Pulsarcast - Scalable and reliable pub-sub over P2P networks

João Antunes
INESC-ID Lisboa
ULisboa / Instituto Superior Técnico
me@jgantunes.com

Abstract

The publish-subscribe paradigm is a wildly popular form of communication in complex distributed systems. The properties offered by it make it an ideal solution for a multitude of applications, ranging from social media to content streaming and stock exchange platforms. Consequently, a lot of research exists around it, with solutions ranging from centralised message brokers, to fully decentralised scenarios (peer to peer).

Within the pub-sub realm not every solution is the same of course and trade-offs are commonly made between the ability to distribute content as fast as possible or having the assurance that all the members of the network will receive the content they have subscribed to. Delivery guarantees is something quite common within the area of centralised pub-sub solutions, there is, however, a clear lack of decentralised systems accounting for this. Specifically, a reliable system with the ability to provide message delivery guarantees and, more importantly, persistence guarantees. To this end, we present Pulsarcast, a decentralised, highly scalable, pub-sub, topic based system seeking to give guarantees that are traditionally associated with a centralised architecture such as persistence and eventual delivery guarantees.

The aim of Pulsarcast is to take advantage of the network infrastructure and protocols already in place. Relying on a structured overlay and a graph based data structure, we build a set of dissemination trees through which our events will be distributed. Our work also encompasses a software module that implements Pulsarcast, with our experimental results showing that is a viable and quite promising solution within the pub-sub and peer to peer ecosystem.

1 Introduction

The publish-subscribe (pub-sub) interaction paradigm is an approach that has received an increasing amount of attention throughout the century [13] [11]. This is mainly due to its unique properties, that allow for full decoupling of time, space and synchronisation of all the communicating parties. 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.

Due to the properties described above, maney applications rely on the publish-subscribe paradigm and much work has been done by companies like Twitter [1], Spotify [18] 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. Most of 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 (P2P) 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 living proof of the high scalability P2P can offer, with pub-sub systems over P2P networks being an active research topic with much attention.

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. There is also, to the best of our knowledge, a lack of pub-sub systems with a strong focus on persistence.

We intend to address this in Pulsarcast by focusing on the following properties:

- Eventual delivery guarantees;
- Data persistence;
- Ability to scale to a vast number of users;
- Take advantage of the network infrastructure and network protocols we have in place today;
- Strong focus on reliability;

Besides the specification and architectural model of our system, we also provide a concrete implementation of it. So,

[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/

in order to validate the solution we purpose, we have created the following:

- A Javascript implementation module of Pulsarcast with a clearly defined API (Application Programming Interface) through which applications can integrate with;
- A distributed test runner capable of running large scale test scenarios and simulate abnormal network conditions;
- An easy to automate test-suite based on a real-world application;

This document is structured as follows: Section 2 presents and analyses our related work. Section 3 introduces and describes Pulsarcast, its architecture, data structures and algorithms. Section 4 covers the implementation of our solution, with a more thorough overview of our Javascript module. Next, Section 5 explains our evaluation methodology and presents those results. Finally, Section 6 provides a set of closing remarks.

2 Related Work

When considering pub-sub systems, there is a set of different options that will lay the 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 [13] [11] and that implementations usually follow:

- Topic-based subscriptions - Clients subscribe to classes of events, usually identified by keywords.[6][26][3][2][19]
- Content-based subscriptions - Clients subscribe to events based on specific values (or ranges of values) on the properties of the events.[21][7][5][12][4][23]
- Type-based subscriptions[10] - Bring the notion of a type scheme to a topic-based subscription model.[15]

The subscription models are tied to the expressiveness of the system as a whole. Case in point, a content-based subscription model allows for a lot more expressiveness in subscription definition. However, it makes it a lot harder to implement a scalable way of filtering messages.

Another critical design dimension, primarily when covering P2P pub-sub systems, is how peers choose to organise and maintain their view of the underlying network (commonly referred to as network overlays). These overlays can usually be divided into structured overlays, using structures as Distributed Hash Tables (DHT) [14] [20] for example, and unstructured overlays that rely on other approaches such as gossip communication protocols [22] [25].

The systems we will cover next have chosen different approaches for the design dimensions described above; however, all of them have played a seminal role for our proposed solution.

Gryphon [21] is a content-based pub-sub system built on top of a centralised broker hierarchy topology, 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]. Developed at IBM, Gryphon uses an interesting approach to match events with subscriptions [1], relying on a distributed broker based network to build a tree structure representing the subscription schema.

Siena [5] 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. Still on the subject of broker based network solutions, HyperPubSub [27] is a recent example of such an approach but focused on bringing verifiability and other forms of decentralised validation to pub-sub operations.

