then, to avoid the scalability issue, the approach we propose is by implementing a Cloud Native application. By leveraging on Kubernetes, it is possible to trigger the creation of Deployments, which are composed of Pods. Each Pod can contain a given quantity of containers, which are processes running in a GNU/Linux Operating System. In this way, we can dedicate a Pod to run the FNAA server and different Pods to run the Processors. This approach provides a convenient process isolation and enables both horizontal and vertical scalability.

Moreover, the way in which the FNAA would bring up and manage Processor instances would be though an integration with the underlaying Kubernetes instance, by means of the Kubernetes API. The result is a Cloud Native application that leverages the power and flexibility of Kubernetes to manage the Processor instances.

On the other hand, the programming language for the FNAA must also be defined. For this, we consider that it must be possible to implement the FNAA and the Flow Processors in different programming languages. For the FNAP it is recommended to employ Golang, since Kubernetes CLI tool is implemented in this language and there are several libraries available for integration. As for the Flow Processors, it must be possible to use any programming language as long as the IPC interface is correctly implemented.

Regarding the IPC interface for the communications between the FNAA and the Flow Processors, the recommendation is to employ gRPC together with Protobuf. The rationale for choosing this this technology is the fact that gRPC enables binary communications, which are the desired type of communication for systems integration. Then, both the FNAA and the Flow Processors must share this Protobuf interface definition and implement it accordingly through gRPC.

Finally, the FNAA must implement the protocol we have named FNAP, which provides the main set of functionalities for the Event Streaming Open Network. **The implementation of FNAP must be stateful**, in the sense that it is connection-based. Additionally, **the implementation must be text-based**, with the goal that humans can interact with FNAA servers in the same way that it is possible for SMTP servers. The transport protocol must be TCP with no special definition for a port number, since the port should be able to be discovered by means of DNS SRV Resource Records.

Regarding security for the FNAA servers, **TLS must be supported**. This means that any client can start a TLS handshake with the FNAA servers before issuing any command.

In conclusion, the implementation of the FNAA over Kubernetes provides the needed flexibility and set of capabilities required for this component. It

is recommended to implement the FNAA in Golang and enable the implementation of Flow Processors in any programming language as long as the Protobuf interface is correctly implemented. Finally, the FNAA must implement the protocol FNAP in a connection-based and text-based manner.

6.3.2. Flow Namespace User Agent

The Flow Namespace User Agent (FNUA) can have different implementations as long as they comply with the protocol FNAP.

We propose the initial availability of a CLI tool that acts as a Flow Namespace User Agent. This CLI tool must provide a client implementation of all the functionalities available in the FNAA server. Among the functionalities to be implemented as a must, we can mention:

- Discover the FNAA server for a given Flow URI.
- Connect to the FNAA server to manage Flow Namespaces and Flows, as exemplified in Figure 14:

Additionally, the FNUA should be able to discover the Authoritative FNAA server for a given Flow Namespace. This discovery shall be performed by leveraging on the DNS-SD specification. Refer to Annex D to review the discovery process.

Regarding the implementation of the CLI tool, it is recommended to employ Golang together with Cobra, a library specialized to create CLI tools. In Figure 21, we have a diagram that shows the different functionalities that the CLI tool should implement.

