PulsarCast

Scalable and reliable pub-sub over P2P networks

João Gonçalo da Silva Antunes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Doctor Luís Manuel Antunes Veiga
David Miguel dos Santos Dias

October 2019

Acknowledgments

TODO

Abstract

The publish-subscribe paradigm is a wildly popular form of communication in complex distributed systems. A lot of research exists around it, with solutions ranging from centralised message brokers, to fully decentralised scenarios (peer to peer). When we are focusing on scalability, decentralisation poses the best option. There is, however, a clear lack of decentralised systems accounting for reliability, message delivery guarantees and, more importantly, persistence. To this end, we present PulsarCast, a decentralised, highly scalable, pub-sub system seeking to give guarantees that are traditionally associated with a centralised architecture such as persistence and eventual delivery guarantees.

Keywords: Publish Subscribe, Peer to peer, Reliability, Persistence, Eventual delivery

Resumo

O teu resumo aqui...

Palavras-Chave: TODO

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The publish-subscribe (pub-sub) interaction paradigm is an approach that has received an increasing amount of attention over the course of the century [2] [3]. This is mainly due to its special properties, that allow for full decoupling of all the communicating parties. Taking a closer look at this definition one can see that this comes hand in hand with the way information is consumed nowadays, with the exponential growth of social networks like Twitter and the usage of feeds such as RSS.

1.1 Motivation

First, we should define what the publish-subscribe pattern is. In this interaction paradigm, subscribers (or consumers) sign up for events, or classes of events, from publishers (or producers) that are subsequently asynchronously delivered. This decoupling can be broken into three different parts. The decoupling in time, space and synchronisation.

The time decoupling comes from the fact that publishers and subscribers do not need to be actively interacting with each other at the same time; this means that the publisher can publish some events while the subscriber is disconnected and the subscriber can be notified of an event whose publisher is disconnected. Space decoupling gives both parties the benefit of not needing to know each other in order to communicate, given that consumers and producers are focused on their specific roles (consuming/producing) and do not care for who is doing what, or how many producers are for example. Synchronization decoupling is a consequence of the asynchronous nature of the pub-sub pattern, as publishers do not need to be blocked while producing events and subscribers can be asynchronously notified. The decoupling that this kind of system offers makes it the ideal candidate for very large networks that need a way to communicate in an efficient way.

Due to the properties described above, a lot of applications rely on the publish-subscribe paradigm and a lot of work has been done by companies like Twitter [1] and LinkedIn into making these systems capable of scaling to a large number of participants, with the creation of tools like Kafka [2], which aim at guaranteeing low latency and high event throughput. Other examples are the multiple message queue systems like Apache Active MQ [3], RabbitMQ [4], Redis [5], etc. These solutions are, of course, centralised and as such suffer from all the common issues that affect centralised solutions: it is quite hard to maintain and scale these systems to a large number of clients. Peer-to-Peer networks, on the other hand, have proven numerous times, that this is where they shine, with examples such as Gnutella, Skype and most

[1] https://www.infoq.com/presentations/Twitter-Timeline-Scalability
[2] http://kafka.apache.org/documentation/#design
[3] http://activemq.apache.org/
[4] https://www.rabbitmq.com/
[5] https://redis.io/topics/pubsub

recently IPFS [6]. All of these systems are a living proof of the high scalability P2P can offer, with pub-sub systems over P2P networks being an active research topic with a lot of attention.

1.2 Goals

As we are going to cover in the next sections, lots of different solutions exist. However, most of them either rely on a centralised or hierarchic network to have a reliable system, with stronger delivery and persistence guarantees, or end up sacrificing these same properties in order to have a decentralised system with the potential to scale to a much larger network.

The solution we propose is Pulsarcast, a pub-sub module with a strong focus on reliability, eventual delivery guarantees and data persistence, while maintaining the ability to scale to a vast number of users. Our solution takes advantage of the network infrastructure and network protocols we have in place today. There is also, to the best of our knowledge, a lack of pub-sub systems with such a strong focus on persistence, which is something our solution does.

1.3 Document Roadmap

TODO

[6] https://ipfs.io/

Chapter 2

Related Work

In this section, we will cover the research work and industry references that can be considered relevant to our initial objective. The following section 2.1 will lay the ground to define how pub-sub systems are structured. In section 2.2 we will cover a set of systems defined as relevant. Finally, section 2.3 will address some of the web technologies of interest in this area.

2.1 Distributed Publish-Subscribe Paradigm

In this section we will cover the basis of the pub-sub paradigm, defining a taxonomy we will later use to classify relevant systems. We will start by covering the *Subscription Model*, followed by the *Network Architecture* and *Overlay Structure*. Finally, we address the *Subscription Management and Event Dissemination*.

2.1.1 Subscription Model

When considering pub-sub systems, there is a set of different options that will lay ground for the behaviour of the whole system. We call these options, design dimensions. Specifically, in our case, one of the biggest decisions when designing a pub-sub system is what kind of subscription model to use.The subscription model determines how subscribers will define which events they are interested in. There are three major approaches covered by relevant literature [2] [3] and that implementations usually follow:

- Topic based subscriptions
- Content based subscriptions
- Type based subscriptions

Topic based subscription model employs, as the name states, the notion of topics or subjects to allow participants to subscribe to relevant content. These topics are identified by keywords and can be naturally viewed as a group or a channel to which participants can send messages (publish) and receive messages (subscribe). This approach was one of the earliest models in the pub-sub paradigm, with references such as TIBCO [1], mainly due to its similarity with the group communication systems already in place at the time. Some examples of the topic based approach allow building a topic hierarchy. A specific one is using a UNIX path like approach, which allows a topic hierarchy just like paths in a file system. Consider as an example:

[1] https://www.tibco.com/

```
/fruits
/fruits/citrus
/fruits/citrus/orange
```

The list above is an example of 3 topics, that act as 3 different tiers on a hierarchy. This allows for specialisation and the possibility to extend the subscription structure already in place. There are numerous solutions that cover the topic based subscription scenario. Specifically, in the distributed/decentralised area, we have solutions like Scribe [4], Bayeux [5], Tera [6] and Poldercast [7].

The content based subscription model brought a different approach that sought to use the content of the event message itself as a way for subscribers to specify the messages they were interested in [8]. Essentially, subscribers could define fields, or conditions on those same fields that would make an event match a subscription or not. Consider the following example of a simple message and subscription, represented using JSON [2].

Message

```
{
  exchange: "Euronext Lisboa",
  company: "CTT",
  order: "buy",
  number: "100",
  price: "5.55",
}
```

Subscription

```
{
  exchange: "Euronext Lisboa",
  order: "buy",
  number: ">50",
  price: "<10",
}
```

The example above translates into a subscription to a stock exchange pub-sub system, where the client would receive all the event messages for *buy orders* of more than 50 stock actions for a maximum price of 10€. The notion of subscription is much more complex in this model but allows for a much more powerful, expressive and accurate message filtering. Usually, in order to implement this, systems rely on the definition of schemas as a way to create subscriptions. Some examples of solutions that follow a content based subscription model are Gryphon [9], Jedi [10], Siena [11], Meghdoot [12], Mercury [13] and Sub-2-sub [14].

Also worth referencing is the **type based subscription model**. [15]. The type based model seeks to use the type scheme of a programming language without introducing a topic hierarchy. Instead, it focuses on the idea that, in practice, messages part of the same topic usually are of the same type and notify the same kind of event. As such we can rely on a straightforward type-safe interpretation of messages belonging to the same topic, since most topic based systems only offer, at most, weakly typed interfaces. This, of course, comes quite handy when working with strongly typed languages such

[2] https://www.json.org/

as Java and C++. One other aspect also worth mentioning is that, similar to topic based systems, the type based system also offers a notion of hierarchy through sub-typing. In this area, Hermes [16] is a reference system implemented on top of a distributed network.

While looking back at these different models it is crucial to understand how they are tied to the expressiveness of the system as a whole. Choosing a topic based subscription model will allow for an easier implementation when it comes to message filtering at each node, but it will clearly affect the capabilities of the system. On the other end, a content based subscription model allows for a lot more expressiveness in subscription definition, but it makes it a lot harder to implement a scalable way of filtering messages. It is also important to note that these three categories are not strict distinct models, but somewhat fluid and subject to hybridisation, as is quite possible to have solutions in between, such as content based filtering through the use of special topics, or content based filtering only for pre-set fields. As such, not all approaches are easy to categorise and, for some specific scenarios and systems, the line is quite thin between the multiple subscription models.

It is also interesting to look at the application space and notice that not all applications have the same expressiveness requirements. This makes the existence of multiple subscription models not only justifiable but required. Consider the example that was given above for a stock exchange system: this kind of applications have a need for a complex set of subscription patterns, quite different from the ones you would probably have for a chat or social media application, which would rely heavily on the notion of topics and groups.

2.1.2 Network Architecture

Independently of the subscription model used, the system approach to the network architecture is crucial as it will, not only set the way clients interact with it, but will also determine a lot of the properties that the solution will benefit from (such as scalability, reliability, etc.). Networks can generally be categorised as centralised or decentralised.

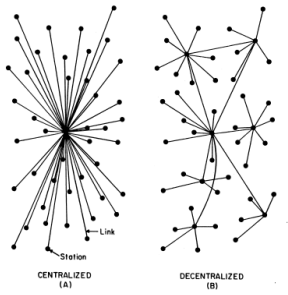

Figure 2.1: Example of a Centralised and a Decentralised network, extracted from [1]

Note that the goal of a pub-sub system is to enable the exchange of events in an asynchronous manner, with the decoupling of producers from consumers as previously discussed. This can be easily achieved using an entity which is responsible for receiving the messages from the producers, storing them and distributing them across all the consumers. This is what we refer to as a **centralised architecture**, motivated by the need of this central entity. This is the approach adopted by a lot of the message queue systems like Apache Active MQ, RabbitMQ and Redis. The usual focus for applications relying on this kind of systems is on reliability and data consistency but with a low data throughput. Typically expected to operate in a more stable environment, such as datacenters. Figure 2.2 is an example to illustrate how would a centralised pub-sub system work.

Being the broad term that it is, centralised encompasses a lot of different solutions. One can have

centralised solutions that employ a distribution of load through different nodes in order to improve the overall scalability of the system. In the pub-sub field, these networks of servers are commonly referred to as brokers. There are multiple pub-sub systems that follow this approach. More precisely Gryphon [9], Siena [11] and Jedi [10]. But, even between them, there are some clear differences on how these broker networks organise. In both Gryphon and Jedi, these nodes organise in a hierarchical fashion, or define what we call a **broker hierarchy**. As for Siena, the nodes resort to not following a specific structure, making it effectively a **broker mesh**.

The asynchronous nature of the pub-sub paradigm also allows for a different approach to message forwarding, with both producers and consumers being responsible for storing and forwarding messages, without the need of an intermediary entity. This approach is referred to as a **decentralised architecture** as there is no central entity that could easily become a bottleneck for the whole system. Additionally, when the network is **fully decentralised** it is commonly referred to as peer to peer (P2P) architecture, for it relies solely on the communication between peers in the same network. An example of a pub-sub system following this approach is Scribe [4]. This kind of systems have a great focus on scalability and, consequently, on efficient message delivery.

2.1.3 Overlay structure

Working with a P2P architecture has its own set of challenges. When we rely on the communication between peers we need a way to create and maintain links between multiple nodes in a network. Hence the overlay networks. The idea is to have a structure of logical links and nodes, independent of the physical network beneath them that actually powers the communications through. Unlike traditional layer-3 networks, the structure of these overlays is not dictated by the fairly statical physical topology (presence and connectivity of hosts), but by logical relationships between peers. This way we have the potential to manipulate the logical network at the application level, without needing to change the network backbone that connects the nodes. This approach was key to deploy P2P applications such as Gnutella [3], Kazaa [4] or Skype [5] on top of the existing Internet infrastructure.

In practical terms, each node maintains a view of its neighbours in the overlay network, which translates into the communication links between them. There are different approaches to the way this state is stored and maintained, with two main categories dominating the P2P ecosystem. At one end of the spectrum we have the **unstructured overlay** networks, where peers form a network with no clear structure or hierarchy (commonly referred to as a network mesh) with each peer connected to a subset of other nodes independent of their ID, localisation, network IP address, etc.

[3] https://web.archive.org/web/20000620113133/http://gnutella.wego.com
[4] https://web.archive.org/web/20040701062605/http://www.kazaa.com:80/us/index.htm
[5] https://www.skype.com

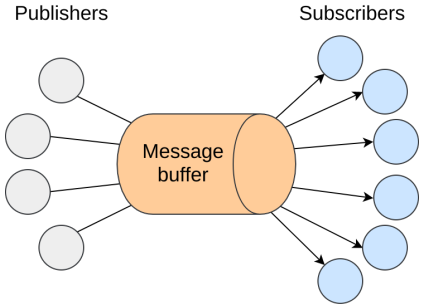

Figure 2.2: Example of a centralised message broker

Unstructured overlays: These rely on membership protocols that try to preserve a couple of key properties, such as the network diameter and its average degree. A great amount of these membership protocols use gossip based (also referred to as epidemic) approaches in order to do this. These approaches exploit properties that arise when information is disseminated in a random, or close to random, way. These probabilistic approaches help to keep the overlay connected in the event of network failures.

One relevant example is Cyclon [17], a membership protocol that uses a gossip based approach to help maintain a network which resembles a random graph in terms of degree distribution, clustering coefficient and path length. In order to do this, the approach followed by Cyclon is, at each node, besides keeping a fixed size of neighbours (other nodes in the network), to also keep information on when for the last time that node was contacted. Periodically, each node contacts the oldest node of its neighbours (i.e. the node which has been the longest time without being contacted) and shares with it a fixed size partial list of its neighbours, to which the contacted peer replies back with its own partial view of its neighbours. Each node updates its neighbours list with the new info (either by filling empty cache slots or by replacing entries that were sent in the previous contact). It is also worth noting that during this exchange, the node that initiated the contact will drop the contacted node of its neighbour list, as the contacted node will inversely add the node that established contact to his. This way we end up with a uniform and organic way to disseminate node information across the network. This approach is based on a technique named shuffling [18].

The unstructured overlay has an interesting set of properties, such as its ability to accommodate a highly dynamic network with a high resilience to network failures and churn (i.e. high volumes of changes in network participants). However, the lack of structure in the network usually limits the kind of queries for content one can run through. The delivery of messages in the network will always follow a probabilistic best effort approach. Finally, unstructured gossip based approaches rely on a pre-set of conditions that, if not met accordingly, may affect the whole behaviour of the system [19]. For example, the selection of neighbours is a key aspect and should assume a random or pseudo-random fashion. If disturbed by a small set of nodes that could either be malfunctioning or behaving selfishly, the basic properties of the network like its resilience could be severely affected.

Structured overlays: On the other end of the spectrum of overlay networks we have the **structured overlay**, where peers are organised according to a specific structure, like a ring, a tree or a

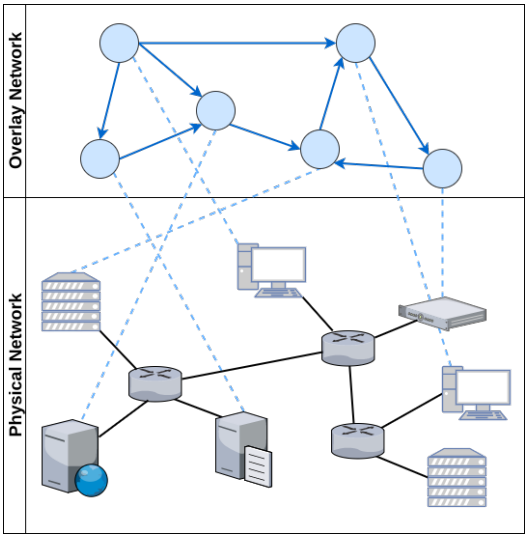

Figure 2.3: A comparison between the physical network and a logical overlay

multi-dimensional space. This is usually achieved by imposing constraints on how the nodes should be organised based on their identifiers. In order to do this, a common approach is to think of the ID space as a hash table to where the content should then be distributed. The distribution of content is then done based the value of the keys generated for each piece of information, keys with values close to a node ID will be stored in that node. This is commonly referred to as a Distributed Hash Table (or DHT for short) since the key space is distributed across multiple nodes. For example, **Chord** [20], one of the first examples of a DHT, organises the nodes in a ring like structure based on their ID (which results from the SHA-1 hash [6] of its IP address). The content is then distributed in this key space, using the same hashing function to produce the content key that was used to produce the node ID.

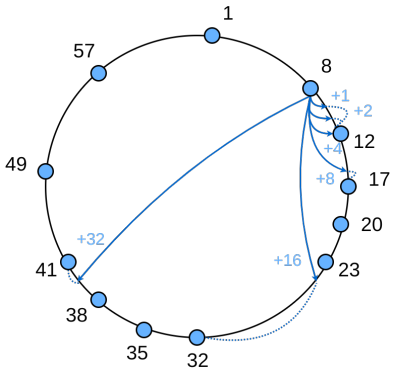

Figure 2.4: Example of a simple Chord ring and the finger table of a node

It is common for Distributed Hash Tables to have a cost of O(logN) in terms of the number of nodes contacted, on average, to search for a given key (where N is the number of nodes in the network). Chord base structure per se only gives us O(N), as such, Chord uses a mechanism to allow for a speedier search. At each node, an additional routing table is kept with m entries, where m is the number of bits in the key space. Each ith entry in this table will be this node's successor (next node in the ring in a clockwise direction) with an ID, at least, bigger than 2^{i-1} (modulo 2^m) in the key space. For example, for a node with ID 8, the 4th entry will be the first node in the ring with an ID larger than 16. This table, also referred to as finger table, will allow for a logarithmic search as demonstrated in Chord's specification.

Another approach is followed by **Kademlia DHT** [21]. Just as in Chord, nodes have 160 bit identifiers and content is stored in the nodes whose IDs are close to the content key (160 bit identifiers too), but the way the routing tables are structured and maintained is quite different. For starters, Kademlia relies on a XOR based distance metric between 2 keys, where the distance between 2 keys is the resulting bitwise XOR operation interpreted as an integer. The XOR metric gives us an interesting set of properties. It is unidirectional (just like Chord clockwise direction) ensuring that lookups for the same key converge along the same path but, unlike Chord, it is symmetric, as such, the distance between x and y is the same as the distance between y and x. This symmetry allows Kademlia queries to give valuable insights along every node they go through, helping out in populating each node's routing table.

Kademlia nodes keep contact information about each other in a list, size m where m is the number of bits used for the keys in the system, and where each entry is a list itself of maximum size k (a system wide parameter) containing all the known nodes of distance between 2^i and 2^{i+1} of itself. These lists are appropriately called k-buckets and are kept sorted by time last seen (least recently seen node at the head). Whenever a node receives a message, it updates the appropriate k-bucket for the sender's node ID, inserting it in the respective k-bucket or moving it to the tail of the list if it is already there. K-buckets aim at implementing a least-recently seen eviction policy, where live nodes are never removed. This

[6] https://tools.ietf.org/html/rfc3174

stems from a careful analysis of Gnutella trace data [22] where the longer a node has been up, the more likely it is to remain up for another hour. Whenever a node wants to retrieve or store content it uses a recursive node lookup procedure in order to find the k closest nodes to a given key. This lookup can be run with multiple queries in parallel, because nodes have the flexibility to forward messages to any of the k nodes in a bucket, aiming for lower latency.