Scribe [6] is a topic-based pub-sub system built on top of a fully decentralised network (P2P). In order to do this, it relies on the Pastry DHT [17] as its overlay structure. This allows it to leverage the robustness, self-organisation, locality and reliability properties of Pastry to build a set of per-topic multicast trees used to disseminate events.

Meghdoot [12] is a content-based pub-sub system. It is built on top of a P2P network, specifically CAN DHT [16]. Meghdoot leverages the multidimensional space provided by the CAN DHT in order to create an expressive content-based system.

Poldercast [19] 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 three different overlays.[24][22] Similar to systems like VCube-PS [8], 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).

Architecturally speaking, one can see the similarities between Poldercast and SELECT [2], as it too relies on a set of three different overlays, working together to fulfil subscriptions. However, SELECT brings a different set of properties to the table, as it maps the social connection graph of the peers to the actual overlays operating underneath. This way,

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

the system can exploit both the social graph and the online activity of each social user (each peer) to establish connections and disseminate messages accordingly and avoid an unnecessary number of hops.

The aim of our work is also to take advantage of the network infrastructure and technologies already in place. One of the best ways of doing so is by leveraging what the Web platform has to offer. 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 to allow the creation of simple interactions and animations in web pages it is now one of the main programming languages for the web [9], with runtimes in browsers and servers thanks to projects such as **NodeJS** [10]. In the application realm, many P2P apps have leveraged these technologies. Such examples are **browserCloud.js**[9], a solution seeking to bring cloud computing to the Web platform, taking advantage of technologies such as WebRTC [11] and Javascript and IPFS [12], 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** [13]. 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 (Merkle links) to other nodes, creating a persistent, chain-like structure that is immutable as documented in figure 1

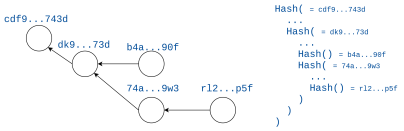

Figure 1. Graph visualisation of a Merkle DAG and its respective hash function dependencies

Having implementations in both Go and Javascript, IPFS leverages the modularity mantra in a fascinating way, focusing on creating standard interfaces that allow for different pieces of the architecture to be changed and selected according to one's needs. These small modules that constitute IPFS have recently been brought together under the same umbrella, as **libp2p** [14], a set of packages that seek to solve everyday 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, named Floodsub. 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 handpick the functionalities we intend to use.

[8] https://www.ecma-international.org/publications/standards/Ecma-262.htm
[9] https://insights.stackoverflow.com/survey/2017
[10] https://nodejs.org
[11] https://www.w3.org/TR/webrtc/
[12] https://ipfs.io
[13] https://github.com/ipld/specs/blob/95df205ca5fdb961ec2c2265a169989fef595db1/FOUNDATIONS.md
[14] https://libp2p.io

3 Pulsarcast

Pulsarcast is a peer to peer, pub-sub, topic-based system focused on reliability, eventual delivery guarantees, and data persistence. Properties usually associated with centralised pub-sub solutions.

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.

Pulsarcast is a fully decentralised solution, which means that each node plays a crucial part in fulfilling the system's purpose, delivering events and ensuring their 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 peer discovery, content discovery and to bootstrap our other overlay, our per-topic dissemination trees. These trees are critical for us to disseminate information across our decentralised network.

When a peer publishes an event or creates a new topic a set of the overlays previously 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.

Every topic and event is stored in the Kadmelia DHT before being forwarded through the topic dissemination trees. This ensures the data 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. Once persisted, we forward the data through the appropriate dissemination trees previously built. 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 has run through when forwarding events across their dissemination trees. If this fails, though, a query to the DHT is in order.

Pulsarcast has a set of two fundamental data structures to which we refer to as **event** and **topic descriptors**. All of our data structures are immutable, content addressable and linked together to form a Directed 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 2 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 so far, 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. This is the core of Pulsarcast's eventual delivery guarantees.

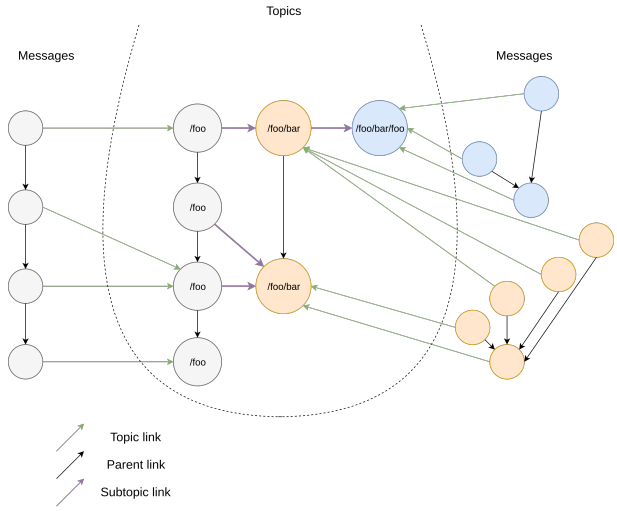

Figure 2. Representation of the Pulsarcast DAG

```
{
  "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 1. Topic descriptor schema in a JSON based format