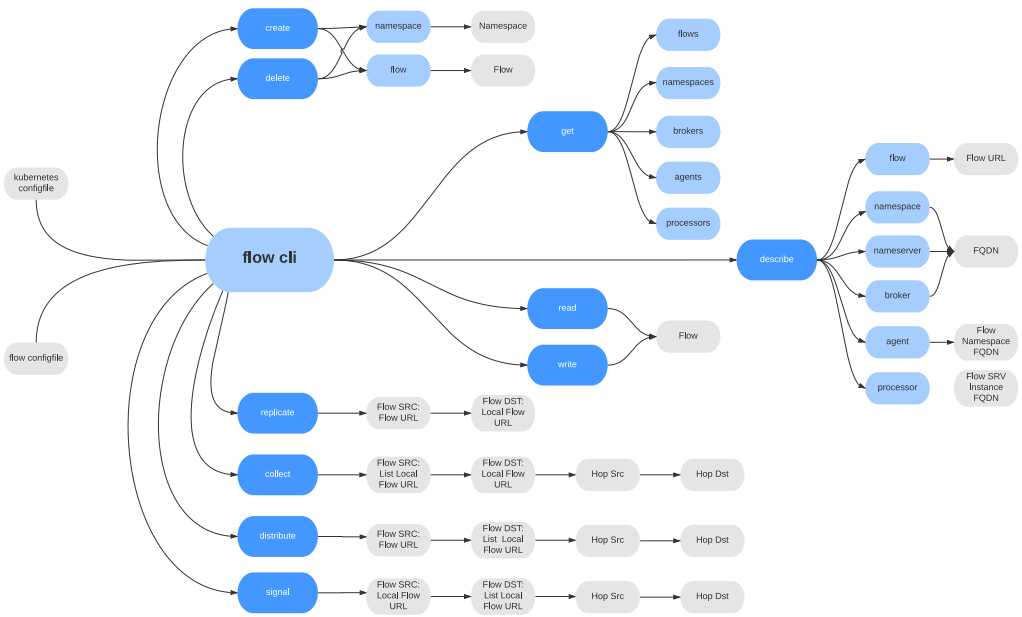

Figure 21. Flow CLI parameters diagram.

7. Proof of Concept

In this section, we will focus on providing a minimum implementation of the main Event Streaming Open Network component: the Flow Namespace Accessing Agent. This implementation should serve as a Proof of Concept of the overall Event Streaming Open Network proposal.

As described in the previous section, the Flow Namespace Accessing Agent (FNAA) is the main and core required component for the Open Network. All Network Participants must deploy an FNAA server instance in order to be part of the network. The FNAA actually implements a server-like application for the Flow Namespace Accessing Protocol (FNAP). **Then, the first objective of this Proof of Concept is to show an initial implementation of the FNAA server component.**

On the other hand, the FNAA is accessed by means of a Flow Namespace User Agent (FUA). This component acts as a client application that connects to a FNAA. Also, this component can take different forms: it could be a web-based application, a desktop application or even a command line tool. For the purposes of this Proof of Concept, we will implement a CLI tool that acts as a client application for the FNAA. **Thus, the second objective of this PoC is to provide an initial implementation of the FNUA client component.**

In the following sections, we will first describe the minimum functionalities considered for validating the overall proposal for the Event Streaming Open Network. This minimum set of requirements for both the FNAA and the FNUA will compose the Proof of Concept.

Afterwards, we will describe the technology chosen for the initial implementation of both the FNAA and the FNUA. Then, a description of how these tools work in isolation will be provided. Subsequently, we will review different use cases to prove how the network could be used by network participants and its users.

Lastly, we will provide a conclusion for this Proof of Concept, where we mentioning if and how the minimum established requirements have been met or not.

6.4. Minimum functionalities

Network Participants system administrators must be able to run a Server Application that acts as FNAA.

Users using a Client Application actiong as a FNUA must be able to:

1. Access the flow account and operate its flows.
2. Create a new flow.
3. Describe an existing flow.
4. Subscribe to an external flow.

7.2. FNAA - Server application

The FNAA server application must implement FNAP as described in Section 6. Basically, the FNAA will open a TCP port on all the IP addresses of the host to listen for new FNUA client connections.

The chosen language for the development of the FNAA is GoLang. The reason for choosing GoLang is because Kubernetes is written in this language and there is a robust set of libraries available for integration. Although there is no integration built with Kubernetes for this Proof of Concept, the usage of GoLang will enable a seamless evolution of the FNAA application. In future versions of the FNAA codebase, new functionalities leveraging Kubernetes will be easier to implement than if using a different programming language.

When the FNAA server application is initialized, it provides debug log messages describing all client interactions. In order to start the server application, a Network Participant system administrator can download the binary and execute it in a terminal:

```
ignatius ~ 0$./fnaad
server.go:146: Listen on [::]:61000
server.go:148: Accept a connection request.
```

Now that the 61000 TCP port is open, we can test the behaviour by means of a raw TCP using telnet command in a different terminal:

```
ignatius ~ 1$telnet localhost 61000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 fnaa.unix.ar FNAA
```

We can now see that the server has provided the first message in the connection: a welcome message indicating its FQDN fnaa.unix.ar.

On the other hand, the server application starts providing debug information for the new connection established:

```
ignatius ~ 0$./fnaad
server.go:146: Listen on [::]:61000
server.go:148: Accept a connection request.
server.go:154: Handle incoming messages.
server.go:148: Accept a connection request.
```

7.3. FNUA - Client application

In order to test the FNAA server application, a CLI-based FNUA application has been developed. The chosen language for this CLI tool is also GoLang. The reason for choosing GoLang for the FNUA is because of its functionalities for building CLI tools, leveraging on the Cobra library.
Thus, the FNUA for the PoC is an executable file that complies with the diagram in Figure 20.