A completely different method is used in the **Content Addressable Network DHT**[23]. In CAN, the key space used to address the content stored in the DHT is a virtual d-dimensional Cartesian coordinate space. In order to store and retrieve content, the generated keys use a uniform hashing function that maps the key into the d-dimensional space, resulting in a point. The overall space is split into different areas referred to as zones. Each node is responsible for a zone and, consequently, for all the keys stored in that zone. Retrieving a key can be done by calculating its corresponding point in the d-dimensional space and, if the point does not belong to this node space or any of its neighbours (nodes responsible for adjacent zones) it can be routed through CAN infrastructure in order to find the node responsible for storing the key. Intuitively, routing in CAN works by following the straight line from the source to the destination coordinate in the Cartesian space. In practice, this is done by forwarding the message to the neighbour closest to the destination coordinate. Interestingly enough, the usage of a multidimensional space as the key space for the DHT, makes the distance metric in the CAN DHT as a simple Cartesian distance between two points.

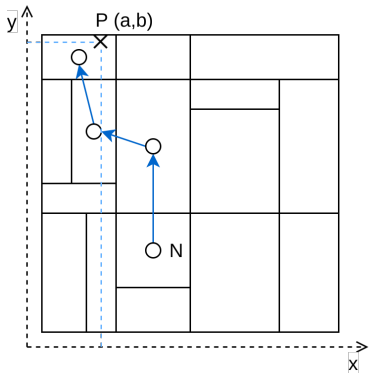

Figure 2.5: Example of a 2 dimensional CAN routing command

Other popular examples in the DHT field are **Pastry** [24] and **Tapestry** [25] (that we kept out for the sake of simplicity, although a lot of the mechanisms described above apply to these). DHTs present a set of interesting benefits, such as good routing performance (usually logarithmic in the number of nodes), the limited size of state kept at each node (usually logarithmic routing tables), a better support for exact match and other complex queries and also present stronger guarantees on message delivery. If the hashing function is properly selected it can also be ensured that the load is balanced properly across the network. However, these networks lack the tolerance for heavy network partitions and network churn that the usual unstructured network can bare with.

Hybrid overlays: As with everything discussed so far, not every solution lies at each end of the spectrum, and overlay structure is no different. Recent research has been pushing more and more towards hybrid solutions that take advantages of both sides. Such example is **Vicinity** [26] which employs Cyclon (discussed above) as a peer sampling service to help out in building an ideal structure that links nodes based on their proximity (for some notion of proximity, e.g. latency, localisation, etc.). In the end, we get a structured overlay, generated from an unstructured, gossip based, overlay (hence the hybrid solution). More importantly, this overlay will have properties that guarantee that it is an almost ideal structure for a

given proximity metric. The Vicinity system discusses that the usage of probabilistic mechanisms helps out in keeping a healthy and reliable structure.

2.1.4 Subscription Management and Event Dissemination

Now that we have set the underlying structures that power up the network, it is time to cover the specific requirements of a pub-sub system. We have two different aspects to cover: subscription management and event dissemination. By subscription management we refer to a set of key factors that will determine the overall performance of the pub-sub system, specifically in terms of matching events with subscribers, the selected representation for subscriptions, registering new subscriptions and deleting subscriptions. Event dissemination dictates how will the events be propagated through the system, in a way that avoids burdening specific nodes, but assures that all the subscription requirements are met. It is natural that in some ways these two aspects are connected (e.g. the way we store our subscriptions will probably impose a set of restrictions on how our events will be propagated) but it is still possible to make a clear distinction between how they work and their role in the overall system.

As discussed before, in order to match subscribers with publishers, some kind of state must be kept (what we refer to as subscriptions). There are plenty of ways of doing this and factors like network architecture and subscription model come into play here. For a system with a centralised architecture, this is not such a big challenge, since the central nodes will be responsible for keeping and managing the state, matching events with the correct subscribers and making sure the event propagation works accordingly. However, in a distributed or a decentralised scenario, this is not such a trivial problem to solve.

One interesting property of topic based systems in a decentralised and distributed scenario is that their subscription management and event dissemination can be easily implemented with an application level multicast system if we cluster subscribers of some topic/group in a single structure (e.g. a multicast tree). For example, consider the topic */foobar* issued by a particular node in a pub-sub system. If, when new subscriptions are issued to this node, a tree like structure is built that allows events related to this topic to flow accordingly, disseminating a new event in */foobar* is just a matter of sending the event to the root of the tree. From there, dissemination can flow blindly through the multiple links. Subscriptions are then represented as simple dissemination trees for each topic, which, interestingly enough, ends up also representing how the actual events will be propagated in each topic. The root node (or nodes) acts as a *rendezvous* which, as the name suggests, it is where events are targeted at and new subscriptions issued to. The core idea is that, by relying on such nodes, eventually, all the system state will be synchronised (all the events will be propagated to the expected nodes and no subscription is left unattended). This does not mean that other nodes cannot cache state though, the idea of the *rendezvous* is to have a basic reassurance in subscription management and event dissemination. Ideally this would be implemented in a distributed fashion, keeping as much pressure out of the *rendezvous* node as possible. This is the approach followed by Scribe and Bayeux.

The usage of *rendezvous* nodes and tree like structures to represent subscriptions is not something particular to topic based systems. There are examples of these techniques in content based systems also, specifically Gryphon and Jedi. Hermes, on the other hand, is an example of the same mechanisms with a type based subscription model. A more detailed description of how this is done in Gryphon and Scribe will be made further along since they have different approaches motivated by their different options in network architecture and subscription model.

For content based systems though, a common approach is to use multidimensional spaces as a way to represent subscriptions. The idea is to have each dimension refer to a specific attribute of the pub-sub schema.

```
{
  exchange: String,
  company: String,
  order: String,
  number: Integer,
  price: Float,
}
```

Considering the example above, we could map each of the given attributes to a given dimension and end up with a 5 dimensional space that we could use to route events accordingly. Meghdoot is an example of a content based pub-sub system that follows an approach close to this one, using a CAN DHT with $2n$ dimensions, where n is the number of attributes in the schema. We will cover Meghdoot further down, but it is worth mentioning that there are other alternatives to using a multidimensional space DHT to replicate this behaviour. Mercury for example relies on the usage of several ring-based DHTs to recreate this multidimensional space and support range queries, using one DHT per attribute.

A different approach to managing subscriptions and disseminating events in topic based systems is by having an overlay for each different topic. The idea is that by clustering nodes one can afford an easier event dissemination as well as an easy way of matching events with subscribers, since it is just a matter of propagating a given event inside its overlay. In order to keep everything connected, a general overlay can be used, that will allow all the nodes to have visibility on the whole set of topics. In this scenario, subscriptions are simply represented as being part of a specific network of peers, that could take any form or shape, or even be unstructured. For an unstructured network, the propagation of events could be a simple flooding algorithm, as it happens in Tera. Tera, a topic based pub-sub system, follows an approach close to this one. It keeps two distinct gossip based overlays, one responsible for keeping state on entrypoints for each topic (peers which are subscribed to a given topic and that can act as dissemination points for it) and another used to keep the subscribers of each topic. This clustering approach, where subscribers of a given topic are kept in a topic specific overlay, helps out in the dissemination step after an event has been published and reached the cluster. Another example following this approach is Poldercast, which uses a set of three different overlays to keep the pub-sub network running. We will cover Poldercast more thoroughly later on.

2.2 Relevant Pub-Sub Systems

We now describe in further detail the systems which most resemble the work we are going to do.

2.2.1 Gryphon

Gryphon [9] is a content based pub-sub system built on top of a centralised broker hierarchy topology. Developed at IBM, Gryphon uses an interesting approach to match events with subscriptions [27]. Gryphon relied on a distributed broker based network to build a tree structure representing the subscription schema. Considering a schema with multiple attributes - $A1, \ldots, An$ - each level on the subscription tree would represent a specific attribute. So, for example, if we were to have an event with a value $V1$ for the attribute $A1$, at the root node (which represents the attribute $A1$) the link followed by the event would be the one that would represent the value $V1$. The event would then be propagated through the multiple branches of the tree until it arrives at the broker node that represented all the specific values for that event. From there it would then be propagated to all the subscribers registered with that broker node. Figure 2.6 illustrates this approach.

When a client issues a new subscription, the same approach will be followed until the subscription arrives at the broker node that represents it. If for some reason, the tree does not have an edge for a specific value of an attribute, a new edge will be created. During both of these approaches (subscription and event propagation), a subscription or event that does not name an attribute at a given level will follow the edge with label $*$ (do not care).

Gryphon has been successfully deployed over the Internet for real-time sports score distribution at the Grand Slam Tennis events, Ryder Cup, and for monitoring and statistics reporting at the Sydney Olympics [7].

2.2.2 Siena

Siena [11] is a content based pub-sub system built on top of a centralised broker mesh topology. Siena does not make any assumptions on how the communication between servers and client-server works, as this is not vital for the system to work. Instead, for server to server communication, it provides a set of options ranging from P2P communication to a more hierarchical structure, each with its respective advantages and shortcomings.

Events in Siena are treated as a set of typed attributes with values. Consequently, subscriptions (or *event filters* as they are referred to in Siena) select events by specifying a set of attributes and constraints on its values. When issuing a new subscription, a client sends its subscription to its broker node, which then forwards it throughout the network. At each node, the subscription leaves some state behind, identifying it and the neighbour which previously forwarded the message. This is crucial, for these will be the dissemination paths that events will follow when travelling through the network. Siena also defines an interesting concept of *subscription coverage*. A subscription *S* is covered by a subscription *M* if, whenever *S* is matched by an event *e*, then *M* is matched by *e* as well. Although a simple concept, it saves a considerable overhead during subscription dissemination and processing. A broker that detects a link with a more general subscription will not need to forward subscriptions that are covered by it.

[7] https://www.research.ibm.com/distributedmessaging/gryphon.html

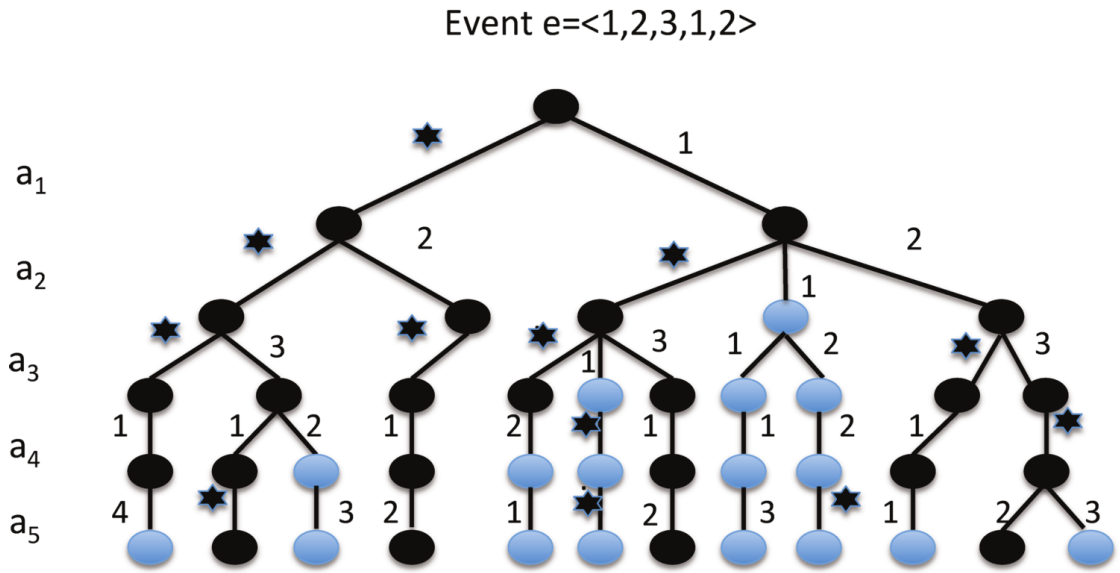

Figure 2.6: extracted from [2]. Each level of the broker tree represents an attribute. When the event $e =< 1, 2, 3, 1, 2 >$ is published, all dark circles (representing brokers) are visited.

Event dissemination will work based on the previously set state at each broker node. In the end, events will technically follow the reverse paths of the subscriptions. A detail worth noting is that Siena optimises for *downstream replication*, that is, events should be routed as one copy as far as possible and should replicate only downstream.

Interestingly enough Siena also proposed another influential idea, which is the idea of *advertisements*. This concept could be viewed as a reverse subscription. The concept is simple, a node advertises to the network the type of content it is producing. In this paradigm, advertisements are propagated throughout the network and when a subscription is issued, it follows the paths previously set by the advertisements, effectively *activating* the path. Events are then forwarded through these activated paths.

2.2.3 Scribe

Scribe [4] is a topic based pub-sub system built on top of a fully decentralised network (P2P). In order to do this, it relies on Pastry DHT as its overlay structure. This allows it to leverage the robustness, self-organisation, locality and reliability properties of Pastry. Pastry DHT is at all similar to the DHTs described in the previous section (Chord and Kademlia), with a specific effort on achieving good network locality and a routing mechanism close to that of Kademlia.

Scribe subscriptions are represented by a multicast tree, with each different tree representing a specific topic (or *group* as it is referred in Scribe). The root of this tree acts as the *rendezvous* node for the group. Each group has a *groupId* assigned to it, as such, the *rendezvous* node will be the one with the closest ID in the network. This multicast tree is built by joining the multiple Pastry routes from each group member to the *rendezvous*. This dynamic process happens whenever a new node decides to join a group. In order to do that, it asks Pastry to route a *JOIN* message with the *groupId* as the key. At each node along the path, the Scribe forward method is invoked to check this node is already part of this group (also called a *forwarder*). If it is, it accepts the *JOIN* request and sets the node as its child, else this node will become a *forwarder* for the group, sets the requesting node as its child and it sends a *JOIN* request to this group. Note that any node can be a *forwarder* for any group, it does not need to be an active part of it (i.e. subscriber or publisher).

Disseminating an event in a group is a matter of disseminating it through its respective multicast tree. Fault tolerance mechanisms can be implemented on top of this system but, out of the box, Scribe provides only best effort delivery. As for the *rendezvous* nodes, their state can be replicated across the k closest nodes in the leaf set of the root node. Whenever a failure is detected by one of the children, it will issue a *JOIN* message which, thanks to Pastry's properties will be routed to a new root node which has access to the previous state of the *rendezvous*.

2.2.4 Meghdoot

Meghdoot [12] is a content based pub-sub system. It is built on top of a P2P network, specifically CAN DHT (already covered in the previous section). Meghdoot leverages the multidimensional space provided by the CAN DHT in order to create an expressive content based system.

In Meghdoot, subscriptions are defined over a schema of *n* attributes. Each attribute has a *name*, *type*, and *domain*, and can be described by a tuple $\{$*Name: Type, Min, Max*$\}$. *Min* and *Max* describe the range of domain values taken by the given attribute. All the peers in the system will use this same model. A subscription will then be a set of predicates over the previously defined attributes. In order to map this to the CAN DHT, Meghdoot defines the n attributes schema as a $2n$ dimensional space in the DHT. Subscriptions will be a point in this multidimensional space, where the range query it defines will be represented as two separate dimensions per attribute in the DHT (hence the $2n$ space). Each

subscription is routed through the CAN DHT until it reaches the peer responsible for managing the zone it is part of.

Event dissemination in Meghdoot will be a matter of routing each event through the CAN DHT. The events too will be defined by points in the multidimensional space. The point will be represented by the value of that same attribute in each of dimensions used to map it. For example, for a 2 dimensional space (x, y) (only one attribute), an event with a value z would be mapped to a point $x = y = z$. Once the event arrives at the node responsible for managing that specific zone in the DHT, it will be up to it to route the event to all of its neighbours that are part of the region affected by it. An interesting property of the $2n$ dimensional space is that half of it is left unexplored by the default subscription algorithm. This allows that space to be used to persist replicas of the subscriptions on the other half, making Meghdoot a system with fault tolerance by default.

2.2.5 Poldercast

Poldercast [7] is a recent pub-sub system with a strong focus on scalability, robustness, efficiency and fault tolerance. It follows a topic based model and follows a fully decentralised architecture. The key detail about this system is that it tries to blend deterministic propagation over a structured overlay, with probabilistic dissemination through gossip based unstructured overlays. In order to do this, Poldercast uses 3 different overlays. Two of them, Cyclon and Vicinity, we covered in the previous section and the third one closely resembles Chord in many ways.

Poldercast subscriptions are represented as a structured ring overlay. Each topic has its own overlay in fact, with all subscribers (and only them) of the corresponding topic connected to it. This overlay is maintained by a module referred to as the *Rings Module* and its overall mechanisms closely resemble Chord's ones. In order for each node to have visibility across the whole pub-sub network, Vicinity, with the help of Cyclon, will be responsible for keeping an updated set of peers participating in each of the available topics in the network. Subscribing to a topic will then be a matter of consulting this set of peers and joining the specific overlay for the topic.

Propagating events will be a matter of forwarding the event through the specific topic overlay. It is important to notice that Poldercast assumes only peers subscribed to a topic can publish to that same topic. The way this propagation works is through the ring overlay that, despite being similar to Chord, it has some important differences. It does not use a finger table at each node to speed up propagation. Instead, with the help of Vicinity, each node keeps a random set of peers for the topics it is part of. With them, whenever a node receives a message from a specific topic, it will propagate the event through a set of these peers. This propagation will be based on a system wide fanout parameter. It will also forward the event to its successor or predecessor (depending on where the event originated from), or will simply ignore if it is not the first time it has received it. These mechanisms, depending on the fanout parameter, guarantee average dissemination paths for each topic to be asymptotically logarithmic.

Through the multiple mechanisms described above, Poldercast attempts to provide a set of guarantees. To start with, only nodes subscribed to a topic will receive events published to that topic. In other words, no relay nodes are used. It also focuses on handling churn through the use of a mixture of gossip mechanisms, ensuring a highly resilient network. Finally, it seeks to reduce message duplication factor (i.e. nodes receiving the same message more than once).

2.2.6 Systems Analysis

Let us now analyse all the relevant pub-sub systems we used as a basis for our work. Table 2.1 will serve as a useful comparison mechanism for this. A couple of notes on the terminology used. We

refer to *delivery guarantees* as the ability to deliver a message under normal working conditions and *fault tolerance* as the ability to keep such guarantees under churn. This, of course, depends on the persistence of subscriptions and mechanisms to replicate these. The rest of the criteria were addressed in the previous sections.

Systems / Properties	Subscription Model	Architecture	Overlay Structure	Subscription Management	Event dis-semination	Relay Free Routing	Delivery Guarantees	Fault Tolerance
Gryphon	Content based	Centralised broker hierarchy	N/A	Each broker responsible for a subscription scheme	Tree hierarchy	N/A	Yes	Best effort
Siena	Content based	Centralised broker mesh	N/A	Keep state at each node	Flood with cached state	N/A	Yes	Best effort
Jedi	Content based	Centralised broker hierarchy	N/A	Keep state at each node	Tree hierarchy	N/A	Yes	Best effort
Bayeux	Topic based	Decentralised	Tapestry DHT	Rendezvous node	Multicast tree	No	Yes	Best effort, no subscription persistence
Scribe	Topic based	Decentralised	Pastry DHT	Rendezvous node	Multicast tree	No	Yes	Best effort, no subscription persistence
Medhdoot	Content based	Decentralised	CAN DHT	Points in CAN DHT	CAN routing	No	Yes	replicated subscrip-tions
Hermes	Type based	Decentralised	Pastry DHT	Rendezvous node	Multicast tree	No	Yes	Best effort
Tera	Topic based	Decentralised	Gossip based overlay	Unstructured overlay per topic	Random walks and flooding	No	no	Best effort
Mercury	Content based	Decentralised	Ring based DHTs	Overlay per attribute in schema	Route through ring overlays	No	Yes	Best effort
Sub-2-Sub	Content based	Decentralised	Ring based DHT and gossip based overlay	Clustering of similar subscriptions	Gossip and ring overlay routing	No	No	Best effort
Poldercast	Topic based	Decentralised	Ring based DHT/ Vicinity / Cyclon	Ring overlay per topic	Ring overlay routing	Yes	Yes (every publisher is a subscriber)	High resilience to churn, no subscription persistence

Table 2.1: Comparison table for the relevant system

2.3 Web Technologies

When building any kind of network focused system nowadays, there is no question that one should take advantage of the full potential that the web has to offer. Browsers evolved a lot over the past years and allow for a vast world of possibilities in terms of applications that can be built on top of it. P2P applications are no exception here. In the next sections, we will cover a set of technologies that allow for a modern distributed application to run, not only on the desktops and servers we are used, but also in browsers running in multiple platforms.