```
{
  "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 2. Event descriptor schema in a JSON based format

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 us to borrow inspiration from systems like IPFS and decided to use *CIDs* (Content Identifiers) [15]. A CID is a self-describing content-addressed identifier. It uses cryptographic hashes to achieve content addressing and is powered by *multihash* [16]. 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 Listings 1 and 2 provide an accurate description of the schema and format of our data structures. We will cover some of the properties.

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 acts 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 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

[15] https://github.com/multiformats/cid
[16] https://github.com/multiformats/multihash

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.

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.

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. Procedure wise, the meta topic is created just like any other topic, with the same properties (except for its own meta topic of course). 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. Algorithm 1 provides an overview of the procedure to create a new topic.

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 child 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 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 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
5              metaTopic ←
                CreateMetaTopic(newTopic);
6          else
7              metaTopic ← parent.subTopics.meta;
8          end
9          topicData ←
            CreateTopic(newTopic, metaTopic);
10         Subscribe(metaTopic);
11         Subscribe(topicData);
12         StoreInDHT(metaTopic);
13         StoreInDHT(topicData);
14         Publish(metaTopic, topicData)
15     end
```

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 kept under the meta property of the topic descriptor. Figures 4 and 5 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. The node then checks

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 = sender node id
2      begin
3          topicData ← GetTopicData(topicId);
4          if fromNodeId ≠ nodeId then
5              AddToChildren(t, fromNodeId);
6              if topicData.author == nodeId then
7                  return
8              end
9              if GetParents(topicId) ≠ null then
10                 return
11             end
12         else
13             if topicData.author == nodeId then
14                 return
15             end
16         end
17         peer ← ClosestLocalPeer(topicData.author);
18         AddToParents(topicData.id, peer);
19         SendRPC(topicData.id, peer);
20     end
```

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 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, whether 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

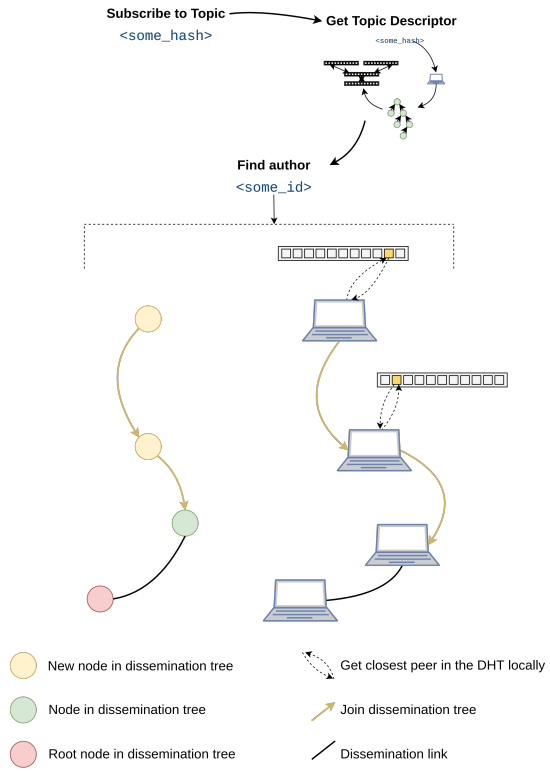

Figure 3. Overview of the flow for creating a new subscription

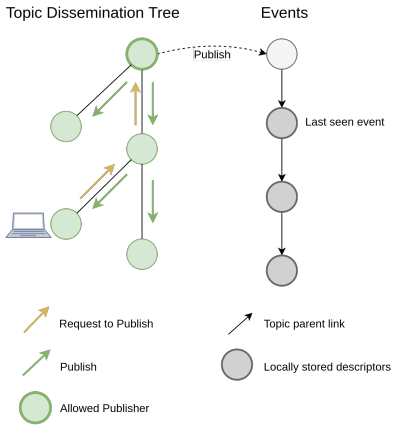

Figure 4. Event dissemination mechanism for a topic with only the author allowed to publish, last seen event linking and request to publish allowed. This scenario provides order guarantee.

