PulsarCast

Scaling PubSub over the Distributed Web

João Antunes, j.goncalo.antunes@tecnico.ulisboa.pt

Instituto Superior Técnico

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. 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 delivery guarantees.

Keywords:
Publish Subscribe, Peer to peer, Web, Reliability, Persistence

Table of Contents

1 Introduction

The publish-subscribe (pub-sub) interaction paradigm is an approach that has received an increasing amount of attention recently [12] [10]. This is mainly due to its special properties, that allow for full decoupling of all the communicating parties. 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. 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.

The previously discussed decoupling can be broken in 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 they are 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 a efficient way.

Due to the properties described above, a lot of applications rely on the publish-subscribe paradigm and a lot of work as been done by companies like Twitter [1] and LinkedIn into making these systems highly scalable, 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 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] 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
[6] https://ipfs.io/

2 Objectives

As we are going to cover in the next sections, lots of different solutions exist in the field. 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 a pub-sub module to IPFS with a strong focus on reliability, delivery guarantees and data persistence, while maintaining the ability to scale to a vast number of users, using the network infrastructure 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.

3 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 3.1 will lay the ground to define how pub-sub systems are structured. In section 3.2 we will cover a set of systems defined as relevant. Finally, section 3.3 will address some of the web technologies of interest in this area.

3.1 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* 3.1, followed by the *Network Architecture* 3.1 and *Overlay Structure* 3.1. Finally we address the *Subscription Management and Event Dissemination* 3.1.

