Pulsarcast: Scalable, Reliable Pub-Sub over P2P Nets

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

David Dias
Protocol Labs
david@protocol.ai

Luís Veiga
INESC-ID / Técnico Lisboa, ULisboa
luis.veiga@inesc-id.pt

***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.

I. 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. 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.

Many 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 aims at guaranteeing low latency and high event throughput. Other examples are the multiple message queue systems like Apache Active MQ, RabbitMQ, Redis, etc. Most of these solutions are centralised and as such, harder to scale to a large number of clients when compared with Peer to Peer (P2P) solutions.

[1]https://www.infoq.com/presentations/Twitter-Timeline-Scalability
[2]http://kafka.apache.org/documentation/#design

ISBN 978-3-903176-15-7

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. In order to validate the solution we propose, 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 II presents and analyses our related work. Section III introduces and describes Pulsarcast, its architecture, data structures, algorithms as well as the implementation of our solution, with a more thorough overview of our Javascript module. Next, Section IV explains our evaluation methodology. Finally, Section V provides a set of closing remarks.

II. 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. 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. 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, 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** [3]. Javascript is a lightweight, interpreted, programming language, known as the scripting language for the web. Initially created to allow simple interactions and animations in web pages it is now one of the main programming languages for the web, with runtimes in browsers and servers thanks to projects such as **Node.js**.

In the application realm, many P2P apps have leveraged these technologies. **browserCloud.js**[9] is such an example, a solution seeking to bring cloud computing to the Web platform, taking advantage of technologies such as WebRTC [4] and Javascript. IPFS is another example, a P2P hypermedia protocol designed to create a persistent, content-addressable network on top of the distributed web, with a **Merkle DAG** at its core. 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.

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**, 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.

III. Pulsarcast

Pulsarcast is a peer to peer, pub-sub, topic-based system focused on reliability, eventual delivery guarantees, and data persistence.

We opted for the more straightforward topic-based subscription model given that, in our view, a well structured and implemented topic-based model is enough for the most common use cases.

Pulsarcast is a fully decentralised solution, meaning 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.

Pulsarcast relies on two different types of overlays to fulfil its needs. Kademlia DHT, used for peer discovery, content discovery and to bootstrap our other overlay, our per-topic dissemination trees. Every topic and event is stored in the Kademlia DHT before being forwarded through the topic dissemination trees. This ensures data persistence at a set of nodes (that might even be extraneous to the topic at hand) and anyone is later able to fetch

[3]https://www.ecma-international.org/publications/standards/Ecma-262.htm
[4]https://www.w3.org/TR/webrtc/

the data using only the DHT. Once persistent, we forward the data through the appropriate dissemination trees. 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.

A. Data Structures

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 and a previous version of themselves. Figure 1 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. 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.

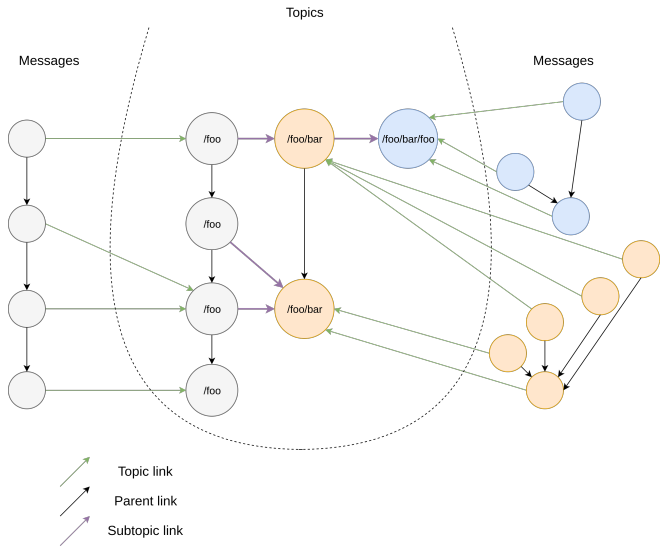

Fig. 1. Representation of the Pulsarcast DAG

Given we are discussing addressability and linking between content, the representation used for our identifiers is an essential part of our system specification. We borrowed inspiration from IPFS and decided to use *CIDs* (Content Identifiers) [5]. A CID is a self-describing content-addressed identifier. 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. 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