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 = sender node id
       Input: eventData = event descriptor
2      begin
3          topicData ← TopicData(eventData.topicId);
4          if AllowedToPublish(nodeId, topicData)
            then
5              SendEvent(fromNodeId, eventData);
6          else
7              if
                AllowedToRequestToPublish(nodeId, topicData
                then
8                  SendRequestToPublish(eventData);
9              end
10         end
11     end
```

We will now highlight some of the properties these 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

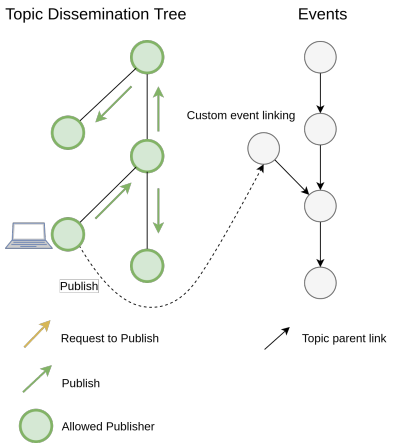

Figure 5. Event dissemination mechanism for a topic with custom event linking and global publishers allowed

Algorithm 4: Event forwarding function

```
1  Function SendEvent(eventData)
       Data: nodeId = node id of this node
       Input: fromNodeId = sender node id
       Input: eventData = event descriptor
2      begin
3          topicData ← TopicData(eventData.topicId);
4          if IsNewEvent(eventData) then
5              linkedEvent ← LinkEvent(eventData);
6              StoreInDHT(linkedEvent);
7          end
8          if IsSubscribed(eventData.topicId) then
9              EmitEvent(eventData.topicId, eventData);
10         end
11         for
            peer ← Children(eventData.topicId) AND
            peer ← Parents(eventData.topicId) do
12             if fromNodeId ≠ peer then
13                 SendRPC(eventData, peer);
14             end
15         end
16     end
```

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.

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. 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.

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. It is worth noting that, much like the work we built on top of, this module is open source [17].

Figure 6 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 [18] which we use in Pulsarcast. Besides the main libp2p module we also use some other utility modules such as the CID module [19] and the Peer-id module [20], both designed to reason with content identifiers (peer identifiers are also content identifiers).

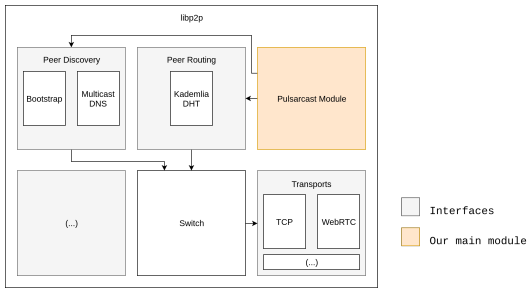

Figure 6. Our Pulsarcast module in the libp2p ecosystem

Pulsarcast has five classes. These are:

- Pulsarcast - Our main class, responsible for holding the state of our node, managing peer lists and connections. It extends the built-in `EventEmitter` class [21], making our class capable of reproducing the event emitter pattern [22], the mechanism we use to bubble up new Pulsarcast events to the application level.
- Peer - An abstraction for the state of each peer, including connections, identifiers, adresses and dissemination trees.
- TopicNode - Abstracts the Topic descriptor for our Merkle DAG.
- EventNode - Just as above, abstracts the Event descriptor for our Merkle DAG.
- EventTree - Manages the state of the event tree for each topic.

Listing 3 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 [23] 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 3. Usage example of our Pulsarcast module

5 Evaluation

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.

To achieve this we relied on containers, specifically Docker [24] containers, which we used to create a containerised version of our module. To orchestrate our containerised application we used Kubernetes [25], an open source orchestration platform based on Google's learnings on running containerised workloads at scale [26], and one of the most popular solutions in the field.

[17] https://github.com/JGAntunes/js-pulsarcast
[18] https://github.com/libp2p/js-libp2
[19] https://github.com/multiformats/js-cid
[20] https://github.com/libp2p/js-peer-id
[21] https://nodejs.org/api/events.html
[22] https://nodejs.dev/the-nodejs-event-emitter

[23] The full documented API for our module - https://github.com/JGAntunes/js-pulsarcast/blob/547ff33527f0df8c751d5fcd73d559fce59cdb77/docs/api.md
[24] https://www.docker.com/products/container-runtime
[25] https://kubernetes.io/