It is indisputable that one cannot think of modern web development without speaking of **Javascript** [8]. Javascript is a lightweight, interpreted, programming language, known as the scripting language for the web. Initially created with the purpose of allowing the creation of simple interactions and animations in web pages it is now one of the main programming languages for the web [9]. It is used, not only for client side programming but also to power server side applications. Since Javascript has different runtimes, it became necessary to create a standardised base from which the multiple browser vendors and runtime maintainers could work from. Hence ECMAScript, the standard for the Javascript implementation.

As it was previously said, nowadays, Javascript is not restricted to browsers only. **NodeJS** [10] was the first successful implementation of a Javascript runtime for the server, built on top of Chrome's V8 JavaScript engine [11]. This allowed developers to write and run Javascript programs in multiple architectures and operating systems, with access to a set of common native libraries that allow to interact with relevant parts of the system [12] such as network, filesystem and others. A key aspect in NodeJS was the way it chose to deal with the lack of support from Javascript for multithreading: the use of an event loop that powers an event-driven architecture capable of asynchronous I/O.

Yet another key element in the NodeJS and Javascript ecosystem is **NPM** [13], its package manager. NPM was one of the main drivers of a philosophy that is deeply ingrained in the JS ecosystem which focuses on building small reusable packages that everyone can use and build on top of. This is heavily inspired by the UNIX philosophy summarised by Doug McIlroy [14] - "Write programs that do one thing and do it well. Write programs that work together". This approach ended up being a major differentiator on how modern web applications are developed. Currently, NPM is one of the largest package registries in the world [15]. This mindset though is really important, for it allows applications to be built on top of previously published packages, making modularity and code reusability core values of the ecosystem. Even more interesting is the sudden possibility offered by having the same programming language supporting different environments (browsers, servers, desktops, etc.), all of this powered by a common way of publishing and sharing code.

When focusing specifically on P2P apps, the past years have brought together a set of new network protocols that empower communication between browsers in a real-time fashion and also provide alternatives to TCP [16]. **WebSockets** [17] aimed at providing a real-time, full-duplex communication between clients and servers over TCP, but it was **WebRTC** [18] that paved the way for new P2P applications that could run in the browser. WebRTC focuses on powering real-time communications, like audio/video stream or just arbitrary data, between browsers, without the need of an intermediary. This, of course,

[8] https://www.ecma-international.org/publications/standards/Ecma-262.htm
[9] https://insights.stackoverflow.com/survey/2017
[10] https://nodejs.org
[11] https://developers.google.com/v8/
[12] https://nodejs.org/api/
[13] https://www.npmjs.com/
[14] https://archive.org/details/bstj57-6-1899
[15] http://blog.npmjs.org/post/143451680695/how-many-npm-users-are-there
[16] https://tools.ietf.org/html/rfc793
[17] https://tools.ietf.org/html/rfc6455
[18] https://www.w3.org/TR/webrtc/

is a real breakthrough in the P2P field as it allows browsers to receive incoming connections. On other hand, alternatives to the TCP transport, such as **uTP** [19] and **QUIC** [20] , came through, seeking to bring reliability and order delivery without the poor latency and congestion control issues of TCP. This provided new suitable alternatives to communication between peers on top of UDP, a transport that has been vital in P2P applications that need an affordable way to perform NAT [21] traversal.

In the application realm, there have been quite a few in the past years that seek to leverage all these new technologies and breakthroughs. One of the examples most worth mentioning is the **InterPlanetary File System** (IPFS) [22], a P2P hypermedia protocol designed to create a persistent, content-addressable network on top of the distributed web.

At the core of IPFS is what they refer to as the **Merkle DAG** [23]. The Merkle DAG is a graph structure used to store and represent data, where each node can be linked to based on the hash of its content. Figure 2.7 provides an example of this.

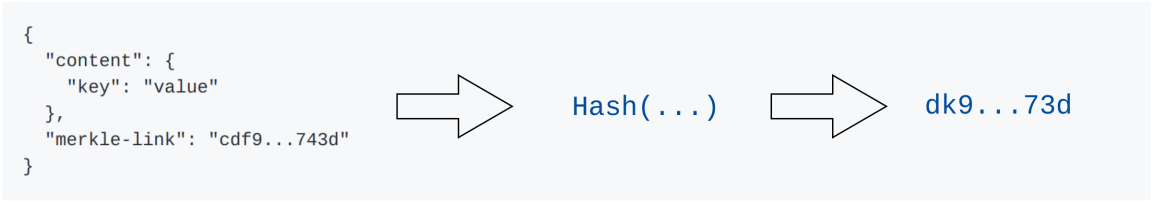

Figure 2.7: JSON representation of a Merkle node with a Merkle link

Each node can have links (Merkle links) to other nodes, creating a persistent, chain like, structure that is immutable as documented in figure 2.8

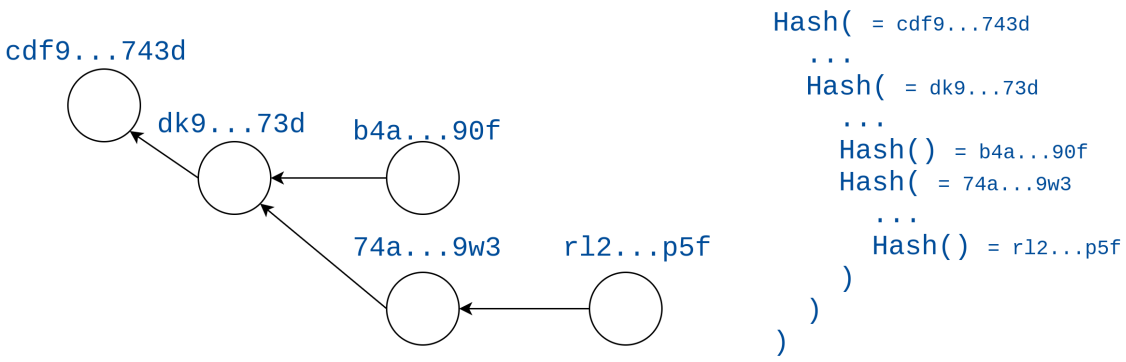

Figure 2.8: Graph visualisation of a Merkle DAG and its respective hash function dependencies

IPFS has an interface around this structure referred to as **InterPlanetary Linked Data** (IPLD) which focuses on bringing together all the hash-linked data structures (e.g. git) under a unified JSON-based model. In order to interact with IPLD, IPFS exposes an API that allows us to insert and request random blobs of data, files, JSON objects and other complex structures. Having implementations in both Go and Javascript, IPFS leverages the modularity mantra in a fascinating way, focusing on creating common interfaces that allow for different pieces of the architecture to be changed and selected according to one's needs. All of this without impacting the overall application and its top level API. These came from the observation that the web we have today is a set of different heterogeneous clients, that have

[19] http://www.bittorrent.org/beps/bep_0029.html
[20] https://datatracker.ietf.org/wg/quic/about/
[21] https://tools.ietf.org/html/rfc2663
[22] https://ipfs.io
[23] https://github.com/ipld/specs/blob/95df205ca5fdb961ec2c2265a169989fef595db1/FOUNDATIONS.md

different needs and resources. As such, not everyone can rely on the same set of transports, storage management and discovery mechanisms. These small modules that constitute IPFS have recently been brought together under the same umbrella, as **libp2p** [24], a set of packages that seek to solve common challenges in P2P applications.

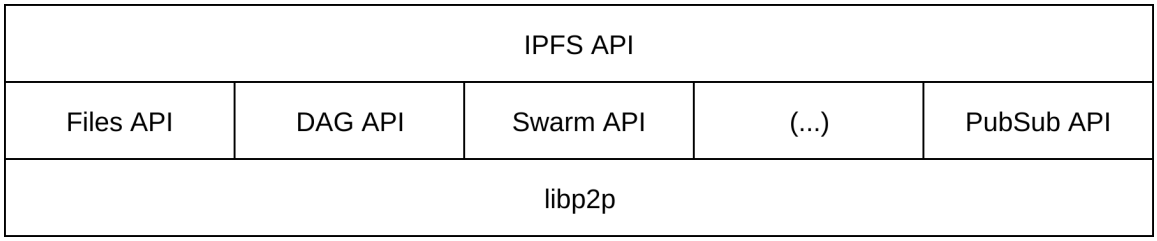

Figure 2.9: An illustration on the IPFS architecture

Interestingly enough, a recent addition to libp2p, and consequently IPFS, was a pub-sub module, with a naive implementation using a simple network flooding technique. Even though libp2p was created with the initial purpose of serving as the foundation of IPFS it is now possible to use libp2p as a standalone module for peer to peer apps, with the possibility to hand pick the functionalities we intend to use.

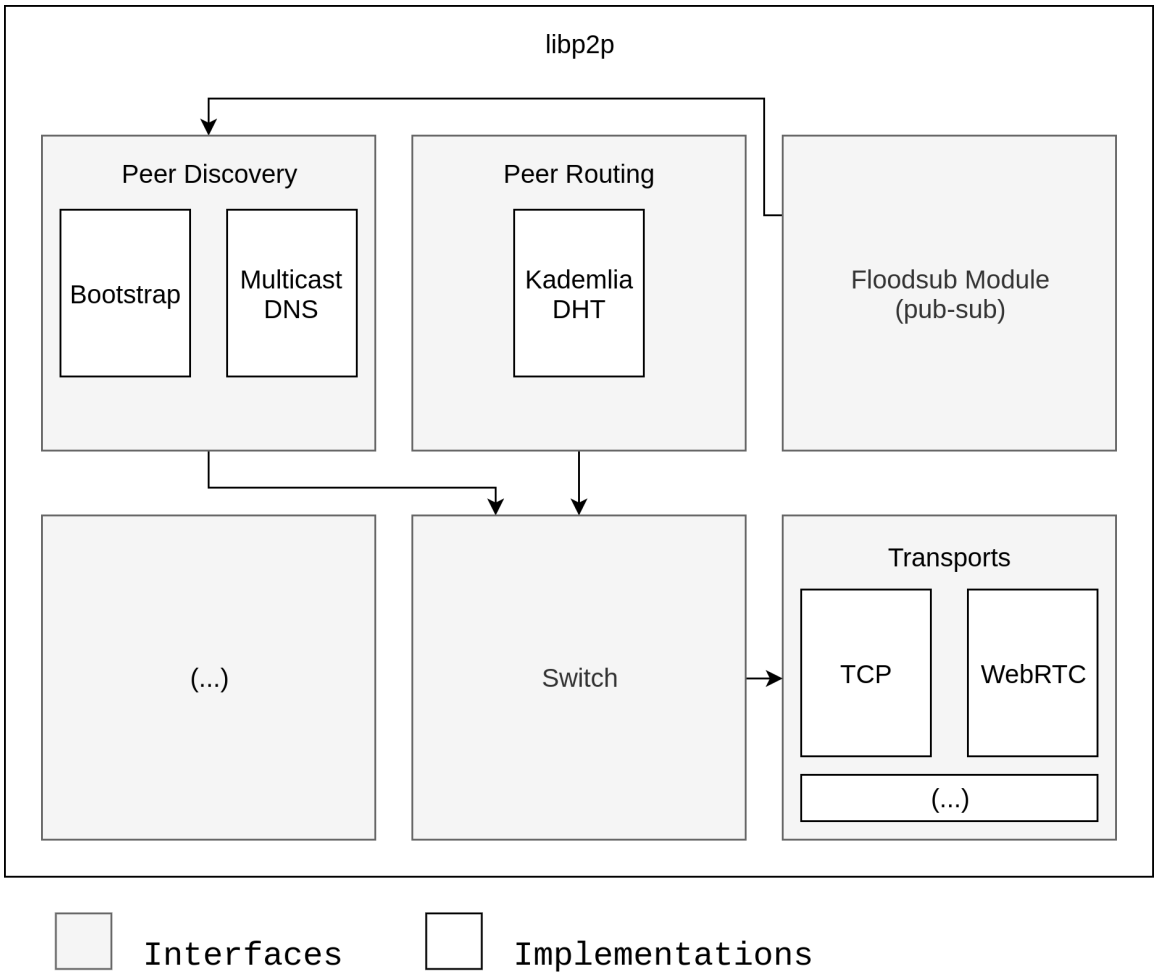

Figure 2.10: An illustration on the libp2p architecture

[24] https://libp2p.io

2.4 Summary

TODO

Chapter 3

Pulsarcast

Pulsarcast is a peer to peer, pub-sub, topic-based system focused on reliability, eventual delivery guarantees, and data persistence. We seek this while not fully compromising the scalability given by the decentralised nature of our architecture. Looking through our related work, it became clear that few fully decentralised solutions exist that try to provide this kind of guarantees. Yet, if we carefully look at the most popular and widely adopted centralised pub-sub solutions, it is clear that most of them heavily rely and depend upon these same guarantees.

We opted for the more straightforward topic-based subscription model given that, in our view, a well structured and implemented topic-based model is more than enough for a significant percentage of our use cases. In the end, we compromise a bit of the expressiveness of the system in order to avoid bringing more complexity in, something we believe will pay off.

We divided this chapter as follows. We start by covering the use case of our system in section 3.1, where we will introduce some of our broader architectural decisions. Next, we will move to section 3.2 where we will deep dive into Pulsarcast's data structure model and how it is distributed across the network. Finally, we will look into one of the most critical parts of our architecture in section 3.3 where, based on the covered related work and the taxonomy we defined, we present the algorithms and mechanisms used for managing the subscriptions and event dissemination. (TODO missing more sections)

3.1 Use Case

Pulsarcast is a fully decentralised solution, which means that each node plays a crucial part in fulfilling the system purpose, delivering events and ensuring its dissemination. Conceptually speaking, Pulsarcast provides four methods for clients and applications to interact with the system, create a topic, subscribe to a topic, unsubscribe from a topic and publish an event in a topic. From a broader perspective, Pulsarcast relies on two overlays to fulfil its needs. Kadmelia DHT, used for a range of purposes from peer discovery, content discovery and to bootstrap our other overlay, the Pulsarcast overlay. The Pulsarcast overlay is actually a set of different overlays or, as we call it, dissemination trees. These trees are on a per topic basis and are the critical factor in the way we disseminate information across our decentralised network. Figure 3.1 illustrates the multiple overlays in action.

The subscription model followed by Pulsarcast is a topic based one, but still allowing for some expressiveness through the usage of sub-topics to enable more complex structures. When a peer publishes an event or creates a new topic a set of the overlays described is used accordingly. For Pulsarcast, both of these actions, happen to take a similar course. That is because the system views these pieces

of information (or descriptors as we call it) as fairly similar, given their importance. Figures 3.2 and 3.3 provide an overview of what flow for creating this information and for accessing it looks like.

Pulsarcast's goal is to give users and any applications built on top of it the reassurance that events reach their destination and that they can rebuild as much of the event or topic history as they see fit. As such, we double down on our efforts to persist and propagate data. Every topic and event is stored in the Kadmelia DHT before being forwarded through the topic dissemination trees. This ensures the data

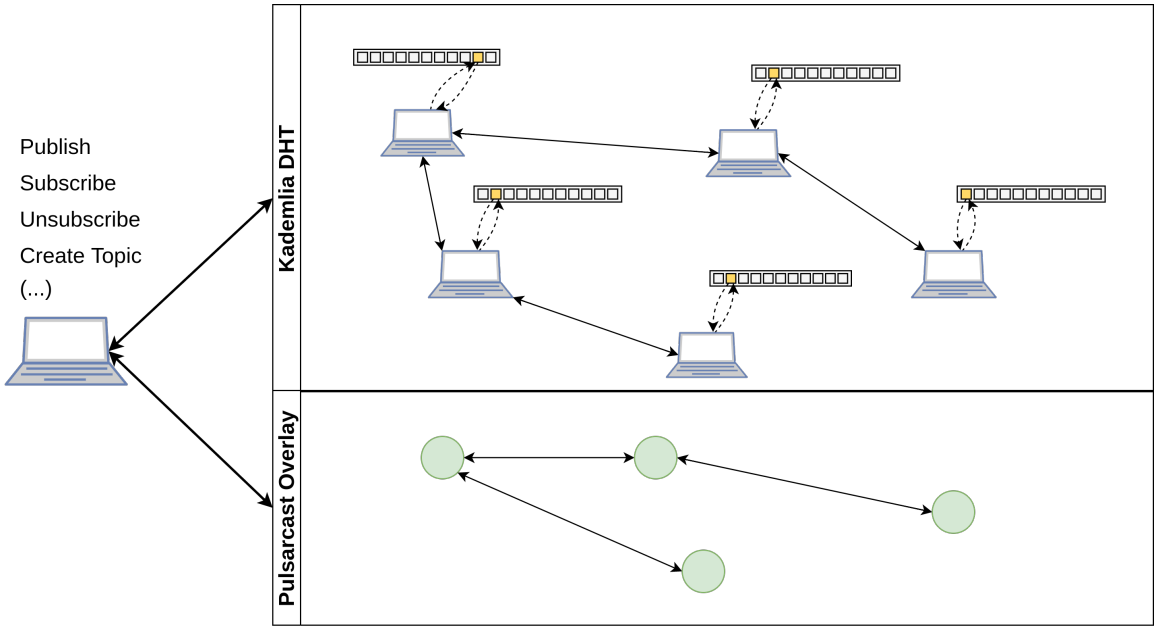

Figure 3.1: Representation of the Pulsarcast overlays

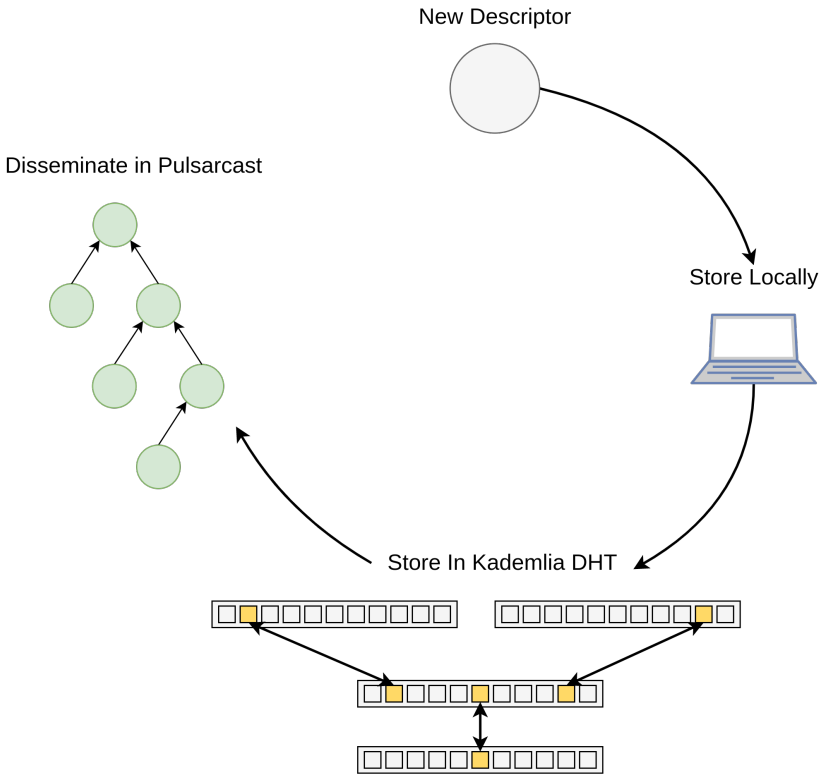

Figure 3.2: Flow for creating a new Topic/Event descriptor

is persisted by a set of nodes (that might even be extraneous to the topic at hand) and anyone is later able to fetch the data using only the DHT if they want to. Afterwards, we forward the data through the appropriate dissemination trees previously built (we will cover this process in section TODO). Currently, every node participating in the data transmission through the dissemination tree stores it indefinitely, although it is something due to being changed in a later revision of our protocol. On the other hand, when someone wants to fetch a piece of data (a topic or an event) it starts by performing a local search in the system, it might have been something that the node as ran through when forwarding events across their dissemination trees. If this fails, though, a query to the DHT is in order.