One of the requirements for the flow CLI tool is a configuration file that defines the different FNAA servers together with the credentials to use. An example of this configuration file follows:

```
ignatius ~/ 0$cat .flow.yml
agents:
  -
    name: fnaa-unix
    fqdn: fnaa.unix.ar
    username: test
    password: test
    prefix: unix.ar-
  -
    name: fnaa-emiliano
    fqdn: fnaa.emiliano.ar
    username: test
    password: test
    prefix: emiliano.ar-

namespaces:
  -
    name: flows.unix.ar
    agent: fnaa-unix
  -
    name: flows.emiliano.ar
    agent: fnaa-emiliano
```

In this file, we can see that there are two FNAA instances described with FQDN fnaa.unix.ar and fnaa.emiliano.ar. Then, there are two namespaces: one called **flow.unix.ar** hosted on **fnaa-unix** and second namespace **flows.emiliano.ar** hosted on **fnaa-emiliano**. This configuration enables the FNUA to interact with two different FNAA, each of which is hosting different Flow Namespaces.

Once the configuration file has been saved, the flow CLI tool can now be used. In the following sections, we will show how to use the minimum functionalities required for the Open Network using this CLI tool.

7.4. Use cases

7.4.1. Use case 1: Authenticating a user

After the connection is established, the first command that the client must execute is the authentication command. As previously defined in Chapter 5, every FNAA client must first authenticate in order to execute commands. Thus, the authentication challenge must be supported both by the FNAA as well as the FNUA.

It is worth mentioning that the chosen authentication mechanism for this PoC is SASL Plain. This command can be extended furtherly with other mechanisms in later versions. However, this simple authentication mechanism is sufficient to demonstrate the authentication step in the FNAP.

The SASL Plain Authentication implies sending the username and the password encoded in Base64. The way to obtain the Base64 if we consider a user *test* with password *test*, is as follows:

```
ignatius ~ 0$echo -en "\0test\0test" | base64
AHRlc3QAdGVzdA==
```

Now, we can use this Base64 string to authenticate with the FNAA. First, we need to launch the FNAA server instance:

```
ignatius~/ $./fnaad --config ./fnaad_flow.unix.ar.yaml
main.go:41: Using config file: ./fnaad_flow.unix.ar.yaml
main.go:57:     Using config file: ./fnaad_flow.unix.ar.yaml
server.go:103: Listen on [::]:61000
server.go:105: Accept a connection request.
```

Then, we can connect to the TCP port in which the FNAA is listening:

```
ignatius ~ 1$telnet localhost 61000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 fnaa.unix.ar FNAA
AUTHENTICATE PLAIN
220 OK
AHRlc3QAdGVzdA==
220 Authenticated
```

Once the client is authenticated, it can start executing FNAP commands to manage the Flow Namespace of the authenticated user. For simplicity purposes, in this Proof of Concept, we will be using a single user.

In the case of the CLI tool, there is no need to perform an authentication step, since every command the user executes will be preceded by an authentication in the server.

7.4.1. Use case 2: Creating a flow

Once the authentication is successful, the client can now create a new Flow. The way to do this using the CLI tool would be:

```
ignatius ~/ 0$./fnua create flow time.flow.unix.ar
Resolving SRV for fnaa.unix.ar. using server 172.17.0.2:53
Executing query fnaa.unix.ar. IN 33 using server 172.17.0.2:53
Executing successful: [fnaa.unix.ar.    604800 IN     SRV    0 0 61000
fnaa.unix.ar.]
Resolving A for fnaa.unix.ar. using server 172.17.0.2:53
Executing query fnaa.unix.ar. IN 1 using server 172.17.0.2:53
Executing successful: [fnaa.unix.ar.    604800 IN     A      127.0.0.1]
Resolved A to 127.0.0.1 for fnaa.unix.ar. using server 172.17.0.2:53
C: Connecting to 127.0.0.1:61000
C: Got a response: 220 fnaa.unix.ar FNAA
C: Sending command AUTHENTICATE PLAIN
C: Wrote (20 bytes written)
C: Got a response: 220 OK
C: Authentication string sent: AHRlc3QAdGVzdA==
C: Wrote (18 bytes written)
C: Got a response: 220 Authenticated
C: Sending command CREATE FLOW time.flow.unix.ar
C: Wrote (31 bytes written)
C: Server sent OK for command CREATE FLOW time.flow.unix.ar
C: Sending command QUIT
C: Wrote (6 bytes written)
```

The client has discovered the FNAA server for Flow Namespace flow.unix.ar by means of SRV DNS records. Thus, it obtained both the FQDN of the FNAA together with the TCP port where it is listening, in this case 61000. Once the resolution process ends, the FNUA connects to the FNAA. First, the FNUA authenticates with the FNAA and then it executes the *create flow* command.