[26] https://research.google/pubs/pub43438/

To aggregate, correlate and analyse metrics and logs we used Elasticsearch [27], Beats [28], Logstash [29] and Kibana [30]. In order to simulate abnormal network conditions we relied on Toxiproxy [31], a TCP proxy that, programatically through an HTTP API, allowed us to inject multiple kinds of faults.

As we know it, Pulsarcast is just a module that applications can use to build on top of. In order to test it we created a fork of JS IPFS [32] where we integrated Pulsarcast. This not only provided us with a command line interfaace (CLI) and an HTTP API to interact with our system, but it also gave us direct access to IPFS' own pub-sub module, Floodsub, to which we wanted to compare our module.

Figure 7 provides an architectural overview of our system. All of the projects and code we created for the testbed are open source [33] [34] [35].

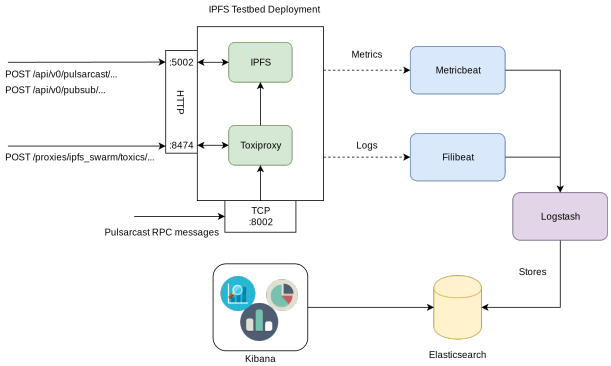

Figure 7. Overview of our ipfs-testbed deployment and our metrics/logs pipeline

Our whole setup consisted of a total of 5 VMs [36] acting as Kubernetes Worker nodes, each with two 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 aboe). Because we wanted to avoid resource starvation and to better take advantage of the Kubernetes scheduler, our testbed deployments allocate 440 MiB of memory 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.

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 [37] comments from 2007 [38] [39] consisting of a sample of approximately 25000 comments in a total of 23 topics (known as subreddits in the platform) [40].

The following graphs give us a distribution analysis of events published per topic (Figure 8) and subscriptions per topic (Figure 9). 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 their popularity.

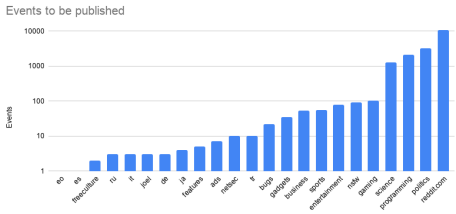

Figure 8. Event distribution per topic with log scale

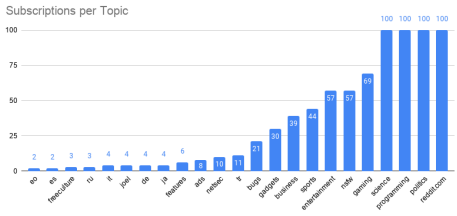

Figure 9. Subscription distribution per topic

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 total number of events injected in the system
  - Number of RPC messages sent per topic and in total

[27] https://www.elastic.co/products/elasticsearch
[28] https://www.elastic.co/products/beats
[29] https://www.elastic.co/products/logstash
[30] https://www.elastic.co/products/kibana
[31] http://toxiproxy.io
[32] https://github.com/jgantunes/js-ipfs
[33] https://github.com/JGAntunes/helm-charts/tree/master/ipfs-testbed
[34] https://github.com/JGAntunes/ipfs-testbed
[35] https://github.com/JGAntunes/ipfs-testbed-cli
[36] Special thanks to Microsoft and the Azure team for supporting our efforts and offering us free credits
[37] https://www.reddit.com/
[38] http://academictorrents.com/details/7690f71ea949b868080401c749e878f98de34d3d
[39] https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/
[40] https://github.com/JGAntunes/pulsarcast-test-harness

  - Average, standard deviation and percentiles (99/95) of the number of RPC messages received and sent by each node

We measure the subscription coverage (number of subscriptions fulfilled) through two distinct metrics. The percentage of fulfillment having the number of events effectively published as a reference and the percentage of fulfillment having the total number of events injected into the system as reference. Given Pulsarcast's nature, when an event is injected into the system, depending on the topic configuration, it may need to be propagated through the dissemination trees before being effectively published (*request to publish*). It also needs to be persisted in the DHT. Having two different metrics allows us to better analyse and distinguish the different behaviours of the system.

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.

We ran 3 different scenarios under 2 different sets of network conditions to an effective total of 6 different executions. For each of the 3 different scenarios we executed one under normal/undisturbed network conditions and another using Toxiproxy's features, adding a latency of 500 milliseconds and 300 milliseconds of jitter to every incoming TCP packet. The scenarios we ran were the following:

- Pulsarcast without order guarantee (basic usage, every node can publish to any topic)
- Floodsub (IPFS' pub-sub implementation)
- Pulsarcast with order guarantees (only one node per topic is allowed to publish and all nodes can request to publish)

For Pulsarcast without order guarantee our fultilment results were the following:

- Under normal network conditions
  - 99% of subscription coverage, having all the events injected into the system as reference
  - 99% of subscription coverage, having the events effectively published as reference
- Under abnormal network conditions
  - 51% of subscription coverage, having all the events injected into the system as reference
  - 86% of subscription coverage, having the events effectively published as reference

Figures 10, 11 and 12 show us a comparison of event fulfilment rates across topics. A couple of factors contributed to the discrepancy between the values of the first and the second executions. The first one is an implementation detail in our Javascript Pulsarcast module, where on each command it receives, it waits on the initial DHT persistence and propagation of events/topics before returning control to the caller (and consequently replying to the clients of our HTTP API). This creates a natural back pressuring system that ended up dragging the second execution for 13h versus the 85 minutes that took for the first execution. The long execution ended up putting a lot of pressure in our testbed which terminated two Pulsarcast nodes due to resource limitations. Nevertheless, our fulfilment rates are almost perfect for the first execution and still considerably high under abnormal network conditions. As for resource usage our first execution had a maximum memory consumption of 31.924 GiB across the cluster, with an average of 0.319 GiB per node and a P99 of 0.378 GiB. For the second execution our memory footprint was fairly higher (responsible for the node terminations), with a total consumption of 35.8 GiB, average of 0.36 per node and a P99 of 0.43 GiB. As expected, given our systems were mostly idle, CPU usage was much lower in the second execution, between 3.5 and 3.8 vCPUs in the first test run and 0.6 and 1.25 in the second one.

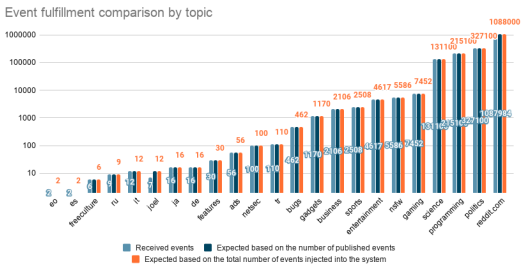

Figure 10. Pulsarcast without order guarantee - Comparison of events fulfilled by topic in a log scale

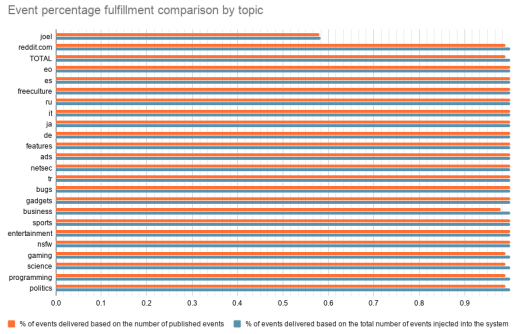

Figure 11. Pulsarcast without order guarantee - Comparison of percentage of events fulfilled by topic

For Pulsarcast with order guarantee our fultilment results were the following:

- Under normal network conditions
  - 37% of subscription coverage, having all the events injected into the system as reference
  - 80% of subscription coverage, having the events effectively published as reference
- Under abnormal network conditions

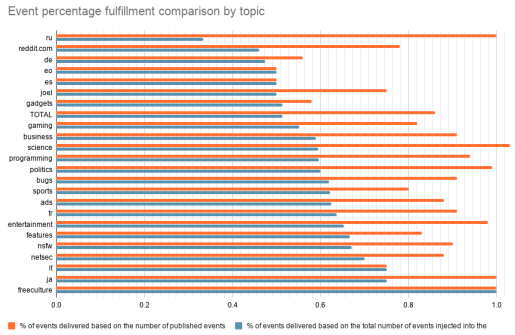

Figure 12. Pulsarcast without order guarantee and latency - Comparison of percentage of events fulfilled by topic

- 32% of subscription coverage, having all the events injected into the system as reference
- 62% of subscription coverage, having the events effectively published as reference

Figures 13, 14 and 15 show us a comparison of event fulfilment rates across topics. Both executions saw 2 nodes being terminated due to limitations on the resources available to the testbed, specifically CPU. These executions put a lot of stress into the root nodes of the most popular topics. Consequently, this affected our fulfilment rates. However, we are aiming at a total different level of QoS in this scenario, so it is important to put these numbers into perspective, as it is unfair to compare them directly with any of the other results in this document. It is interesting to see though that, despite the added network latency in the second execution, our QoS did not suffer a clear impact. As for resource usage our first execution had a maximum memory consumption of 17.84 GiB across the cluster, with an average of 0.178 GiB per node and a P99 of 0.207 GiB. For the second execution our memory footprint was quite similar, with a total consumption of 19.99 GiB, average of 0.2 per node and a P99 of 0.35 GiB. As expected, given our systems were mostly idle, CPU usage was much lower in the second execution, between 3 and 4.5 vCPUs in the first test run and 0.93 and 4.33 in the second one.

