Pulsarcast - Scalable and reliable pub-sub over P2P networks

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

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

David Dias
INESC-ID Lisboa
ULisboa / Instituto Superior Técnico
mail@daviddias.me

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 over the course of the century [2] [1]. This is mainly due to its special properties, that allow for full decoupling of all the communicating parties. Taking a closer look at this definition one can see that this comes hand in hand with the way information is consumed nowadays, with the exponential growth of social networks like Twitter and the usage of feeds such as RSS.

In the publish-subscribe communication paradigm, subscribers (or consumers) sign up for events, or classes of events, from publishers (or producers) that are subsequently asynchronously delivered. This decoupling can be broken into three different parts:

- Decoupling in time - publishers and subscribers do not need to be interacting with each other at the same time;
- Decoupling in space - both parties do not need to need to know each other or be in the same space in order to communicate;
- Synchronisation - publishers and subscribers do not need to be blocked by each other in order to produce or consume content:

Due to the properties described above, a lot of applications rely on the publish-subscribe paradigm and a lot of work has been done by companies like Twitter [1], Spotify [3] 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 a living proof of the high scalability P2P can offer, with pub-sub systems over P2P networks being an active research topic with a lot of attention.

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.

[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/

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

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

Besides the specification and architectural model of our system we also provide a concrete implementation of it. So, 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 **??** explains our evaluation methodology and presents those results. Finally, Section 5 provides a set of closing remarks and a set of improvements and future work.

2 Related Work

3 Pulsarcast

Pulsarcast is a peer to peer, pub-sub, topic-based system focused on reliability, eventual delivery guarantees, and data persistence. Properties that 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. Figure 1 illustrates the multiple overlays in action.

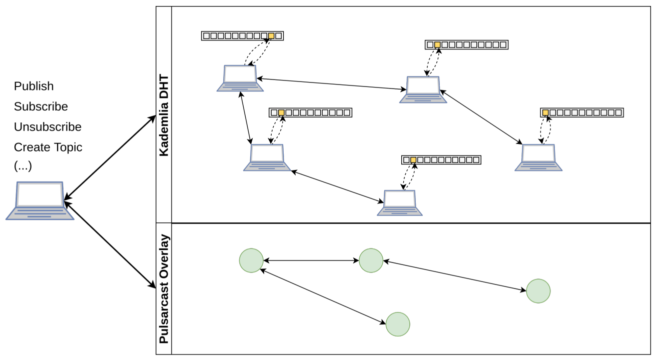

Figure 1. Representation of the Pulsarcast overlays

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. Figures 2 and 3 provide an overview of how the flows for creating this information and for accessing it look like.

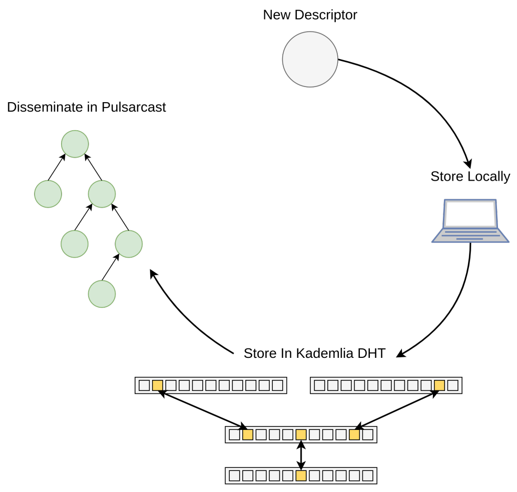

Figure 2. Flow for creating a new Topic/Event descriptor

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

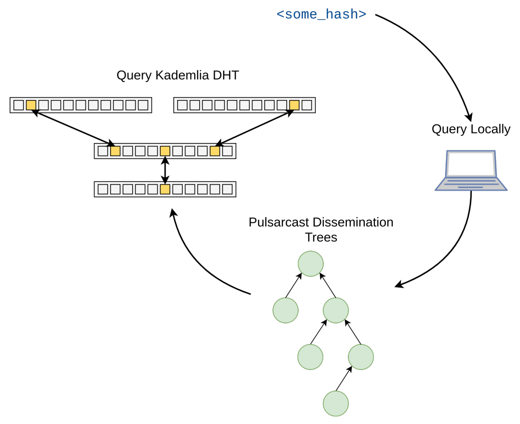

Figure 3. Flow for querying a Topic/Event descriptor

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 4 provides a broader picture of how it all fits together. Immutability and content-addressability give us verifiability. Consequently, the assurance that the state of our distributed system is the same no matter where we are accessing it from or who is viewing it. It also allows us to build a notion of history which plays nicely into a pub-sub scenario. Through these links and the mechanisms described 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.

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)[7]. A CID is a self-describing content-addressed identifier. It uses cryptographic hashes to achieve content addressing and is powered by *multihash*[8]. 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.

[7]https://github.com/multiformats/cid
[8]https://github.com/multiformats/multihash

```
{
  "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

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

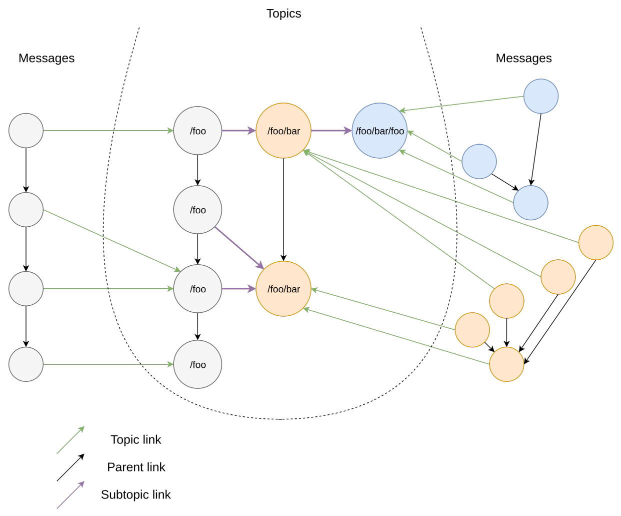

Figure 4. Representation of the Pulsarcast DAG

```
{
  "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

(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 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.