If we were to simulate the same behavior using a raw TCP connection, the following steps would be executed:

```
ignatius ~ 1$telnet localhost 61000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 fnaa.unix.ar FNAA
AUTHENTICATE PLAIN
220 OK
AHRlc3QAdGVzdA==
220 Authenticated
CREATE FLOW time.flows.unix.ar
220 OK time.flows.unix.ar
```

Now, the client has created a new flow called time.flows.unix.ar located in the flows.unix.ar namespace. The FNAA in background has created a Kafka Topic as well as the necessary DNS entries for name resolution.

7.4.2. Use case 3: Describing a flow

Once a flow has been created, we can obtain information of if by executing the following command using the CLI tool:

```
ignatius ~/ 1$./fnua describe flow time.flow.unix.ar
Resolving SRV for fnaa.unix.ar. using server 172.17.0.2:53
Executing query fnaa.unix.ar. IN 33 using server 172.17.0.2:53
Executing successful: [fnaa.unix.ar.    604800 IN     SRV    0 0 61000
fnaa.unix.ar.]
Nameserver to be used: 172.17.0.2
Resolving A for fnaa.unix.ar. using server 172.17.0.2:53
Executing query fnaa.unix.ar. IN 1 using server 172.17.0.2:53
Executing successful: [fnaa.unix.ar.    604800 IN     A      127.0.0.1]
Resolved A to 127.0.0.1 for fnaa.unix.ar. using server 172.17.0.2:53
C: Connecting to 127.0.0.1:61000
C: Got a response: 220 fnaa.unix.ar FNAA
C: Sending command AUTHENTICATE PLAIN
C: Wrote (20 bytes written)
C: Got a response: 220 OK
C: Authentication string sent: AHRlc3QAdGVzdA==
C: Wrote (18 bytes written)
C: Got a response: 220 Authenticated
C: Sending command DESCRIBE FLOW time.flow.unix.ar
C: Wrote (33 bytes written)
C: Server sent OK for command DESCRIBE FLOW time.flow.unix.ar
Flow time.flow.unix.ar description:
flow=time.flow.unix.ar
type=kafka
topic=time.flow.unix.ar
server=kf1.unix.ar:9092
Flow time.flow.unix.ar described successfully
Quitting
C: Sending command QUIT
C: Wrote (6 bytes written)
```

In the output of the *describe* command we can see all the necessary information to connect to the Flow called *time.flow.unix.ar*: (i) the type of Event Broker is Kafka, (ii) the Kafka topic has the same name of the flow and (iii) the Kafka Bootstrap server with port is provided. If we were to obtain this information using a manual connection, the steps would be:

```
ignatius ~ 1$telnet localhost 61000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 fnaa.unix.ar FNAA
AUTHENTICATE PLAIN
220 OK
AHRlc3QAdGVzdA==
220 Authenticated
DESCRIBE FLOW time.flows.unix.ar
220 DATA
flow=time.flows.unix.ar
type=kafka
topic=time.flows.unix.ar
server=kf1.unix.ar:9092
220 OK
```

Now, we can use this information to connect to the Kafka topic and start producing or consuming events.

7.4.3. Use case 4: Subscribing to a remote flow

In this section, we will show how a subscription can be set up. When a user commands the FNAA to create a new subscription to a remote Flow, the local FNAA server first needs to discover the remote FNAA server. Once the server is discovered by means of DNS resolution, the local FNAA contacts the remote FNAA, authenticates the user and then executes a subscription command.