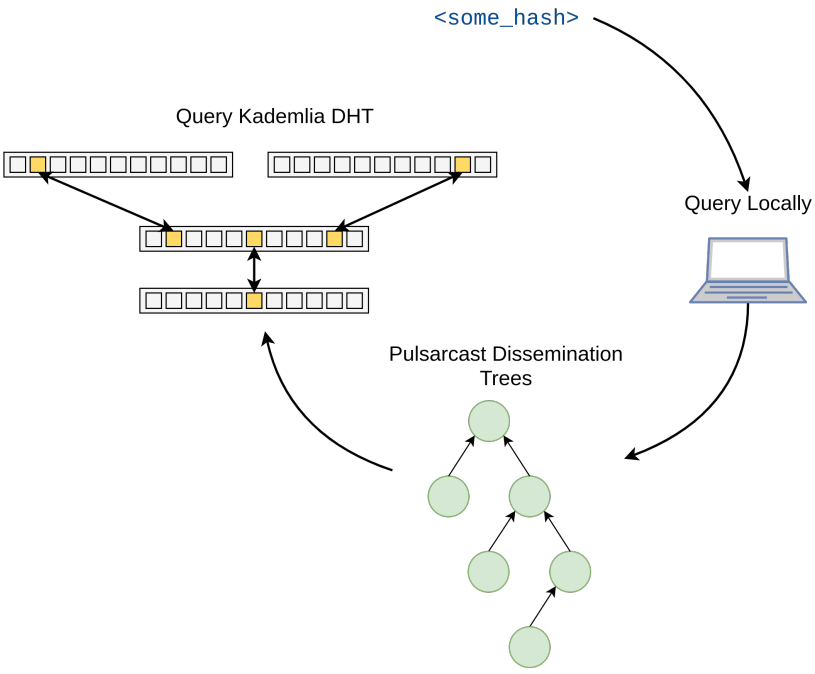

Figure 3.3: Flow for querying a Topic/Event descriptor

3.2 Data Structures

Pulsarcast has a set of two fundamental data structures to which we refer to as event and topic descriptors. To help us represent this data, we rely on a concept already introduced in our related work, the Merkle DAG.

3.2.1 Content-Addressability

All of our data structures are immutable, content addressable and linked together to form a Direct Acyclic Graph (Merkle DAG). Events link both to their respective topic descriptor and a past event in that topic. Topics, on the other hand, link to their sub-topics (if any) and a previous version of themselves. Figure 3.4 provides a broader picture of how it all fits together. Immutability and content-addressability give us verifiability. Consequently, the assurance that the state of our distributed system is the same no matter where we are accessing it from or who is viewing it. It also allows us to build a notion of history which plays nicely into a pub-sub scenario. Through these links and the mechanisms described in the previous section, users and applications are free to rebuild their topic and event history to any point they wish. Be that because they were not part of the network at the time or because they missed out due to some system or network failure, acting as a NACK (not acknowledged) for relevant events.

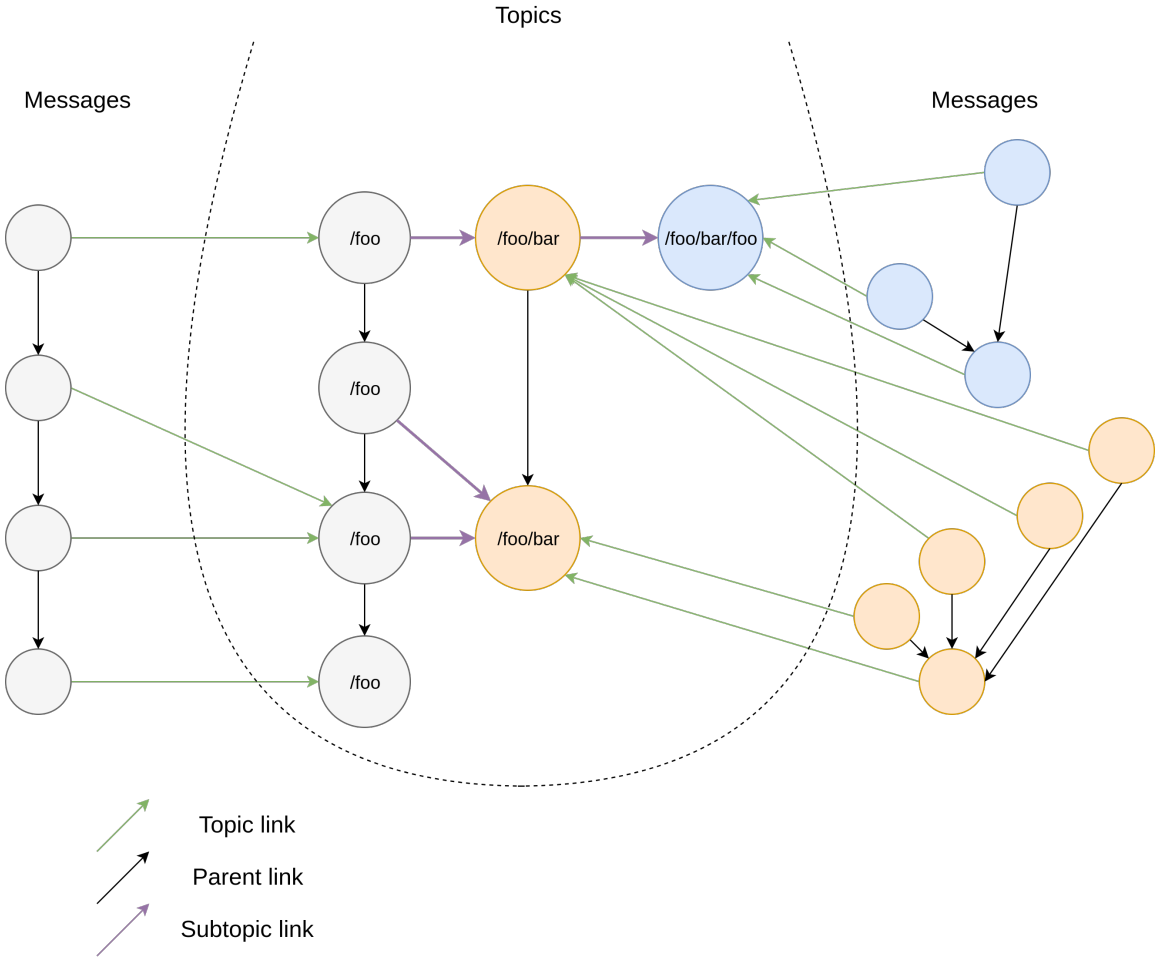

Figure 3.4: Representation of the Pulsarcast DAG

Given we are discussing addressability and linking between content, the representation used for our identifiers is an important part of our system specification. That was one of the main reasons for

```
{
  "name": <string>,
  "author": <peer-id>,
  "parent": {                          //The parent link for this topic
    "/": <topic-id>
  },
  "#": {                               //Sub topic links
    "meta": {                            //Meta topic
      "/": "zdpuAkx9dPaPve3H9ezrtSipCSUhBCGt53EENDv8PrfZNmRnk"
    },
    <topic-name>: {
      "/": <topic-id>
    },
    ...
  },
  "metadata": {
    "created": <date-iso-8601>,
    "protocolVersion": <string>,  //Pulsarcast protocol version
    "allowedPublishers": {        //If enabled, whitelist of allowed publishers
      "enabled": <boolean>,
      "peers": [ <peer-id> ]
    },
    "requestToPublish": {         //Enable request to publish
      "enabled": <boolean>,
      "peers": [ <peer-id> ]      //Optional whitelist able to request
    },
    "eventLinking": <string>,     //One of: LAST_SEEN, CUSTOM
  }
}
```

Listing 3.1: Topic descriptor schema in a JSON based format

us to borrow inspiration from systems like IPFS and decided to use *CIDs* (Content Identifiers) [1]. A CID is a self-describing content-addressed identifier. It uses cryptographic hashes to achieve content addressing and is powered by *multihash* [2]. Multihash is a convention for representing the output of many different cryptographic hash functions in a compact, deterministic encoding that is accommodating of future change. This is because multihash encodes the type of hash function used to produce the output. All of the relevant identifiers in our system are CIDs. This includes node identifiers as well as the identifiers for both event descriptors and topic descriptors themselves (given they are the hash of its content). The descriptors contain a set of relevant metadata as well as the actual information that they refer to. The following JSON like representations 3.1 and 3.2 provide an accurate description of the schema and format of our data structures. We will cover some of the properties.

3.2.2 Data links (Merkle links)

Parent links in the event descriptor serve as a reference to previous events in the topic tree. A Pulsarcast node that has just received an event can, through its parent link, know a previous event of this same topic and act on it accordingly (fetch it or not). Depending on the type of topic we have at hand (something we will cover further in this document) this parent link can have different meanings and relevance.

The parent links in the topic descriptor act as a reference to a previous version of this same topic. Keep in mind that data in Pulsarcast is immutable. As such, one cannot update content that has already been published and disseminated. We can, however, create a new reference of it and link to what we

[1] https://github.com/multiformats/cid
[2] https://github.com/multiformats/multihash

```
{
  "name": <string>,
  "publisher": <peer-id>,          //Peer who published the event
  "author": <peer-id>,             //Author of the event
  "parent": {                      //The parent link for this event
    "/": <topic-id>
  },
  "topic": {
    "/": <topic-id>
  },
  "payload": <binary-data>
  "metadata": {
    "created": <date-iso-8601>,
    "protocolVersion": <string>,   //Pulsarcast protocol version
  }
}
```

Listing 3.2: Event descriptor schema in a JSON based format

consider to be a previous version. This is the exact use case for the parent links in the topic descriptor, to act as a link to previous versions of this same topic. Possible changes to the topic descriptor can encompass changes to the topic metadata for example or additions of new sub-topics.

In topic descriptors, sub-topic links are indexed under a # key. Commonly, these are indexed by name, but it is not mandatory, it is actually up to the topic and consequently its owner to choose accordingly. There is no limit to how many sub-topics a topic can have. One significant note though is that every topic comes with a default meta topic as a sub-topic. The idea is for this meta topic to be used to disseminate changes for the original topic descriptor, something we will cover in section 3.3.

Both descriptors have an author field that is self-descriptive, essentially meaning the peer responsible for creating and, in the case of the topic, maintaining this descriptor. The topic descriptor, however, has an extra field which is the publisher field. This is because the producer of the content (author) and the peer responsible for actually pushing this into the Pulsarcast dissemination trees (publisher) might not be the same peer.

3.2.3 Metadata

Metadata for the event descriptor is quite simple. It includes a creation timestamp in ISO8601 [3] format and the version of the protocol at the time the event was published. Remember, these are content-addressable immutable data structures, as such the info we provide in these metadata fields will be persistent and verifiable, something we take advantage. On the other hand, the topic descriptor is a bit more complicated. Besides the timestamp and version, it includes further configuration options that tell us how event dissemination and event linking will be handled for of the topic, something we will cover in section 3.3.

A key objective for these data structures and, specifically, to the metadata design has been for it to be easily extensible. The protocol version aids us with this, efficiently conveying any breaking changes. Examples of fields and properties that could be added in the future include things such as author and publisher signatures so that peers could verify the authenticity of all the content.

[3] https://www.iso.org/iso-8601-date-and-time-format.html

3.2.4 Distributed and local state

The data in Pulsarcast, however, is distributed, as illustrated by figure 3.5, which means we have to deal with an extra layer of complexity in our system. We again tackle this with the way we represent our data, the Merkle DAG. The fact that all data is addressed based on its content means that the representation and indexing done at each node is the same we do system-wide. Descriptors are persisted using the Kadmelia DHT and disseminated through the Pulsarcast overlay, with each node then keeping this state locally. This allows us to abstract state at each node in a manner where clients can refer to the same piece of data in the same way and expect the same representation and result, weather looking locally or system-wide.

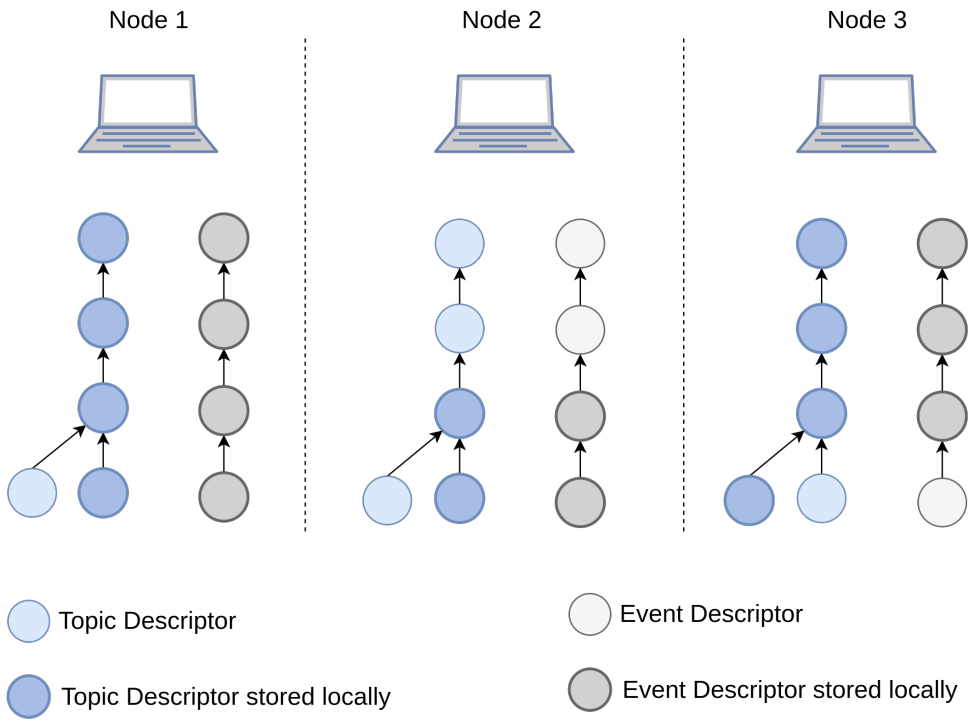

Figure 3.5: Overview of how state is kept across the network

3.3 Subscription Management and Event Dissemination

In Pulsarcast, both the subscription management and the event dissemination lie on top of the multiple overlays built on a per topic basis or, as we call it, the dissemination trees. These trees represent the path traversed by the events in order to reach the necessary subscribers, so, in the end, these end up being the actual representation of both subscriptions and dissemination paths. In order to better understand them we will start by understanding how the system handles the creation of topics, new subscriptions, followed by how events are propagated (with a detailed view of the algorithms used). We will see some of the configurations the topic descriptor allows for that will change the way events are propagated and published. Finally, we will run through the messages sent by each node in order to perform these operations, a crucial part in our distributed system.

3.3.1 Topic creation

Before we can speak about a new subscription, a topic must already exist. In order for this to happen a node starts by creating the meta topic descriptor. This meta topic descriptor is to be used to disseminate any changes relative to the topic descriptor at hand and is linked as a sub-topic of it. Only after it has been created and stored in the Kadmelia DHT does the node proceed to create the actual topic descriptor (with the meta topic linked as a sub-topic), which is then also persisted in the DHT. When any change to the original topic descriptor is in order, the node creates a new topic descriptor (remember the immutability of our data structures) but with the original topic descriptor linked as a parent and with the same meta topic linked as sub-topic. When these changes happen, the node publishes the new topic as an event in the meta topic. The algorithm 1 provides an overview of the procedure to create a new topic.

Algorithm 1: Create a new topic

```
1  Function CreateTopic(newTopic)
       Input: newTopic = data for new topic creation
2      begin
3          parent ← newTopic.parent;
4          if parent == null then ;                    // Check if the topic has a parent link
5
6              metaTopic ← CreateMetaTopicDescriptor(newTopic);
7          else
8              metaTopic ← parent.subTopics.meta;
9          end
10         topicData ← CreateTopicDescriptor(newTopic, metaTopic);
11         Subscribe(metaTopic);
12         Subscribe(topicData);
13         StoreInDHT(metaTopic);
14         StoreInDHT(topicData);
15         Publish(metaTopic, topicData);          // Publish the new topic in the meta topic
16
17     end
```

3.3.2 Subscribing

With the topic descriptor stored and available to the whole network, its creator will act as the root node in this newly created topic dissemination tree. When a node wants to subscribe to this topic, it starts by fetching its descriptor from the Kademlia DHT. After some sanity checks, such as checking if the node is already part of the dissemination tree, we use the Kadmelia DHT to find the closest known peer to the author of the topic. Keep in mind that we are not hitting the network and performing a Kadmelia lookup operation, we are resorting to information previously stored locally by the DHT in its K buckets. The node stores the closest known peer as its parent in this topic dissemination tree. The join request is then forwarded to it where the sender peer id is extracted and used as its children in this topic dissemination tree, followed by repeating the whole process. This recursive operation, across multiple nodes in the network, ends when the join request hits a node that is either already part of the dissemination tree for this topic or, the actual author of the topic. Algorithm 2 provides a more detailed generic procedure to be used at every node when receiving or sending a subscription request (or a join

request as we call it) and figure 3.6 tries to provide a visual representation of the whole subscription flow. In order to maintain the dissemination trees, every node must keep some state of its neighbours for every topic. If by some chance a node is unable to connect to a neighbour, a retry mechanism is in place for a limited amount of retries (a configurable parameter). If the node is still unable to connect, then it goes through the subscription procedure again.

Algorithm 2: Join request handler for each node

```
1  Function ReceivedJoin(fromNodeId, topicId)
       Data: nodeId = node id of this node
       Input: topicId = topic id
       Input: fromNodeId = node who we got the join request from
2      begin
3          topicData ← GetTopicData(topicId);
4          if fromNodeId ≠ nodeId then
5              AddToChildren(t, fromNodeId);        // Add as children in dissemination tree
6              if topicData.author == nodeId then ;        // This node is author of the topic
7
8                  return
9              end
10             if GetParents(topicId) ≠ null then ;        // Already part of dissemination tree
11
12                 return
13             end
14         else
15             if topicData.author == nodeId then ;        // This node is author of the topic
16
17                 return
18             end
19         end
20         peer ← GetClosestKnownPeer(topicData.author);
21         AddToParents(topicData.id, peer);            // Add as parent in dissemination tree
22         SendRPC(topicData.id, peer);
23     end
```

3.3.3 Publishing and event dissemination

Considering the topic creation and subscription management previously discussed we can see that event dissemination becomes easier to handle, almost as a consequence of the way the subscription management is built, and dissemination trees again play their key part here. Pulsarcast, however, allows for some additional customisation and configuration at the topic level focused on providing a lot more flexibility to our system. When a node is creating a topic, it can configure:

- Which nodes are allowed to publish
- If and which nodes can request to publish
- How events are linked together (through the parent link)

These options are *requestToPublish*, *allowedPublishers* and *eventLinking*, all mentioned in the previous section TODO, we will see how each one of these fits together with the way Pulsarcast nodes dissminate events. Figures 3.7 and 3.8 provide visual aids to how these options come together for event dissemination.