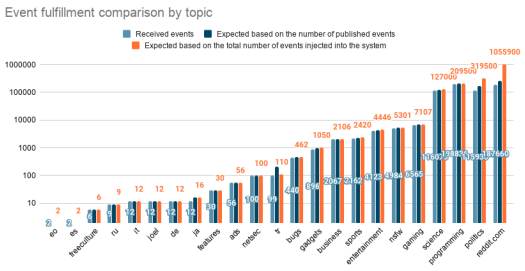

Figure 13. Pulsarcast with order guarantee - Comparison of events fulfilled by topic in a log scale

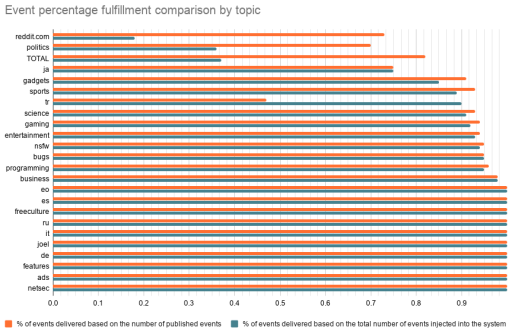

Figure 14. Pulsarcast with order guarantee - Comparison of percentage of events fulfilled by topic

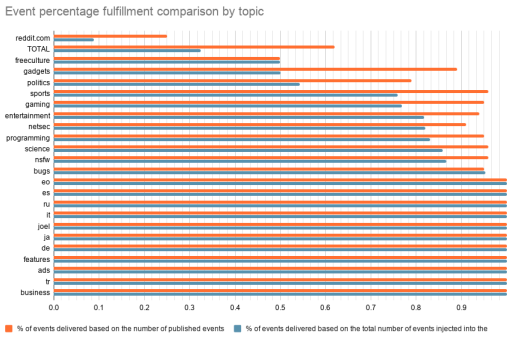

Figure 15. Pulsarcast with order guarantee and latency - Comparison of percentage of events fulfilled by topic

It is important to highlight that, for all of the Pulsarcast executions we have described so far, network and memory usage across the cluster always grew linearly with the number of events received. Same for the RPC messages sent and received, as we can see from Figure 16

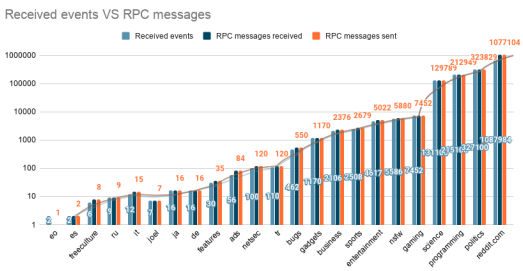

Figure 16. Pulsarcast without order guarantee - Comparison of events received and RPC injected in the system

Finally our Floodsub scenario executions gave us the following results [41]:

[41] For Floodsub there is no distinction between an event injected into the system and an event published, given there is no acknowledgement for it

- 41% of subscription coverage under normal network conditions
- 31% of subscription coverage, under abnormal network conditions

Figures 17 and 18 show us a comparison of event fulfilment rates across topics. For both experiments, the network as a whole was unable to cope with the load of our execution and shortly after starting, some nodes became unresponsive and were terminated, going as low as only 15 node running at a given time. It took about 50 minutes for the network to fully recover to 100 nodes again. This is a clear indicator of Floodsub's inability to handle the same workload Pulsarcast did in the previous scenarios. Which is expected, as the way Floodsub operates is by forwarding messages to all of its peers, creating a huge strain in the network (both CPU wise and in network data transmission). In terms of resource usage, memory consumption has been lower than the Pulsarcast experiments,hitting a maximum of 21.15 GiB for the first experiment and 15.95 for the second one. However, CPU has been way higher, picking at 5.53 vCPUs across the cluster. Network wise, Floodsub experiments have transmitted much more data for a lower QoS level, specifically 6552 MiB and 6474 MiB for both experiments across the cluster, three times as much for the respective Pulsarcast experiments.

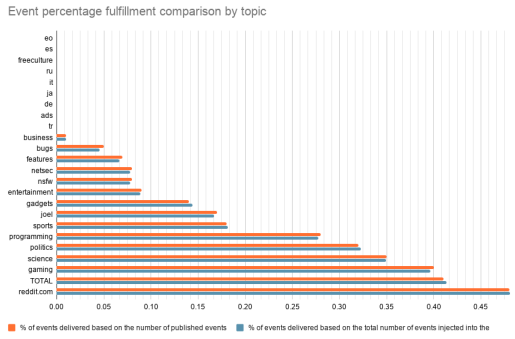

Figure 17. Floodsub - Comparison of percentage of events fulfilled by topic

One aspect that we would like to highlight is the fact that we are not only compressing months of data into a largely shorter timepsan, we are also simulating interactions made by thousands of users into a much smaller set of nodes (one hundred). All of this of course in an environment based on virtualisation techniques and with a limited set of resources. This experiment is essentially pushing the boundaries of what both systems would handle on a real world scenario. Equally important to note is that, for Pulsarcast, every event effectively published, is stored in the DHT. So, it is possible for any application using Pulsarcast to resolve past or missing events from the event tree. This is the cornerstone of Pulsarcast's eventual delivery guarantees, hence why it

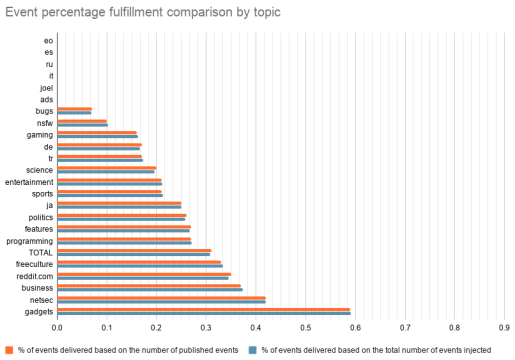

Figure 18. Floodsub with latency - Comparison of percentage of events fulfilled by topic

is essential to look at the percentage of events effectively published as well as the subscription coverage for these same events. Taking those same numbers into consideration we can see a considerably high coverage percentage, with the lowest being 62% for the order guarantee test with latency injected.