Thus, the initial communication between the FNUA and the FNAA, in which the user indicates to subscribe to a remote flow, would be as follows:

```
ignatius ~ 1$telnet localhost 61000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 fnaa.unix.ar FNAA
AUTHENTICATE PLAIN
220 OK
AHRlc3QAdGVzdA==
220 Authenticated
SUBSCRIBE time.flows.unix.ar LOCAL emiliano.ar-time.flows.unix.ar
220 DATA
ksdj898.time.flows.unix.ar
220 OK
```

Once the user is authenticated, a SUBSCRIBE command is executed. This command indicates first the remote flow to subscribe to. Then, it also specifies with LOCAL the flow where the remote events will be written. In this example, the remote flow to subscribe to is **time.flows.unix.ar**, and the local flow is emiliano.ar-time.flows.unix.ar. Basically, a new flow has been created,

emiliano.ar-time.flows.unix.ar, where all the events of flow time.flows.unix.ar will be written.

The server answers back with a new Flow URI, in this case ksdj898.time.flows.unix.ar. This Flow URI indicates a copy of the original flow time.flows.unix.ar created for this subscription. Thus, the remote FNAA has full control over this subscription, being able to revoke it by simply deleting this flow or applying Quality of Service rules.

The remote FNAA has set up a Bridge Processor to transcribe messages in topic time.flows.unix.ar to the new topic ksdj898.time.flows.unix.ar. Another alternative to a Bridge Processor would be a Distributor Processor, which could be optimized for a Flow with high demand. Moreover, instead of creating a single Bridge Processor per subscription, a Distributor Processor could be used, in order to have a single consumer of the source flow and write the events to several subscription flows.

The user could use the FNUA CLI tool to execute this command in the following manner:

```
Ignatius ~ 0$./fnua --config=./flow.yml subscribe time.flows.unix.ar --
nameserver 172.17.0.2 -d --agent fnaa-emiliano
Initializing initConfig
    Using config file: ./flow.yml
Subscribe to flow
Agent selected: fnaa-emiliano
Resolving FNAA FQDN fnaa.emiliano.ar
Starting FQDN resolution with 172.17.0.2
Resolving SRV for fnaa.emiliano.ar. using server 172.17.0.2:53
Executing query fnaa.emiliano.ar. IN 33 using server 172.17.0.2:53
FNAA FQDN Resolved to fnaa.emiliano.ar. port 51000
Resolving A for fnaa.emiliano.ar. using server 172.17.0.2:53
Resolved A to 127.0.0.1 for fnaa.emiliano.ar. using server 172.17.0.2:53
C: Connecting to 127.0.0.1:51000
C: Got a response: 220 fnaa.unix.ar FNAA
Connected to FNAA
Authenticating with PLAIN mechanism
C: Sending command AUTHENTICATE PLAIN
C: Wrote (20 bytes written)
C: Got a response: 220 OK
C: Authentication string sent: AHRlc3QAdGVzdA==
C: Wrote (18 bytes written)
C: Got a response: 220 Authenticated
Authenticated
Executing command SUBSCRIBE time.flows.unix.ar LOCAL emiliano.ar-
time.flows.unix.ar
C: Sending command SUBSCRIBE time.flows.unix.ar LOCAL emiliano.ar-
time.flows.unix.ar
C: Wrote (67 bytes written)
C: Server sent OK for command SUBSCRIBE time.flows.unix.ar LOCAL emiliano.ar-
time.flows.unix.ar
Flow emiliano.ar-time.flows.unix.ar subscription created successfully
Server responded: emiliano.ar-time.flows.unix.ar SUBSCRIBED TO
ksdj898.time.flows.unix.ar
Quitting
C: Sending command QUIT
C: Wrote (6 bytes written)
Connection closed
```

This interaction of the FNUA with the FNAA of the Flow Namespace emiliano.ar (fnaa-emiliano) has trigger an interaction with the FNAA of unix.ar Flow Namespace (fnaa-unix). This means that before fnaa-emiliano was able to respond to the FNUA, a new connection was opened to the remote FNAA and the SUBSCRIBE command was executed.