When a node wants to publish an event in a topic, it starts by fetching the topic descriptor, first locally and then, if it is not present, from the Kadmelia DHT. This way, we have access to the topic data as well as its configurations. The node then checks if it is allowed to publish through the topic configuration whitelist mechanism. This option, *allowedPublishers*, can either be enabled and, if so, a list of nodes is provided that is checked before publishing, or it can be disabled, and in that scenario, every node can

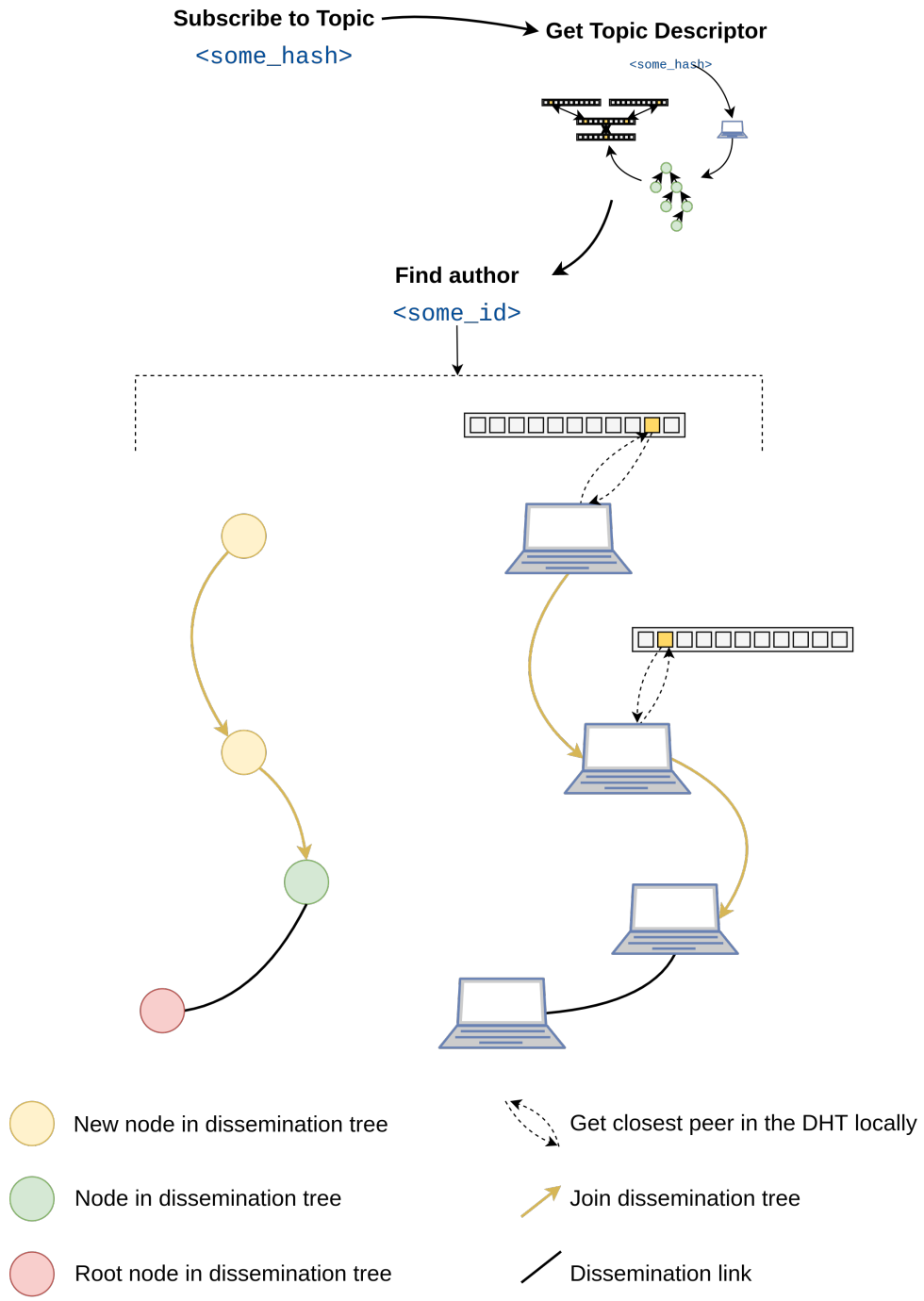

Figure 3.6: Overview of the flow for creating a new subscription

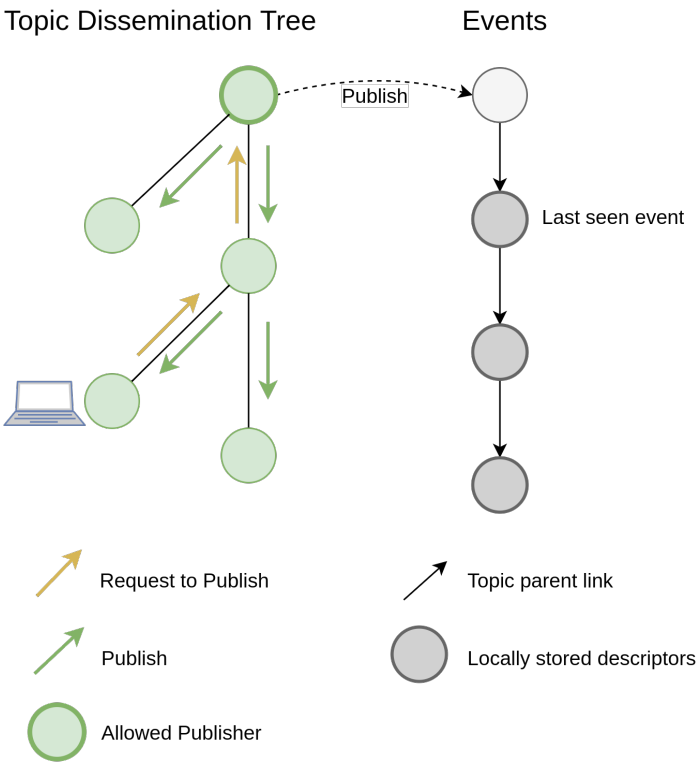

Figure 3.7: Event dissemination mechanism for a topic with only the author allowed to publish, last seen event linking and request to publish allowed. This scenario provides order guarantee.

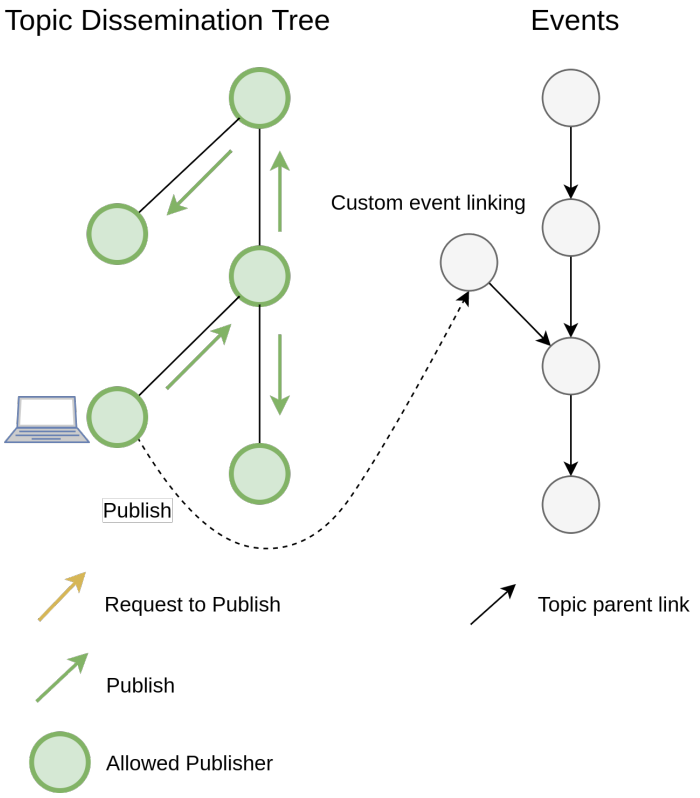

Figure 3.8: Event dissemination mechanism for a topic with custom event linking and global publishers allowed

publish a message. If the node cannot publish the message, it will check if it can submit a request to publish. This request to publish is another option set in the topic descriptor, through the *requestToPublish* field, that, if enabled, allows every node in the network to submit these special requests. Optionally, it can also be a whitelist of nodes allowed to submit these. When a node forwards a request to publish across the network, it propagates across the dissemination tree (from children nodes to parents) until it eventually finds a node which is allowed to publish this event. This will dictate the difference in the publisher (node who actually publishes the content) and the author (node responsible for creating the content in the first place).

Upon receiving a publish event request, weather if it was initiated at this node or through a remote request to publish, the node starts by appropriately linking the new event to a parent event. This is where the *eventLinking* option in our topic descriptor comes into play. Right now this option can either be *CUSTOM* or *LAST_SEEN*. When the topic allows for custom linking, the client application can set a custom parent event, as long as it exists. With the last seen option, however, the Pulsarcast node takes care of linking the given event to the event last seen by it. After the linking is done, the node can safely store the event descriptor in the Kademlia DHT, followed by disseminating it through its children and parent nodes in this topic dissemination tree. From this point forward, nodes along the dissemination tree will forward the event across branches of the tree where this has not gone through. All of the logic we have covered around event dissemination is better detailed in the algorithms 3 and 4.

Algorithm 3: Event handler for each node

```
1  Function ReceivedEvent(fromNodeId, eventData)
       Data: nodeId = node id of this node
       Input: fromNodeId = node who we got the event from
       Input: eventData = event descriptor
2      begin
3          topicData ← GetTopicData(eventData.topicId);
4          if AllowedToPublish(nodeId, topicData) then
5              SendEvent(fromNodeId, eventData);
6          else
7              if AllowedToRequestToPublish(nodeId, topicData then
8                  SendRequestToPublish(eventData);    // Send request to publish to parent
                    node
9
10             end
11         end
12     end
```

Algorithm 4: Event forwarding function

```
1  Function SendEvent(eventData)
       Data: nodeId = node id of this node
       Input: fromNodeId = node who we got the event from
       Input: eventData = event descriptor
2      begin
3          topicData ← GetTopicData(eventData.topicId);
4          if IsNewEvent(eventData) then
5              linkedEvent ← LinkEvent(eventData);                    // Add parent link
6              StoreInDHT(linkedEvent);
7          end
8          if (IsSubscribed(eventData.topicId) == true) then
9              EmitEvent(eventData.topicId, eventData);
10         end
11         for peer ← GetChildren(eventData.topicId), GetParents(eventData.topicId) do
12             if fromNodeId ≠ peer then ;                    // Do not send the event back
13
14                 SendRPC(eventData, peer);
15             end
16         end
17     end
```

It is essential to understand some of the properties that these multiple configuration options allow. The simplest example would be a scenario where only the author of a topic is allowed to publish, event linking is based on the last seen event and request to publish is allowed. In this example, despite every node being allowed to create content, we can achieve order guarantee, with a single stream of events all linked together. Another example would be a scenario where we have a whitelist of allowed publishers, no request to publish allowed and last seen event linking taking place. With this, we get a simple producer/consumer scenario, with a list of a few selected and vouched for producers that every node is aware of (that could even be expanded later on by the topic author). Finally, on the other end of the spectrum, we have a scenario where everyone is allowed to publish, and custom event linking is allowed. Here, we are essentially giving the ability for clients and applications to use event trees to represent data in however they see fit given that, with custom event linking, applications can shape the event trees however they like. Links can go as far as to imply event causality if applications are programmed and configured as such. All of these scenarios came from a realisation that it did not make sense to limit Pulsarcast's uses out of the box, especially taking account that through simple configuration we could cater for a broader set of use cases and applications.

3.4 RPC message protocol

We have run through the overall architecture of Pulsarcast. However, we still have not detailed how the nodes communicate with each other. The fact is, Pulsarcast does not favour or define the usage of a specific wire protocol. In our view Pulsarcast can and should work independently of what we use in the transport layer. It does, however, specify a well-defined communication protocol built using Protocol Buffers [4]. Protocol Buffers (or protobuf for short) are Google's open-source, language-neutral, platform-

[4] https://developers.google.com/protocol-buffers/

```
message RPC {
  enum Operation {
    PUBLISH_EVENT = 2;
    JOIN_TOPIC = 3;
    LEAVE_TOPIC = 4;
    NEW_TOPIC = 5;
    REQUEST_TO_PUBLISH = 6;
  }

  message Message {
    optional Operation op = 1;
    oneof payload {
      TopicDescriptor topic = 2;
      EventDescriptor event = 3;
      bytes topicId = 5;
    }
    optional MetaData metadata = 6;
  }

  message MetaData {
    optional string created = 1;
    optional string protocolVersion = 2;
  }

  repeated Message msgs = 1;
}
```

Listing 3.3: Protobuf schema for our RPC messages

neutral, extensible mechanism for serializing structured data. It allows us to define common lightweight interfaces that, through source code generation, work across platforms. Plus, it is fast and quite small once encoded to the protocol buffer binary format (what actually runs in the wire), while also supporting a text-based representation, similar to JSON, for readability purposes. With protobuf's, we have defined a set of Remote Procedure Call (RPC) schemas used by our system to communicate between nodes.

We started by defining a common interface for all of our RPC messages as showed in the listing 3.3. In protobufs, every field is associated with a number. These are used to identify a field in the protobuf binary format [5] uniquely. Our message format is quite straight forward, with three fields, *op* (for operation), *payload* representing a generic payload and finally *metadata* containing relevant metadata such as the creation date of this message and the protocol version used. The supported operations for now essentially translate all the actions a node might undertake, which are publishing an event, joining a topic, leaving a topic, creating a topic and requesting to publish. The payload is an abstraction, and it can hold a topic descriptor, an event descriptor or a topic id depending on the operation field of this message. Finally, the RPC schema itself is an array of the messages, so that if needed a node can forward multiple operations in one go.

The topic and event descriptors also have a specific protobuf schema, essentially a representation of what we have already covered in section 3.2. Listings 3.4 and 3.5 show us the specification of these. In order to keep our messages as lightweight as possible and avoid unnecessarily burdening the network, we opted to have all of our content identifiers (such as the author and links) sent in binary format. Otherwise, the remaining fields end up being a one to one mapping from the data structure schemas of the event descriptor and topic descriptor.

[5] https://developers.google.com/protocol-buffers/docs/encoding

```
message Link {
  optional bytes / = 1;
}

message TopicDescriptor {

  optional string name = 1;
  optional bytes author = 2;
  optional Link parent = 2;
  map<string, Link> # = 3;
  optional MetaData metadata = 4;

  message MetaData {

    enum EventLinking {
      LAST_SEEN = 0;
      CUSTOM = 1;
    }

    message PublishersList {
      optional bool enabled = 1;
      repeated bytes publishers = 2;
    }

    optional string created = 1;
    optional string protocolVersion = 2;
    optional PublishersList allowedPublishers = 3;
    optional PublishersList requestToPublish = 4;
    optional EventLinking eventLinking = 5;
  }
}
```

Listing 3.4: Protobuf schema of the topic descriptor

```
message Link {
  optional bytes / = 1;
}

message EventDescriptor {

  message MetaData {
    optional string created = 1;
    optional string protocolVersion = 2;
  }

  optional bytes author = 1;
  optional Link topic = 2;
  optional bytes payload = 3;
  optional bytes publisher = 4;
  optional Link parent = 5;
  optional MetaData metadata = 6;
}
```

Listing 3.5: Protobuf schema of the event descriptor

3.5 Summary

TODO

Chapter 4

Implementation

For our Pulsarcast implementation, we decided to take advantage of the *libp2p* ecosystem as it solves a lot of the underlying issues of building a peer to peer system, not specific to our pub-sub scenario. This includes dealing with connection multiplexing, NAT traversal, discovery mechanisms and others. All of which libp2p, a community-focused project with implementations in multiple languages already solves. We can also take advantage of the utility modules it has and the advantage of having an already working implementation of the Kadmelia DHT. Our focus is then to build a module, implementing the Pulsarcast specification that clients and apps can take advantage of. Section 4.1 provides a detailed description of it.

Besides our module, we also needed to find a suitable way to test our system as a whole. Given our specific needs, we opted to build a custom testbed, detailed in section 4.2 and the basis of the evaluation detailed in chapter 5.

4.1 Pulscarcast Javascript module

We chose to implement our Pulsarcast module in Javascript. As we covered in our related work, Javascript is ubiquitous, running in browsers, servers and many different kinds of devices and architectures. Through it, we can run our Pulscarcast nodes in a multitude of systems and most importantly, direct its usage for the World Wide Web. Plus, libp2p has a Javascript implementation focused on cross-compatibility between server and browser. This is not to say that in the future we will not have other implementations in different languages, that is, in fact, one of the reasons for the clear separation between the Pulcarcast specification and its actual implementation. However, we had to choose, and in our view, Javascript is the clear winner. It is worth noting that, much like the work we built on top of, this module is open source [1].

4.1.1 Dependencies

Figure 4.1 gives us an overview of how our module fits in the libp2p ecosystem. libp2p defines interfaces responsible for routing content (peer routing), discovering other peers in the network (peer discovery), network transports and leveraging multiple network connections (switch). These all come bundled in the libp2p javascript module [2] which we use in Pulsarcast. Besides the main libp2p module we also use

[1] https://github.com/JGAntunes/js-pulsarcast
[2] https://github.com/libp2p/js-libp2

```
{
  ...
  "dependencies": {
    "async": "^2.6.1",
    "bs58": "^4.0.1",
    "cids": "^0.5.5",
    "debug": "^3.1.0",
    "ipld-dag-cbor": "^0.13.0",
    "joi": "^13.4.0",
    "joi-browser": "^13.4.0",
    "peer-id": "^0.12.2",
    "peer-info": "^0.14.1",
    "protons": "^1.0.1",
    "pull-length-prefixed": "^1.3.1",
    "pull-stream": "^3.6.9"
  },
  ...
}
```

Listing 4.1: Pulsarcast module dependency list

some other utility modules such as the CID module [3] and the Peer-id module [4], both designed to reason with content identifiers (peer identifiers are also content identifiers).