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 [12] [10] 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 [7], mainly due to its similarity with the group communication systems already in place at the time. Some examples of the topic based approach allow to build a topic hierarchy. A specific one is using a UNIX path like approach, which allows to build topic hierarchy just like paths in a file system. Consider as an example:

```
/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 [7], Bayeux [26], Tera [3] and Poldercast [18].

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 [4]. 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 [8].

Message

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

Subscription

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

[7] https://www.tibco.com/
[8] https://www.json.org/

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 [21], Jedi [8], Siena [6], Meghdoot [11], Mercury [5] and Sub-2-sub [23].

Also worth referencing is the **type based subscription model**. [9]. 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 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 [14] is a reference system implemented on top of a distributed network.

While looking back at these different models its 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 given above for a stock exchange system: these 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 in the notion of topics and groups.

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.

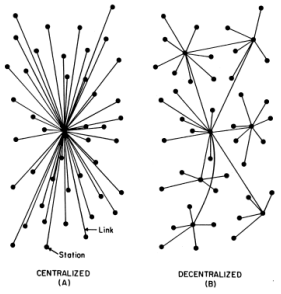

Fig. 1. Example of a Centralised and a Decentralised network

Note that the goal of a pub-sub system is to enable the exchange of events in a 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 is an example to illustrate how would a centralised pub-sub system work.

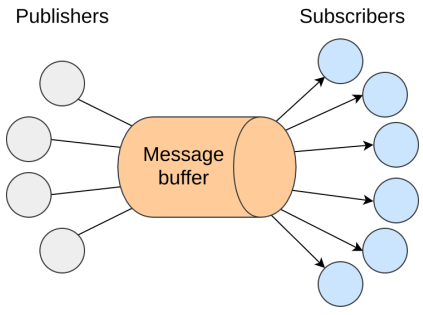

Fig. 2. Example of a centralised message broker

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 [21], Siena [6] and Jedi [8]. But, even between them there are some clear differences on how these broker networks organise. In both Gryphon and Jedi these nodes organise in 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 on 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 [7]. These kind of systems have a great focus on scalability and, consequently, on efficient message delivery.

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 [9], Kazaa [10] or Skype [11] 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.

[9] https://web.archive.org/web/20000620113133/http://gnutella.wego.com
[10] https://web.archive.org/web/20040701062605/http://www.kazaa.com:80/us/index.htm
[11] https://www.skype.com

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 keeping the overlay connected in the event of network failures.

One relevant example is Cyclon [22], 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 [19].

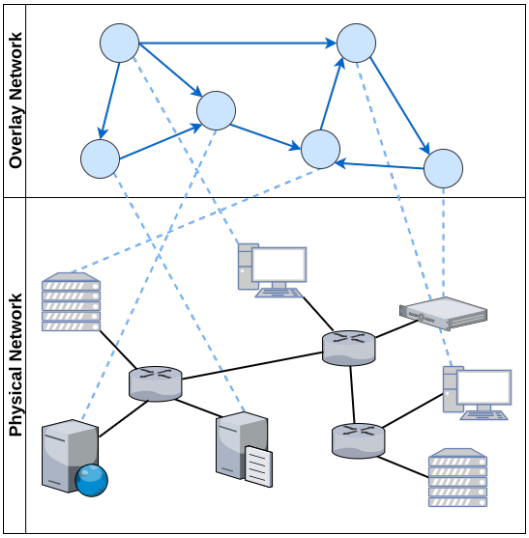

Fig. 3. A comparison between the physical network and a logical overlay

The unstructured overlay has an interesting set of properties, such as its ability to accommodate for 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 [2]. 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 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 [12] of its ip adress). 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.

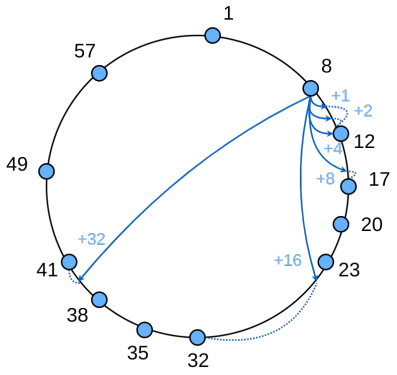

Fig. 4. Example of a simple Chord ring and the finger table of a node

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

It is common for Distributed Hash Tables to have a cost of O(logN) in terms of 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** [13]. 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 stems from a careful analysis of Gnutella trace data [17] 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**[15]. 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.

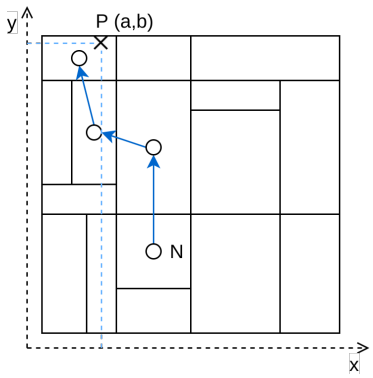

Fig. 5. Example of a 2 dimensional CAN routing command

Other popular examples in the DHT field are **Pastry** [16] 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 in 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** [24] 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 a 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 an healthy and reliable structure.

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 in 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 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 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.

3.2 Relevant Pub-Sub Systems

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

Gryphon Gryphon [21] 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 [1]. 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 6 illustrates this approach.

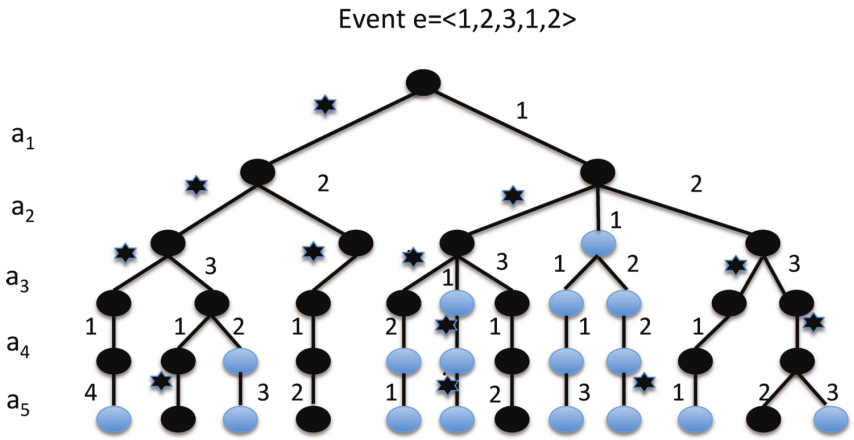

Fig. 6. extracted from [12]. Each level of the broker tree represents an attribute. When the event $e =< 1, 2, 3, 1, 2 >$ is published, all dark circles (representing brokers) are visited.

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 by 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 [13].

Siena Siena [6] 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.

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 trough these activated paths.

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

Scribe Scribe [7] 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 a 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*.

Meghdoot Meghdoot [11] 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 the 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 persist replicas of the subscriptions on the other half, making Meghdoot a system with fault tolerance by default.

Poldercast Poldercast [18] 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 where the event originated from), or will simply ignore if it is not the first time it has received it. These mechanisms, depending of 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).

Systems Analysis We will now compare some of the systems we have mentioned so far. Table 1 will serve as a useful comparison mechanism for this. A couple of notes on the terminology used. We address *delivery guarantees* as the ability to deliver a message under normal working conditions and refer to *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 have been addressed throughout the previous sections.