[5]https://github.com/multiformats/cid

```
{
  "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

received an event can, through its parent link, know a previous event of this same topic.

The parent links in the topic descriptor act as a reference to a previous version of this same topic. Keep in mind that data in Pulsarcast is immutable. As such, one cannot update content that has already been published and disseminated. We can, however, create a new reference of it and link to what we consider to be a previous version. Possible changes to the topic descriptor include changing the topic metadata for example or addition of new sub-topics.

In topic descriptors, sub-topic links are indexed under a # key (commonly indexed by name but not mandatory). One important 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. 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.

B. Subscription Management

Before we can speak about a new subscription, a topic must already exist. 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 DHT does the node proceed to create the actual

topic descriptor (with the meta topic linked), which is then also persisted in the DHT. When any change to the topic happens, 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.

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

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
```

The topic creator acts 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, we use the DHT to find (locally, within its K buckets) the closest known peer to the author of the topic. 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. The process is then repeated. 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 general procedure to be used at every node when receiving or sending a subscription request. 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.

C. Event Dissemination

Pulsarcast allows for some additional customisation and configuration at the topic level that will dictate how events are disseminated through it. This brings 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.

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 Kademlia DHT. The node then checks if it is allowed to publish through the topic configuration whitelist mechanism. This option, *allowedPublishers*, can be enabled and, if so, a list of nodes is provided that is checked before publishing. Alternatively, 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. 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 publishes the content) and the author (node responsible for creating the content in the first place).

Upon receiving a publish event request 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 DHT, followed by disseminating it through its topic dissemination tree. From this point forward, nodes along the dissemination tree will forward the event across its branches. All the event dissemination logic is better detailed in the Algorithms 3 and 4.

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

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
```

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 allowed. 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 producers that every node is aware of. Finally, another scenario is an example of a topic where custom event linking is allowed and applications rely on it to create links that imply event causality (such as a chat with multiple rooms and threads).

D. Implementation Details

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 its utility modules and working implementation of a Kademlia 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 [6].

Figure 2 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 which we use in Pulsarcast.

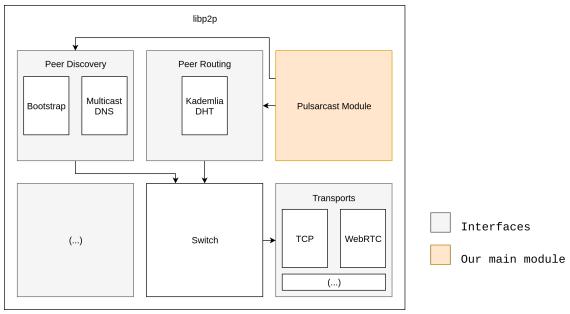

Fig. 2. Our Pulsarcast module in the libp2p ecosystem

IV. Evaluation Methodology

As part of our implementation, we built a testbed [7]. It relied on Docker containers, Kubernetes, an Elastic stack to collect results and metrics and Toxiproxy, a TCP proxy that allowed to simulate abnormal network conditions.

Our test setup consisted of 5 VMs [8] acting as Kubernetes Worker nodes, each with two vCPUs, 16 GiB of RAM and 32 GiB of storage. In them, we ran a total of 100 IPFS Testbed deployments containing our own Pulsarcast module.

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 comments from 2007 [9] consisting of a sample of approximately 25000 comments in a total of 23 topics (known as subreddits in the platform). 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.

For each execution, we wanted 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) for CPU, Memory (GiB) and Network (MiB transmitted)
- QoS

[6] https://github.com/JGAntunes/js-pulsarcast

[7] https://github.com/JGAntunes/ipfs-testbed

[8] Special thanks to Microsoft and the Azure team for supporting our efforts and offering us free credits

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

- Events published/received 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
- 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: i) the percentage of fulfilment having the number of events effectively published as a reference; ii) the percentage of fulfilment 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 the different behaviours of the system.

A preliminary assessment of our results demonstrated a subscription coverage as high as 99% for Pulsarcast under normal network conditions. When 500 milliseconds of latency and 300 milliseconds of jitter were introduced to every incoming TCP packet, our results dropped to an upper bound limit of 86% . In comparison, under those same conditions, Floodsub had a subscription coverage of 41% and 31% respectively.

V. Conclusion

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 Kademlia 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. Our initial assessment results are encouraging: Pulsarcast compares most favourably with IPFS's current implementation (Floodsub), providing a better QoS with a smaller resource footprint.

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.

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] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. TERA. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems - DEBS '07*, page 2, New York, New York, USA, 2007. ACM Press.

[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. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429, pages 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[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. In Rachid Guerraoui, editor, *Middleware 2001*, number November 2001, pages 329–350. Springer Berlin Heidelberg, Berlin, Heidelberg, 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 Riviere, Anne-marie Kermarrec, Maarten Van Steen, Others, 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, INRIA, 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: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, number June in NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. Association for Computing Machinery.

[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.