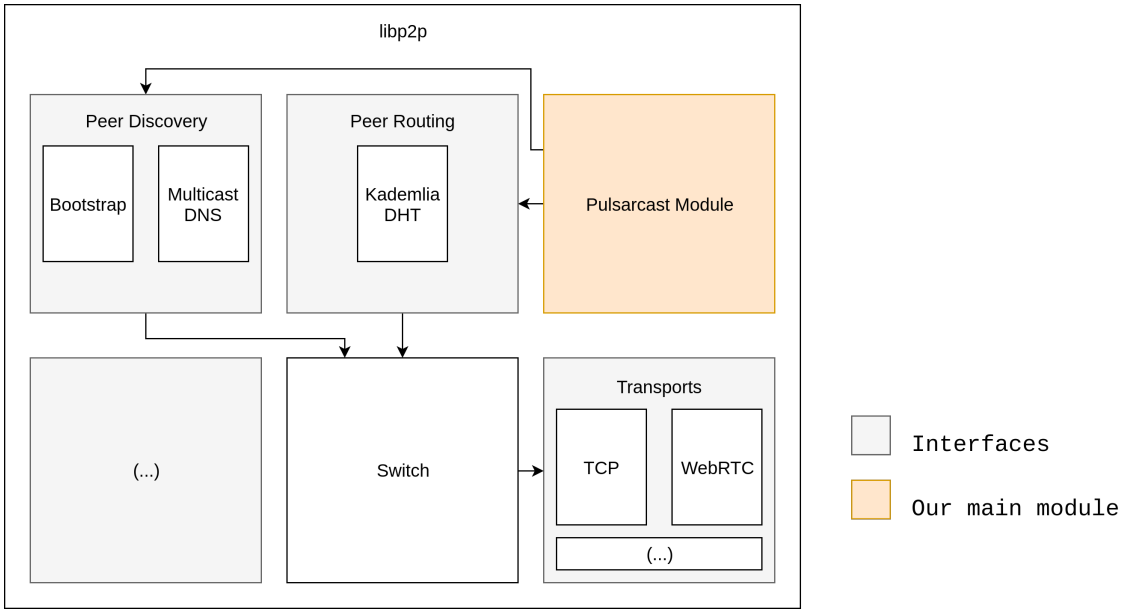

Figure 4.1: Our Pulsarcast module in the libp2p ecosystem

Other than the libp2p dependencies, we have used other open source community modules. The listing 4.1 shows our dependency list, pulled from our *package.json* (our module manifest). We have used a couple of helpers for handling data streams [5] as well as a library to help us with the asynchronous control flow [6]. To help us with protocol buffers, we use protons [7] and finally, joi [8] is our validation library of choice.

[3] https://github.com/multiformats/js-cid
[4] https://github.com/libp2p/js-peer-id
[5] https://github.com/pull-stream/pull-stream
[6] https://github.com/caolan/async
[7] https://github.com/ipfs/protons
[8] https://github.com/hapijs/joi

4.1.2 Code Organisation

When it comes to code organisation, and despite not existing a clear standard for modules in Javascript, we follow what is most generally conceived as best practices. Listing 4.2 gives us an overview of the file structure of our project.

Our *src* (source) directory has all the components that constitute our Pulsarcast node. *src/index.js* holds our top-level class, Pulsarcast, that is exported and listed as the main component to import when applications *require* our module. We have split internal components into folders according to its functionality. The classes for our event and topic descriptors, as well as its trees, are under the *dag* directory. The *messages* directory keeps all the important pieces of functionality relative to our RPC messages, including its Protocol buffers, validators and marshalling logic. Our RPC handlers, containing Pulsarcast's core algorithms and logic, are kept under *rpc*. Finally, on the root of our source directory, we keep our default node configuration, our peer class and an *utils* directory, where we store a series of utility functions used across our codebase.

Apart from our source directory, we have a series of unit and integration tests, currently kept under *test*. These tests exercise the critical components of our system, running a series of nodes locally, making sure it is functionally sound. Besides being able to run locally, these tests are what we currently run on our *CI* [9] [10] environment on every code submission we perform.

4.1.3 Classes

Pulsarcast has five classes. These are:

- Pulsarcast
- Peer
- TopicNode
- EventNode
- EventTree

Figure 4.2 shows the Unified Modelling Language (UML) class diagram of our system, but we will run through those classes which we consider more relevant for a thorough look into its functioning.

4.1.3.a Pulsarcast

Pulsarcast is our main class and essentially what makes a Pulsarcast node. It contains the state of subscriptions, topics, connections and event dissemination trees relevant to this node as well as the needed methods to interact with the Pulsarcast underlying system. It extends the built-in `EventEmitter` class [11], making our class capable of reproducing the event emitter pattern [12], the mechanism we use to bubble up new Pulsarcast events to the application level. It has a fairly simple constructor as one can see in the listing 4.3. The class has seven key methods exposed:

- `start(callback)` - Start the Pulsarcast node by registering our protocol with the *libp2p* protocol connection handler.

[9] https://www.thoughtworks.com/continuous-integration
[10] https://gitlab.com/jgantunes/js-pulsarcast/pipelines
[11] https://nodejs.org/api/events.html
[12] https://nodejs.dev/the-nodejs-event-emitter

```
.
|-- docs
|   `-- api.md
|-- LICENSE
|-- package.json
|-- package-lock.json
|-- README.md
|-- src
|   |-- config.js
|   |-- dag
|   |   |-- event-node.js
|   |   |-- event-tree.js
|   |   |-- topic-node.js
|   |   |-- topic-tree.js
|   |   `-- utils.js
|   |-- index.js
|   |-- messages
|   |   |-- create-rpc.js
|   |   |-- index.js
|   |   |-- marshalling.js
|   |   |-- protobuffers
|   |   |   |-- index.js
|   |   |   `-- messages.proto.js
|   |   `-- schemas
|   |       |-- event-descriptor.js
|   |       |-- index.js
|   |       |-- peer-tree.js
|   |       |-- rpc.js
|   |       `-- topic-descriptor.js
|   |-- peer.js
|   |-- rpc
|   |   |-- index.js
|   |   |-- receive.js
|   |   `-- send.js
|   `-- utils
|       |-- dht-helpers.js
|       `-- logger.js
`-- test
    |-- integration
    |   |-- 2-nodes.js
    |   `-- multiple-nodes.js
    |-- test-node.js
    `-- utils.js
```

Listing 4.2: File tree for our Pulsarcast implementation

- `stop(callback)` - Stop the Pulsarcast node, gracefully close connections we might have to any peer and de-register our protocol from the *libp2p* protocol connection handler.
- `createTopic(topicName, [options], callback)` - Create a Pulsarcast topic with the specified name. An optional object can be provided through which one can add `subTopics}, \verb`parent—link and topic metadata configuration with `allowedPublishers`, `requestToPublish` and `eventLinking`. By default, no `parent` link is added and only the topic author is allowed to publish (essentially creating an event dissemination tree with order guarantee).
- `subscribe(topicB58Str, [options], callback)` - Subscribe to a topic with the given base58 CID. An option `subscribeToMeta` can be provided which will determine if this node will subscribe to this topic's meta-topic also (defaults to true).
- `unsubscribe(topicB58Str, callback)` - Unsubscribe from the topic with the given base58 CID.

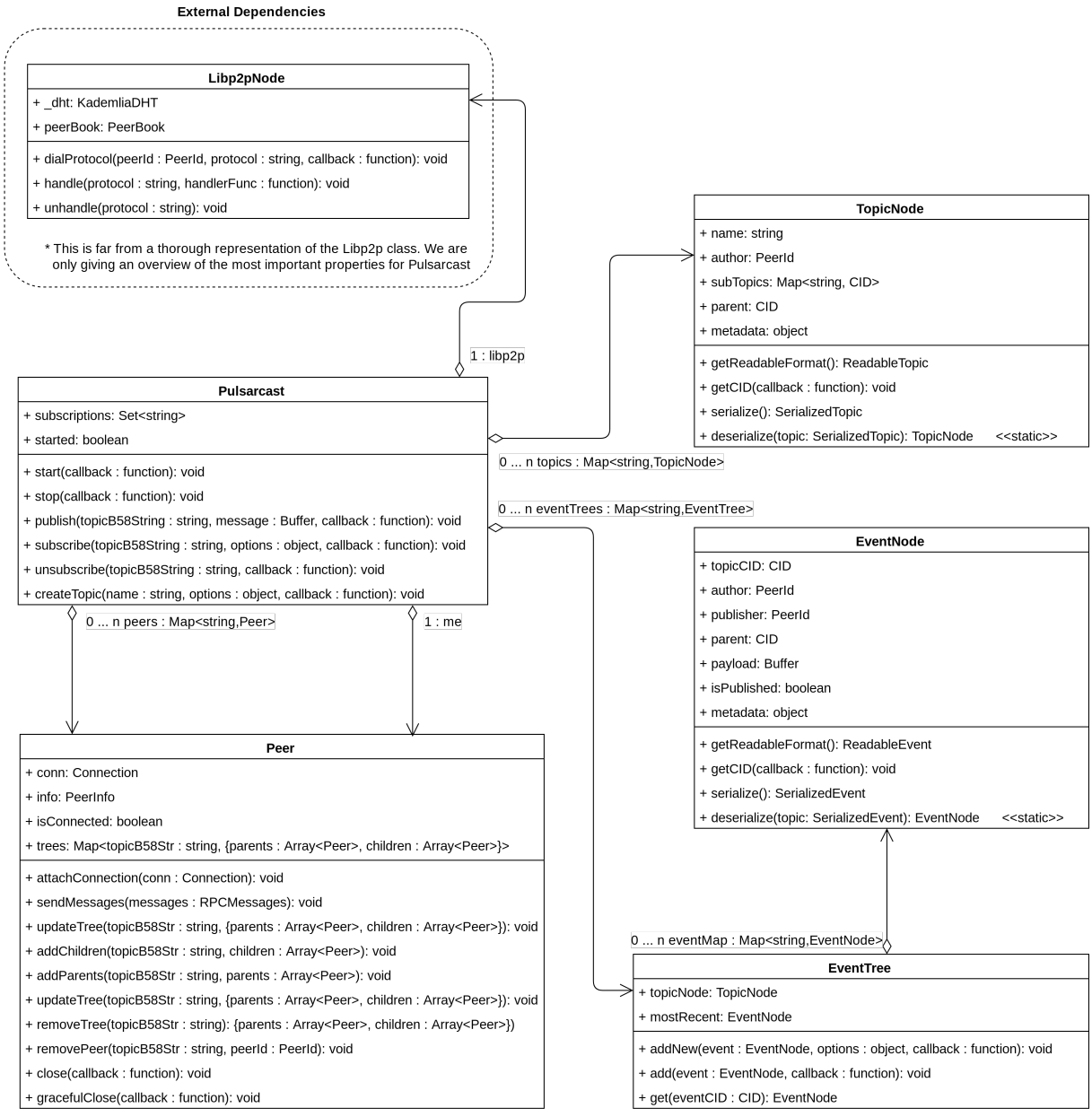

Figure 4.2: UML representation of the classes in our Pulsarcast system

```
class Pulsarcast extends EventEmitter {

  /**
   * Create a new PulsarCast node.
   *
   * @param {external:Libp2pNode} libp2p
   * @param {object} [options={}] - PulsarCast options
   */
  constructor(libp2p, options = {}) {
    super()

    this.libp2p = libp2p
    this.started = false

    this.peers = new Map()
    this.topics = new Map()
    this.subscriptions = new Set()
    this.eventTrees = new Map()

    this.me = new Peer(libp2p.peerInfo)
    this._onConnection = this._onConnection.bind(this)

    // Create our handlers to receive and send RPC messages
    this.rpc = createRpcHandlers(this)
  }

  (...)

}
```

Listing 4.3: Pulsarcast constructor

An option `unsubscribeFromMeta` can be provided which will determine if this node will unsubscribe to this topic's meta-topic also (defaults to true).

- `publish(topicB58Str, message, callback)` - Publish the provided message in the topic with the given base58 string as a CID.

Pulscarcast's topic creation, subscribe/unsubscribe and publish methods are relatively simple, given our separation of concerns. All of them consist of a set of input sanity checks with control, in the end, being handover to our RPC handlers, responsible for storing any relevant information in the DHT and forwarding/creating any RPC request that we might need. Listing 4.4 shows us an example of the logic above, the subscribe method. New subscription calls in Pulsarcast are done using the topic's base58 CID. The Pulscarcast node keeps all the state relative to the topics we are subscribed. As such, before performing any action, we always check to see if we are subscribed to the topic given. If all the input checks are right, we hand over to our RPC handlers to create the appropriate RPC messages and forward our request.

Although not part of the public methods we have mentioned, one key part of the Pulsarcast class is the actual management of peers and connections (outgoing and incoming). When the Pulsarcast node is started via the `start` method, we register our protocol `/pulsarcast/1.0.0` with the *libp2p* protocol handlers. When an incoming connection that matches our protocol reaches our node, the Pulsarcast class is responsible for internally handling it, creating the necessary Peer objects and registering the appropriate connection handlers so that when/if the connection drops we clear all the relevant state. The same goes for when the node needs to contact another peer. If a connection needs to be established, it

```
subscribe(topicB58Str, options, callback) {
  assert(this.started, 'Pulsarcast is not started')
  if (!callback) {
    callback = options
    options = {}
  }
  const topicCID = new CID(topicB58Str)
  // By default subscribe to meta topic
  const defaultSubscribeOptions = {
    subscribeToMeta: true
  }
  const subscribeOptions = { ...defaultSubscribeOptions, ...options }
  if (this.subscriptions.has(topicB58Str)) {
    return this._getTopic(topicCID, (err, topicNode) => {
      callback(err, topicNode, topicCID)
    })
  }
  this.subscriptions.add(topicB58Str)
  this.rpc.receive.topic.join(
    this.me.info.id.toB58String(),
    topicCID,
    subscribeOptions,
    callback
  )
}
```

Listing 4.4: Pulsarcast class subscribe method

is up to the Pulsarcast class to do it properly. Listing 4.5 gives us a thorough look of all of the methods used for this purpose.

4.1.3.b Peer

The Peer class provides an abstraction for the state of each peer, including connections, information such its id and address and the dissemination trees it is part of. Listing 4.6 shows us the constructor of our class. Its `trees` property is a `Map` indexed by topic identifier, that keeps the peer's current dissemination trees neighbours. Listing 4.7 shows the methods that interact with this property as well as the connection and message forwarding handlers that help our node forward messages to the necessary peers.

4.1.3.c TopicNode and EventNode

The `TopicNode` and `EventNode` classes refer to our topic and event descriptor dag nodes respectively. They are quite similar given that, in the end, they both abstract Merkle DAG nodes. Listings 4.8 and 4.9 show us its constructors. Despite the similarities, though, some aspects are worth noting. Such example is the `isPublished` field on the event descriptor. Which is crucial for our system to understand if this event has or has not been published yet.

Both of the classes have a series of methods which we feel like are worth describing:

- `getCID(callback)` - Given our descriptors are content-addressable, their identifiers are ever-changing depending on the values of their properties. This method returns, through the callback function, the CID for this descriptor.
- `getReadableFormat()` - Return an object whose properties are human-readable representations of the values of this object in the canonical format described in chapter 3. Consists mostly of

```
_addPeer(peerInfo, conn) {
  const idB58Str = peerInfo.id.toB58String()
  // Check to see if we already have the peer registered
  let peer = this.peers.get(idB58Str)
  if (peer) {
    peer.attachConnection(conn)
    // Attach peer conn to our rpc handlers
    this._listenToPeerConn(idB58Str, conn, peer)
    return peer
  }
  peer = new Peer(peerInfo, conn)
  // If connection closes, remove peer from list
  peer.once('close', () => { this.peers.delete(idB58Str) })
  // Insert peer in list
  this.peers.set(idB58Str, peer)
  this._listenToPeerConn(idB58Str, conn, peer)
  return peer
}

_getPeer(peerId, callback) {
  const idB58Str = peerId.toB58String()
  const peer = this.peers.get(idB58Str)
  // We already have the peer in our list
  if (peer) return setImmediate(callback, null, peer)
  // Let's dial to it
  this.libp2p.dialProtocol(peerId, protocol, (err, conn) => {
    if (err) return callback(err)
    const peerInfo = this.libp2p.peerBook.get(peerId)
    const peer = this._addPeer(peerInfo, conn)
    callback(null, peer)
  })
}

_onConnection(protocol, conn) {
  conn.getPeerInfo((err, peerInfo) => {
    if (err) {
      // Terminate the pull stream
      return pull(pull.empty(), conn)
    }
    this._addPeer(peerInfo, conn)
  })
}

_listenToPeerConn(idB58Str, conn, peer) {
  pull(
    conn,
    lp.decode(),
    pull.map(data => protobuffers.RPC.decode(data)),
    pull.drain(
      rpc => this._onRPC(idB58Str, rpc),
      err => this._onConnectionEnd(err, idB58Str, peer)
    )
  )
}
```

Listing 4.5: Pulsarcast internal peer and connection handlers. These methods abstract the logic of connection handling to all the other parts of the Pulsarcast system.

```
class Peer extends EventEmitter {
  constructor(peerInfo, conn = null) {
    log.trace('New peer registered %j', { peer: peerInfo.id.toB58String() })
    assert(peerInfo, 'Need a peerInfo object to initiate the peer')
    super()

    this.stream = null
    this.conn = conn
    this.info = peerInfo
    this.trees = new Map()

    if (conn) {
      this.attachConnection(conn)
    }
  }
  (...)
}
```

Listing 4.6: Peer class constructor

```
  attachConnection(conn) {
    this.stream = new Pushable()
    this.conn = conn
    pull(this.stream, lp.encode(), conn)
    this.emit('connection')
  }

  sendMessages(messages) {
    this.stream.push(messages)
  }

  addChildren(topic, children) {
    const tree = this.trees.get(topic)
    if (!tree) {
      this.trees.set(topic, { children, parents: [] })
      return
    }
    children.forEach(child => {
      const exists = tree.children.find(peer => {
        return peer.info.id.isEqual(child.info.id)
      })
      if (!exists)tree.children.push(child)
    })
  }

  addParents(topic, parents) {
    const tree = this.trees.get(topic)
    if (!tree) {
      this.trees.set(topic, { parents, children: [] })
      return
    }
    parents.forEach(parent => {
      const exists = tree.parents.find(peer => {
        return peer.info.id.isEqual(parent.info.id)
      })
      if (!exists) tree.parents.push(parent)
    })
  }
```

Listing 4.7: Peer class relevant methods

```
class TopicNode {
  constructor(name, author, options = {}) {
    assert(author, 'Need an author to create a topic node')
    this.name = name
    this.author = author
    this.subTopics = {}
    this.parent = options.parent ? new CID(options.parent) : null
    if (options.subTopics) {
      this.subTopics = Object.entries(options.subTopics)
        .map(([name, topicB58Str]) => ({ [name]: new CID(topicB58Str) }))
        .reduce((topics, topic) => ({ ...topic, ...topics }), {})
    }
    this.metadata = createMetadata(options.metadata)
  }
  (...)
}

function createMetadata({
  allowedPublishers = false,
  requestToPublish = true,
  eventLinking = 'LAST_SEEN',
  created = new Date(),
  protocolVersion = config.protocol
} = {}) {
  return {
    protocolVersion,
    created: new Date(created),
    allowedPublishers,
    requestToPublish,
    eventLinking
  }
}
```

Listing 4.8: TopicNode class constructor

```
class EventNode {
  constructor(topicCID, author, payload, options = {}) {
    assert(topicCID, 'Need a topicCID object to create an event node')
    assert(author, 'Need an author to create an event node')
    assert(payload, 'Need a payload to create an event node')
    this.topicCID = topicCID ? new CID(topicCID) : null
    this.author = author
    this.payload = payload
    this.publisher = options.publisher
    this.parent = options.parent ? new CID(options.parent) : null
    this.metadata = createMetadata(options.metadata)
  }

  get isPublished() {
    return Boolean(this.publisher)
  }
  (...)
}

function createMetadata({
  created = new Date(),
  protocolVersion = config.protocol
} = {}) {
  return { protocolVersion, created: new Date(created) }
}
```

Listing 4.9: EventNode class constructor

returning all the CIDs as base 58 strings.

- `serialize()` - Return an object in the canonical format described in chapter 3 of the descriptor in question, with all the CIDs in binary format (buffers). Mostly useful for when we are about to convert this descriptor to a protocol buffer and forward it.