The log of fnaa-emiliano when the SUBCRIBE command was issued looks as follows:

```
server.go:111: Handle incoming messages.
server.go:105: Accept a connection request.
server.go:253: User authenticated
server.go:347: FULL COMMAND: SUBSCRIBE time.flows.unix.ar LOCAL emiliano.ar-
time.flows.unix.ar
server.go:401: Flow is REMOTE
client.go:280: **#Resolving SRV for time.flows.unix.ar. using server
172.17.0.2:53
server.go:417: FNAA FQDN Resolved to fnaa.unix.ar. port 61000
client.go:42: C: Connecting to 127.0.0.1:61000
client.go:69: C: Got a response: 220 fnaa.unix.ar FNAA
server.go:435: Connected to FNAA
server.go:436: Authenticating with PLAIN mechanism
client.go:126: C: Sending command AUTHENTICATE PLAIN
client.go:133: C: Wrote (20 bytes written)
client.go:144: C: Got a response: 220 OK
client.go:154: C: Authentication string sent: AHRlc3QAdGVzdA==
client.go:159: C: Wrote (18 bytes written)
client.go:170: C: Got a response: 220 Authenticated
server.go:444: Authenticated
client.go:82: C: Sending command SUBSCRIBE time.flows.unix.ar
client.go:88: C: Wrote (30 bytes written)
client.go:112: C: Server sent OK for command SUBSCRIBE time.flows.unix.ar
server.go:456: Flow time.flows.unix.ar subscribed successfully
server.go:457: Server responded: ksdj898.time.flows.unix.ar
server.go:459: Quitting
```

We can see how fnaa-emiliano had to trigger a client subroutine to contact the remote fnaa-unix. Once the server FQDN, IP and Port is discovered by means of DNS, a new connection is established and the SUBSCRIBE command is issued. Here we can see the log of fnaa-unix:

```
server.go:111: Handle incoming messages.
server.go:105: Accept a connection request.
server.go:253: User authenticated
server.go:139: Received command: subscribe
server.go:348: [SUBSCRIBE time.flows.unix.ar]
server.go:367: Creating flow endpoint time.flows.unix.ar
server.go:368: Creating new topic ksdj898.time.flows.unix.ar in Apache Kafka
instance kafka_local
server.go:369: Creating Flow Processor src=time.flows.unix.ar
dst=ksdj898.time.flows.unix.ar
server.go:370: Adding DNS Records for ksdj898.time.flows.unix.ar
server.go:372: Flow enabled ksdj898.time.flows.unix.ar
server.go:139: Received command: quit
```

Thus, we were able to set up a new subscription in fnaa-emiliano that trigger a background interaction with fnaa-unix.

7.5. Results of the PoC

We can confirm the feasibility of the overall Event Streaming Open Network architecture. The test of the proposed protocol FNAP and its implementation, both in the FNAA and FNUA (CLI application), show that the architecture can be employed for the purpose of distributed subscription management among Network Participants.

The minimum functionalities defined both for the Network Participants and the Users were met. Network Participants can run this type of service by means of a server application, the FNAA server. Also, the CLI-tool resulted in a convenient low-level method to interact with a FNAA server.

In further implementations, the server application should be optimized as well as secured, for instance with a TLS handshake. Also, the CLI-tool could be enhanced by a web-based application with a friendly user interface.

Nevertheless, the challenge for a stable implementation of both components is the possibility of supporting different Event Brokers and their evolution. Not only Apache Kafka should be supported but also the main Public Cloud providers events solutions, such as AWS SQS or Google Cloud Pub/Sub. Since the Event Brokers are continuously evolving, the implementation of the FNAA component should keep up both with the API and new functionalities of these vendors.

Regarding the protocol design, it would be needed to enhance the serialization of the exchanged data. In this sense, it could be convenient to define a packet header for the overall interaction between the FNAA both with remote FNAA as well as with FNUA.

Regarding the subscription use case, it would be necessary to establish a convenient format for the server response. Currently, the server is returning a key/value structure with the details of the Flow. This structure may not be the most adequate, since it may differ depending on the Event Broker used.

Also, the security aspect needs further analysis and design since its fragility could lead to great economical damage for organizations. Thus, it would be recommended to review the different security controls needed for this solution as part of an Information Security Management System.

Finally, the implementation should leverage the Cloud Native functionalities provided by the Kubernetes API. For example, the FNAA should trigger the deployment of Flow Processors on demand, in order to provide isolated computing resources for each subscription. Also, a Kubernetes resource could be developed to use the **kubectl** CLI tool for management, instead of a custom CLI tool.

9. Summary & Conclusions

In this chapter we will provide a summary of everything that has been done in this work, as well as some conclusions about it.

First, we described the current landscape of the Event Streaming technology, mentioning the main tools and actors in the market. While describing the state-of-the-art of the technology, Apache Kafka was identified as the leading tool for Event Streaming. Not only Apache Kafka enables convenient functionalities for real-time events communications but also is currently being widely adopted for this purpose.