3.3 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 are a lot more complex 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 to, but also in browsers running in multiple platforms.

It is indisputable that one cannot think of modern web development without speaking of **Javascript** [14]. 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 [15]. 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 restrict to browsers only. **NodeJS** [16] was the first successful implementation of a Javascript runtime for the server, built on top of Chrome's V8 JavaScript engine [17]. 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 [18] such as network, filesystem and others. A key aspect in NodeJS was the way it chose to deal with the lack of

[14] https://www.ecma-international.org/publications/standards/Ecma-262.htm
[15] https://insights.stackoverflow.com/survey/2017
[16] https://nodejs.org
[17] https://developers.google.com/v8/
[18] https://nodejs.org/api/

Systems / Properties	Subscription Model	Architecture	Overlay Structure	Subscription Management	Event dissemination	Relay Free Routing	Delivery Guarantees	Fault Tolerance
Gryphon	content based	centralised broker hierarchy	N/A	route to broker node responsible for subscription	tree hierarchy	N/A	yes	best effort
Siena	content based	centralised broker mesh	N/A	route through the whole system, keep state at each node	flood with cached state at each node	N/A	yes	best effort
Jedi	content based	centralised broker hierarchy	N/A	route through the whole system, keep state at each node	tree hierarchy	N/A	yes	best effort
Bayeux	topic based	decentralised	Tapestry DHT	route to rendezvous node	multicast tree	no	yes	best effort, no subscription persistence
Scribe	topic based	decentralised	Pastry DHT	route to rendezvous node	multicast tree	no	yes	best effort, no subscription persistence
Medhdoot	content based	decentralised	CAN DHT	points in CAN DHT	CAN route (line in multi-dimensional space)	no	yes	replicated subscritions
Hermes	type based	decentralised	Pastry DHT	rendezvous node	multicast tree	no	yes	best effort
Tera	topic based	decentralised	gossip based overlay		random walk outside cluster, flood after	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/ gossip based overlay	ring overlay with subscribers	gossip outside cluster, route through ring after	no	no	best effort
Poldercast	topic based	decentralised	ring based DHT/ Vicinity / Cyclon	ring overlay with subscribers	route through ring overlay	yes	yes (every publisher is a subscriber)	high resillience to churn, no subscription persistence

Table 1. Comparison table for the relevant system

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** [19], 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 [20] - "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 [21]. 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 [22]. **WebSockets** [23] aimed at providing a real-time, full-duplex communication between clients and servers over TCP, but it was **WebRTC** [24] 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 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** [25] and **QUIC** [26], 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 [27] 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) [28], 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

[19] https://www.npmjs.com/
[20] https://archive.org/details/bstj57-6-1899
[21] http://blog.npmjs.org/post/143451680695/how-many-npm-users-are-there
[22] https://tools.ietf.org/html/rfc793
[23] https://tools.ietf.org/html/rfc6455
[24] https://www.w3.org/TR/webrtc/
[25] http://www.bittorrent.org/beps/bep_0029.html
[26] https://datatracker.ietf.org/wg/quic/about/
[27] https://tools.ietf.org/html/rfc2663
[28] https://ipfs.io

as the Merkle DAG [29]. 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. Each node can have links to other nodes, creating a persistent, chain like, structure that is immutable. 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, blockchains, etc.) 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 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** [30], a set of packages that seek to solve common challenges in P2P applications. 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.

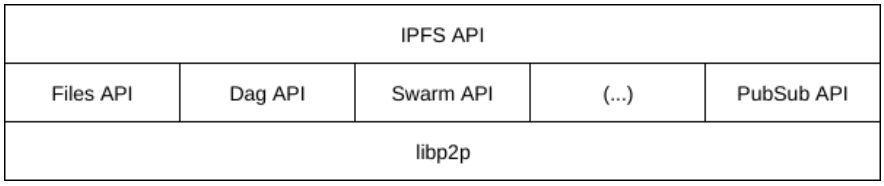

Fig. 7. An illustration on the IPFS architecture

4 Proposed Solution

We now describe our proposed solution. Since our goal is to have a highly scalable system with reliability and persistence in mind, we decided to take advantage of the IPFS ecosystem and all of its different modules. Our pub-sub module will provide an alternative to the naive implementation of pub-sub, currently in place in IPFS. The modularity of the IPFS system allows users to choose what is more convenient for them. Besides, the existence of a base implementation allows us to have a baseline for improvement, one from which we can extract metrics and relevant data.

[29] https://github.com/ipld/specs/tree/master/ipld
[30] https://libp2p.io

We will start by covering our subscription model and the multiple structures that describe events and subscriptions. Afterwards we will take a closer look at the overlay structure used. We will then cover how the system works in terms of subscription management, focusing on how new subscriptions are handled, how new topics are issued and how the topic hierarchy works. Finally we will cover event dissemination followed by quality of service, focusing on the mechanisms we have used to bring persistence, fault tolerance and delivery guarantees to the network.

4.1 Subscription Model and Data Structures

Our subscription model follows a topic based approach. It has however some nice properties that make it far more expressive than what would be expected of a regular topic based system. To do this, we take advantage of the core structure of IPFS, the Merkle DAG, through its previously described interface IPLD. We start by defining two basic structures: the *topic descriptor* and the *event descriptor*.

The **topic descriptor** defines the basic structure of our topics. It is *versioned* by default and possesses links (or merkle-links as it is referred to in the IPLD specification) to its sub-topics. A *metadata* object also helps to store other relevant information such as the protocol version. The following is an example using the JSON format which IPLD uses.

