SmartPubSub@IPFS

Pedro Eduardo Reis Agostinho

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Eng. David Miguel dos Santos Dias

November 2021

Acknowledgments

I would like to thanks both my thesis supervisors Luís Veiga and David Dias, for their support. I appreciate the meetings with Professor Veiga at the beginning of this project, where he provided me with his useful guidance and time to discuss ideas for the project's architecture.

I would like to give a very special thanks to all members of the IPFS community that assisted me through GitHub issues, online office hours, and slack chats during this project.

I would also like to thank all my professors and colleagues that accompanied me during my journey at IST. In special my course colleagues, that during the lockdown phase spared some time to share their own experiences in their projects and always motivated me.

I want to finish by thanking all those who were part of my life, especially two important people in my life. I must thank my parents, that through the years, showed me the best they possibly could, how to live a meaningful life, and gave me several chances, even when I felt undeserving, to progress my studies. This simple project is thanks to you, mom, and dad.

Abstract

The InterPlanetary File System (IPFS) is a hypermedia distribution protocol addressed by content and identities. IPFS enables the creation of completely distributed applications. One of the most efficient and effective ways to distribute information is through the use of notifications or other methods, which involve a producer of content (publisher) that shares content with other interested parts (subscribers).

Currently, IPFS has some working implementation of topic-based pub-sub systems under an experimental flag. The goal of this work is to develop a content-based pub-sub system (with subscriptions based on predicates about event content) to disseminate information on top of IPFS in an efficient and decentralized way, by exploring its current infrastructure.

In this work, we present a survey with the state-of-the-art in the Publish-Subscribe and Peer-to-Peer areas and present, develop, and evaluate a content-based pub-sub system to work over IPFS.

We design two protocols: ScoutSubs protocol that is completely decentralized; FastDelivery protocol that is centered in the publisher. With these two approaches, we pretend to show the different advantages of having each of these protocols simultaneously by comparing ScoutSubs' complete decentralization and FastDelivery's centralization at the data source.

Keywords

IPFS, Publish-Subscribe, content-based, p2p

Resumo

O Interplanetary File System (IPFS) é um protocol de distribuição de dados na Web, que endereça os dados pelo seu conteúdo e tipo . O IPFS permite a criação de aplicações distribuidas. Uma das maneiras mais efetivas de distribuir informação é através de notificações ou outros métodos, os quais envolvem o produtor do conteúdos que partilha este mesmo conteúdo com outras partes interessadas (subscritores).

Atualmente, o IPFS tem algumas implementações de *topic-based pub-subs* em fase experimental. O objetivo deste trabalho é desenvolver um *content-based pub-sub* (com subscrições baseadas em predicatos descritivos do conteúdo dos eventos) para disseminar informação sobre o IPFS de forma eficiente e descentralizada, explorando a sua atual infrastrutura.

Neste trabalho apresentamos a nossa pesquisa do estado da arte nas áreas de publish-subscribe e peer-to-peer e apresentamos, desenvolvemos, e avaliamos o nosso *content-based pub-sub* que trabalhará sobre o IPFS.

Nós desenvolvemos dois protocolos: protocolo ScoutSubs que é completamente descentralizado; protocolo FastDelivery que se centra no produtor de informação. Com estas duas abordagens, pretendemos mostrar as diferentes vantagens de cada um destes em cenários diferentes comparando a descentralização total do ScoutSubs com a centralização na fonte dos dados do FastDelivery.

Palavras Chave

IPFS, Publish-Subscribe, content-based, p2p

Contents

List of Figures

List of Tables

Acronyms

IPFS	Interplanetary File System
p2p	Peer-to-Peer
CPU	Central Processing Unit
DHT	Distributed Hash Table
RPC	Remote Procedure Call
URL	Uniform Resource Locator
RAM	Random Access Memory
XML	eXtensible Markup Language
Id	Identifier
HTML	HypertText Markup Language
IoT	Internet of Things
IP	Internet Protocol
CCTV	Closed-Circuit TeleVision
CID	Content Identifier
DAG	Directed Acyclic Graph
IPLD	InterPlanetary Linked Data
UDP	User Datagram Protocol
API	Application Programming Interface
RN	Rendezvous Node
QoS	Quality of Service
VM	Virtual Machine
DDoS	Distributed Denial of Service

1 Introduction

Contents

Introduction

1.1 Motivation

At the start, the Web was used as an alternative medium for sharing information worldwide. What started as a sharing medium used by colleges and tech/government organizations started to be used by mainstream businesses, and new technologies such as social media began emerging.

Today's Web is managed by a few big players (Google, Amazon, Microsoft) that, throughout the years, added their mark to the Web and started incorporating smaller adversary companies to maintain their relevance. These major tech corporate giants make the current Web format highly centralized in a few data-centers own by them, making information easily censorable and allowing the use and control of their users' private data. Web decentralization is a hot topic of discussion in today's world, but there are currently several decentralized web applications operating. Two examples are the decentralized file repositories like torrent [1] and the growing number of cryptocurrencies.

The InterPlanetary File System [2] was born to help create a decentralized Web, where users play a central role in the distribution and storage of information without the direct intervention of big corporate organizations. We can draw a parallel between what Interplanetary File System (IPFS) is trying to achieve and what web broadcast services like YouTube and others have done to diversify the current media, taking the market's control from the mainstream media companies and placing it on the content producers.

Currently, IPFS is a file-sharing system operating over a peer-to-peer network, and it has, under an experimental flag, some topic-based pub-sub systems. This project's goal is to provide the best possible content-based pub-sub alternative to work over IPFS. To do so, we investigated relevant Publish-Subscribe and peer-to-peer content distribution systems.

Publish and Subscribe This research area was active in the last two decades (especially in the early 2000s), where several topics and content-based architectures such as Hermes [3] and Meghdoot [4] appeared [5] [6]. We take special interest in the role of pub-sub systems in network communication and its different variants to understand what solution is ideal to provide an efficient content-based system over IPFS.

Peer-to-Peer Peer-to-Peer (p2p) has been an active area of research for a long time, and it is still highly relevant today. This is due to the increase of worldwide devices connected to the internet with higher bandwidth links and this devices increasing CPU power and memory capacity. Inside the peer-to-peer systems, we will concentrate on those systems that provide infrastructures that allow content distribution among peers [7]. Some systems in this category are Kademlia [8] and BitTorrent. We will try to analyze these content distribution p2p infrastructures, to better understand their functioning and how p2p file sharing systems, such as IPFS, work to provide insights into how to implement our solution.

This project can be considered a ”marriage” between these two main research areas as they complement themselves. On one side, we have Publish-Subscribe systems implement a communication paradigm that allows a total decoupling between the event source and the interested parties of that event. On the other, we have peer-to-peer systems, which are the most scalable and fault-tolerant networks. We may conclude that a pub-sub system over p2p networks would allow data dissemination over an environment like the internet, which requires a highly scalable and fault-tolerant system [6].

A smart pub-sub Currently, IPFS content-addressing system works with Kademlia's Distributed Hash Table (DHT) and the Bitswap protocol [9]. This mechanism is a static one meaning it needs a search effort to trigger a data object retrieval. Another important detail about this static content-addressing is that the content is organized based on its physical properties. By physical properties, we mainly mean its raw bits and data type, so to an IPFS user, a byte hashing carries no meaning to him when compared with a semantic expression like photo/surf/portugal.

So the goal of this work is to provide a dynamic semantic-addressing layer (meaning content-addressing a human can understand) where the information is routed through the network depending on the users' interests. A pub-sub system is what allows dynamic addressing, meaning users expresses to the network their interests, and information of interest to them upon production is forwarded towards them. Both layers can work together, for example, once an HTML page is published using our pub-sub with its semantic meaning (e.g. news about a topic), it may contain images that can then be fetched using the static layer. Figure 1.1 shows an example of a simple interaction between peers and IPFS using both mentioned layers (the image contains only the main steps). The Kademlia static layer is still preferable to work as decentralized file storage and be used by pub-sub to minimize the size of its events.

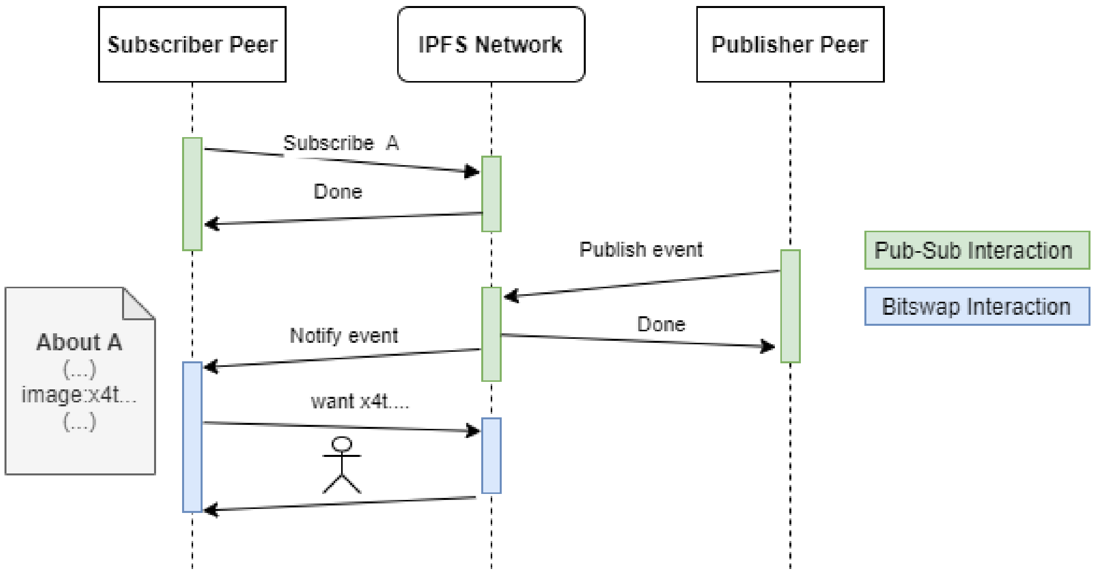


Figure 1.1: Peers interacting with the current static IPFS layer and a dynamic pub-sub layer

1.2 Current Shortcomings

When analyzing the state-of-the-art regarding content-based pub-sub, no relevant systems are running at the scale of the Web. One of the most relevant cases is Wormhole [10] from Facebook, but this one is not a completely decentralized approach.

The most relevant architectures of decentralized content-based pub-subs are mostly theoretical or not being used on large scale on the internet. In the IPFS community, some topic-based systems were created (e.g., Pulsarcast [11]), being GossipSub [12] the main one being tested/improved. The nonexistence of a content-based pub-sub in IPFS and the Web opens space for our work to build a content-based approach for it.

1.3 Contributions

This work comprises extensive research, development, and testing effort made to try to the best of our knowledge to produce an efficient and scalable content-based pub-sub. Some of this project's major contributions are:

- **Survey:** Firstly, in this document, readers can find in section 2 a survey both about pub-sub and p2p content distribution systems, resulting from the research that is the basis of this work.
- **A content-based pub-sub architecture:** We designed a content-based pub-sub architecture capable of being resistant to peer failure, scalable so it may perform steadily even in a network of millions of users.

- **A content-based pub-sub implementation:** We developed in golang a working content-based pub-sub system, that works over any bootstrapped IPFS-like network. Our code is open and available for everyone to try it and modify it [13].

- **A testbed:** We developed in golang several testing scenarios that can prove our pub-sub correctness and analyze its performance. Our test environment uses Testground [14] and is available for everyone who wants to make other types of scenarios or test it with a higher number of nodes [15].

1.4 Document Roadmap

This document is structured as follows: Chapter 2 presents a survey of the current state-of-the-art in pub-sub and p2p and detail some relevant systems in the context of this work. Chapter 3 describes our proposed architecture, and in Chapter 4 we describe the development choices made and explain our codebase structure and user interface. Chapter 5 explains how we planned and executed the test environment and shows and analyzes the obtained results. Chapter 6 finishes with a final summarizing thought of our work and where new researchers can focus their attention.

2

Related Work

Contents

Related Work

In this section, we will introduce the two main areas that cover this project, which are the Publish-Subscribe systems in Section 2.1 and Peer-to-Peer systems in Section 2.2. In Section 2.3, we present several fundamental systems for our project that are directly connected with one or both of the above main topics, and that were relevant for understanding the problem and propose the solution presented in Section 3.

2.1 Publish-Subscribe

In Section 2.1.1, we present Pub/Sub communication paradigm by comparing it with other alternative message paradigms to provide a clearer view of its properties and advantages [5]. After characterizing pub-sub alternatives, we will specify the communication paradigm in detail in Section 2.1.2 and its variations in Section 2.1.3. In Section 2.1.4 and 2.1.5, we analyze some implementation details and challenges of pub-sub systems.

2.1.1 A Communication paradigm

Before presenting the different communication paradigms, we will briefly define the actors in computer network information exchange. We will have an event producer, an event consumer, and an event manager representing the medium where the events are forwarded. The event itself is a particular piece of data exchanged between interested peers/users.

We will shortly describe each one and analyze them over three main properties: space, when consumers or producers know each other physical addresses (IP-address); time, when they need to be active in the transmission at the same time; synchronization coupling when one of these needs to be active at a certain time to participate in the information exchange process.

- **Message passing** is the ancestor and foundation stone of distributed communication. In this paradigm, participants send and receive messages from one another. Message passing participants are coupled both in space and time when exchanging messages, and while the sender

Communication paradigms	Space decoupling	Time decoupling	Synchronization decoupling
Message Passing	No	No	Producer-side
RPC	No	No	Yes
Notifications	No	No	Yes
Shared Spaces	Yes	Yes	Producer-side
Message Queuing	Yes	Yes	Producer-side
Pub-Sub	Yes	Yes	Yes

Table 2.1: Decoupling between producers and consumer using different communication paradigms

may be asynchronous, the listener needs to be synchronous, because it needs to active when the message reaches it.

- **Remote Procedural Calls** have been applied to object-oriented contexts in the form of remote method invocations in several systems (e.g. Java RMI), simplifying a lot distributed programming. Regarding coupling properties, Remote Procedure Call (RPC) is both space and time coupled at the client-side since consumers need a reference to the producer but might have synchronization decoupling if the client process is asynchronous.
- **Notification systems** get closer to pub-sub in a way that consumers subscribe to events from producers. This system can be achieved by two asynchronous calls: one with the subscription of interest to a certain event from a consumer, containing a callback for the producer to respond; another from a producer to a consumer, using the consumer's callback it kept, to send the event the consumer is interested in. With these two messages, this system manages synchronization decoupling but, because the relation producer-consumer is direct, both parties will be coupled both in space and time.
- **Shared Spaces** paradigm provides hosts in a distributed system with the view of a common shared space across disjoint address spaces, in which synchronization and communication between participants take place through operations on shared data. One popular type of shared space is the notion of tuple space. In tuple space, a producer creates a tuple in shared memory space. This tuple then has no necessary connection with its producer and the future consumer of the tuple, providing this way space and time decoupling. But there is still the need to synchronize consumers in a way that there cannot be more than one reading or removing a certain tuple.
- **Message Queuing** is similar to the shared space's paradigm but allows the additional transaction, timing, and ordering guarantees because messages are ordered in a queue. So, producers add messages to the queue, and consumers get messages from it synchronously. This way, the message queue achieves the same decoupling as shared space paradigms.

The Publish-Subscribe paradigm was born from the necessity of having complete decoupling, both in space, time, and synchronization (see Table 1), to separate the producers of certain events from its consumers. These properties make pub-sub the most flexible and easy to scale paradigm in theory.

This decoupling in pub-sub is achieved by delegating the responsibility of forwarding the events to an information bus, often distributed, that provides these decoupling properties.

2.1.2 Pub/Sub interaction schema

In publish-subscribe systems, event producers are called publishers and consumers subscribers. A publisher produces an event with particular characteristics and publishes it at the information bus.

A subscriber consumes events of certain characteristics of his/her interest and must formalize his/her interest by subscribing to that event at the information bus. The expression that details the subscriber's interest is a subscription predicate. A predicate is a group of attributes that describe one or more events.

So, in pub-sub we have three roles, each with the following possible actions:

- **Publisher**
  - **publish(event e)** - publish event ”e” at an information bus.
  - **advertise(predicate e)** - share with the information bus that he publishes events with certain characteristics.
- **Subscriber**
  - **subscribe(predicate e)** - shares user's interest to a specific subset of events identified by a particular predicate or topic, to the information bus.
  - **unsubscribe(predicate e)** - undoes the above operation.
- **Information bus**
  - **notify(event e)** - delivers to the interested subscribers a notification.

2.1.3 Pub-Sub variations

After understanding the overall structure of a pub-sub environment, we need to define how a Pub/Sub system associates a subscription to one event or a subset of them.