- `deserialize(node)` - Convert a descriptor in a serialized format to an instance of its class, returning that same instance. A static method for each class.

4.1.4 RPC Handlers

Pulsarcast's RPC handlers are responsible for containing the implementation of the critical algorithms mentioned in chapter 3. These functions are not only used by the Pulsarcast class when a new command is invoked but are also registered to handle incoming messages from other nodes. A generic handler is responsible for receiving every incoming message and, depending on the type of operation (see 3.3), forward it to the correct handler.

It is also the generic handler responsibility to validate incoming and outgoing messages, making sure they obey our pre-defined schemas. This is important because specific requirements and rules cannot be enforced at the protocol buffer level. For this purpose, we use the already mentioned validation library, Joi. Joi allows us to write up validation schemas that are easy to cope with, in Javascript. Listing 4.10 provides an example of such schema in practice.

```
const metadata = Joi.object()
  .keys({
    created: Joi.date().iso().required(),
    protocolVersion: Joi.string().required()
  }).required()

const eventDescriptor = Joi.object().keys({
  publisher: Joi.binary().required().allow(null),
  author: Joi.binary().required(),
  parent: Joi.object()
    .keys({
      '/': Joi.binary().allow(null)
    }).required(),
  payload: Joi.binary().required(),
  topic: Joi.object()
    .keys({
      '/': Joi.binary().required()
    }).required(),
  metadata
})
```

Listing 4.10: Joi schema for and event descriptor

We have split our RPC handlers into outgoing and incoming and tried to reuse as much logic as possible. Hence why when the client invokes a new command through the Pulsarcat API, control is handed over to the respective RPC handler as if we had just received an RPC message from another node. Consequently, some of the logic described in the algorithms in chapter 3 is split between incoming/outgoing RPC handlers. The following list provides an overview of the functions we have created:

- `rpc.receive.genericHandler()` - The generic handle responsible for receiving all the incoming RPC messages, validating these and forwarding it accordingly.

- `rpc.receive.publish()` - The receiving RPC handler for publish events, responsible for partially implementing algorithm 3. Upon receiving the `eventNode` and the id of the peer sending it (`idB58Str`), it starts by checking if the publish originated in that same node. Checks for the existence of the topic descriptor the event refers to and checks this node's permission to publish. Depending on the result, the function hands control over to the `rpc.send.publish` function or `rpc.receive.requestToPublish`.
- `rpc.receive.requestToPublish()` - The receiving RPC handler for request to publish events, responsible for partially implementing algorithm 3. Upon receiving the `eventNode` and the id of the peer sending it (`idB58Str`) checks for the existence of the topic descriptor the event refers to and checks this node's permission to request to publish. It then checks this node's permission to publish the event. Depending on the result, the function hands control over to the `rpc.send.publish` function or the `rpc.send.requestToPublish`.
- `rpc.receive.join()` - The receiving RPC handler for a new subscription, responsible for partially implementing algorithm 2. Upon receiving the `topicCID` and the id of the peer sending, it (`idB58Str`) checks for the existence of the topic descriptor we want to join to and if the join command originated at this node. If not, it means we have received this join request from another node, as such, we need to add it to our event dissemination tree for this topic. With the new state appropriately set, we check if we are already part of this dissemination tree, if not we need to generate a join request ourselves and forward it, handing over control to `rpc.send.publish`.
- `rpc.receive.leave()` - The receiving RPC handler for unscribe requests. This function is similar to its counterpart responsible for subscribing to a topic.
- `rpc.send.publish()` - The outgoing RPC handler for event publishing, responsible for partially implementing algorithm 4. It starts by checking if the event is being created at this node if it is, it links it to its parent event accordingly and stores it in the DHT. If the node is subscribed to this topic, the event is emitted to the application level. Finally, the event is forwarded to the branches of the dissemination tree who have not received it yet.
- `rpc.send.requestToPublish()` - The outgoing RPC handler for request to publish events, responsible for partially implementing algorithm 4, mainly consisting of forwarding the requests to the appropriate dissemination tree branches.
- `rpc.send.join()` - The outgoing RPC handler for subscription requests, responsible for partially implementing algorithm 2. Upon being called, it looks up for the closest known peer to the topic author, locally stored in DHT. With the closest peer id, we update our node's dissemination tree parent for this topic and send our join RPC request.
- `rpc.send.leave()` - The outgoing RPC handler for unsubscribe requests. Mostly the same logic as its counterpart `join` but for leaving a topic.
- `rpc.send.newTopic()` - This function handles the topic creation logic, responsible for implementing the algorithm 1. When called, it starts by validating the new topic, checking if its sub-topics and parent are valid and exist, creating a meta topic if one has not been provided already, storing these in the DHT and finally propagating the new topic through the meta-topic.

4.1.5 Usage

Listing 4.11 provides a usage example of our Javascript module. Keep in mind that this is a oversimplified example of course, with a single node, its purpose is to understand how the API comes together and

allows applications to integrate with it.

```
const Pulsarcast = require('pulsarcast')

// node is a libp2p Node
const pulsarcastNode = new Pulsarcast(node)

const pulsarcastNode.start((err) => {
  if (err) console.log('No!!!', err)

  pulsarcastNode.createTopic('fuuuuun', (err, cid, topicNode) => {
    if (err) console.log('No!!!', err)
    console.log('Our new topic \o/', topicNode)

    pulsarcastNode.on(cid.toBaseEncodedString(), (eventNode) => {
      console.log('event', eventNode)
    })

    pulsarcastNode.publish(cid.toBaseEncodedString(), new Buffer('super fun!'),
    (err, eventCID) => {
      if (err) console.log('No!!!', err)
      console.log('published', eventCID.toBaseEncodedString())
    })
  })
})
```

Listing 4.11: Usage example of our Pulsarcast module

4.2 Testbed

As part of our implementation, we needed a way to test our Pulsarcast system. However we had a set of specific requirements that made our choice of tools harder. We needed something that fulfilled the following:

- Easily deploy and test different versions of our module.
- Run tests not only on Pulsarcast but also on IPFS' own pub-sub implementation.
- Able to extract relevant usage metrics.
- Simulate network constraints such as latency.
- Able to run locally but easily scalable to a large network.
- Can be controlled from a central point, while being able to interact with specific nodes in the system.
- Easy to create scripts for, so that we could automate as much of our test suite as possible.

Unfortunately for us, we could not find a tool that, straight out of the box, ticked all of these requirements. We were then faced with the need to create our own testbed which we further detail in the next sections.

4.2.1 Architecture

If we look carefully into the set of requirements above we see that a core principle that can be extracted is reproducibility. Reproducibility allows us to run our testbed locally or in a cloud service somewhere in the world and still expect the same behaviour and results. Reproducibility is usually achieved through virtualisation mechanisms that allow a certain workload to behave and act the same, whether we are running it locally in our workstation or in a machine with different specifications in the other end of the world. Nowadays this is most commonly done through containerisation techniques [13] [14] with tools such as Docker [15], which we used to create a containerised version of our module. To orchestrate our containerised application we used Kubernetes [16], an open source orchestration platform based on Google's learnings on running containerised workloads at scale [17], and one of the most popular solutions in the field.

Kubernetes exposes a set of relevant usage metrics [18] [19] however we still need to process, aggregate and correlate these with the data from our test runs. In order to do this we rely on Elasticsearch [20] to store metrics and test result data. Prior to consumption by the Elasticsearch, the data is piped through Logstash [21] for adequate processing. At each Kubernetes node, a couple of agents entitled Beats [22] are responsible for forwarding metrics and test data acquired through logs (events published, messages sent, etc.). The final piece of the puzzle is Kibana [23], used to query data and build useful visualisations. Together, all these tools are commonly referred to as the Elastic Stack [24] and are the key to how we gather results from our test runs. Figure 4.3 provides an overview of this pipeline in practice.

As we know it, Pulsarcast is just a module that applications can use to build on top of. In order to test it properly we needed something to allow us to perform operations on demand in a Pulsarcast network (create a topic, subscribe publish, etc.). For both simplicity sake but also because we wanted to use it as our own test baseline, we've created a fork of the Javascript IPFS module, and some of its dependencies, that instantiates and exposes a Pulsarcast node. We stripped it down of anything that it didn't need in order to run Pulsarcast and its own version of pub-sub, so that we could avoid having extraneous logic disturb our test results and/or increase resource consumption. This way we got an application which exposed an HTTP API that allowed us to interact with both systems (Pulsarcast and IPFS' pub-sub). The source code for this fork is available and open source [25].

In order to accurately test our solution we needed a way to simulate abnormal network conditions. Toxiproxy [26] ended up being our tool of choice. Toxiproxy worked by acting as a TCP proxy that, programmatically through an HTTP API, allowed us to inject multiple kinds of faults (known as toxics in the project). By putting it in front of our IPFS nodes we were able to simulate different network conditions. Figure 4.3 provides a visual explanation on how our IPFS deployment and Toxiproxy fit together.

Our IPFS deployment together with Toxiproxy makes up a configurable deployment to which we call IPFS Testbed. The template for this deployment (built using Helm [27]) can be seen in our Helm Charts

[13] https://www.freebsd.org/doc/handbook/jails.html
[14] https://blog.jessfraz.com/post/containers-zones-jails-vms
[15] https://www.docker.com/products/container-runtime
[16] https://kubernetes.io/
[17] https://research.google/pubs/pub43438/
[18] https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/
[19] https://github.com/kubernetes/kube-state-metrics
[20] https://www.elastic.co/products/elasticsearch
[21] https://www.elastic.co/products/logstash
[22] https://www.elastic.co/products/beats
[23] https://www.elastic.co/products/kibana
[24] https://www.elastic.co/products/elastic-stack
[25] https://github.com/jgantunes/js-ipfs
[26] http://toxiproxy.io
[27] https://helm.sh/

repository [28].

This whole setup has been coded so that we can run it an automated fashion, being able to bootstrap the Kubernetes cluster, deploy the Elastic Stack and deploy any amount of IPFS Testbed nodes. Figure 4.4 gives us an idea of what this would look like. As usual, the logic for this is open source and accessible [29].

4.2.2 Usage

[28] https://github.com/JGAntunes/helm-charts/tree/master/ipfs-testbed
[29] https://github.com/JGAntunes/ipfs-testbed

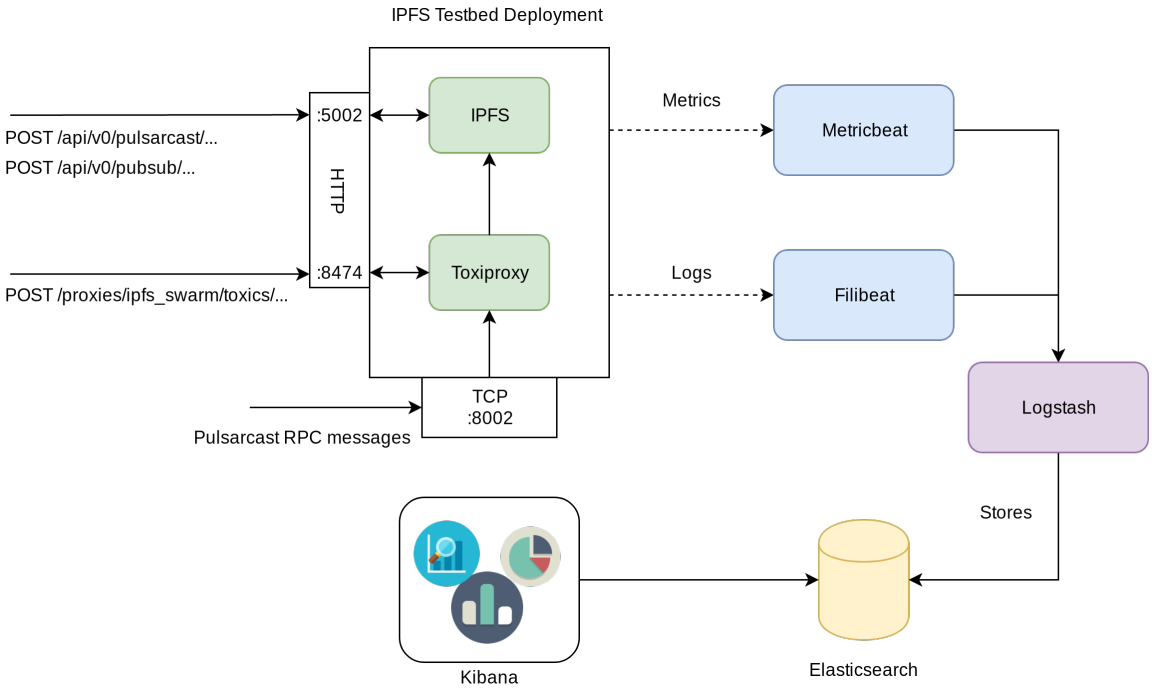

Figure 4.3: Overview of our ipfs-testbed deployment and our metrics/logs pipeline

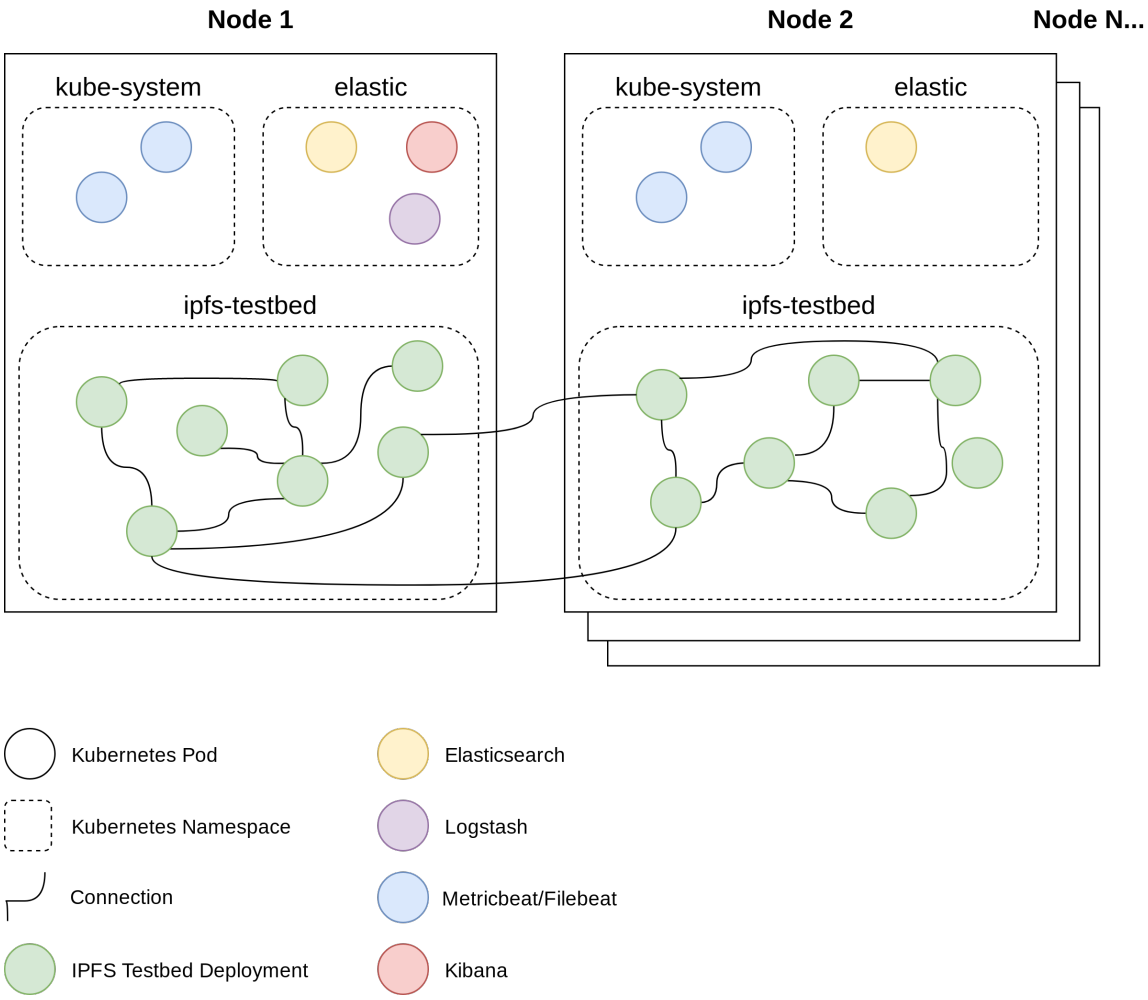

Figure 4.4: Example of our system deployed in a Kubernetes cluster

4.3 Summary

TODO

Chapter 5

Evaluation

In order to evaluate our system we start by looking at its core focus, its eventual delivery guarantees and overall performance. We not only test our system by itself but we also compare it with the current solution tied to libp2p, entitled floodsub. To do this we rely on the tools previously described in 4.2. TODO roadmap

Afterwards we seek to evaluate Pulsarcast's new functionality, such as the ability to rebuild our topic stream history through data immutability and persistence.

5.1 Testbed configuration

Our test runs were designed to be performed in managed infrastructure (commonly known as cloud services). For the initial runs and general fine tuning of the platform we relied in Google Cloud's managed Kubernetes solution. Later on, and for our actual test executions, we ran all of our tests in Microsoft's Azure Kubernetes solution, thanks to Microsoft and the Azure team we were kind enough to support our efforts and offer us free credits.

Our whole setup consisted of a total of 5 VMs acting as Kubernetes Worker nodes, each with 2 vCPUs, 16 GiB of RAM and 32 GiB of storage. In our cluster, besides other operational bits, we ran 3 Elasticsearch instances, 1 Logstash instance, 1 Kibana and a total of 100 IPFS Testbed deployments (as described in 4.2.1). Because we wanted to avoid resource starvation and to better take advantage of the Kubernetes scheduler, our testbed deployments allocate 440 MiB per deployment, each burstable to a maximum of 500 MiB. During our whole test execution, periodic HTTP health checks (part of the Kubernetes platform) make sure our deployments are working accordingly.

Test executions are managed through a single machine, from where all the commands are sent. During execution, a max of 5 commands are performed in parallel, with a slight 10 millisecond delay added between each bulk execution. All the requests are subject to retries, in case of failure, to a maximum of 5 attempts, after which the failure is registered and the execution moves on.

5.2 Dataset

To test our system accordingly, we wanted a dataset that could simulate a real life scenario as much as possible. We chose to use a dataset of Reddit's [1] comments from 2007 [2] [3] consisting of a sample

[1] https://www.reddit.com/

[2] http://academictorrents.com/details/7690f71ea949b868080401c749e878f98de34d3d

[3] `https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/`

```
// Topic
{
  "type": "topic",
  "node": "node-71",
  "name": "reddit.com",
  "author": "test-user",
  "totalNumberEvents": 1
}

// User
{
  "type": "user",
  "name": "foobar",
  "node": "node-71",
  "events": [
    {
      "internalId": 0,
      "topic": "reddit.com",
      "body": "test",
      "ups": 1,
      "downs": 0,
      "controversiality": 0
    },
    {
      "internalId": 176,
      "topic": "reddit.com",
      "body": "test-123",
      "ups": 1,
      "downs": 4,
      "controversiality": 0
    }
  ],
  "subscriptions": {
    "reddit.com": true,
    "politics": true,
    "business": true,
  }
}
```

Listing 5.1: Data example to be used in testbed

of approximately 25000 comments in a total of 23 topics (known as sub-reddits in the platform). For the purpose of our test runs, we used the comments as events to be injected in our system and the sub-reddits as the topics in which these were published.