Then, we identified a use case for which there is currently no adequate solution provided by existing tools. This use case is based on the cross-organization integration of real-time event streams. Nowadays, organizations intending to integrate these kind of data streams struggle with offline communication to achieve a common interface for integration. In this context, we proposed an Open Network for Event Streaming as a possible solution for this difficulty.

For this Open Network, we have followed the main necessities from the technical perspective. While there already exist many components that can be leveraged, some components require analysis, design, and implementation. Then, we referred to the Commons Infrastructure literature in order to show how Event Streaming can be considered an Infrastructure Resource that can enable downstream productive activities. Finally, we established the main guidelines that an Open Network should follow, basing these definitions on Free, Open & Neutral Networks.

Using the previous definitions, we have designed an architecture for the Event Streaming Open Network, establishing the components that the different Network Participants should implement in order to participate in the network. After providing a thorough description of all the components, we showed some use cases of integration among different Network Participants.

Once the architecture was defined, we proposed an implementation approach which describes the existing components that can be leveraged as well as those that need to be developed from scratch. The outcome was that a server-side application called FNAA had to be developed. This application implements the protocol FNAP and can be accessed by a client application, which we named FNUA.

Finally, we proved the feasibility of the proposed architecture by providing an implementation of the minimum functionalities required, in the form of a Proof of Concept. The results of this PoC were encouraging since it was possible to implement the initial functionalities for the FNAA and FNUA components.

As conclusion, we can mention that there is great potential for an Open Network for Event Streaming among organizations. In the same way the email infrastructure acts as an open network for electronic communications among

people, this kind of network would enable developers to integrate real-time event streams while minimizing offline agreement of interfaces and technologies.

However, there are many difficulties that could be furtherly worked on. First, a robust implementation for the Event Streaming Open Network main components must be provided, mainly for the FNAA and FNUA. In order to achieve an acceptable level of quality and stability, the development of a community around the project is needed.

Secondly, we found that the proposed architecture is a convenient starting point. However, it can suffer modifications based on the learning process during the implementation. For example, while designing the architecture, we avoided the need of a database for the FNAA component, leveraging on the DNS infrastructure. While this can be sufficient for the minimum functionalities described, it will most probably be necessary for the FNAA to persist data in a database of its own. In this sense, we believe that leveraging the Kubernetes resources model could be a convenient alternative.

Thirdly, during the PoC execution, we identified some difficulties implementing the security functionalities of authentication and authorization. Although we were able to implement an authentication mechanism, the reality indicates that integration with well-established protocols is needed (i.e., OAuth, GSSAPI, etc.).

Finally, there is also the need to leverage on the Cloud Native architecture, basically Kubernetes, to provide hyper-scalability and enable Network Participants to agnostically choose the underlaying infrastructure. The selection of Golang for the PoC implementation showed to be convenient, given the vast number of available libraries for integration of third-party components and services.

Notwithstanding the difficulties, we firmly believe that cross-organization real-time event integration can provide great benefits for society. It would enhance the efficiency of business processes throughout organizations. Also, it would provide broad visibility to the final users, enabling experimentation and entrepreneurship. New business models for existing productive activities could be developed, as well as enabling innovation, which in turn would conform the positive externalities of the Event Streaming Open Network.

10. Bibliography

FRISCHMANN B. (2007) [Online] Infrastructure Commons in Economic Perspective < https://firstmonday.org/article/view/1901/1783>

URQUHART J. (2021) Flow Architectures

WIDL M. (2013), Guided Merging of Sequence Diagrams

NAVARRO L. (2018) [Online] Network Infrastructures: The commons model for local participation, governance and sustainability <https://www.apc.org/en/pubs/network-infrastructures-commons-model-local-participation-governance-and-sustainability>

BRINO A. (2019) Towards an Event Streaming Service for ATLAS data processing.

GUTTRIDGE, Gartner (2021) “Modern Data Strategies for the Real-time Enterprise” Big Data Quarterly 2021

11. License

Annex A – Traditional synchronous archetype

Traditional communications among systems include a service integration layer as well as a database integration one. We can see in Figure 1 a traditional archetype for traditional software development.

The System User Space in the cloud symbolizes the group of people, applications, devices, etc.; that make use of the System. The System exposes a Service Layer that permits users to access data, which can be built on network and presentation standards such as REST, GraphQL, gRPC, Web Sockets, etc. Moreover, the behavior exposed by this Service Layer is predominantly synchronic.