- **Topic-based**: The earliest pub-sub scheme was based on the notion of topics. These systems can be organized into groups of subscribers interested in a specific topic. Using the information bus abstraction, we have for each topic an information bus responsible to forward the events with the same topic to the subscribers interested in them. This way, a publisher forwards an event to the

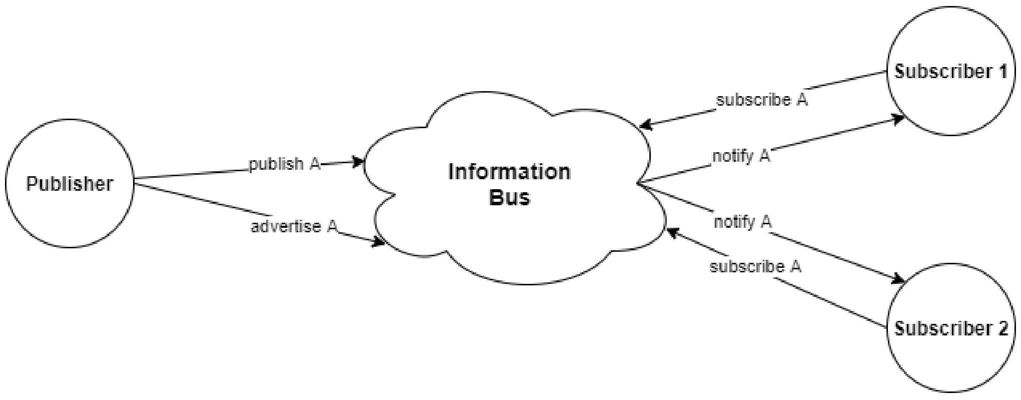


Figure 2.1: Pub-Sub interaction Schema

information bus responsible for that event's topic; a subscriber subscribes to the information bus that represents the topics of his interest; and when the information bus receives the event, it sends it to all interested subscribers kept at its records. In topic-based pub-sub, topics are commonly defined by a string. More advanced topic-based systems provide the idea of hierarchies. These topics might be represented by Uniform Resource Locator (URL)-like strings (e.g. fruits/banana), being a subscriber interested in a high topic of the hierarchy, he/she will receive all the events of the below subtopics (if he subscribes to fruits he/she will receive the events fruits/banana and fruits/apple for example). Examples of relevant topic-based pub/sub architectures are Scribe [16], SpiderCast [17], and Rappel [18].

- **Content-based**: Even in a hierarchical topic-based approach, if a node is only interested to be notified of PCs with 16 GB of Random Access Memory (RAM) when the price is below 1000 €, it will have to subscribe to PC/16GB and still would receive several notifications that it is not interested in (PCs price $>$ 1000€). Besides being an inaccurate approach (false positives at the subscriber), this congests the network by sending unwanted events. In these circumstances, a content-based alternative will prove more precise and might even be more efficient regarding bandwidth consumption. In a content-based approach, an event is mainly characterized by its content and not just associated with a topic. This way, subscriptions should be represented by predicates that not only represent topics and subtopics but also range queries. Predicates might be composed of symbol-operator-value triplets where a symbol might be "price", the operator "$<$" and the value "1000". Examples of relevant content-based pub-sub architectures are Hermes [3], Meghdoot [4], Sub-2-Sub [19], PastryStrings [20], Wormhole [10] and PopSub [21].

- **Type-based**: In a topic-based pub/sub, events are not just characterized by their content. They are grouped with others sharing the same structure/meaning linearly, making topic characterization inaccurate. With a type-based approach, an event is strictly characterized by a scheme that associates it with its type, enabling a closer integration of the language and the middleware (information bus). Moreover, type safety can be ensured at compile-time by parameterizing the resulting abstraction interface with the type of the corresponding events. One example of a type-based pub-sub system is FlexPath [22].

2.1.4 Architectural Details

Pub/Sub systems can be used for several purposes and implemented by several architectures, but will always consist of information exchange over a network infrastructure. Analyzing first the context of the information itself, we may represent two scenarios:

- **Messages** are any data transferred on a network. Some messages have a header containing meta-data information such as Identifier (Id), source, priority, and the respective payload used for routing or application purposes. Other systems treat messages as opaque bytes arrays and, others treat them has text or typified file (e.g. eXtensible Markup Language (XML)). Some types of messages might be self-describing and might not even need header information.
- **Invocations** are requests specifically designed to a certain kind of interface and add an application-related logical component to simple messaging exchange. An example of this in Internet of Things (IoT) is when a sensor subscribes to a system alteration like temperature change. Once it sends an invocation to another actor on the network (another sensor), the receiver triggers an immediate response to it, like turning on the air conditioner if the temperature is high.

The transmission of data between producers and consumers is the task of the middleware medium. This medium can be classified according to characteristics like its architecture. We can divide pub-sub media architectures into three main types:

- **Centralized Server:** here, to provide decoupling between subscribers and publishers, both publish and subscribe at a central server. Although this solution might not provide optimal dissemination and turn the server into a single point of failure that could easily bottleneck, having only one server could bring several advantages. With only one logically centralized server (possibly physically replicated), properties such as reliability, data consistency, or transactional support, are more easily guaranteed.
- **Decentralized:** a decentralized approach has no central control of the distribution of events, allowing faster delivery of transient data, and has no single point of failure. But now, properties easily managed by a centralized approach become much harder to assure in/with this architecture.
- **Hybrid:** we can mix both architectures by having an intermediate solution composed of several main servers that together manage properties like reliability in message delivery and consistency

of data. The servers can then exchange messages about their status, distribute their workloads and allow the system to tolerate the failure of some of them.

Besides a pub-sub level of centralization, there are some main techniques to structure event and subscription dissemination over more decentralized approaches. We mention here those we found more relevant:

- **Rendezvous:** in this approach, some nodes on the network will provide a point of contact between publishers and subscribers. Rendezvous can be assigned for each type of event by hashing an event header or content to a network's addressable space or be preconfigured and then accessed by checking a predefined list of rendezvous nodes. Examples of systems using this approach are Scribe [16], and Meghdoot [4].
- **Filtering:** a filtering approach can be used in different scenarios, but its goal is to transform a subscription into a filter. This way, once an intermediate node receives a filter, it may attempt to merge/combine it with previous ones, reducing the number of filters/subs it needs to check. This approach needs to implement a mechanism of distribution of the filters. It may redirect the filters to a rendezvous or event's publishers. Either way, it is more efficient to filter closer to the event's source (source-side filtering). Wormhole [10] uses this approach.
- **Gossip:** one alternative approach is to diffuse events via gossiping. Here subscribers are grouped by interest, and publishers somehow forward their events to subscribers interested in them. These subscribers organize themselves to guarantee the event's forwarding between them and the remaining interested ones. Typically this approach relies on the ability of the subscriber to find the best neighboring peers based on its interests. Some of the architectures using gossip are Spidercast [17], GossipSub [12], and Rappel [18].
- **Flooding:** the last approach is the flooding of events or subscriptions around an entire network, requiring a cache to prevent duplication of events. This approach is the most bandwidth-intensive and, on subscription flooding, it may also become CPU/memory intensive because it is hard to manage the entire state of an internet-like network at a single node.

In Table 2.2 we resume every mentioned pub-sub's characteristics.

2.1.5 Implementation Challenges

We already analyzed the different types of publish-subscribe systems and now will take a closer look at what those changes may translate. The guarantees provided by the medium for every message vary between the different systems. Among the most common qualities of service considered in pub-sub, we have the following:

PubSub Systems	Subscription Granularity	Centralization Degree	Dissemination Techniques
Scribe [16]	topic-based	decentralized	rendezvous
SpiderCast [17]	topic-based	decentralized	gossip
Rappel [18]	topic-based	decentralized	gossip
GossipSub [12]	topic-based	decentralized	gossip
Hermes [3]	content-based	decentralized	rendezvous + filtering
Meghdoot [4]	content-based	decentralized	rendezvous
Sub2Sub [19]	content-based	decentralized	gossip
PastryStrings [20]	content-based	decentralized	rendezvous + filtering
PopSub [21]	content-based	decentralized	different strategies depending on event popularity
Wormhole [10]	content-based	publisher-centred	filtering
FlexPath [22]	type-based	mesh-like	direct delivery
ScoutSubs*	content-based	decentralized	rendezvous + filtering
FastDelivery*	content-based	publisher-centred	direct delivery (1 or 2 overlay hops)

Table 2.2: Presenting the main characteristics of each mentioned pub-sub system. (*) Are those presented in this work, for reference.

- **Reliability:** to ensure reliable message exchange, a system must assure each subscriber receives all events of its interest. Several protocols allow reliable communication on centralized and hybrid networks but, it is harder to achieve in a decentralized approach. The solution may pass through the help of a structured overlay over this decentralized network that would somehow guarantee the reliability of all messages sent or through more complex recursive reliable protocols.
- **Persistence:** to guarantee data persistence in a pub-sub system, we must have a reliable message exchange algorithm and a safe copy, allowing the system to inform those who were down when the dissemination process was running. Normally, persistence is only provided in a centralized approach or semi-centralized where there is a central or distributed repository that keeps a copy of the events.
- **Priorities:** message prioritization is important when we have events related to run-time processes. This way messages with a higher priority could be treated earlier than other messages.
- **Transactions:** in messaging systems, transactions are used to group messages into atomic units, either a complete sequence of messages is sent and delivered, or none of them are.

We already mentioned that a centralized approach provides easier management of properties such as persistence and prioritization. But on the other hand, these systems do not scale well, are vulnerable to failures, and do not provide efficient event dissemination, which are the advantages of a decentralized approach. Although, for this approach to assure the same guarantees, it would need to have more complex algorithms that may increase the complexity of the system and decrease its previous benefits.

Similar reasoning can be made when comparing a topic-based with a content-based approach. Content-based allows fewer false positives on the subscriber's side by filtering each event more in detail. This approach also reduces the amount of wasted bandwidth due to traffic of unwanted events. On the other hand, although more imprecise and network expensive, topic-based is easily implemented using multicast protocols, while content-based approaches require more complex dissemination algorithms that increase each node's workload due to the extensive filtering of events.

So, we may conclude that there are no ideal pub-sub implementations but instead a relation of cost/benefit that, depending on our objectives and problem context, we may want to implement a more precise/network-lighter pub-sub or a faster/node-lighter one for example.

2.2 Peer-to-Peer

In this section, we will present the other important area of this work - Peer-to-Peer Systems [7]. In Section 2.2.1, we try to define p2p, then continue by presenting the different applications possible with p2p architectures in Section 2.2.2. In Sections 2.2.3 and 2.2.4, we will refer to the different levels of centralization and structure of p2p systems, respectively, and finish with an analysis of the challenges p2p systems face in Section 2.2.5.

2.2.1 What are Peer-to-Peer systems

The definition of what is a peer-to-peer system or which systems are p2p is not exact, but to clarify this concept, the following two main properties are central in most definitions of what is a p2p system:

- Decentralized sharing of computational resources on a particular network.
- The ability to maintain a properly functioning network even in the presence of node failures, connectivity problems, and churn (nodes leaving and entering the network).

After the turn of the century, p2p systems were opposed to several grid computing solutions. Grid computing is typically a geographically distributed network of super-computers loosely coupled. With these networks' growth, problems relating to fault-tolerance and scalability started to emerge.

So p2p systems were seen as its antagonistic environment because they are highly tolerant to node failure/departure and can easily scale. This is due to:

- Every node being both a client and a server, and so, the number of servers grows linearly with the number of clients, preventing server bottlenecks.
- Nodes are self-organizing and do not need to have global knowledge of the system, facilitating joins, departures, and failures/recoveries of nodes, which are gracefully handled below a certain churn threshold.

Still, having computation power centralized in fewer machines has its advantages, but today with the amount of processing power available at the edge, on millions of computers and the hundreds of millions of smartphones and other Web-connected devices, p2p systems have become extremely attractive.

To regulate large-scale p2p systems, these have overlay structures providing logical support to nodes of the system, ensuring routing between them as also join/departure/recovery mechanisms to all system nodes. The "overlay" concept comes from the p2p system's overlay topology being built on top of an underlying physical computer network infrastructure (typically Internet Protocol (IP)-routing).

2.2.2 Peer-to-Peer Applications

There are many types of p2p applications using p2p architectures to work, from the most traditional file-sharing systems to the new cryptocurrency ones we list the most relevant:

- **Communication and Collaboration:** includes systems that provide the infrastructure for facilitating direct, usually real-time, communication and collaboration between peer computers.

- **Distributed Computation:** includes systems whose aim is to take advantage of the available peer's processing power (CPU cycles). This is achieved by breaking down computer-intensive tasks into small working units and distribute them to different peers, who execute them and return the results. Central coordination is required at the task level, being the node responsible for delegating its parallel computation, its coordinator. Other approaches might leverage idle resources from p2p networks [23], including purely web-based solutions (e.g., browsercloud.js [24]).

- **Internet Service Support:** different applications based on p2p infrastructures have emerged for supporting a variety of internet services. Examples of such applications include p2p multicast systems, internet indirection infrastructures, security applications that protect against denial of service or other attacks that may place a server unavailable, and cloud services [25].

- **Database Systems:** considerable work was done on designing distributed database systems based on p2p infrastructures. This may vary from sets of data stored in a p2p network comprised of inconsistent local relational databases interconnected to each other or scalable distributed query engines built on top of a p2p overlay network topology that allows relational queries to run across thousands of peers [26].

- **Sensors' Networks:** with the continuous development in IoT and the increase in interconnected smart devices, being on Closed-Circuit TeleVision (CCTV) surveillance systems or at smart-Housing, p2p IoT systems will become even more common after 5G networks go online [27].

- **Cryptocurrency networks:** some of today's most business-relevant p2p networks are those used by Bitcoin and Ethereum [28] to propagate blocks and transaction information of their block-chains. These networks may goal is to spread cryptocurrency transactional data among miners and other participants.

- **Content Distribution:** this area includes systems and infrastructures designed for the sharing of digital media and other data between users. Peer-to-peer content distribution systems range from relatively simple direct file-sharing applications to more sophisticated systems that create a distributed storage medium for securely and efficiently publishing, organizing, indexing, searching, updating, and retrieving data. These types of p2p systems are the focus of this project.

2.2.3 Overlay Network Centralization

Although peer-to-peer overlay networks were supposed to be decentralized, in practice, this is not always true, and systems with various degrees of centralization have emerged. Specifically, the following three categories may be identified:

- **Purely Decentralized:** all nodes in the network perform the same tasks, acting both as servers and clients, and there is no central coordination of their activities. Some architecture examples are Kademlia [8], CAN [29] and Pastry [30].

- **Partially Centralized:** is similar to purely decentralized systems, although some nodes assume a more important role, acting as local indexes for files shared by neighboring peers. Since these super-nodes are dynamically assigned by the overlay and they do not constitute single points of failure and, if they fail, the network will automatically replace them with others. One example of this is Kazaa [31].

- **Hybrid Decentralized:** in these systems, there is a central server facilitating the interaction between peers by maintaining directories of metadata, describing the shared files stored by the peer nodes. Although the end-to-end interaction and file exchanges may take place directly between two peer nodes, the central servers facilitate this interaction by performing the lookups and identifying the nodes storing the files. Obviously, in these architectures, there is a single point of failure (the central server). This typically renders them inherently unscalable and vulnerable to censorship, technical failure, or malicious attack. One example of this is Publius [32].

2.2.4 Overlay Network Structure

We can build network overlays using non-deterministic methods as nodes and content are added or using specific rules restricting a node's actions. We can categorize peer-to-peer networks in terms of their structure as:

- **Unstructured:** the placement of content is unrelated to the overlay's topology. In an unstructured network, content typically needs to be located. Searching mechanisms range from brute force methods, such as flooding the network with propagating queries in a breadth-first or depth-first manner until the desired content is located, to more sophisticated and resource-preserving strategies that include the use of random walks and routing indices. These systems are generally more appropriate for accommodating highly transient node populations. An example is BitTorrent's system [1] which allows a node to join a p2p graph to share/acquire a file. These systems require some kind of bootstrapping mechanism, but after the node joins the network, it becomes purely unstructured.

- **Structured:** these alternatives emerged mainly to address the scalability issues that unstructured systems were facing. In structured networks, the overlay topology is tightly controlled, and content is precisely placed at a specific location in the ID space of the overlay. These systems essentially provide a mapping between content and location, in the form of a distributed routing table, so that

Network Overlays	Centralization Degree	Structure Degree
Publius [32]	hybrid decentralized	unstructured
Kazaa [31]	partially centralized	unstructured
Kademlia [8]	purely decentralized	structured
Pastry [30]	purely decentralized	structured
CAN [29]	purely decentralized	structured
Bittorrent [1]	purely decentralized	unstructured

Table 2.3: Presenting the main characteristics of each mentioned network overlay.

queries can be efficiently routed to the node with the desired content. Structured systems offer a scalable solution for exact-match queries, where the identifier of the requested data object is known. A disadvantage of structured systems is that it is hard to maintain the structure required for efficiently routing messages in the face of a very transient node population, where nodes are joining and leaving at a high rate. Examples of these systems can be those based on prefix-based routing has in Plaxton [33].

At Table 2.3 we can see in summary the characterization of the mentioned content distribution systems.

2.2.5 Challenges in P2P Content Distribution Systems

Due to their loosely coupling and liberal policies, these systems tend to create challenges to their use when specifying stricter requisites for:

- **Security:** if we want to keep a file only accessible for a certain number of people, but at the same time, we want to make it available and replicated on a p2p infrastructure, we need ways of assuring secure store, routing, and access control. Having these properties without recurring to a centralized solution is challenging. Other security properties become relevant when talking about cryptocurrency in p2p networks and protecting against Sybil and other attacks that may be tackled via local peer reputation metrics [12].