Algorithm 1: Create a new topic

```
1  Function CreateTopic(newTopic)
       Input: newTopic = data for new topic creation
2      begin
3          $parent \leftarrow newTopic.parent$;
4          if $parent == null$ then
5              $metaTopic \leftarrow CreateMetaTopic(newTopic)$;
6          else
7              $metaTopic \leftarrow parent.subTopics.meta$;
8          end
9          $topicData \leftarrow CreateTopic(newTopic, metaTopic)$;
10         $Subscribe(metaTopic)$;
11         $Subscribe(topicData)$;
12         $StoreInDHT(metaTopic)$;
13         $StoreInDHT(topicData)$;
14         $Publish(metaTopic, topicData)$
15     end
```

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

Algorithm 2: Join request handler for each node

```
1  Function ReceivedJoin(fromNodeId, topicId)
       Data: $nodeId$ = node id of this node
       Input: $topicId$ = topic id
       Input: $fromNodeId$ = sender node id
2      begin
3          $topicData \leftarrow GetTopicData(topicId)$;
4          if $fromNodeId \neq nodeId$ then
5              $AddToChildren(t, fromNodeId)$;
6              if $topicData.author == nodeId$ then
7                  return
8              end
9              if $GetParents(topicId) \neq null$ then
10                 return
11             end
12         else
13             if $topicData.author == nodeId$ then
14                 return
15             end
16         end
17         $peer \leftarrow ClosestLocalPeer(topicData.author)$;
18         $AddToParents(topicData.id, peer)$;
19         $SendRPC(topicData.id, peer)$;
20     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 6 and 7 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 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

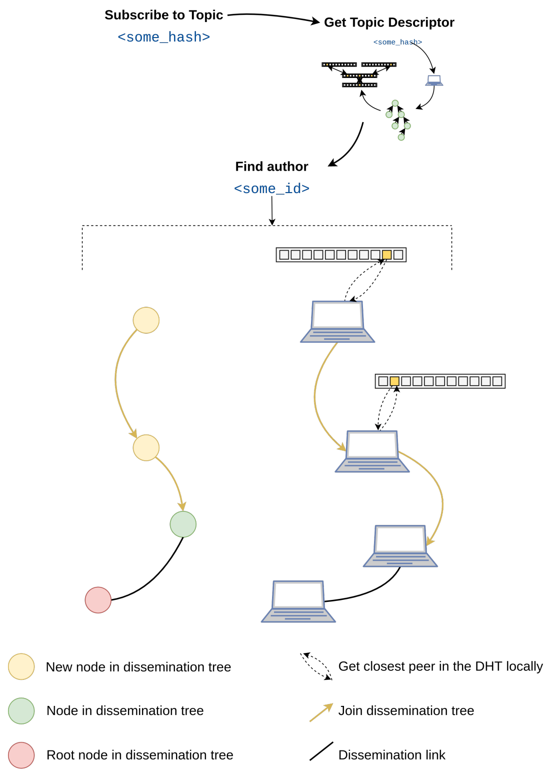
Subscribe to Topic
<some_hash>
Get Topic Descriptor
<some_hash>
Find author
<some_id>
New node in dissemination tree
Node in dissemination tree
Root node in dissemination tree
Get closest peer in the DHT locally
Join dissemination tree
Dissemination link

Figure 5. Overview of the flow for creating a new subscription

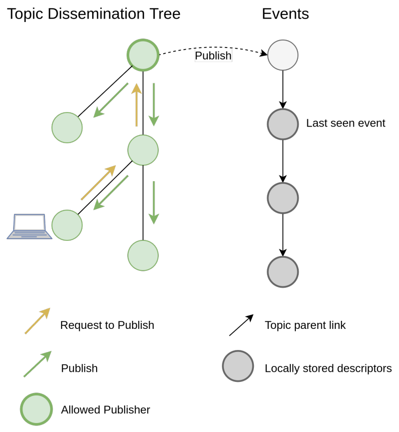

Figure 6. Event dissemination mechanism for a topic with only the author allowed to publish, last seen event linking and request to publish allowed. This scenario provides order guarantee.

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 gone through. All of the logic we have covered around event dissemination is better detailed in the Algorithms 3 and 4.

It is essential to understand some of the properties that these multiple configuration options allow. The simplest

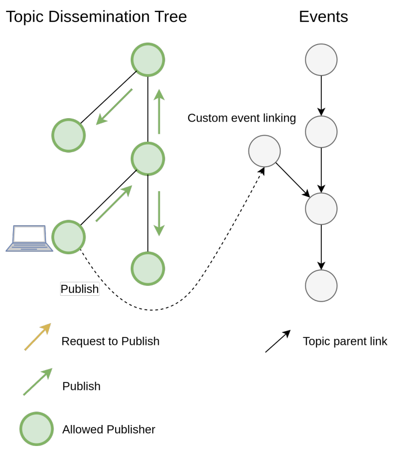

Figure 7. Event dissemination mechanism for a topic with custom event linking and global publishers allowed

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
```

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

4 Implementation

5 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] 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.

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

[3] 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.