The synchrony can be of two different types: (i) tight synchrony or (ii) loose synchrony. The first one implies one or more synchronous tasks in the service layer, for example operations against a RDMS (Relational Database Management System) such as MySQL, PostgreSQL, MongoDB, etc. The latter, loose synchrony refers to behavior that does not include nested synchronic behavior or includes asynchronous nested behavior. The main objective of differentiating these categories is due its convenience to illustrate the importance of latency in the traditional communication archetype.

Nevertheless, this archetype has an Achilles Heel. While this archetype can effectively solve unlimited use cases it cannot solve all of them with the same level of efficiency. This weakness is located in the System User Space. In the last few years, there has been a tremendous growth in users and devices, together with the growth of client network applications installed on computers and mobile phones. The result is what literature on the subject calls “firehose of data”, meaning an intense and rich source of data.

When an instance system of the traditional archetype is exposed to large data demands from the network, the weaknesses are risky as well as costly. There are two main threatening scenarios. First scenario in which the application cannot cope with the demand and collapses. In the second scenario, the system is not fully ready (and cannot be made fully ready in the middle term) to start receiving large flows of data for various reasons, for instance software development costs, know-how, etc.

In any case, whether the application has been made ready to receive intense data streams or not, the system will need to be transformed to a distributed system. Then a large refactoring problem emerges since transforming an standalone system to a distributed one is not a straight-forward task.

However, there is the alternative of deploying a tool that guarantees the reception of the data streams independently of the business reader application. These tools are referred to Event Streaming Platforms and basically expose both a service and storage layer. The ESP offers the great benefit of serving all entities (people, devices, apps, etc.) that need to asynchronously write event data. On the other side, ESPs also low-latency and high-throughput communications for the readers of data streams.

Annex B – Flow URI Resolution

In this case, the query for PTR records would be as follows:

```
;; QUESTION SECTION:
;notifications.calendar.people.syndeno.com.           IN    PTR
```

The response would be in the following form:

```
;; ANSWER SECTION:
notifications.calendar.people.syndeno.com. 21600 IN   PTR
_flow._tcp.notifications.calendar.people.syndeno.com.
```

Using the FQDN returned by this query, an additional query asking for SRV records is made:

```
;; QUESTION SECTION:
;_flow._tcp.notifications.calendar.people.syndeno.com.      IN    SRV

;; ANSWER SECTION:
_flow._tcp.notifications.calendar.people.syndeno.com. 875 IN      SRV
      30 30 65432 fnaa.syndeno.com.
_flow._tcp.notifications.calendar.people.syndeno.com. 875 IN TXT "tls"

_queue._flow._tcp.notifications.calendar.people.syndeno.com.   875   IN
      SRV   30 30 9092 kafka.syndeno.com.
_queue._flow._tcp.notifications.calendar.people.syndeno.com. 875 IN TXT
"broker-type=kafka tls"
```

First, the response informs the network location of the FNAA server, in this case a connection should be opened to TCP port 65432 of the IP resulting of resolving fnaa.syndeno.com:

```
;; QUESTION SECTION:
;fnaa.syndeno.com.      IN    A

;; ANSWER SECTION:
fnaa.syndeno.com. 21600 IN    A     208.68.163.200
```

Secondly, this response offers other relevant information, like the TCP port where the queue service is located (9092). It also includes a TXT Resource Record that establishes the protocol of the Event Queue Broker, defined in the variable "broker-type=kafka".

Now, using the returned FQDN for the queue, kafka.syndeno.com, the resolver can perform an additional query:

```
;; QUESTION SECTION:
;kafka.syndeno.com.           IN    A

;; ANSWER SECTION:
kafka.syndeno.com.21600 IN    A     208.68.163.218
```

Annex C – Flow URI Syntax

flow://flowName.flowCategory.myNameSpace.domain.tld

- **Flow Namespace FQDN:** myNameSpace.domain.tld
- **Flow Name:** flowName.flowCategory
- **Flow FQDN:** flowName.flowCategory.myNameSpace.domain.tld

The following are examples of this URI Syntax:

flow://notifications.calendar.people.syndeno.com

- **Flow Namespace FQDN:** people.syndeno.com
- **Flow Name**: notifications.calendar
- **Flow FQDN:** notifications.calendar.people.syndeno.com

flow://created.invoice.finance.syndeno.com:

- **Flow Namespace FQDN:** finance.syndeno.com
- **Flow Name**: created.invoice
- **Flow FQDN:** created.invoice.finance.syndeno.com