- **Deniability:** in p2p content sharing, a node may host data that he does not own/published, and so he must not be legally responsible for that data. Nowadays, p2p networks have been used to avoid state censorship and, on the other side to evade legal measures (online piracy). Deniability can be applied both to the content stored and to content in transit.

- **Resource sharing:** in p2p content sharing networks, we may find “free-riders”, which are nodes/users of the system that use those system's resources and do not contribute to it. To tackle this, many systems may implement one of these mechanisms:

  - **Trust-based incentives** - in this approach, there must be a way to know the reputation of a node. In centralized systems, the central server can easily calculate every peer reputation (rating), but in p2p systems, the information can only be local to a neighborhood. To have a global idea of a peer reputation it would be necessary to aggregate several local reputation values. A peer's reputation can be kept by logging its interactions locally.
  - **Trade-based incentives** - in this approach, users can establish contracts with other users by assuring disk space, hosting a file, or caching other users' content. This way, peer relations are merely contractual.

- **Resources management:** these p2p systems provide the users with a content sharing infrastructure, but operations like content deletion or update are much harder to be efficiently implemented.

2.3 Relevant Systems

In this section, we present some systems related to our problem's context and inspired the solution presented in Section 3. In Section 2.3.1, we resume the Interplanetary File System [2]. In Section 2.3.2, we present and explain in detail Kademlia's DHT [8] and finish with the Hermes architecture [3] in Section 2.3.3, which inspired part of our solution.

2.3.1 Interplanetary File System

The Interplanetary File System is a peer-to-peer file-sharing system [2]. IPFS' core ideas are to provide a more decentralized and content-oriented approach to the current Web, which is centralized in a few big players (Google, Amazon, Microsoft, ...).

Providing a more decentralized Web would bring several benefits, such as:

- Increased data resilience, since data could be saved in several independent devices all around the world instead of centralized repositories.
- Data would become harder to censor.
- Can decrease bandwidth consumption at the Web backbone because it is no longer necessary to communicate with centralized servers if one IPFS node next to you already has the file you want.
- Can improve the performance of more isolated networks with low throughput gateways to the Web, for the same reason of the above topic.

At IPFS, files are not identified by their location (e.g. tecnico.pt/file) but by their content. This way, when we search for a file in IPFS, we do not need to know its location but its Content Identifier. IPFS uses these Content Identifier (CID)s because a file cannot be represented by a physical address when it is replicated around the Web.

To better explain IPFS, we may divide IPFS core architecture into how it addresses its content using CIDs, content is correlated using Directed Acyclic Graph (DAG)s, and how the content is forward using DHT-like routing.

Content Addressing: Every file at IPFS has a CID, which is generated by hashing a file content. To address data by its content, IPFS uses InterPlanetary Linked Data for connecting and addressing different types of data. This way, a CID is not a simple file's raw byte hashing but also contains its data structure. This means it contains its raw bytes hash, multihash (hashing algorithm used), and codec (data format, ex: JSON). With InterPlanetary Linked Data (IPLD), every type of data will have a unique content representation on IPFS.

Distributed Acyclic Graphs: DAGs are used in IPFS (Merkle DAGs) as a way of structuring complex data objects by using their acyclic leaves to parent node properties. This way, a file inserted at IPFS will be represented by a Merkle DAG (tree-like structure), where each leaf will be a part (block of data) of the original file and the root of the DAG is a CID identifying the entire file. This would make it possible, like in other file-sharing systems, that being a file sub-divided, once one node of IPFS wants a Web page, it will get first the root CID, and then he could ask for each child of the root in the DAG to be downloaded from different peers as long as they have those files. Also, if an HypertText Markup Language (HTML) page has an image, it could be downloaded from someone who has the HTML page or someone that only has that image because CIDs are equal for files of the same content.

Distributed hash tables: For information lookup and routing, IPFS has a Kademlia DHT implemented. We will not go on particular details around Kademlia's design and protocol in this section because we find it nuclear enough for our work and will deserve a space of its own at the relevant systems.

2.3.2 Kademlia

The authors of Kademlia introduced a new DHT for peer-to-peer systems that provide performance and consistency in fault-prone environments, querying and rooting through the network's topology using a novel XOR distance metric. We present this system because IPFS routing is implemented using an adaptation of Kademlia's DHT, and so it is important to understand this p2p structured overlay system.

Design Overview: A peer in Kademlia is assigned a 160 bit opaque Id (describing version [8]), which becomes its position in the DHT's addressable space. The system provides a lookup algorithm that allows this peer to reach any Id in Kademlia in a logarithmic number of steps. Kademlia's addressable space is represented by its authors as a binary tree in which each peer is a leaf.

The routing algorithm makes sure that a node knows at least one other peer from each existing subtree. A subtree of the node 0011 will be trees 0, 00, and 001. With this property, peer 0011 can reach every node (available) in the tree by requesting recursive queries to peers he knows from the subtree that node X belongs to. An example of a peer and one of its subtrees is in Figure 2.2.

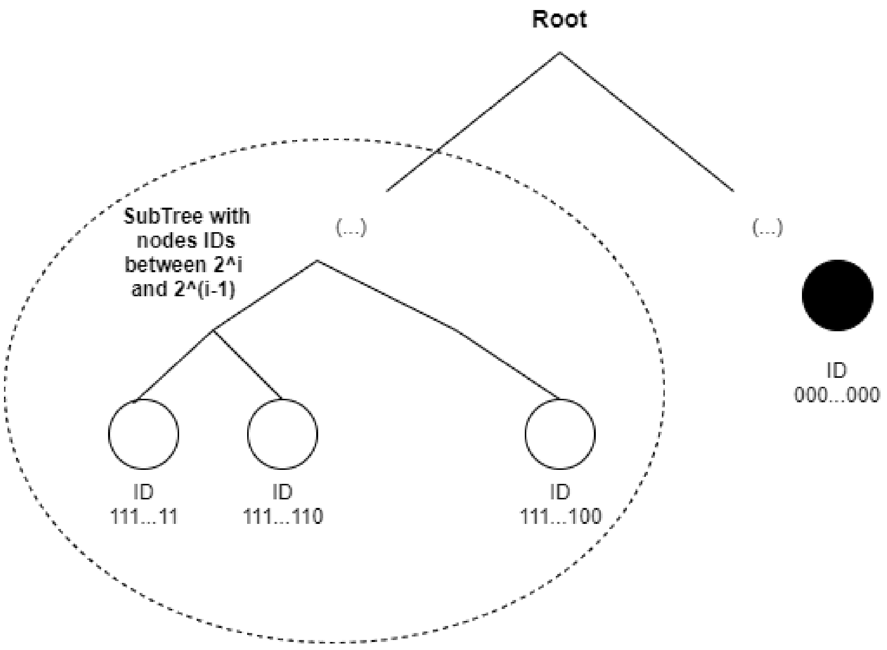


Figure 2.2: Kademlia's addressable tree-like space

XOR Metric: As said before, in this system a peer has a 160-bit Id, but files also have an identifying key with a 160-bit Id, being a file a pair (key, value). So, to have the notion of closure, it's necessary to calculate distances between peers and between peers and keys. The notion of distance in Kademlia is calculated by an XOR operation between two peers or key Ids. The XOR operation has several useful properties, but the reason why it is perfectly suited to Kademlia is that it captures the essence of its tree-stretching structure.

Node State: Each node will have several lists with nodes that are a certain distance from itself. One list is called a k-bucket and could comprise k peers with an XOR distance between 2^i and $2^i + 1$ of the node, corresponding to a subtree. The peers in the bucket are identified by the triplet (IP address, User Datagram Protocol (UDP) port, peer Id). There may be one k-bucket for each sub-tree. To keep each bucket record fresh, the peers are organized in a queue from least used to the most recently used. This way, when the lookup algorithm is running, it refreshes the used buckets. This also improves the bucket's peer availability over time due to a verified preposition in other distributed file-sharing systems that the longer the nodes are available, the more probable they are to continue being available [34].

Kademlia Protocol: The Kademlia protocol consists of four RPCs: PING, STORE, FIND NODE and FIND VALUE. The PING RPC probes a node to see if it is online. STORE instructs a node to store a (key, value) pair for later retrieval.

The FIND NODE RPC receives a 160-bit Id. He then asks for the k closest peers to the requested Id to the peers of the k-bucket closer to the ID. Each receiver returns k triplets (IP, port, Id) from the closest k-bucket to the requested Id or from several if the preferential k-bucket is empty or with less than k peers. The FIND VALUE is similar, but once he reaches the file with the wished key, he immediately returns and terminates the operation. This way, to find peers or files, Kademlia uses as lookup algorithm the FIND NODE and FIND VALUE recursively applied initially to an alpha number of nodes closest to the pretended Id. This process in FIND NODE only stops if the node is found or the closest node to it (this will finish by sending FIND NODE to all the k closest known nodes to the Id by the initiator). In the case of FIND VALUE, this successfully stops once the process returns the file or it fails after reaching the closest nodes and does not find it.

During the lookup process, nodes that fail to quickly reply to a query are assigned as stale nodes. To efficiently replace them, there will be cached good nodes when the buckets are full, so that once a node is marked as stale, it is immediately replaced with a cached one if possible. A stale node is not immediately removed, because if the bucket is incomplete it might still be valuable to use one with a bad connection. To assure data availability, each data owner must store every file every 24 hours if it has not been accessed.

To make sure k-bucket's peers are in good condition if there are no operations performed by those peers, every hour it will be chosen a random Id to perform the FIND VALUE on every peer of that bucket. One way to decrease traffic at hot spots is to cache temporarily along the way the popular file.

Routing Table: Kademlia routing is composed of several k-buckets. Each k-bucket represents a subsection of the 160-bit Kademlia tree's foliage (peers). Each bucket has optimally k nodes. One of the main problems in the Kademlia system is if the binary tree becomes too unbalanced (for example there are 100 nodes with the prefix 1... and only 5 with the prefix 0...).

Efficient Key Re-publishing: Keys need to be re-published due to two types of events: the nodes that contain those keys leave or fail and new nodes might enter the system closer to that key. To tackle this problem a key needs to be re-published from time to time. One way to do this is setting up a timer, which increases the number of packets on that k closest nodes but being the probability of them being all synchronized low, they will not do it simultaneously. After the first re-publishes, the others' keys would restart their timers.

IPFS' Kademlia Implementation: Regarding Kademlia's version on IPFS, libp2p [35] provides several implementations in different programming languages, being the main distinction between it and the old version presented the 256-bit addressable space used by IPFS. IPFS also uses constants (like k-bucket, with k=20) that allow optimal performance results at its p2p network environment, and further research is being done to improve its performance.

2.3.3 Hermes

The Hermes [3] content-based pub-sub system was chosen as one of the most relevant systems for this project because its overall design inspired part of the proposed solution has we will see later in Section 3. There are two types of participants in Hermes: event clients (publishers and subscribers) and event brokers. Communication between participants is made with message passing, and subscriptions are passed through the network to filter events as closer to the publisher as possible (source-side filtering).

Routing Overlay Network: To provide a scalable overlay structure capable of IP routing and tolerant to node failures, Hermes works on top of a Pastry-like overlay routing network, where every node has a unique random identifier. This overlay provides Hermes the following message passing abstraction route (message, destination Id). To have efficient event forwarding, there needs to be a way to build dissemination trees for each event, which could be created when advertising or subscribing to a subset of them. Although many systems use this by broadcasting subscriptions or advertisements to the network, Hermes implements rendezvous nodes for each event type. This way subscriptions and advertisements are forward using the route abstraction to the broker with an Id determined by a hashing function.

Type and attribute-based Pub/Sub: In many content-based approaches, there is a mismatch between events and application programming languages. This is because events are mainly untyped collections of pairs attribute/value, and programming languages only support statically or dynamically typed objects. This way, Hermes associates to each event and subscriptions an event type, allowing the application to perform runtime checks to the events. Each event type is managed by its event broker that acts as the rendezvous node for that event. Event types can be specialized by applying attribute-based filtering. In the same way, a subscriber may have a subscription that encompasses an entire event type or a specialized subscription that represents his interest at a particular event subset, providing a content-based pub/sub approach.

Architecture: Hermes follows this layered scheme on top of an IP unicast network (e.g. Internet) as seen in Figure 2.3:

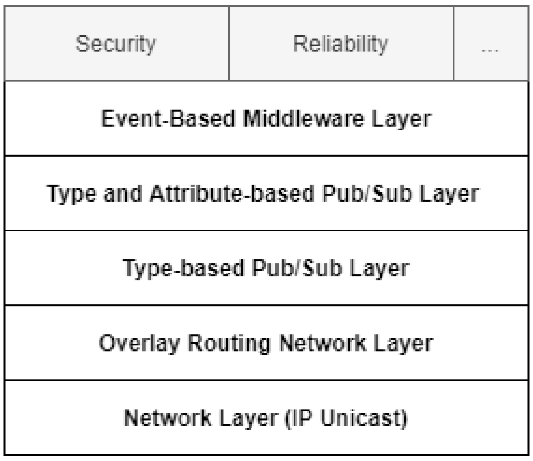
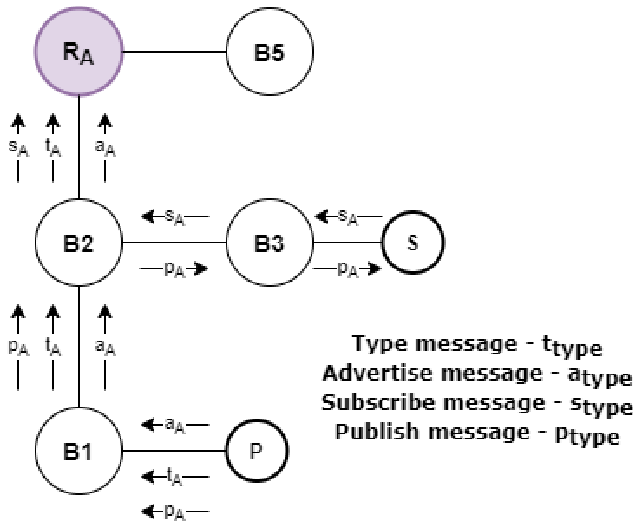

Security
Reliability
...
Event-Based Middleware Layer
Type and Attribute-based Pub/Sub Layer
Type-based Pub/Sub Layer
Overlay Routing Network Layer
Network Layer (IP Unicast)

Figure 2.3: Hermes' Layered Scheme

- Overlay network structure – responsible for routing messages between brokers.
- Type-based pub/sub layer – responsible for attributing brokers as rendezvous nodes for certain event types, providing a topic-based pub-sub layer.
- Type and attribute-based pub-sub layer - it distributes filter expressions through the network of event brokers to achieve efficient source-side filtering on event attributes [36]. This allows it to provide a content-based approach on top of a topic-based.
- Event-based middleware layer – provides an Application Programming Interface (API) that allows a programmer to advertise, subscribe, or publish events, to add or remove event types from the system, and it performs type-checking of events and subscriptions. The middleware layer consists of several modules that implement further middleware functionality such as fault-tolerance, reliable event delivery, event type discovery, security, transactions, mobility support, etc.

Event Routing Algorithms: The layers responsible for routing events are the type-based and the type + attribute-based layers. To understand the pub-sub interactions follow Figure 2.4 Both layers will use the following messages:

- Type messages – add new event types to the system and set up rendezvous nodes for them. A typed message contains the definition of an event type stored at a rendezvous node. Published events can be type-checked against this definition.
- Advertisement messages – denote an event publisher's capability of publishing a certain event type. They set up event dissemination paths in the broker network, routing them towards rendezvous nodes.
- Subscription messages – express a subscriber's interest in certain events. Together with advertisements, they are routed towards rendezvous nodes creating paths for events through the network of brokers.
- Publication messages – carry published events in message form. They are sent by event publishers and follow the paths created by advertisement and subscription messages.



R_A
B5
s_A
t_A
a_A
B2
s_A
B3
s_A
S
p_A
p_A
p_A
t_A
a_A
Type message - t_{type}
Advertise message - a_{type}
Subscribe message - s_{type}
Publish message - p_{type}
B1
a_A
P
t_A
p_A

Figure 2.4: Hermes Pub-Sub Protocol

Type-Based Routing: Type-based routing is the core of Hermes' dissemination of events. The routing paths are formed by the reverse-path forwarding [37] of the publisher advertisements and subscriber subscriptions to the rendezvous node of that event type. Events do not always need to go to the rendezvous node and might be forward to the subscribers directly if subscriptions paths were already formed. By allowing this, Hermes reduces the bottleneck at the rendezvous node.

Type and Attribute-Based Routing: While advertisements create a path for the subscriptions to follow to the rendezvous nodes, subscriptions while traversing the network on their way to the rendezvous node leave their filter at intermediate brokers. A subscription filter corresponds to the attribute-based layer, which works over the type-based one. This way, Hermes improves event delivery precision by enriching the topic-based approach, turning it into a content-based one. Filters on intermediate nodes will be summarized, and subscriptions already covered by these brokers will not be forwarded to the rendezvous node.