```
{
  name: <topic-name>,
  metadata: <json-object>, // creation date, protocol version, etc.
  #: {
    <sub-topic-name>: <merkle-link to topic descriptor>,
    ...
  },
  parent: <merkle-link to topic descriptor>
}
```

These structures are addressed by the hash of its content, which in IPFS are referred to as Content Identifiers [31] or CID. In fact, given the definition of these objects, all the content of the structure itself is addressable based on its CID. For example, if we had a topic descriptor with a CID *foobar-hash* we could address it using a UNIX path approach such as */foobar-hash*. However we could access its properties directly such as */foobar-hash/#* to get the sub-topics list, or */foobar-hash/parent* to get the previous version of this topic. These paths are referred to under IPFS as *merkle-paths*.

Given the immutable nature of the IPLD structure, the parent link allows us to create new updated versions of the topic descriptor (with a new sub-topic for example) while maintaining a history of previous topic descriptors for this

[31] https://github.com/ipld/cid

topic. The key # on the other hand contains a JSON object with merkle-links for all of sub-topics indexed by name.

The topic name is a string following a UNIX like path pattern. For example */sports*. If we are speaking about sub-topics though, there is an extra requirement, given that a sub-topic name needs to be consistent with its parent hierarchy. This means that, for a topic */sports*, it cannot have a sub-topic */fruits*, or */fruits/apples*. */sports/football* however is a valid example.

The **event descriptor** defines the structure of our events. Each object has links to its *topic* and its *parent* which represents the previous event in this stream. A *metadata* object is used to store creation timestamps an other relevant information. The *publisher* key will be a reference to the ID of the publisher node. Finally we have the *payload* object, which will contain the actual information of the event. The following is an example of it.