6 Conclusion

In this work, we introduced Pulsarcast, a decentralised, topic-based, pub-sub solution that seeks to bring reliability and eventual delivery guarantees (commonly associated with centralised solutions) to the P2P realm. We analysed how Pulsarcast provides a feature rich API on top of a system that leverages a Kadmelia structured overlay to build immutable and content-addressable data structures (Merkle DAG) representing both topics and events. These structures power Pulsarcast's eventual delivery guarantees.

We observed that Pulsarcast surpassed IPS's current implementation (Floodsub) in every aspect, providing a better QoS with a smaller resource footprint. The only exception being the order guarantee scenarios, however we are looking at total different levels of QoS. Resource wise, Floodsub is far more network-intensive than Pulsarcast (with six times more usage in some cases) and generally requires more CPU power. It is also essential to consider Pulsarcast's high publish rates, given that for each event published we store it in the DHT. This is the cornerstone of its eventual delivery guarantees, giving applications the ability to fetch missing events from their event tree.

We concluded that our system provides a good alternative to applications that seek a better QoS level as well as a feature-rich topology setting, that allows to restrict publishers and configure topics to one's needs. Despite being heavily reliant on a structured overlay, Pulsarcast did not underperform under adverse network conditions, making it suitable for multiple scenarios.

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] Nuno Apolonia, Stefanos Antaris, Sarunas Girdzijauskas, George Pallis, and Marios Dikaiakos. SELECT: A distributed publish/subscribe notification system for online social networks. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*, pages 970–979, 2018.

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] João Paulo De Araujo, Luciana Arantes, Elias P. Duarte, Luiz A. Rodrigues, and Pierre Sens. A Publish/Subscribe System Using Causal Broadcast over Dynamically Built Spanning Trees. *Proceedings - 29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017*, pages 161–168, 2017.

[9] David Dias and Luís Veiga. BrowserCloud.js: A distributed computing fabric powered by a P2P overlay network on top of the web platform. *Proceedings of the ACM Symposium on Applied Computing*, pages 2175–2184, 2018.

[10] Patrick Eugster, Rachid Guerraoui, Joe Sventek, and Agilent Laboratories Scotland. Type-Based Publish/Subscribe. Technical report, Swiss Federal Institute of Technology, Lausanne, 2000.

[11] 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.

[12] 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.

[13] Anne-Marie Kermarrec and Peter Triantafillou. XL peer-to-peer pub/-sub systems. *ACM Computing Surveys*, 46(2):1–45, 2013.

[14] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. pages 53–65. 2002.

[15] 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.

[16] 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.

[17] 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.

[18] Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimåker. The hidden pub/sub of spotify. In *Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS '13*, page 231, New York, New York, USA, 2013. ACM Press.

[19] 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.

[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] Huanyang Zheng and Jie Wu. NSFA: Nested Scale-Free Architecture for scalable publish/subscribe over P2P networks. *Proceedings - International Conference on Network Protocols, ICNP*, 2016-Decem:1–10, 2016.

[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.

[27] Nejc Zupan, Kaiwen Zhang, and Hans Arno Jacobsen. Demo: HyperPubSub: a decentralized, permissioned, publish/subscribe service using blockchains. *Middleware 2017 - Proceedings of the 2017 Middleware Posters and Demos 2017: Proceedings of the Posters and Demos Session of the 18th International Middleware Conference*, pages 15–16, 2017.