Fault-Tolerance: In a peer-to-peer system, nodes may fail or leave, so the system itself needs to adapt to these changes. So, Hermes refreshes the soft-state of the network nodes by having a heartbeat protocol that regenerates the nodes' state. Besides that, it also relies on the overlay structure to manage node/link failure. The fault scenarios are link failure, event broker failure, and event client failure. The most complex fault on the system is a rendezvous node failure, bringing the necessity of having more than one rendezvous node. With this, the complexity of the system increases but the possibility of load balancing solutions emerge. To support multiple RNs, it would be needed to add a salt to the hash function to have as many RN replicas as different salts + 1. So now the authors propose four different approaches to manage this situation:

- Clients subscribe to all RNs: When a client issues a subscription, it must separately send this subscription to all replicated rendezvous nodes. An event publisher may then advertise to any rendezvous node. This measure makes subscriptions more expensive.
- Clients advertise to all RNs: An event publisher advertises its event types to all replicas, and subscribers may subscribe to any RN replica. This measure creates an overhead for every event publisher in the system.
- RNs exchange subscriptions: Event clients only subscribe to a single replica, but this one then subscribes to all other ones. In the case of network partitions between different RN replicas, the subscriber will potentially not receive all events but, it will be able to subscribe if at least one replica is reachable. Replicas need to support a reconciliation protocol when they re-join after a network partition.
- RNs exchange advertisements: In this scheme, the replicas exchange advertisements. This approach is more scalable than the previous scheme when the number of subscriptions exceeds the number of advertisements in the system.

Summary

In this chapter, we presented our two main areas of interest. We showed the objective of a publish-subscribe system and its different variations and ways of being implemented. In second, we presented peer-to-peer systems capable of providing a content distribution overlay and analyzed the different types of architectures and the challenges these systems face.

At last, we analyzed the Interplanetary File System because it is the environment where our pub-sub will work, we presented Kademlia's DHT since it underlies the IPFS content routing and will be used by our pub-sub, and showcased a summary of Hermes' pub-sub since it inspired our pub-sub's architecture.

We may conclude through the research made that pub-sub and p2p systems have already been combined before, since both are decentralized architectures by nature, to provide ways to disseminate data in environments with large number of users, with a transient node population. IPFS is an example of a p2p system with a Kademlia's DHT as its content distribution system, that with our work's effort will have a content-based pub-sub system also operating on top of it.

3 Architecture

Contents

Architecture

In this section, we propose a content-based pub-sub middleware able to disseminate events over IPFS. First, in Section 3.1, we present the main requirements to guide our system's design, then we introduce the overall system design in Section 3.2, and then in Sections 3.3 and 3.4, we will present their algorithms/protocols in detail.

3.1 Requirements

From the beginning of the thought process, we decided to drive the design of our solution to aim at fulfilling the following requirements:

- **Content-based:** our system needs to give the subscriber the option to specify his interest with finer granularity, to minimize the amount of uninteresting information received.
- **Decentralized:** our solution must not rely on any central authority/peer to work properly, and no peer on the network should need to have a global view of the system.
- **Scalability:** our solution needs to work in an environment of millions of users with ease.
- **Efficiency:** our system should use the minimal network (bandwidth) and computational (CPU and memory) resources possible.
- **Fault Tolerance:** our system needs to tolerate a churn rate (peers going in and out of the network) without compromising the system's functioning and resist the failure of several of its nodes.
- **Quality of Service (QoS):** our pub-sub should focus on delivering the information to the subscribers as fast as possible.
- **Flexibility/Extensibility:** due to the limited time and scope of our project, we decided to build a system that can be partially changed/improved by anyone.

Also, due to time limitations, we decide to focus our pub-sub middleware on a non-byzantine fault scenario. To integrate our system with IPFS and open its use to everyone, we must protect the system against malicious actors. We will leave that job for future projects.

3.2 Overall Design

From the start, we tried to design one optimal solution using the existing IPFS routing overlay (Kademlia DHT). Because Kademlia [8] is a structured approach, we saw two options: build a structured approach over it or only using it as a bootstrapping mechanism to enter a gossip approach. After sketching up some architectures, we decided to go with a more structured solution, due to their theretical higher scalability and efficiency (Section 2.2.4), and the fact that IPFS' main pub-sub topic-based alternative is unstructured (GossipSub [12]).

But there is a particularity with Kademlia peer Id distribution. In a network with Kademlia as its routing overlay, peers are assigned an Id and establish connections with other based only on Id distancing between them. That makes IPFS peer connection not geographically oriented, increasing the network's resilience, but reducing performance by allowing redundant hops like US-CHINA-US-CHINA.

Taken that into account, we decided to proposed an event middleware with the two following protocols, also represented in Figure 3.1:

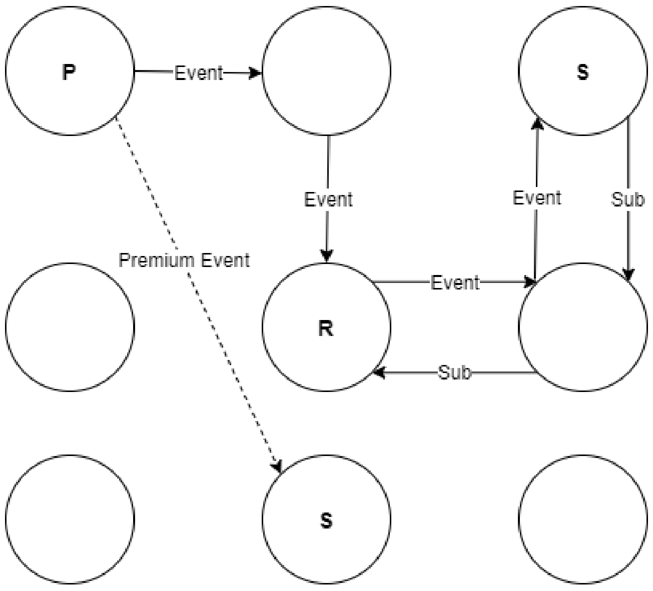


Figure 3.1: Solution Overview

- **Decentralized Protocol - ScoutSubs:** This protocol will provide a pub-sub system using the existing IPFS routing tables and adding a filtering structure inspired by the Hermes system [3] to disseminate events from pub to sub based on their content and the content of the subscribers' subscriptions. ScoutSubs allows its users to possess a global semantic-based knowledge of their network.

- **Publisher Centered Protocol - FastDelivery:** This protocol's focus is to deliver events as fast as possible. Theoretically, the fastest way to disseminate events would be to use IP-multicast [38], but this solution needs previously configured routers to work. So, we provide an application-level multicast, geographically oriented, to provide low latency event delivery. The publisher coordinates its pub-sub service and may request its subscribers to disseminate its events making the IPFS' overlay used only for advertising. This protocol becomes interesting when subscribers are interested in a source of information instead of a global notion of content.

3.2.1 Protocol Stack

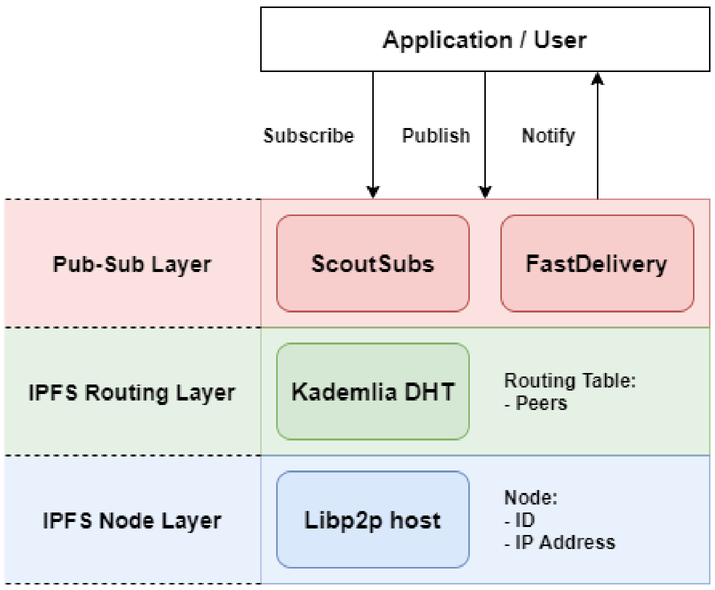


Figure 3.2: Pub-Sub Protocol Stack

Our system's network stack is comprised of three main layers illustrated in Figure 3.2. At the bottom we have the libp2p host, which represents a node at IPFS and includes all its properties and information. For our pub-sub we will use a node's Id and its address.

Representing IPFS content routing we have an instance of its Kademlia DHT, in which we will use its routing table to reach every key (rendezvous) closest peer in the network in a logarithmic number of steps.

Our pub-sub layer will have both our protocols that will use the ones below to provide an user or external application the possibility of publishing or subscribing to events in a content/meaningful way.

3.2.2 Expressing Content

In both protocols, the way a subscriber expresses its interest and the publisher expresses the content of its event is by assigning it a predicate.

A predicate is an expression assigned to a piece of data (event/message/file) that attributes a semantic meaning to it, allowing human comprehension of its content. This expression is composed by attributes that add specific meaning to a predicate.

For our project, we only use two types of attributes:

- **Topic:** should be single word key phrases capable of capturing the essence of the described data. Common examples can be names of countries, companies, bands, clubs or sports.
- **Range Queries:** are attributes with numerical meaning, composed by a characteristic and its numerical interval/value. Common examples of these numerical characteristics can be price, temperature, height or dates. We cannot forget that for this to work, all predicates need to use the same units (e.g. use dollars for prices and Celsius for temperature).

Predicates need to be assigned to events published on the system and to the subscriptions, so that our pub-sub may understand who is interested in what. To simplify, predicates are assigned to events by their publishers and to subscriptions by the subscribers.

3.3 ScoutSubs Protocol

We now introduce the ScoutSubs protocol and specify its fault tolerance mechanism at Section 3.3.2, the redirect mechanism that allows unpopular events to jump unnecessary hops at Section 3.3.1, how we achieve reliability at Section 3.3.3, and how we maintain and recover our system's state at Section 3.3.4. This protocol is the center of our work, since it provides a decentralized content-based pub-sub alternative both scalable and efficient.

As mentioned before, the ScoutSubs protocol will provide a pub-sub middleware over IPFS' content routing overlay. The base design of this system was inspired by Hermes [3], for it shares a topic and a filtering layer over it that provides a content-based approach.

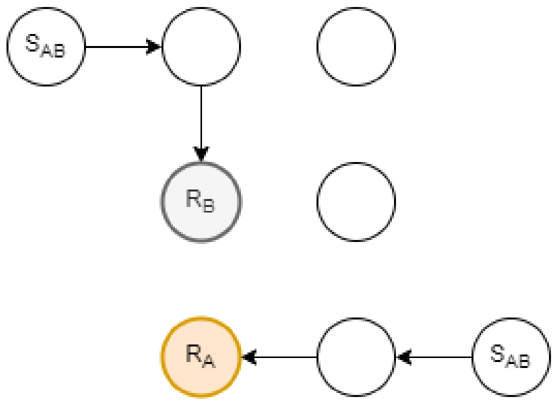
Rendezvous The topic-based layer is created by the rendezvous nodes. There are as many rendezvous nodes as attributes, being the one representing the attribute football the closest peer to the key generated by its string.

The purpose of these nodes is to provide a point of reference for the subscription forwarding. So, if we have the subscription (football, Tom Brady), we first see which of these attributes has the closest Id (XOR-Distance) to ours, and then we will forward the subscription towards it, minimizing the number of hops. On the other hand, the publisher needs to send its events towards all the rendezvous of its event attributes. Figure 3.3(a) illustrates that.

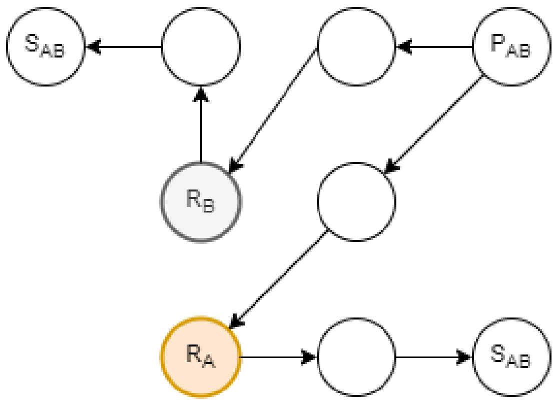
Filtering To provide a content-based approach, we implemented a filtering mechanism over the rendezvous nodes. Before a subscription is sent towards the rendezvous node, it leaves its filter (subscription) at each intermediate node. This way, when a publisher forwards the event to the rendezvous node, once it arrives at it, it will follow the reverse path of the subscriptions back to the interested subscribers, as Figure 3.3(b) shows.

Here the idea of the filter is separate from subscription because an intermediate node may merge a subscription or even ignore it if it does not change the outcome of the forwarding decision. This happens when a intermediate node receives a subscription/filter it already has or has other filter that contains this one. For example, consider that a node A receives from B a filter (portugal); if it receives a filter (portugal, weather) from B it will ignore it because the first one already encompasses the second. Filters may also be merged if they have range query attributes that intersect themselves and all other topic attributes in common. These two actions decrease the number of filters an intermediate node needs to analyze when it receives an event.

All the filtering information is kept at a filtertable which initially is a replica of the kademlia routing table [8]. Upon receiving subscriptions from those nodes, it adds filters to those node's entries. When receiving a subscription from a new node, it needs to create a new entry at the table and register the filter.



(a) Subscription forwarding



(b) Event Forwarding

Figure 3.3: Event and Subscription dissemination

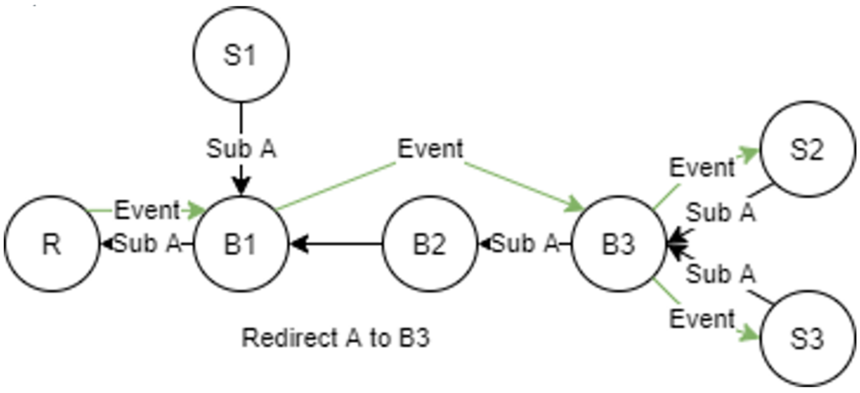


Figure 3.4: Forwarding using shortcut

3.3.1 Redirect Mechanism

To optimize event forwarding, we decided to add a mechanism that would allow an event to jump as many hops as possible without compromising its delivery to all interested subscribers.

To make this happen, when a node receives a filter from one peer towards a rendezvous node, it will always forward upstream the option to provide a redirect (jump over itself), if there were no filters forwarded from other peers to that same rendezvous, as shown in Figure 3.4. We need then to keep track of how many filters were forwarded to each rendezvous. If the number of filters is below two, we may provide a shortcut option, but if the number gets bigger or equal to two, we must warn the node upstream that the shortcut is no longer valid, forcing it to forward these events through the regular path.

Advantages The first immediate advantage is that we reduce the number of hops on the network, saving bandwidth and reducing event delivery time. The other important point is that the peers that are jumped over no longer have to check their filtertables, saving their precious CPU cycles. This fact is even more interesting because shortcuts are used more frequently on unpopular events, meaning that searching a filter table with lots of filters to then have only one hit avoids wasting CPU unnecessarily. Besides these points, it is a simple structure to implement, meaning it would only require to keep track of the number of subscriptions forwarded per rendezvous, and to save the redirect option.

3.3.2 Fault Tolerance

To tolerate the failure of any nodes along the dissemination path (rendezvous to subscribers), the filtering information must be backed up somehow.

To achieve this, we decided that every node will have f backups and that these are the peers closer to the node in question by Id (XOR Distance). This backing up mechanism allows our system to tolerate f failures of consecutive nodes by Id, meaning that in a large network where node failures occur inde-

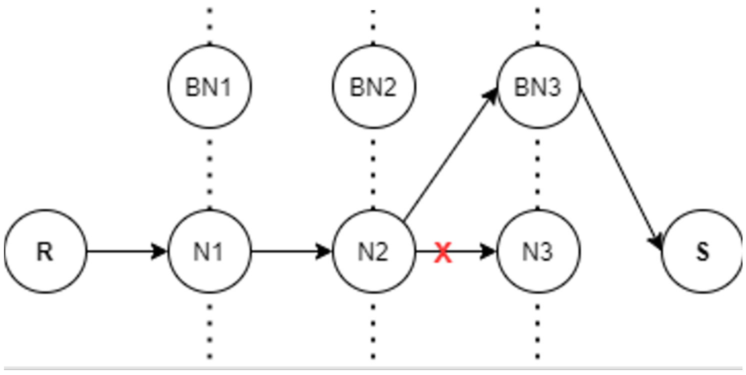


Figure 3.5: Forwarding event using backup

pendently of a peer's Id, our system can withstand several node failures. This system is also designed to have a dynamic tolerance, meaning we can initialize our system depending on the churn rate we want to tolerate.