```
{
  topic: {
    name: <topic-name>, // Name of the topic
    link: <merkle-link> // Link to the topic of this message
  },
  publisher: <publisher node ID>
  parent: <merkle-link to previous event>,
  metadata: <json-object>, // Timestamp and other relevant info
  payload: <json-object>, // The actual message content
}
```

It is worth noticing the importance of both *parent* links in each structure. These allow both graph structures to build a complex history. On one side we have the topic descriptors with a versioning system and on the other we have a stream of events represented as a chain of immutable objects. These concepts are very important for the overall system.

We still need however one more structure to help us build our system. Given the immutability of the objects above, we need a way to point to the latest version of a given topic descriptor so that other peers can easily find it. Fortunately, this scenario is already covered by the IPFS ecosystem through IPNS [32], which aims at providing mutability over all these immutable structures. An IPNS record has a special CID, generated through the hash of a public key from a cryptographic key pair. These records can be used to point to a specific CID and are generated using asymmetric cryptography to sign the record. The records are ephemeral though, so it requires the owner of the asymmetric keys to republish the record every 24 hours. We have two important guarantees with this structure: mutability, which allow creators of topics to point to a new version of a topic descriptor seamlessly, keeping the same CID as an entry point for the topic; authenticity of the given topic, given that peers can check the IPNS record signature and decide if they trust the peer owning that record or not.

[32] https://github.com/ipfs/specs/tree/master/iprs

4.2 Overlay Structure

Since our work will be a libp2p compatible module, we will be able to leverage the multiple modules that already exist in libp2p ecosystem. This includes network transports, discovery and routing mechanisms as well as other useful data types and utility methods. Figure 9 illustrates where our work will take place and some of the other modules we will be able to use. In order to understand it though there are some key aspects around libp2p that we need to cover first. libp2p tries to separate concerns of peer communication and data transports. In order to do that, it has different transport implementations under a common interface [33], which can then be leveraged through lip2p-swarm [34], a connection abstraction that can deal with multiple connections under different protocols. On top of this we then have the peer communication, which can be split into two big mechanisms. On one end we have the discovery mechanisms [35], which focus on ways of finding and connecting to new peers. On the other end we have peer routing [36], which focus on transferring data between already connected peers.

[33] https://github.com/libp2p/interface-transport
[34] https://github.com/libp2p/js-libp2p-swarm
[35] https://github.com/libp2p/interface-peer-discovery
[36] https://github.com/libp2p/interface-peer-routing

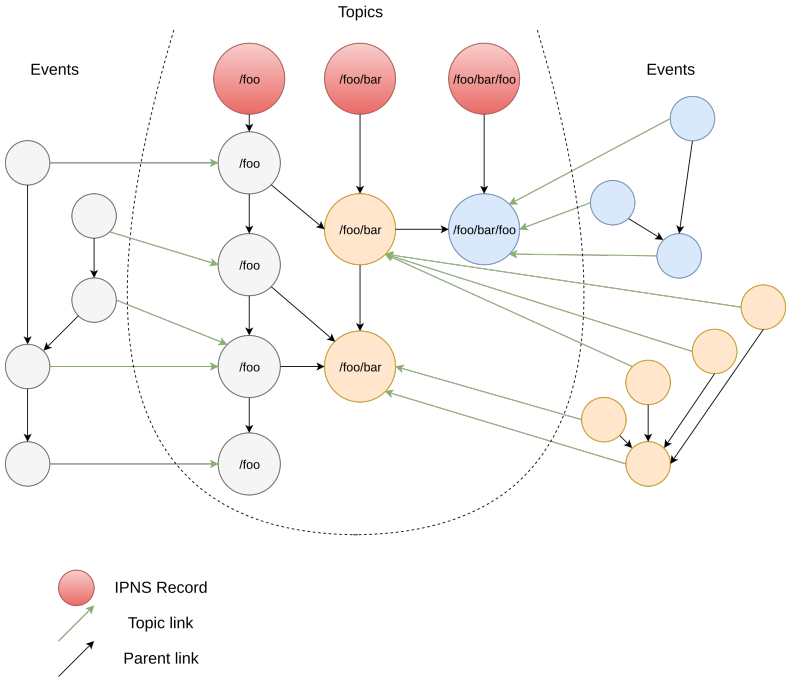

Fig. 8. A digram with an example of how these structures relate to each other

Our work will mostly reside in three different modules: the pub-sub module implementation; adding extra functionality to the Kademlia DHT peer routing module; building a small module to power gossip based communication between peers.

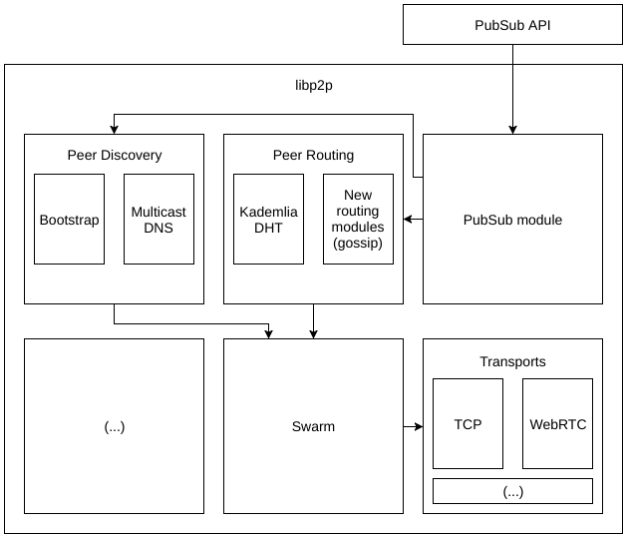

Fig. 9. The libp2p architecture and where will our work take place

IPFS currently relies on a Kademlia DHT implementation to provide a structured overlay mechanism through which it can route messages. We plan on using this same overlay as a routing mechanism for our system. In order to do that we have to understand the methods already provided by it.

- **put**: insert a value with a given key in the DHT.
- **get**: get a value of given key from the DHT.
- **findPeer**: find the peer with the given peer ID in the DHT.
- **findPeerLocal**: find the peer with the given peer ID in the list of peers to which we are already connected to.
- **getClosestPeers**: find the k(system wide parameter) closest peers to a given key.
- **provide**: let the network know that this peer can also distribute a given key.
- **findProviders**: find providers for a given key.

Some methods are basic functions of Kademlia as explained in the previous section (e.g. *getClosestPeers* is basically a *node lookup* operation). However, some of these methods are not part of the Kademlia definition, specifically *provide* and *findProviders*. The way the *provide* method works is by setting special records at the node that is housing the key we want to provide. This way, when peers