5.2.1 Filtering and Normalisation

Our dataset consisted in line separated JSON structures, each describing a comment. Given the large set of data, we started by sampling a set of 25000 messages. Following this we needed to first, remove comments from unknown users (users that had deleted the account at the time when the comments were scrapped), followed by normalising our user number (reduce the number of publishers in the dataset to the number of active nodes in our pulsarcast system), as well as correlating all of our data. We ended up creating a CLI tool that consumed data from this dataset and generated a document of multiple JSON objects separated by newlines [4], ready to be used by our ipfs-testbed-cli, described in chapter 4.2.2. Examples of the output produced can be seen in 5.1

[4] https://github.com/JGAntunes/pulsarcast-test-harness

5.2.2 Data distribution

The following graphs give us a distribution analysis of events published per topic 5.1, subscriptions per topic 5.2 and subscription distribution per nodes 5.3. Given our dataset choice, we aimed for a non uniform subscription distribution per topic and, as it would be expected in a real world scenario, the distribution of events follows a power law based on its popularity.

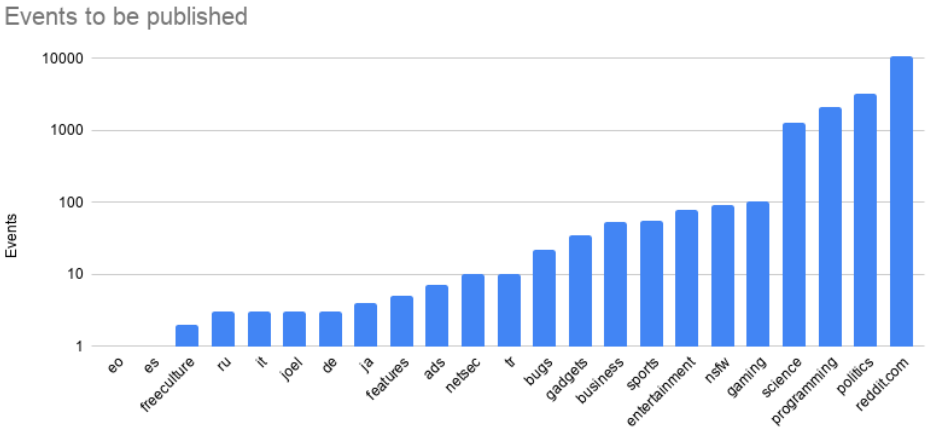

Figure 5.1: Event distribution per topic with log scale

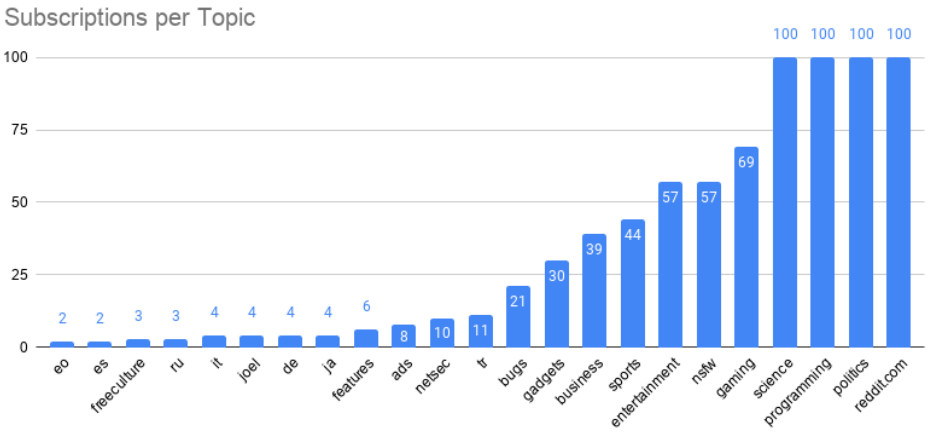

Figure 5.2: Subscription distribution per topic

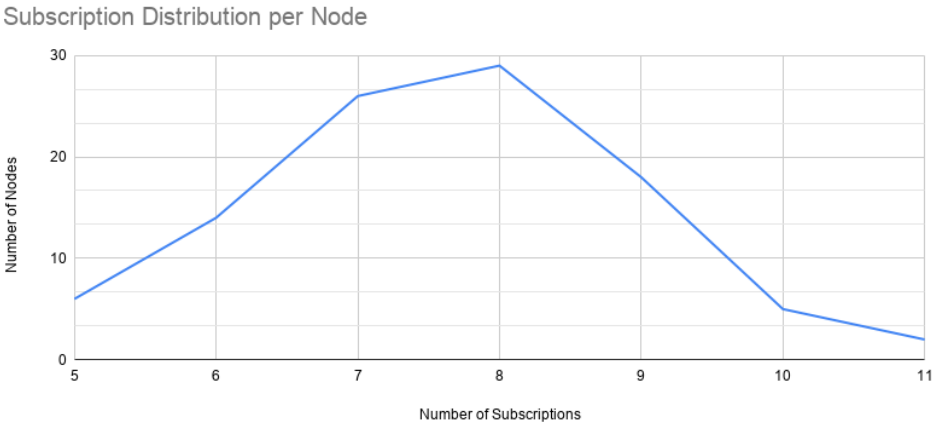

Figure 5.3: Subscription distribution per number of nodes

5.3 Metrics

For each execution we look to extract two key groups of data. Resource usage data and QoS data. The following list describes these in more detail:

- Resource usage as a total in the whole cluster, and per node (95/99 percentile and average)
  - CPU Usage (CPU number)
  - Memory Usage (GiB)
  - Network Usage (MiB transmitted)
- QoS
  - Events published by topic and in total
  - Events received by topic and in total
  - Percentage of subscriptions fulfilled based on the number of events successfully published
  - Percentage of subscriptions fulfilled based on the number of events initially injected in the system
  - Number of RPC messages sent per topic and in total
  - Average, standard deviation and percentiles (99/95) of RPC messages sent by node

It's important to keep in mind that some of the metrics under the QoS group only make sense in Pulsarcast test runs, hence will be ignored when running the baseline Floodsub solution.

5.4 Executions

We have devised a total of 6 test executions we wanted to go through and compare results. Starting with the 3 base scenarios we wanted to test:

- Pulsarcast without order delivery guarantee
- Pulsarcast with order delivery guarantee
- Floodsub (baseline solution)

For the first one, we run a scenario where every created topic in Pulsarcast allows every participating node to publish events. For the second execution, we configure the topics so that only the creator of the topic is allowed to publish, all the events from other nodes will need to be forwarded as a request to publish (as described in 3.3.3). Floodsub was used as it is, as no configuration is required.

For each of the executions described above, we performed two tests, one without any network disturbances, and another using Toxiproxy's features, adding a latency of 500 milliseconds and 300 milliseconds of jitter to every incoming TCP packet.

5.5 Results

In this section we will evaluate the results for each of the executions we described, followed by a comparison of these.

5.5.1 Pulsarcast With Order Guarantee

For this test we explored how the Pulsarcast system performed when only a single node (the creator of the topic), was allowed to effectively publish events. The execution took a total of 38 minutes, however, at the 10 minute mark, one of our nodes (the root node for the reddit.com topic) became unresponsive due to the load it was dealing with and the lack of CPU power to handle it, eventually leading the Kubernetes scheduler to restart it. Given that Pulsarcast does not provide a recovery mechanism for root nodes for a topic out of the box and the fact that this was the only node allowed to publish, we ended up seeing a clear impact in our results.

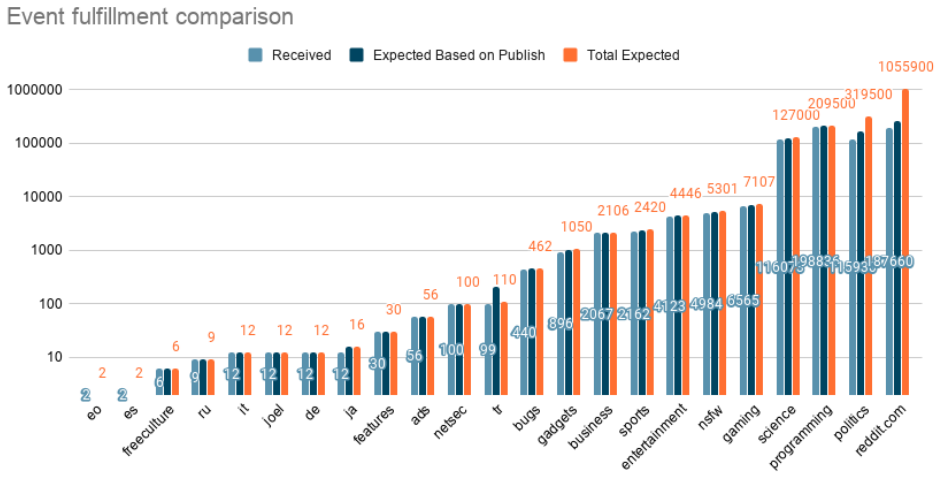

Figure 5.4: Comparison of of events fulfilled by topic in a log scale

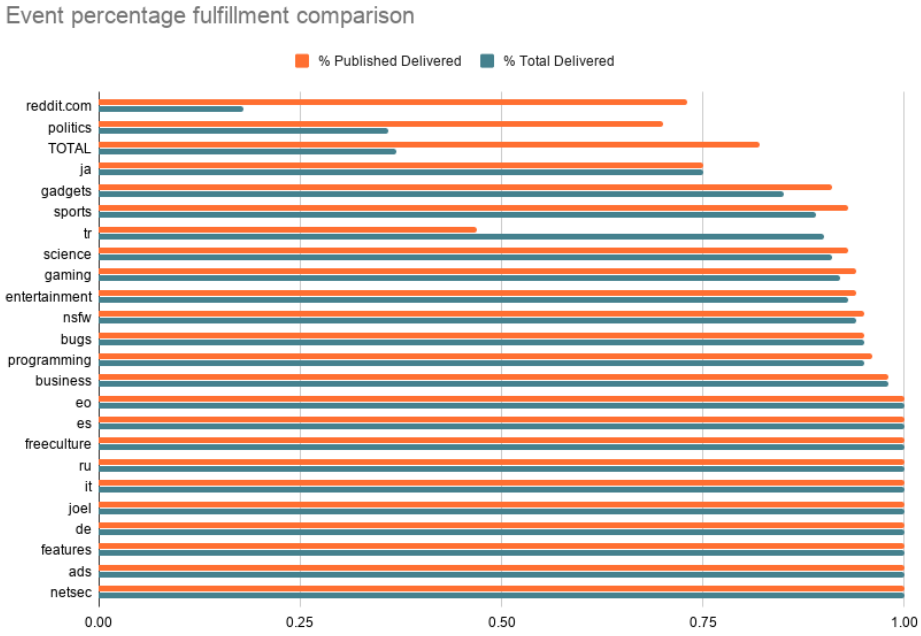

Figure 5.5: Comparison of percentage of events fulfilled by topic

Looking at 5.4 and 5.5 we can see that, of the total of events injected into the system, Pulsarcast fulfilled 37% of its subscriptions. For the events events published, Pulsarcast fulfilled 80% of its sub-

Table 5.1: Resource utilisation metrics

/	P95	P99	Average	Total
Memory (GiB)	0.207	0.358	0.178	17.84
Network (MiB)	26.26	99.41	10.5	1050.4
RPC Messages Received	8440	8961	7315.18	[NA]
RPC Messages Sent	33655.5	80058.5	5889.29	[NA]

scriptions. The biggest contributors for these lower percentages were the reddit.com and the politics topic, effectively due to the drop-out node.

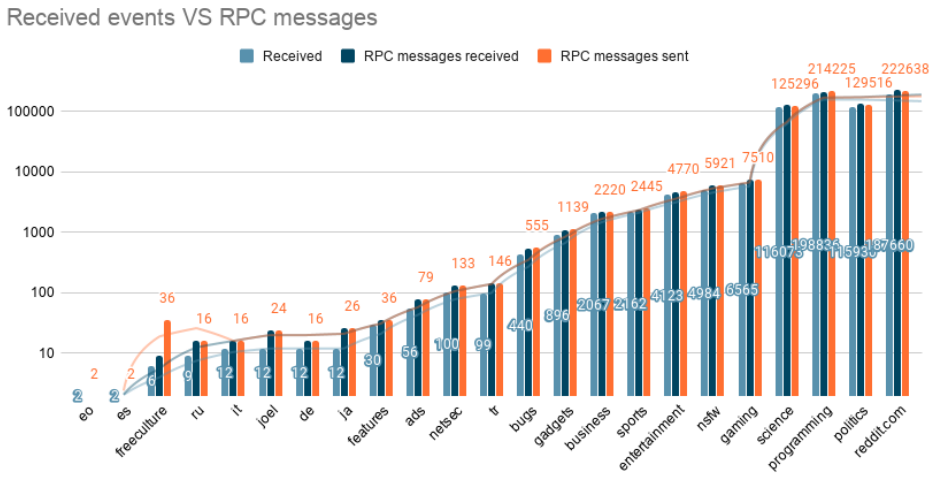

Figure 5.6: Comparison of events received and RPC injected in the system

Considering figure 5.6, we see that the RPC messages injected in the system grow linearly with the number of events received.

Table 5.1 provides an overview of the Memory and Network utilisation by node. Given these grew linearly throw the test execution we are only presenting the final values. We also provide an RPC message analysis by node. Despite having a low consumption overall (Network and Memory wise), we can see the presence of a relatively large set of outliers, a possible consequence of the unfairness of this execution, given its tendency to overload the owners of the topics. Now, looking at graph 5.7 which shows us the CPU usage across time, we can see the same outlier pattern as before, as well as the moment (at the 10 minute mark) when the CPU usage picked for the node which eventually became unresponsive.

5.5.2 Pulsarcast Without Order Guarantee

5.5.3 Floodsub

5.5.4 Pulsarcast With Order Guarantee and Latency

5.5.5 Pulsarcast Without Order Guarantee and Latency

5.5.6 Floodsub With Latency

5.5.7 Comparison and Discussion

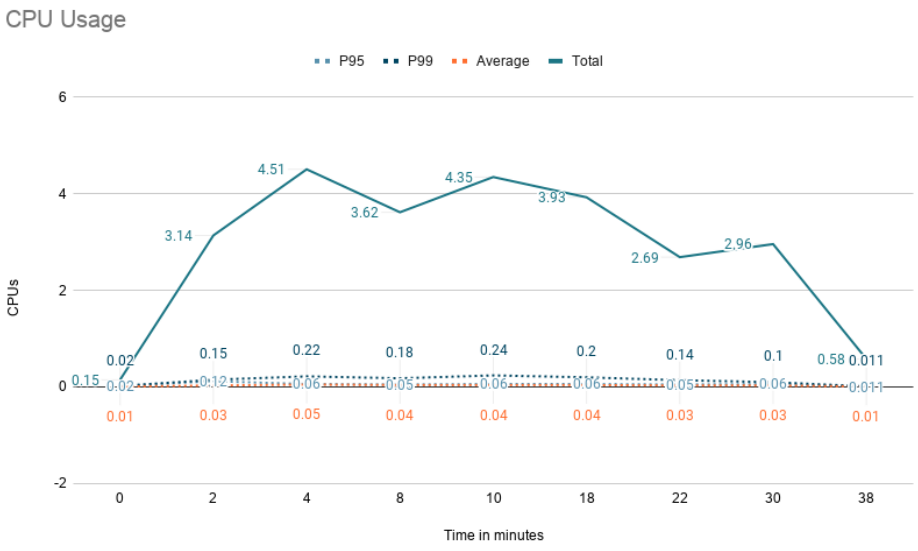

Figure 5.7: CPU usage across time

Chapter 6

Conclusion

Conclusion here...

Bibliography

[1] P. Baran, “On Distributed Communications Networks,” 1964.

[2] A.-M. Kermarrec and P. Triantafillou, “XL peer-to-peer pub/sub systems,” *ACM Computing Surveys*, vol. 46, no. 2, pp. 1–45, 2013.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, “Scribe:A large-scale and decentralized application-level multicast infrastructure,” *IEEE Journal on Selected Areas in Communication*, vol. 20, 2002.

[5] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, “Bayeux,” in *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video - NOSSDAV ’01*, no. June, (New York, New York, USA), pp. 11–20, ACM Press, 2001.

[6] R. Baldoni, R. Beraldi, V. Q. Ema, L. Querzoni, and S. Tucci-Piergiovanni, “TERA: Topic-based Event Routing for peer-to-peer Architectures,” 2007.

[7] V. Setty and M. V. Steen, “Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub,” *Proceedings of the 13th . . .* , pp. 271–291, 2012.

[8] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, “An efficient multicast protocol for content-based publish-subscribe systems,” *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pp. 262–272, 1999.

[9] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, “Gryphon: An Information Flow Based Approach to Message Brokering,” *Arxiv preprint cs9810019*, vol. cs.DC/9810, pp. 1–2, 1998.

[10] G. Cugola, E. Di Nitto, and A. Fuggetta, “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS,” *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 827–850, 2001.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, “Design and evaluation of a wide-area event notification service,” *Foundations of Intrusion Tolerant Systems, OASIS 2003*, vol. 19, no. 3, pp. 283–334, 2003.

[12] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi, “Meghdoot: Content-Based Publish/Subscribe over P2P Networks,” *Springer LNCS*, vol. 3231/2004, no. Middleware 2004, pp. 254–273, 2004.

[13] A. Bharambe, S. Rao, and S. Seshan, “Mercury: a scalable publish-subscribe system for internet games,” *1st Workshop on Network and Systems Support for Games (NetGames ’02)*, pp. 3–9, 2002.

[14] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. V. Steen, “Sub-2-Sub: Self-Organizing Content-Based Publish and Subscribe for Dynamic and Large Scale Collaborative Networks,” tech. rep., 2005.

[15] P. Eugster, R. Guerraoui, J. Sventek, and A. L. Scotland, “Type-Based Publish/Subscribe,” tech. rep., Swiss Federal Institute of Technology, Lausanne, 2000.

[16] P. R. Pietzuch and J. M. Bacon, “Hermes: A distributed event-based middleware architecture,” *Proceedings - International Conference on Distributed Computing Systems*, vol. 2002-Janua, pp. 611–618, 2002.

[17] S. Voulgaris, D. Gavidia, and M. Van Steen, “CYCLON: Inexpensive membership management for unstructured P2P overlays,” *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–216, 2005.

[18] A. Stavrou, D. Rubenstein, and S. Sahu, “A Lightweight, Robust P2P System to Handle Flash Crowds,” vol. 22, no. 1, pp. 6–17, 2002.

[19] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. Li, R. van Renesse, and G. Tredan, “How robust are gossip-based communication protocols?,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, p. 14, 2007.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A Scalable Peer-to-peer Pookup Service for Internet Applications,” *Sigcomm*, pp. 1–14, 2001.

[21] P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” pp. 53–65, 2002.

[22] S. Saroiu, P. K. Gummadi, and S. Gribble, “A Measurement Study of Peer-to-Peer File Sharing Systems,” *SPIE MMCN ’02: Proc. of the Annual Multimedia Computing and Networking*, vol. 4673, pp. 156–170, 2002.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, “A scalable content-addressable network,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 161–172, 2001.

[24] A. Rowstron and P. Druschel, “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems,” No. November 2001, pp. 329–350, 2001.

[25] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, “Tapestry: A Resilient Global-Scale Overlay for Service Deployment,” *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, jan 2004.

[26] S. Voulgaris and M. Van Steen, “VICINITY: A pinch of randomness brings out the structure,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8275 LNCS, pp. 21–40, 2013.

[27] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, “Matching events in a content-based subscription system,” *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing - PODC ’99*, pp. 53–61, 1999.

Appendix A

Appendix chapter