Backing up The information relevant to each node are the filters it contains in its filtertable, being then essential to back up the filtertable to those peer's backups. The exception to the rule is the filtering information arriving at the rendezvous since it needs to be backed up to the closest nodes to the rendezvous attribute key, instead of the closest ones to its Id.

Another important detail is that the node upstream (closer to the rendezvous) also needs to know the backup nodes of the peers it receives the filters. Having this necessity means that each entry of the filtertable, besides having Id, endpoint address, and filters, also needs to have that node's backups endpoints.

With all of this, once an event arrives at a node, it first checks his filtertable to know which ones are interested in a particular event and then tries to forward it to them. If one node is not responding, it will send the event to the first working backup of that node. Once the backup receives the event with a backup flag activated, it will check the copied filtertable of the failed node and forward the event downstream. We can see the backup chain working in Figure 3.5.

Rendezvous management The role of the rendezvous node has one particularity because even if it is properly functioning, it also needs to be a long-lived node, meaning it needs to be working for an entire refreshing cycle or epoch (explained at Section 3.3.4). This working time requirement is necessary since a new node entering the network and closer to the key, e.g. surf, than its previous rendezvous node will become the new rendezvous node for the attribute surf. But the new rendezvous node will be missing most of the filtering data, so until it has not worked for a complete refreshing cycle, besides sending to the ones he already knows, it needs to redirect the events to an old rendezvous or one of its old backups.

Backups of intermediate nodes do not need to be long-lived nodes because these are fixed and refreshed upon change.

Building upon underlying Kademlia's module The path between the subscribers and rendezvous nodes needs to be backed up, but all other functions and properties are inherited from Kademlia.

The pathway from the publisher to the rendezvous node only uses information from the Kademlia's module, not needing any backup mechanism besides the fault tolerance already present at Kademlia's operations. Because we always use IPFS DHT to guide our subscriptions and events towards rendezvous, appointed by its hashing functions we also inherit its tolerance to network partitioning. After a network being partitioned, it only needs a refreshing cycle to make the new separated networks fully operational internally.

3.3.3 Achieving Reliability

With all mechanisms mentioned above, our pub-sub system is not reliable. When a peer crashes mid forwarding process, the subscribers downstream may not receive the event. To ensure this does not happen, we need to implement a tracking mechanism.

Tracking Mechanism To assure the system is reliable, we need to implement some acknowledging chain. The trickiest part is that implementing it in a completely decentralized network is a bit inefficient, but still, we pretended to build a fully reliable version of our system.

Between the publisher and the rendezvous node, the publisher will keep forwarding the event towards the rendezvous node until it receives and sends an acknowledgment back at him. The next stage (rendezvous to subs) is completely controlled by the rendezvous node. Before acknowledging the reception and process of the event to the publisher, the rendezvous node will track the event and send tracking requests to all its backups. Tracking means that the rendezvous will check its filtertable and create a map with all interested peers of that event. This way, once it receives an acknowledgment from every peer, the event will be considered successfully delivered. In the worst-case scenario, it will resend the event to the peers that have not confirmed yet.

Part of this mechanism is repeated at the peers downstream, where they keep a map with the interested peers, but upon receiving all acknowledgments, they forward their acks upstream. This structure also allows resending each event only to those who have not received it yet. The intermediate nodes have a passive role, being the rendezvous node, the manager of the resending process, the one that forwards the events back if their were not confirmed at all peers before a timeout.

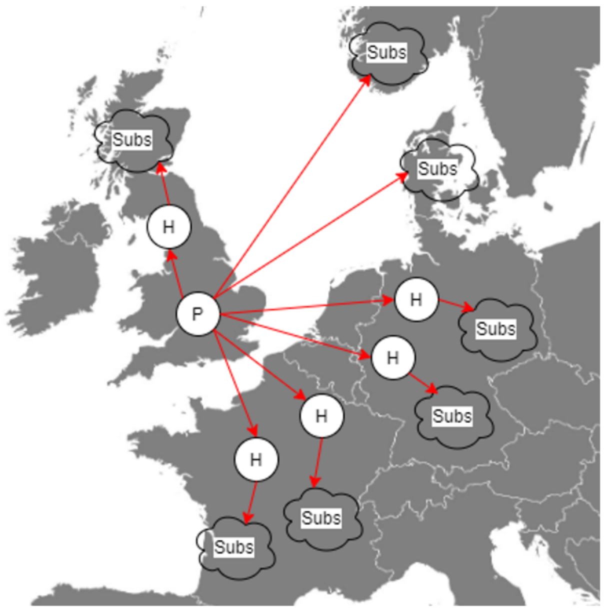


Figure 3.6: FastDelivery's geo-oriented hops

3.3.4 Protocol Maintenance

To maintaining the protocol working over time, there needs to be management of the filtering information. Because ScoutSubs does not allow unsubscribing operations due to a subscription being a filter that can be merged or omitted, we need to establish a refreshing routine.

Heartbeat Routine To ensure that all subscriptions on the system are still relevant, we force the subscriber to resubscribe to the predicates it is still interested in every period of time t. Every 2t time, every peer on the system will replace its main filtertable with a secondary one that was being compiled. A new secondary filter table is then created to receive new and resubscribed filters.

Besides providing an option to abandon a previous subscription and avoiding unbounded growth use of storage usage, it also allows the system to regenerate completely its fault tolerance capacity back to its f maximum every 2t time. The objective is to have the Kademlia DHT module synchronized with the pub-sub to provide full structural refreshing and restoration of its fault-tolerance properties.

3.4 FastDelivery Protocol

As previously mentioned, FastDelivery's main objective is to disseminate events as fast as possible, and provide a different kind of decentralization that comlements ScoutSubs. To do so, we need to escape IPFS' overlay structure and in part centralize the event dissemination at the publisher. The publisher could still provide events to the subscribers via ScoutSubs but would have to manage a group

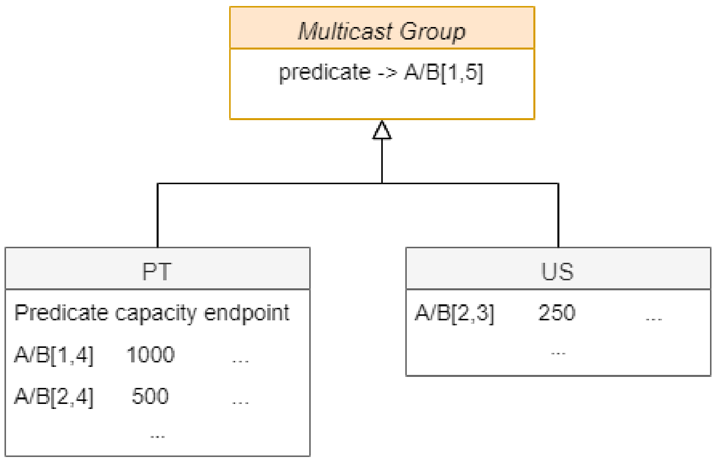


Figure 3.7: Multicast group main structure

of premium subs, to which it sends events directly or in 2 geographically oriented hops (user hops, not physical hops) as shown in Figure 3.6.

These publishers should have more computational and network capacity than the average publisher to manage all the subscribers. Premium subscribers need to provide the publisher their endpoint, location (Region/Country), and resources (network and CPU-wise). For a subscriber to access this service, a contract can be established between both parties. This last part is outside our project's scope but relevant to keep in mind.

Motivation Our motivation with designing this protocol alongside ScoutSubs was to reflect when a more centralized approach to disseminate information is the best option. In this case, when a subscriber is not searching for a topic/content of a publication but a particular publisher with a certain reputation or popularity, this alternative becomes a better option. Because the goal of this project is to design a decentralized content-based pub-sub over p2p, we decided to develop a really simple protocol for reflecting the mentioned point.

Design To design this publisher-centered protocol, we decided that a premium publisher can have one or more multicast groups. A multicast group is a structure a publisher manages containing its interested subscribers and their subscriptions predicates. Each multicast group is represented by its publisher Id and the group's predicate (e.g.: soccer/ajax or apple/france/price[0,1]).

Premium Publisher's Role To manage the multicast group's subscribers and recruit them if the publisher needs assistance, we decided to group subscribers, as shown in Figure 3.7. We group them into regions, ordered by capacity so that once one gets too many subscribers, the most powerful one gets recruited to help the publisher.

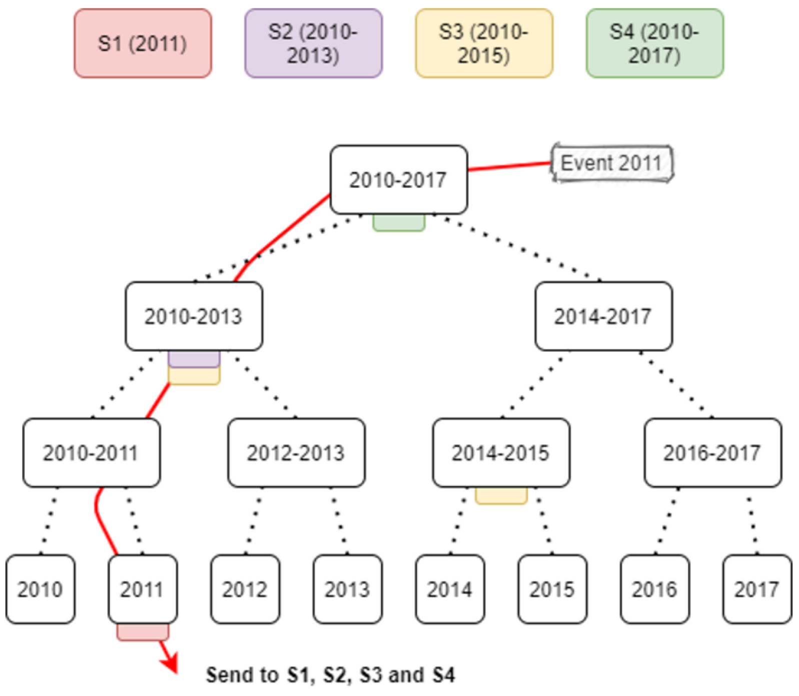


Figure 3.8: Range attribute tree

For organizing the subscribers' predicates, we saved their subscriptions in a simple list in the case of all predicate's attributes being of the topic type. If there are any range type attributes, we will use a binary tree to organize the subscription. As we can see in Figure 3.8, this tree we called range attribute tree allows the publisher to know which are the interested subs by doing a depth search of the tree. If the multicast group's predicate has several range attributes, he would need the same number of range trees and intercept their query results.

Helper's Role Upon subscribing, the subscriber needs to send its location, endpoint address, and capacity to the premium publisher. For the sake of keeping the protocol simple, capacity is the number of subs a subscriber can help the publisher manage. After agreeing to help the publisher, the helper will only support the structure that manages the subscriptions' predicates to forward the publisher's events to the interested subs. The support structure is a list or range trees with the subscribers delegated by the publisher.

Advertisement In terms of advertising a publisher's multicast group, we use advertisement boards at the rendezvous nodes of the attributes of the group's predicate. This way, for a sub to discover the premium publishers, it only needs to query the closest (XOR Distance) rendezvous of one of its subscription predicate's attributes to know their endpoint. A publisher may also prefer to keep its endpoint address private. Therefore, we partially centralize event dissemination for the sake of efficiency but advertisement and discovery remain decentralized for flexibility and scalability.

Advantages Even though this protocol is centered at the publisher and cannot provide a global view of event content, it is still a p2p content-based approach to pub-sub. Some other advantages of this protocol are:

- **Reliability:** it is much simpler with this protocol to guarantee that all subscribers receive events.
- **Resource consumption:** because all the actors in this protocol are interested and benefiting directly from their contribution, this protocol is fairer regarding resource consumption. It also provides the lowest bandwidth consumption possible and computational resources at the subscriber's side.

Summary

In this chapter, we presented our two pub-sub protocols: ScoutSubs and FastDelivery.

In ScoutSubs, we offer a pub-sub capable of providing a global semantic-based layer where all pub-sub users can participate. We described how we route events and subscriptions through the network, manage to tolerate nodes' failure, and maintain our system's state and fault-tolerance properties.

In FastDelivery, we offer, in addition, a simple protocol complementary to ScoutSubs where the publisher has a central role in its events dissemination. We describe the structure the publisher and its helpers have to assist them in forwarding events to the subscribers.

4 Implementation

Contents

Implementation

In this section, we will first present the technical aspects surrounding IPFS that molded our system's implementation at Section 4.1, and provide an overview of how both protocols were implemented at Section 4.2 and 4.3.

4.1 Technical Background

Before implementing the pub-sub system, we needed to view how IPFS systems and technologies are implemented. We needed to understand which programming language to use, how to integrate with IPFS, and which interface to use in order to connect peers.

Programming Language Currently, IPFS' main codebase has been standardized to be used by other p2p applications. This attempt to provide standards for all p2p applications led to the creation of libp2p [35]. So far, libp2p is implemented in golang, javascript, and rust. Due to my lack of experience programming in all these languages and the need to be familiar with golang to create a testground testbed, we have chosen to implement our pub-sub in golang.

Kademlia Module To implement our pub-sub module, we need to have access to IPFS' routing system. Particularly, we need to know a node's neighboring peers (Id and endpoint).

As mentioned before, libp2p (IPFS code library) has a golang implementation, more precisely, it has a golang implementation of the Kademlia module, the go-libp2p-kad-dht [39]. We will use this module as the routing base for our pub-sub to inherit from Kademlia [8] the property that every node can reach another one in log(n) hops up to a churn rate.

gRPC The remote interface we will use is gRPC, although this was not a choice more a requirement to integrate with IPFS. The main feature of gRPC is to provide a remote interface between applications running on different types of machines and written with different programming languages, which is IPFS' example.

Shortly, this tool generates code that handles the remote calls depending on the coding language by reading a simple file containing the RPC's headers with the description of its inputs and outputs. These inputs and outputs types are restricted to avoid serialization problems between the different programming languages.

4.2 ScoutSubs

This protocol is by far the most complex to implement between the two. We first present at Section 4.2.1 how many variants of the protocol we implemented, presented an overview of the protocol at Section 4.2.2, and share particularities of the rendezvous role at Section 4.2.3.

4.2.1 The 4 different protocol variants

Upon building ScoutSubs' protocol, we would like to analyze the impact some of its mechanisms would have on the system. So we decided to develop four different variants:

- **Base-Unreliable:** this variant does not contain the redirect and tracking mechanisms. Another relevant aspect of the unreliable variants is that the event dissemination starts at the publisher, meaning that an event in the worst-case scenario is delivered in log(n) user-hops.
- **Base-Reliable:** in this variant, there is a tracking mechanism at the rendezvous nodes, assuring that reliable event delivery. Although in this case, events take $2 \times \log(n)$ user-hops to reach the subscriber because the rendezvous node is the responsible authority of event dissemination, forcing all events to first reach it.
- **Redirect-Unreliable:** at this variant, we try to measure the impact of the redirect mechanism but not assuring reliability.
- **Redirect-Reliable:** here we combine all mechanisms we developed.

During our evaluation stage, we will analyze the benefits and downfalls of each variant by measuring and comparing each variant's performance.

4.2.2 An overview of the codebase

In this section, we present how we implemented our system [13], explaining first what we use from IPFS' Kademlia DHT implementation. Then we continue with the predicate and filtering modules and start addressing what actions a pub-sub user can perform to publish and receive events to how the protocol works/cooperates with the other peers to guarantee these services. In Figure 4.1 we present the

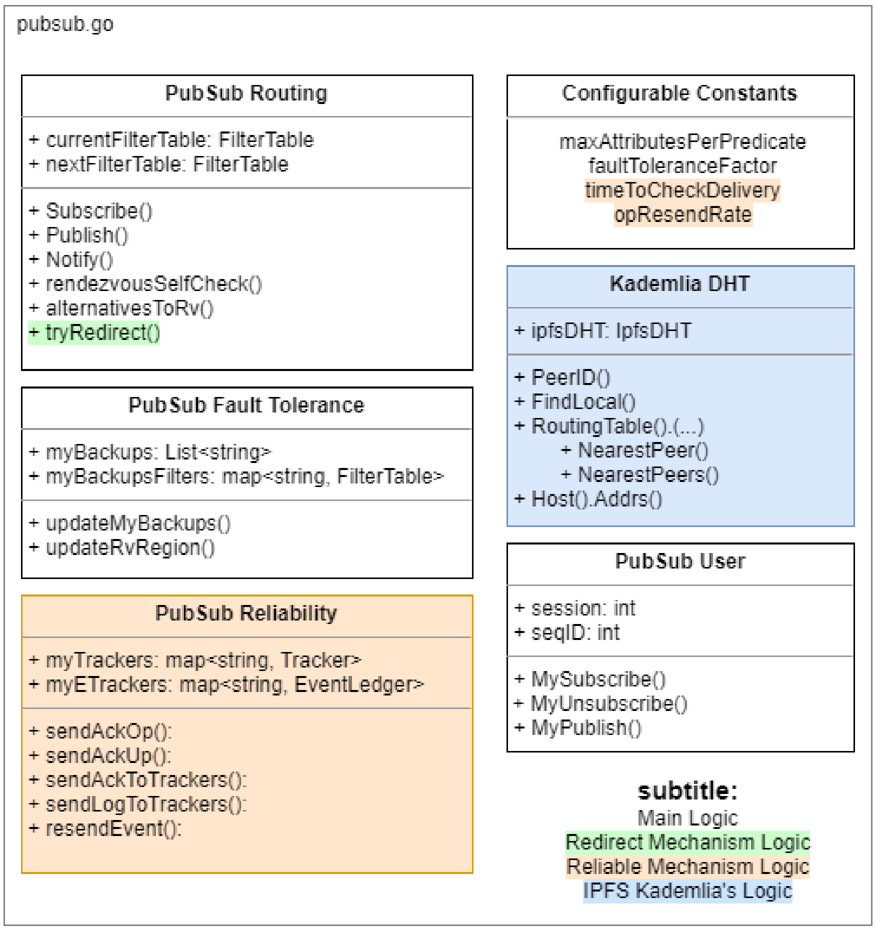


Figure 4.1: Main PubSub Logic

overall structure of our application code related to ScoutSubs, and in Figure 4.2 we present a structured view of the filtering and predicate logic. At the exception of the Kademlia Module, which represents the functions we use from the go-libp2p-kad-dht implementation [39], the remaining functions were all implemented for this project and are present in our codebase [13].