query for providers or for the actual content, the Kademlia routing mechanism will ensure that the message will arrive at the same set of nodes consistently.

We also make use of a gossip based communication mechanism to build a simple unstructured overlay. We will cover its use case later but in functionality terms it is quite simple. At each node a fixed size list of peers should be kept, with its size being a system wide parameter. Periodically, the nodes should exchange its lists and update info accordingly. Finally the module should support a simple network flood, disseminating info across all the nodes.

4.3 Subscription Management

In our system, subscriptions are represented as multicast trees, with a different tree per topic. When a new subscription is issued, a peer, having the CID of an IPNS record for a given topic, will issue a *getClosestPeers* method for that given CID. These peers are in charge of acting as a *rendezvous* for incoming subscriptions and events. Similar to Scribe, when performing the recursive node lookup mechanism, at each step, the initiator peer checks to see if some of the peers that resulted from this step are already part of the multicast tree. In an affirmative case, the initiator node joins the tree as a child of this specific node, if not, it chooses the closest one and issues a special command for this peer to join the multicast tree. The node, upon receiving a command to join the multicast tree, registers the initiator node as its child on the multicast tree and repeats the same process until eventually a node belonging to the tree is found.

Peers that belong to a multicast tree are responsible for making sure that both parent and children nodes are healthy (through periodic pings) and reconstructing the tree at their level if needed be. For that, they keep extra state on extra levels of the tree. However, if a network partition causes these mechanisms to fail, nodes can always rejoin the network through the procedure detailed above.

Since our system accounts for topic hierarchy, it is possible to create sub-topics. If a peer wants to add a sub-topic to a topic it created, all it needs to do is add a new sub-topic link to the parent topic descriptor and afterwards issue a new IPNS record pointing to the new version of the parent topic descriptor. If the node does not own the parent topic it will need to request for the new sub-topic to be added to the node responsible for the parent topic. If the request is accepted the procedure will be the same as the one described previously.

It is important to notice that, in our topic hierarchy, subscribers of a parent topic will not automatically be subscribed to the sub-topics of it. It is easy however to subscribe to these, since whenever a new update to the topic descriptor is made, the node responsible for the topic will issue a special event and disseminate it through the multicast tree. Since updates to the sub-topic list will trigger topic descriptor updates, subscribers only have to monitor for changes on the special key # of the topic descriptor.

4.4 Event Dissemination

Event dissemination in our system is a matter of propagating the event through the multicast tree. Nodes that are already part of the tree just need to send the event through the different links the participant has. For nodes not part of the tree it is just a matter of targeting the *rendezvous* nodes. A really important note though is that before disseminating the event through the multicast tree, the publisher should always invoke the *put* method of the DHT with the event and respective CID. This will ensure the persistence of the event and help with the delivery guarantees which we will later discuss.

4.5 Quality of Service

We will now discuss the mechanisms we employ in order to provide the quality of service guarantees we set as goals for this system. We will cover the fault tolerance mechanisms, the delivery guarantees and data persistence.

Looking at **fault tolerance** we should start by the *rendezvous* nodes. We need to make sure these do not become a bottleneck for the system. Hence the usage of the k closest nodes of the given topic CID, this way, through the existing DHT, we get a natural mechanism for selecting replicas for the *rendezvous* node. When a new topic is created, the node responsible for creating it is also responsible for calculating the k closest nodes to the topic CID and communicating with the peers in order for them to become part of the network. To keep the *rendezvous* nodes synchronised, a simple gossip overlay is used (described in the previous sections). Another approach is to use the *provide* method of the DHT as a mechanism for a node to register as a *rendezvous*.