Kademlia DHT Our pub-sub system will operate over an IPFS-like network, meaning that we need to integrate its routing and peer identification systems with ours, allowing our pub-sub to work at any IPFS node. We identify any peer in our pub-sub system by the Id it inherited from its Kademlia DHT instance. The operations we use from Kademlia are:

- **PeerID():** returns the peer Id of himself.
- **FindLocal(p):** checks locally for peer information (such as endpoint address) from the peer with Id = p.
- **RoutingTable().NearestPeer(key):** returns the closest node he knows to a key. This method is

used to discover the rendezvous node of an attribute by converting that attribute into a key (hash) and check which neighboring peer is closer to that key (XOR-Distance).

In this stage, we did not implement this as a DHT protocol because this system is centered on performance issues, and so it needs to be hardened before being integrated at IPFS. This way, the pub/sub uses a different port than the DHT (details on codebase's file utils.go).

Predicate Everything in the system works around the notion of the predicate. The predicate is how subscribers expresses the subset of events they are interested in and how the publisher describes the events it produces. Here we opted to control the maximum number of attributes each predicate in the system can have, which can be set when initializing the pub-sub instance (like the other constants). The operations this module allows are:

- **SimplePredicateMatch(eventPredicate):** this function returns true if the event's predicate is contained by the subscription filter and false in the other case.
- **TryMergePredicates(p2):** this operation allows the possibility of merging a predicate with a second predicate. For example stock/apple/price[10,15] and stock/apple/price[12,20] can merge into stock/apple/price[10,20] that encompasses all the events of the two previous filters and none other unnecessarily. Merging filters can minimize the number of filters a node needs to analyze when receiving a subscription or event and prevent the propagation of redundant filters towards the rendevous.

Besides these two operations and some auxiliary ones, we have the NewPredicate(rawPredicate) constructor and the ToString() function that pass a string into a Predicate and vice-versa. These operations come in handy because RPCs with gRPC only use basic data types, meaning we will send Predicates in string format. Currently, our system supports attributes such as Topic and Range Query, but others can be added, or even the entire logic may be replaced by an alternative way of representing event content on subscriptions and events.

Filtering All of the pub-sub system's routing data is concentrated at the filtertable, meaning all received filters from neighboring peers. A filtertable has one entry for each neighboring peer, identified by that peer's Id and with a RouteStat. The RouteStat has all filters received from that peer, its backup nodes, and endpoint address. Because this infrastructure is read and written by several threads (multiple events arriving and subscriptions), each RouteStat entry has a lock. Currently, the matching mechanism between the events' and subscriptions' predicates is implemented by checking each filter at each RouteStat entry of a node's filtertable. It starts by analyzing the smaller filters (those with one attribute) so

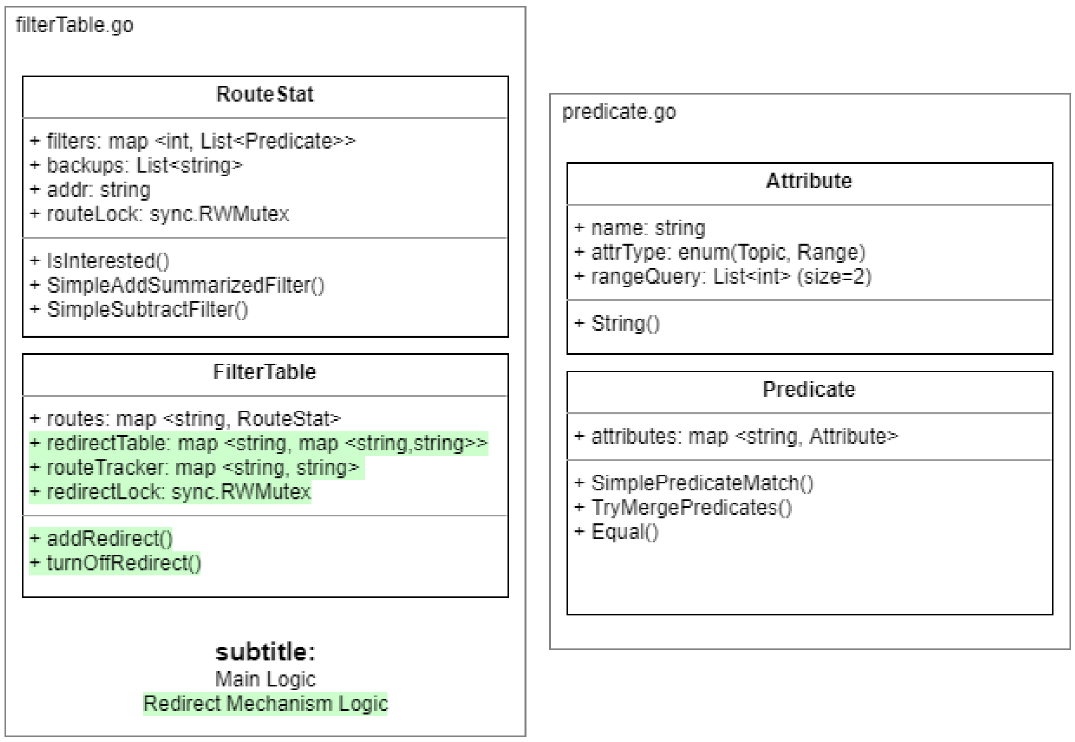

filterTable.go
RouteStat
+ filters: map <int, List<Predicate>>
+ backups: List<string>
+ addr: string
+ routeLock: sync.RWMutex
+ IsInterested()
+ SimpleAddSummarizedFilter()
+ SimpleSubtractFilter()
FilterTable
+ routes: map <string, RouteStat>
+ redirectTable: map <string, map <string,string>>
+ routeTracker: map <string, string>
+ redirectLock: sync.RWMutex
+ addRedirect()
+ turnOffRedirect()
subtitle:
Main Logic
Redirect Mechanism Logic
predicate.go
Attribute
+ name: string
+ attrType: enum(Topic, Range)
+ rangeQuery: List<int> (size=2)
+ String()
Predicate
+ attributes: map <string, Attribute>
+ SimplePredicateMatch()
+ TryMergePredicates()
+ Equal()

Figure 4.2: Filtering and Predicate Logic

that the first match stops the entire matching process, and in the worst scenario, it will have to check all filters before ignoring that route. The functions at a RouteStat are:

- **IsInterested(eventPredicate):** this function checks if that peer is interested in receiving an event by verifying if any filter at the RouteStat contains the event's Predicate, returning true once one contains it and false otherwise.
- **SimpleAddSummarizedFilter(predicate):** it is used to insert a filter in a RouteStat and verifies if there is already a filter that contains this one or the possibility of merging with another one to minimize the number of filters.

The SimpleSubtractFilter() is mainly used to erase a personal subscription on its subscriptions RouteStat. Besides this, the Redirect variants also possess additional structure to support shortcuts on the event dissemination. The additional filtertable fields are:

- **redirectTable:** this table stores the valid redirect options available. The storage is made by route (interested peer), rendezvous, and shortcut endpoint address.
- **routeTracker:** keeps track of how many subscriptions were received at the same rendezvous from different peers. Information's storage is made by rendezvous and the peer Ids of all the nodes that subscribed towards it. This way, a shortcut is deactivated once this value is higher than 1.

User Operations These are the only functions a user or application should use from our pub-sub system.

- **MySubscribe(rawPredicate)** this operation's goal is to inform the network that this peer is interested in a subset of events represented by a rawPredicate. If this peer is not the rendezvous of any attribute of the rawPredicate, it would need to forward the subscription towards the closest rendezvous node.
- **MyPublish(data, rawPredicate)** this operation allows a node to publish an event with content=data and predicate=rawPredicate. Depending on whether the node is a rendezvous of an attribute of rawPredicate, it will check its filtertable and forward downstream besides forwarding towards all remaining rendezvous nodes of the rawPredicate's attributes.

PubSub Routing and Fault Tolerance Two operations a pub-sub user can use are MySubscribe and MyPublish, but the core of the protocol lies in these three remote functions.

- **Subscribe(sub):** this RPC is executed by all peers between the initiator with MySubscribe() until it reaches the pretended rendezvous. At each intermediate node, it leaves a filter at its filterTable. It also propagates the added filter to its backups by using the updateMyBackups() function. When it reaches the rendezvous node, the process is finished, and the rendezvous updates its zone using updateRvRegion().
- **Publish(event):** this RPC is executed by all peers between the initiator of MyPublish() and all the event attributes' rendezvous. In the non-reliable variants, this function checks the filtertable at the intermediate nodes and starts disseminating the events downstream with the Notify() RPC. While the reliable ones only start the downstream process at the rendezvous to simplify the acknowledgment chain.
- **Notify(event):** this RPC is called upon having an interested peer on an event. If a peer wants to notify one of its interested neighbors, but their connection is not working, it needs to use one of its backups (saved at that peer's RouteStat) by sending a Notify() RPC with a backup flag activated. This way, the receiving node will check the backup's filtertable and not its own.

The system's fault tolerance can be switched by configuring the faultToleranceFactor constant. This constant is the number of failed consecutive peers the system can tolerate in a refreshing cycle or epoch (the cycle's period is also configurable).

Implementing Reliability At the reliable variants, we assure that each event is delivered to all working subscribers. We built an acknowledgement chain from the subscribers to the rendezvous to which they forward their subscriptions. The main functions used are:

- **sendAckOp():** once the rendezvous receives an event, it warns the publisher that the event already reached it and will eventually be delivered to all working subscribers. This premise means, before sending an ack back to the publisher, it needs to forward the event to the interested neighbors and send an eventLedger to all trackers (sendLogToTracker() job).
- **sendLogToTracker(eventLog):** to keep track of who already received the event and who did not, we build an eventLog containing the event and all interested peers. This eventLog is sent to all trackers, being the tracker leader the current rendezvous.
- **sendAckUp():** sends an ack towards the rendezvous who forwarded the event downstream. This ack can be made if the receiving peer does not have neighbors interested in the event or if all the interested ones have already sent AckUp() RPCs. For this at a Notify() RPC, each intermediate node builds an eventLog, that once all interested peers have filled the Log with acks, the node sends its AckUp().
- **sendAckToTrackers():** it is used by the rendezvous to propagate the received ack to all trackers.
- **resendEvent():** once timeToCheckDelivery passes and not all acks were received, the leading tracker (rendezvous) uses this function to resend the event to all unacknowledged peers. These peers on their side also send the event only to those that still did not send an ack.

The tracker logic is similar to all the eventLogs peers have, but the leading tracker is the one that periodically triggers resendEvent() to all unacked or partially unacked events. Tracker logic was implemented in file tracker.go, not for being independent of the main logic, only to not turn pubsub.go into a very large source file.

Refreshing The system has a processLoop that, besides executing functions that send subs or events concurrently, has two tickers at t and 2t time (refreshing cycle or epoch), trigger the resub of all personal subscriptions and replace the filtertables, respectively. This way, we assure that the system properties, such as its fault tolerance, are restored.

4.2.3 The details of the Rendezvous Role

For a node to be considered the rendezvous node, it executes the rendezvousSelfCheck() function that compares if its Id is closer to a key than the closest node it knows.

One good consequence of this is that the protocol is resistant to network partitioning, meaning that the rendezvous role will be self-assigned. But during our implementation stage, we saw that this is not enough for the rendezvous role. There are scenarios that can happen where a new node joins and

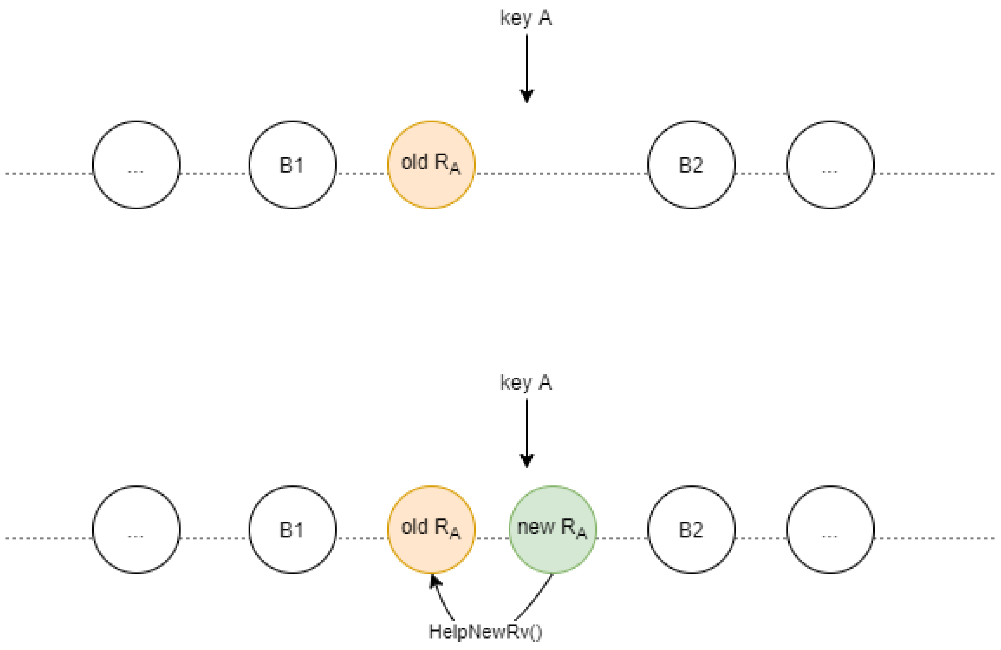


Figure 4.3: Detail of new RN appearing and the need to check an old RN or backup

becomes closer to the rendezvous than the current rendezvous node, or a rendezvous node crashing and the next rendezvous in line being too fresh.

To have a correct rendezvous role, we added the idea of age to the system. A valid rendezvous is the closest node to the key that already lived for at least a refreshing cycle or epoch (renewed its filtertable). This age requirement is needed so that we assure that all subscriptions were received by the rendezvous.

If the rendezvous in question is not old, it needs to find one old backup of the previous one, by using HelpNewRv function as shown in Figure 4.3.

4.3 FastDelivery

In this section, we will briefly present FastDelivery, starting from the MulticastGroups' logic shown in Figure 4.4.

MulticastGroups As mentioned at Section 3.4, the multicastGroup is the center of FastDelivery. Each multicastGroup contains:

- **predicate:** it identifies the subset of events this multicastGroup forwards. It is assigned to the group upon its creation.

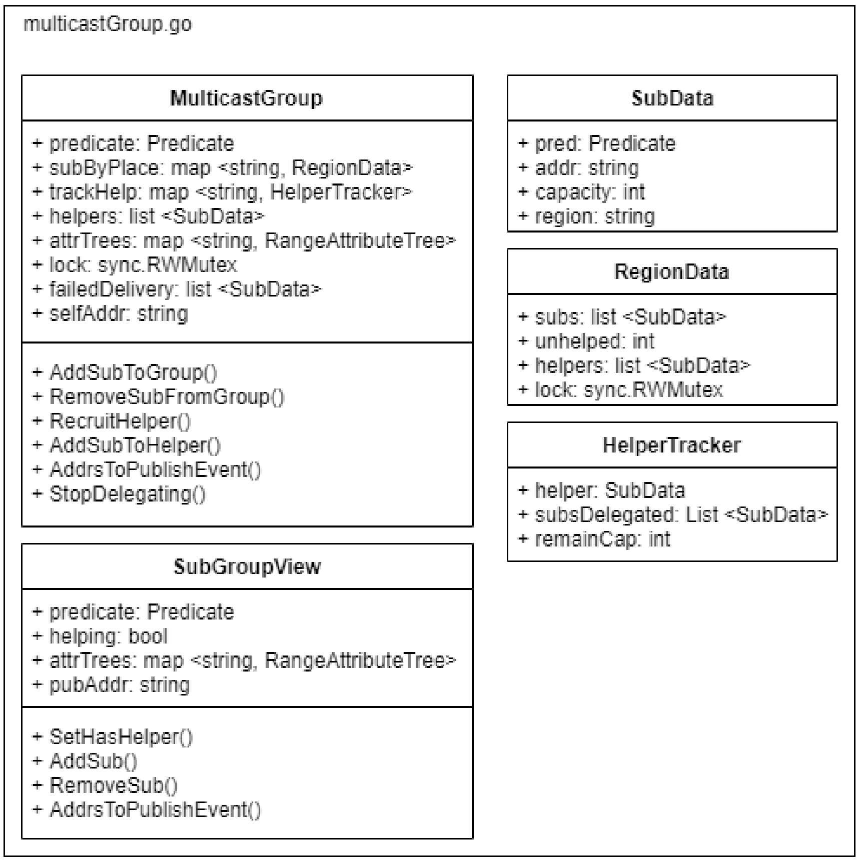


Figure 4.4: Multicast Group Logic

- **subByPlace:** this structure organizes subscribers by RegionData. Each RegionData stores subscribers by capacity at the sub's list, at the helpers' list, those helping the publisher, and it is kept track of how many users are unhelped in that region.

- **trackHelp:** it is the list of all regions' helper info. Each helper has a HelperTracker that contains the subs it is helping and its remaining capacity.

- **attrTrees:** these are the range attribute trees that contain all subscriptions of sub's/pub's responsibility so that for an event, the publisher/helper can rapidly fetch all the addresses of all interested subscribers.

- **failedDelivery:** it is the list with the subscribers that were supported by a failed helper node. The publisher reintegrates the previously delegated nodes and resends the events to the nodes in question.

Some of the main operations used around the MulticastGroup are:

- **AddSubToGroup(rawSubData):** in this function are sent as parameters the sub's data (predicate,

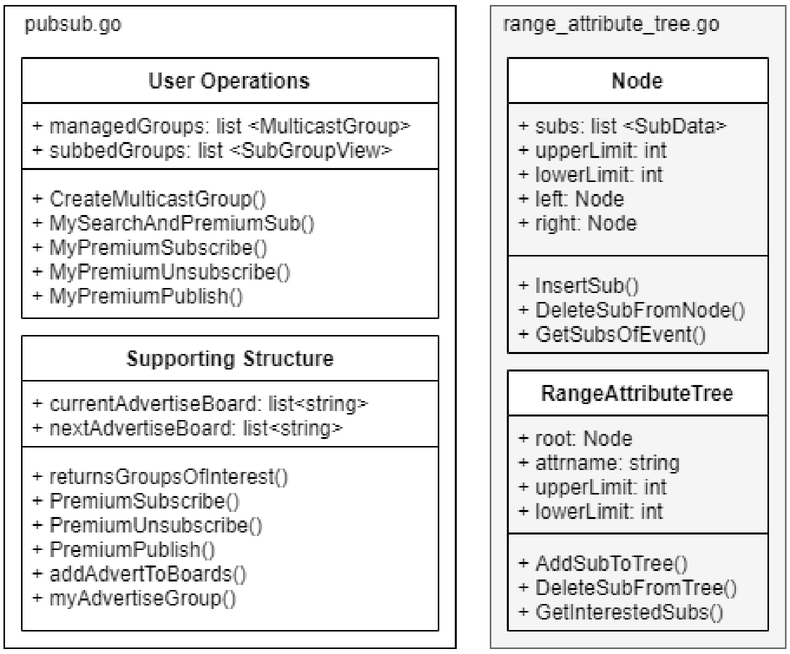


Figure 4.5: Range trees and auxiliary logic

address, capacity, and region) for the subscription to be inserted at its region, or if necessary, it is delegated to a helper or a new helper is recruited.

- **RemoveSubFromGroup(sub):** it eliminates a sub from the group independently if it is a helper or helped by another node.

- **RecruitHelper(helper, subs):** it recruits a helper and delegates to it some subscribers.

- **AddrsToPublishEvent(predicate):** it returns the subscribers' addresses of those interested in an event.

Every node creates a SubGroupView for each MulticastGroup it joins. The SubGroupView is later used to support the publisher if this decides to recruit it as a helper. A helper will only maintain range attribute trees to know which delegated subscribers are interested in an event.

The dissemination infrastructure was already dealt with, but there needs to be an advertising mechanism unless we rely on other ways of sharing the premium publishers' MulticastGroups or leave them to the publisher's responsibility. Either way, the publisher in FastDelivery has the option of advertising its MulticastGroup at the rendezvous of its group predicate, to ensure complete reach. Here are the operations the users can use:

- **CreateMulticastGroup(rawPredicate):** it is used to create a MulticastGroup for events represented by a rawPredicate. Besides creating, this function also calls myAdvertiseGroup() function that advertises the group to the rawPredicate attributes' rendezvous.

- **MySearchAndPremiumSub(rawPredicate):** this function sends a query to the rawPredicate's closest rendezvous and receives the interesting multicast groups. Knowing the premium publishers and their addresses, it calls MyPremiumSubscribe() to each one of them. The MyPremiumSubscribe function can also be called separately if the subscriber already possesses the necessary information about the publisher. In this case, it is the address and the group's predicate.
- **MyPremiumPublish(rawGroupPredicate, eventData, rawEventPredicate):** with this, a premium publisher publishes an event in one of its multicastGroups.

The logic regarding the range tree attributes is separated from the rest and can be replaced by any other structure that can store and return the subscriptions that match certain predicates.

Summary

In this chapter, we address the technical details of implementing our system and describe our codebase structure so others may easily understand and extend it.

During our implementation stage, we tried to make modules like ScoutSubs' filtering and predicate and FastDelivery's range attribute trees independent from the remaining logic, allowing other developers to change/improve our system on those areas.

5 Evaluation

Contents

Evaluation

In this Section, we first explain our evaluation approach in Section 5.1, describe in Section 5.2 the testing environment used, and finish by presenting and analyzing the results of our pub-sub's performance in Section 5.3.

5.1 Methodology

In this Section, we present the evaluation methodology used to assess our system's performance and correctness. In Section 5.1.1, we mention the main metrics we chose to analyze from our system's performance, and in Section 5.1.2, we present the scenarios/workloads we will use as the baseline for our experiences and their goal.

5.1.1 Metrics

We will use the following metrics to evaluate the performance, correctness, and peer-load of our system:

- **Event Latency:** is one of the most relevant metrics, allowing us to analyze the speed at which events are exchanged between publishers and subscribers.
- **Memory usage:** how much additional memory an IPFS peer spends with our pub/sub approach.
- **CPU usage:** is used to evaluate if the proposed solution scales well with increased traffic (subscriptions, publications) and network size (number of participants). We assess this by tracking the CPU user-time spent during our experiences. This metric evaluates mainly the filter matching and summarizing processes.
- **Missing Events:** evaluates if the system is doing its job correctly by checking if our pub-sub is delivering all events to their interested subscribers on the network.
- **Duplicated Events:** we can also analyze how many events are reaching repeatedly (i.e. redundantly and wastefully) the subscribers.

5.1.2 Workloads and comparable systems

In this section, we will mention the different sets of experiences we will do to understand performance fluctuations that might occur depending on the system's state and users' behavior.

Variant's scenarios First, we built different scenarios that, besides assessing our system performance, could provide us with insight on the results/performance of the different variants of our pub-sub.

- **Subscription Burst:** after the system reaches a balanced state, we should trigger several subscription requests and see how event delivery performs during this interval.
- **Event Burst:** after the system reaches a balanced state, we should try to generate several events in almost all peers and see what happens to the system's performance. This scenario may occur when a global event occurs in the world (breaking news or major discovery) that may generate lots of events in a short period.
- **Fault effect on performance:** analyze how the system performs when nodes fail.

Assessing reliable performance After assessing the performance of our pub-sub variants, we separately evaluate our best reliable variant under a stricter evaluation scenario. This experience's goals are:

- **Performance vs replication:** here we will test our system with different values of *FaultToleranceFactor*, so we may understand how performance is influenced by the overhead associated with increasing the system's resilience.
- **Performance vs subscriptions:** while assessing the topic above, we will evaluate the system in several stages. At each stage, the subscribers will perform one more subscription, and the publishers publish an event. Here we could understand how our system handles an increasing number of subscriptions.

Regarding relevant systems to compare performances, besides our variants, we only have in IPFS' universe topic-based approaches. Although these can also be used as a means of comparison, even knowing that our system results should be worst because it provides a finer event granularity.

The variants will be independently evaluated to assess if the tracking and redirect mechanisms are worth adding to our system. The hardest to measure is the tracking one because we need to interrupt an intermediate node mid-transmission, which is hard to achieve in any but very small deployments (and for that less relevant), and even harder to assess the outcome. We will focus instead on the additional load the tracking mechanism adds to the system.

5.2 Test Environment

5.2.1 Testground

To test our pub-sub system, we used a tool created by the IPFS community to test applications designed for its environment. This tool is called Testground [14] and allows us to test our system with multiple libp2p nodes (IPFS nodes). Its main objective is to test applications in scale before they are deployed/used alongside IPFS. Test runs may reach the hundreds of nodes using docker at a local machine and thousands with Kubernetes.

Testground allowed us to execute nodes as a container, isolating the IPFS nodes being tested. It also handles networking between containers and allows us to exchange information between instances in a separated network, permitting us to sync the node instances if necessary. Network properties between containers such as latency, jitter, and bandwidth are also configurable.

5.2.2 Developing Testbed

We developed several test cases [15] running over testground to test our system in the different scenarios. Before building the testing scenarios themselves, first, we need to initialize the libp2p nodes and bootstrap a working network between them to have our IPFS network.

Simulating an IPFS network As mentioned before in Section 4.1, we use IPFS' Kademlia module as routing base for our pub-sub [39]. To simulate an IPFS network we adapted the existing test plan provided by libp2p [35] that initializes a network of libp2p nodes with a Kademlia DHT and provides several bootstrapping mechanisms to connect the nodes.

After ensuring the correct functioning of the IPFS network, we initialize a pub-sub instance at each node. Then we proceeded to implement test cases to simulate each pretended scenario:

- **Normal Scenario:** in the first stage, subscriber nodes subscribe to a subset of events. In a second stage, publisher nodes publish events of the interest of the subscribers. Here, publishers are around 1/10 of the nodes, and subscribers are the remaining ones.
- **Subscription Burst Scenario:** here, we use the same proportion of subscribers/publishers as in the normal scenario, and in a second stage, besides the publisher publishing the events, we will have the subscribers making new subscriptions simultaneously. The subscriptions made do not affect a events' forwarding since their objective in this scenario is to load the network and peers.
- **Event Burst Scenario:** here the subscribers are also publishers, meaning that 9/10 of the peers will publish events of interest to a big part of the network. To have a higher load, we may execute

this scenario after the subscription burst one to have more filters at the peer's filterTables. This scenario is our system's stress test.

- **Fault Scenario:** in this scenario, before a publishing round, f (*FaulToleranceFactor*) peers are forced to crash. Besides that, it is similar to the normal scenario.

- **Performance vs replication vs subscriptions:** for our final test with the redirect-reliable variant, we divided the test into four stages, and in each one, a subscription and a publishing round take place. We will measure the CPU and memory used at each stage and record the latency of the events received. We will perform a set of runs for each different faulttolerancefactor (from 0 to 3).

Collecting Metrics To measure the CPU and memory usage during each scenario, we used a tool(psutil for golang [40]) for capturing both memory and CPU time used in each container. These metrics are relative to all user applications and not only our pub-sub's resource consumption but inside these containers there are no other applications running. That is why our scenarios take few seconds to run, so we can have a closer estimation of how many of these resources were used.

At the end of each scenario, we record all the resulting metrics. Besides CPU and memory, we also record event latency and correctness metrics, such as duplicated and missing events. To analyze and compare results, we used a python script to compile graphics with the data obtained by the test runs, comparing all metrics between each variant at each scenario.

To record and obtain the metrics we save per event received its content to detect missing/duplicated events and two timestamps, one for the operation starting time and another with the finishing time. This last one is used to calculate the time it took an event to reach a subscriber or how much time it took to finish a subscription operation (only on reliable variants).

5.3 Results and Analysis

In this Section, we present the most relevant results of our pub-sub test runs. The following results were achieved using an Ubuntu Virtual Machine (VM) with 126GB of RAM and a 16-core CPU. Besides the graphics and data presented here, more detailed data of this and other experiences can be found on our results page [41].

In Section 5.3.1, we analyze our variants' results by comparing them side by side, and in Section 5.3.2, we analyze our system's performance when increasing its replication and the number of subscriptions each user has.

Variant	Normal	Sub Burst	Event Burst	Fault
Base-Unreliable	16.97%	16.97%	6.954%	16.67%
Redirect-Unreliable	11.61%	11.61%	4.611%	12.04%
Base-Reliable	10.72%	10.72%	12.33%	8.334%
Redirect-Reliable	9.822%	9.822%	6.728%	8.334%

Table 5.1: Percentage of duplicated events received when comparing with the number of supposed received events

5.3.1 Variant's testing

Here we will present the results of each variant tested through the same scenarios mentioned in Section 5.1.2. The goal is to analyze if the redirect and reliable mechanisms are working and if they are efficient. This test battery allowed us to analyze and correct some bugs our system had in the development phase.

Each test run in this Section had a 60 node network, and we programmed the network interfaces to have 1MB of bandwidth and 10ms of latency. We only tested with these nodes due to some complications we had to maintain our bootstrapped network with Kademlia stable. The testbed we used was adapted from a public testbed made by Kademlia's developers, but this was last updated more than a year ago, while libp2p [35] and testground's codebases are constantly being developed and changed.

To minimize the number of runs we merged all tests scenarios into one big test run where we can still analyze the different scenarios separately.

Correctness To know how correct our pub-sub is, we compared the duplicated and missing events by each variant at each scenario.

In all variants and scenarios, our pub-sub performed with 100% reliability and produced in all variants some duplicated events. The FastDelivery approach had perfect results for it manages its subscribers directly. We can see the amount of duplicated events received when comparing with the number of received events in Table 5.1.

Looking at the number of duplicated events between scenarios we can see in the normal and the subscriptions burst one, the value is the same. In both these scenarios, the produced events had similar content, and the network composition remained unchanged. We observe that duplicated events are not influenced by network traffic or race conditions on the locks of the filtertables since with more subscriptions being made, the number of duplicated events stays the same. At the event burst scenario, the percentage of duplicates is much lower, although that is due to the higher number of published events and not in a difference in the way our pub-sub handles them. This factor becomes clearer in the fault scenario because our network loses two nodes resulting in fewer events being recorded at peers

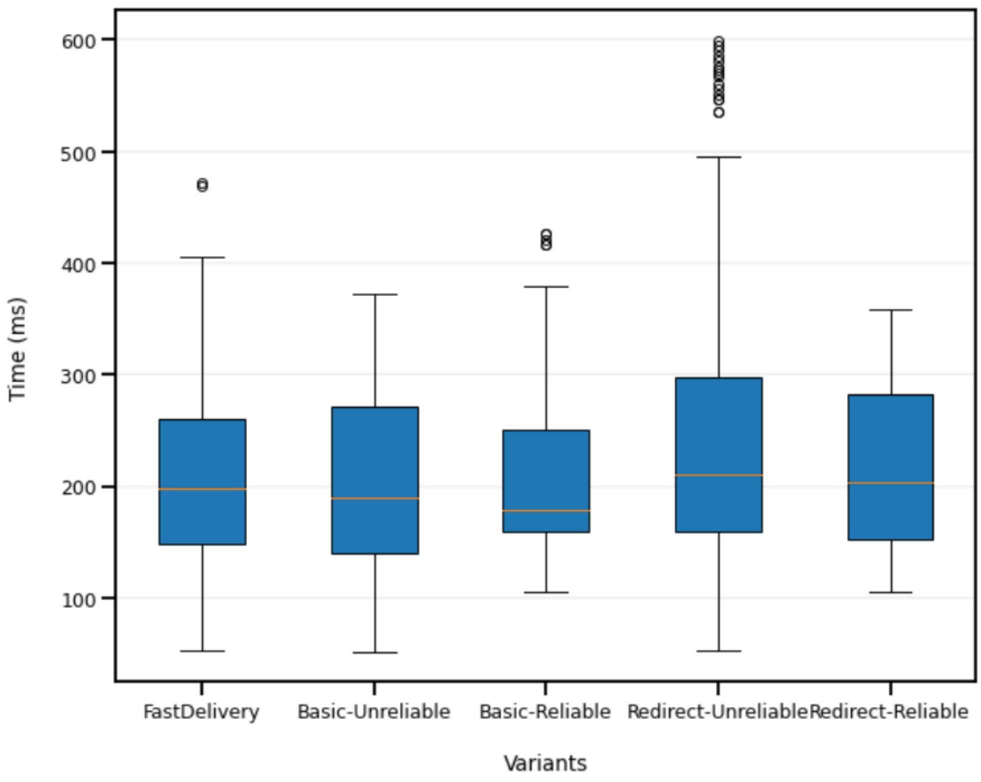


Figure 5.1: Event Latency - Normal Scenario

and so a relatively higher event duplicates percentage. Still, it also depends if these peers had a higher chance of receiving duplicates due to their Id and not because they failed.