We then need to cover the IPNS record issue, for it can also become a bottleneck. An important thing to keep in mind though is that the IPNS records come with a notion of ownership which, if you take a closer look, actually makes sense here. Since the peer that created the topic is actually responsible for updating the respective descriptor and serving as *rendezvous* for the network. This ownership however does not translate at all in lock in though, the usage of these graph structures allows anyone who wants to, to create a new topic based on any previous topic that you do not even need to own. Better yet, you get all the topic history and previous event streams connected to this new graph, for free. With that said, if a set of peers really wants to create a mechanism where the failure of the node that created the topic does not imply that the current IPNS record will eventually disappear, they will need to share the key pair responsible for the record among them, so that they can keep renovating it in case of failure. This of course will imply some kind of consensus between the peers, which is outside of the scope of this work.

On the **delivery guarantee** front we developed a simple mechanism that ensures that every node subscribed to a given topic will eventually get the same event stream. This is because of the way the event stream is linked. Given that through the *parent* key of an event descriptor, a node can check if it is missing any message from the stream. If it is, it will query the Kademlia DHT using the

get method for that specific key. This approach can ultimately been seen as a simple negative ACK mechanism.

Finally, on the **persistence** front, we see these graph structures as the key to create a P2P pub-sub system that finally addresses data persistence properly. In order to this, having easily addressable structures is not enough. As such, we devised a way to guarantee a smooth mechanism for replicating event data. When propagating an event through the multicast tree, each peer will, with a given probability p (a system wide parameter) invoke the *provide* method for this specific event. This ensures data is persisted across multiple nodes which will allow for peers to eventually build their event stream from any given point. Hence our last mechanism, when a peer subscribes to a new topic, it will almost immediately receive the set of leaf events from the graph stream. This simple approach will give it the ability to rebuild the stream of events as far as it likes.

5 Evaluation

We now define the metrics that are going to help us evaluate the overall system. These are focused on, not only testing the overall architecture, but also on testing the fulfilment (or not) of the objectives we set in the beginning of this work.

We want to keep track of four important metrics at each node during all of our tests. CPU load, memory usage, bandwidth usage and disk usage. This will help us attest the efficiency of the system. As comparison we are going to use two systems. A baseline one where publishers send messages directly to all of its subscribers. And naturally, the current pub-sub system in IPFS.

The metrics we are going to track are:

- Ratio of messages sent by each node, correlated with the CPU, memory and bandwidth usages.
- Ratio of throughput speedup vs disk storage used at each node.
- Ratio between latency (total propagation time of an event) reduction vs disk storage used at each node.
- Ratio of subscriptions covered under heavy network churn (network partitions and arrival of new peers).
- Ratio of subscriptions covered after a severe network partition and its recovery.
- State of event streams at each node under heavy network churn (only applicable to our system). Monitor discrepancies in state across nodes and how long does it take to converge.
- State of event streams at each node after a network partition and its recovery (only applicable to our system). Monitor discrepancies and how long does it take to converge.

From all of these metrics we want to extract average, median and percentiles (75, 90, 95 and 99). As input data we are going to use synthetic datasets of both events and subscriptions, of different sizes and with different kinds of distributions (uniform, non-uniform, Zipfian, etc.).

In order to test these assumptions, we are going to run it on a simulator, either using PeerSim [37] or using IPTB [38] to run a cluster of sandboxed IPFS nodes.

6 Conclusions

In this work we have presented a pub-sub system for the distributed web. We started out by presenting the motivation that has led us to this work, defining a set of objectives that we seek for our system. We then documented our literature review in the subject, classifying the reviewed systems accordingly. After that we described our solution, considering all the literature and systems studied previously. Finally, we presented the metrics that will allow us to asses our solution and evaluate the quality of the work done.

[37] http://peersim.sourceforge.net/
[38] https://github.com/whyrusleeping/iptb

A Appendix

A.1 Work Scheduling Example

Estimated schedule for the remaining work presented in table 2.

Table 2. Work Scheduling

Month	**Work**
February (2 weeks)	Explore the pub-sub solution of IPFS
February (2 weeks)	Extract relevant metrics form the pub-sub solution of IPFS
March (1 week)	Plan the libp2p needed changes
March (3 week)	Implement the libp2p changes
April	Implement the new pub-sub module
May	Implement the new pub-sub module
June	Evaluate implemented solution, compare with base module
July	Write thesis report
August	Write thesis report
September	Report review and submission

References

1. Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing - PODC '99*, pages 53–61, 1999.
2. Lorenzo Alvisi, Jeroen Doumen, Rachid Guerraoui, Boris Koldehofe, Harry Li, Robbert van Renesse, and Gilles Tredan. How robust are gossip-based communication protocols? *ACM SIGOPS Operating Systems Review*, 41(5):14, 2007.
3. R Baldoni, R Beraldi, V Q Ema, L Querzoni, and S Tucci-Piergiovanni. TERA: Topic-based Event Routing for peer-to-peer Architectures. 2007.
4. G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 262–272, 1999.
5. Ar Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. *1st Workshop on Network and Systems Support for Games (NetGames '02)*, pages 3–9, 2002.
6. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *Foundations of Intrusion Tolerant Systems, OASIS 2003*, 19(3):283–334, 2003.
7. Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe:A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, 20, 2002.
8. 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*, 27(9):827–850, 2001.
9. Patrick Eugster, Rachid Guerraoui, Joe Sventek, and Agilent Laboratories Scotland. Type-Based Publish/Subscribe. Technical report, Swiss Federal Institute of Technology, Lausanne, 2000.
10. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
11. Abhishek Gupta, Ozgur D Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-Based Publish/Subscribe over P2P Networks. *Springer LNCS*, 3231/2004(Middleware 2004):254–273, 2004.
12. Anne-Marie Kermarrec and Peter Triantafillou. XL peer-to-peer pub/sub systems. *ACM Computing Surveys*, 46(2):1–45, 2013.
13. Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. pages 53–65. 2002.
14. P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. *Proceedings - International Conference on Distributed Computing Systems*, 2002-Janua:611–618, 2002.
15. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review*, 31(4):161–172, 2001.
16. Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Number November 2001, pages 329–350. 2001.

17. Stefan Saroiu, P Krishna Gummadi, and Steven Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. *SPIE MMCN '02: Proc. of the Annual Multimedia Computing and Networking*, 4673:156–170, 2002.
18. Vinay Setty and Maarten Van Steen. Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub. *Proceedings of the 13th ...*, pages 271–291, 2012.
19. Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. 22(1):6–17, 2002.
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*, pages 1–14, 2001.
21. Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An Information Flow Based Approach to Message Brokering. *Arxiv preprint cs9810019*, cs.DC/9810:1–2, 1998.
22. Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–216, 2005.
23. Spyros Voulgaris, Etienne Rivière, Anne-Marie Kermarrec, and Maarten Van Steen. Sub-2-Sub: Self-Organizing Content-Based Publish and Subscribe for Dynamic and Large Scale Collaborative Networks. Technical report, 2005.
24. Spyros Voulgaris and Maarten 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)*, 8275 LNCS:21–40, 2013.
25. B.Y. Zhao, Ling Huang, Jeremy Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, jan 2004.
26. Shelley Q Zhuang, Ben Y Zhao, Anthony D Joseph, Randy H Katz, and John D Kubiatowicz. Bayeux. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video - NOSSDAV '01*, number June, pages 11–20, New York, New York, USA, 2001. ACM Press.