Regarding the different variants, we don't observe any correlation that could lead us to any conclusion, since event duplication changes between runs due to different peers having different connections at each run. Our experiences used a 60 node network, and each node had around 30-40 peers, making communication mostly direct and redundant paths common. These redundant paths are the main reason for event duplication, and only with a 1k+ node network, we can confirm this because routing tables will not have more than 100-200 peers.

Event Latency The most relevant metric for the user regarding the system's performance, besides its correctness, is the time it takes to receive an event.

By observing the results from the normal scenario in Figure 5.1, we can see that the average event latency for the used network composition and configuration is from 200-250 ms. In the subscription burst scenario with the results presented in Figure 5.2, the average event latency is from 200-250 ms. In the event burst scenario with results presented in Figure 5.3, the average event latency is from 1780-2500 ms. In the fault scenario with results presented in Figure 5.4, the average event latency is from 200-270 ms. This results reflect that event delivery is independent of the subscriptions being made and

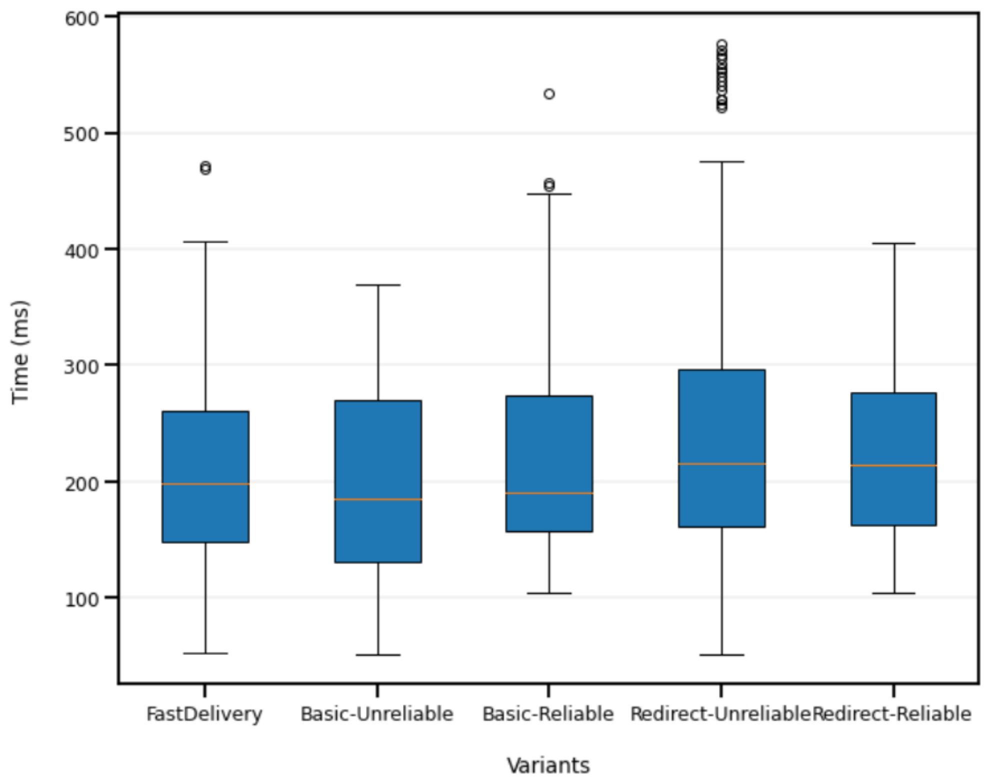


Figure 5.2: Event Latency - Subscription Burst Scenario

only dependent on the number of events being produced and delivered (depending on the number of interested subscribers).

Although, one case that saw us worrying was our pub-sub performance during stress, since it shows a scenario where the entire network is producing events and a big part of the network is also interested. One contributing factor might be the small dimension of our testing network, because IPFS' bootstrapping mechanism will try to fill its routing table buckets of his Kademlia instance, making the average number of connections of a peer in a network of 60-nodes around 30-40. This number of connections is not the ideal network constitution for IPFS's DHT, which for a network of hundreds of thousands of peers has less than a thousand connections. This factor reduces the workload distribution and filter summarizing and may benefit our normal scenario results but overload our system when under stress.

Looking at the results by variant, it is hard to draw conclusions on the scenarios with normal event production, but in the event burst scenario we start to see significant differences. In this scenario we can see that unreliable variants have substantially better results with events being delivered 400 ms faster in average compared with the reliable ones. This difference is due to the additional workload reliable variants have from managing the acknowledgement chains. Another detail visible in the more normal scenarios is that the unreliable variants, because they do not have to reach the rendezvous to start deliver events, have events delivered closer to the rendezvous (visible in the latency distribution

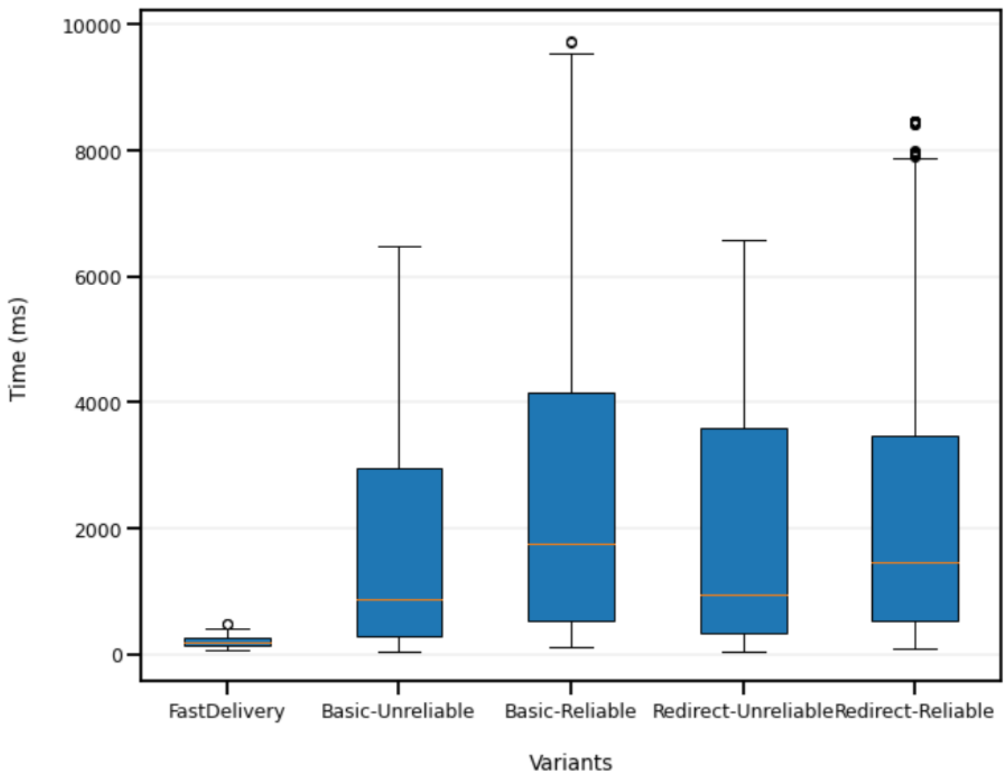


Figure 5.3: Event Latency - Event Burst Scenario

graphics by the lowest event latency values shown).

It is hard to understand the impact of the redirect mechanism in these experiences because all nodes are in 2-3 hops of each other, and events/subscriptions diversity is low due to the size of the network we were testing in.

Resource Consumption We finalize by looking at the memory and CPU consumption of our pub-sub during our test runs. In all test runs, higher memory usage is followed by a higher CPU time usage. We must also acknowledge that each scenario has different periods, as in the event burst scenario, where its testing period is around 12 seconds, and the normal one is less than half that.

When looking to the Table 5.2, we can see that the reliable variants, especially in the event burst scenario, have a substantially higher CPU and memory usage. The main reasons for this are the extra relation between tracker and rendezvous node and the management of the acknowledgement chains. In terms of consultations to the filtertable, these are the same as in an unreliable variant. As mentioned before, the impact of the redirect mechanism is not palpable with a small network, and so we cannot comment on its performance.

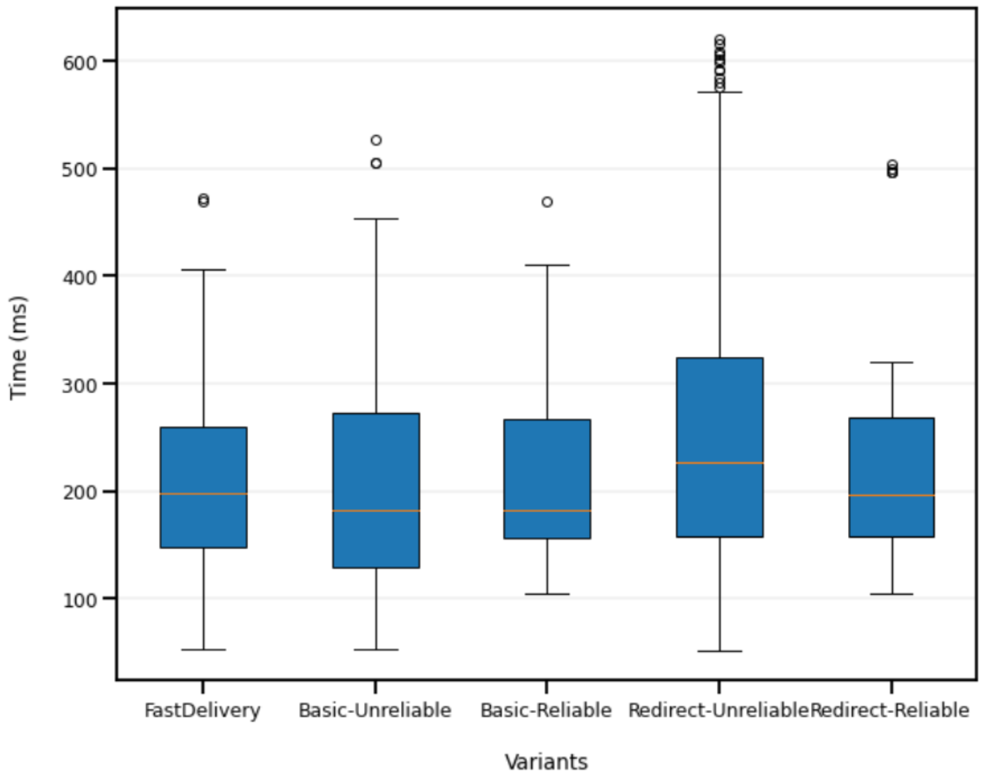


Figure 5.4: Event Latency - Fault Scenario

5.3.2 Replication Performance

After analyzing each variant of our pub-sub, we decided to test with our Redirect-Reliable variant how its performance changes if we increase the *FaultToleranceFactor*. We also took the chance to analyze how the system performs with an increase in the number of subscriptions each subscriber does.

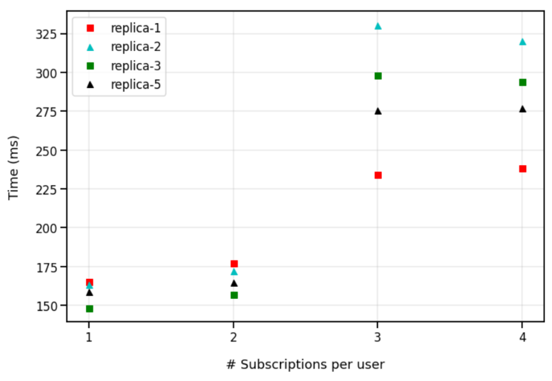
Because this experience is shorter we manage to keep Kademlia stable with 75-nodes and performed a test run for several values of *FaultToleranceFactor*. In this experience, we compare the results of a network with *FaultToleranceFactor* set to 1, 2, 3, and 5, increasing through the experience the number of subscriptions each sub does.

Event and Subscription latency We can start looking at the Figures 5.5(a) and 5.5(b) to analyze the results regarding the event and subscription latency, respectively.

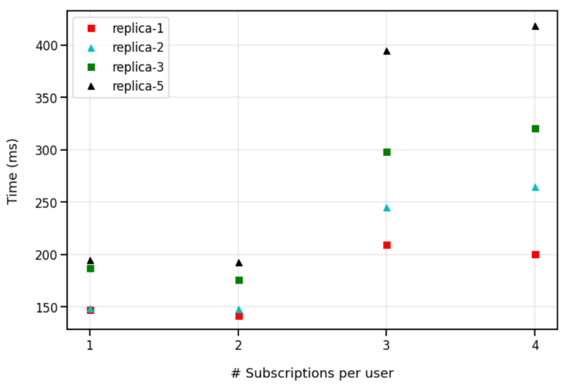
The results of our pub-sub subscription latency are as predicted since the bigger the replicated factor (*FaultToleranceFactor*), the longer it takes to subscribe. The same can be said of the number of subscriptions per sub since each subscribing operation needs to check all the filters of a filtertable entry. Filter checking is necessary to merge a subscription filter with others or ignore it (because one of those in the entry already encompasses it).

Variant	Normal	Sub Burst	Event Burst	Fault
Base-Unreliable	57.3 MB 4.01 s	36.0 MB 2.36 s	173 MB 11.45 s	19.5 MB 1.94 s
Redirect-Unreliable	117 MB 8.13 s	26.7 MB 3.82 s	179 MB 14.62 s	5.10 MB 2.15 s
Base-Reliable	99.2 MB 6.54 s	62.0 MB 3.88 s	310 MB 20.21 s	22.5 MB 3.46 s
Redirect-Reliable	108 MB 8.31 s	37.0 MB 3.61 s	268 MB 21.38 s	21.5 MB 3.84 s

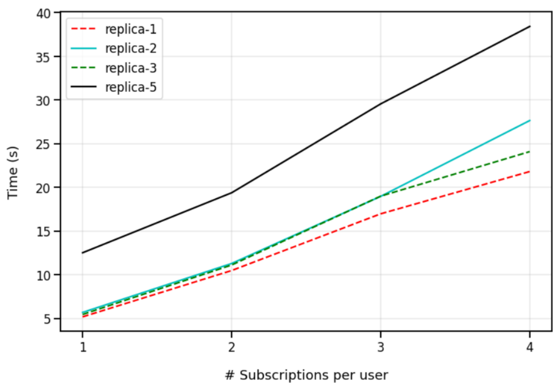
Table 5.2: Average Memory and CPU user-time used per node



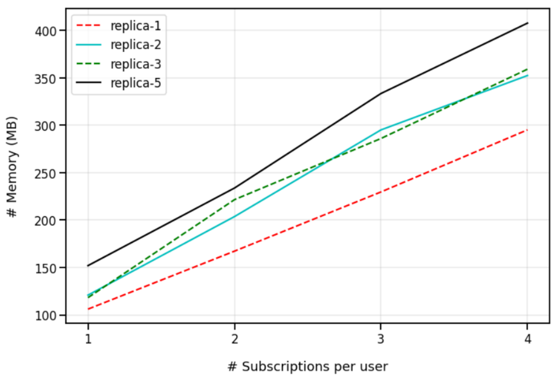
(a) Average Event Latency per replica factor at each stage



(b) Average Subscription Latency per replica factor at each stage



(c) Accumulated CPU user-time used for each replica factor



(d) Accumulated Memory used for each replica factor

Figure 5.5: Results of the replication experience

When looking at the event latency, we see that the correlation is not as strong as in subscriptions. A subscription needs to be sent to a node's backups, and they need to add and summarize the subscription filter. In an event's forwarding, the only interaction between the main path and the backups is at the rendezvous between the different trackers. The independence between fault-tolerance and event forwarding mechanisms in a non-failure scenario was built to achieve a faster event forwarding to the detriment of the subscription operation. As seen in the figures, the event latency difference is 40 ms maximum, and the higher replica factor may not have the higher latency, while in subscription, the difference can reach 80ms, and the bigger the replica factor, the bigger the operation time. Higher replication does not cause higher event latency because the last depends on the order of the filters in the filtertable. In the replica-5 test run, the most popular events by luck (were the first subscription to arrive the node) ended up at the start of a filtertable entry (opposite of what happen at replica-2). Because when receiving the event less filters had to be checked, since at the first hit (event matching filter) the event is forwarded to the route correspondent to that filtertable's entry.

CPU and memory usage We then analyze the resources used in this experience by checking Figures 5.5(c) and 5.5(d) for the CPU and memory usage, respectively.

Memory consumption is straightforward since it increases with the replication factor and remains constant with the increase of the number of subscriptions since their size is small. CPU usage does not increase with the number of subscriptions per user but tends to increase with the replica factor.

Regarding the scalability of our system, when analyzing our system's performance with the increase of subscriptions per user, we can confirm that in terms of resource usage, the system is scalable. When looking at the memory and CPU usage, we can see that they do not follow the increase in subscriptions as the graphic tends to a linear regression (being an accumulative graphic, it means that resource consumption is approximately the same at each stage). In terms of performance, we can see a change, although far from affecting the user perception of the event speed delivery, and may be reduced when running in a larger network with more spaced rendezvous and distributed workload.

Summary

During our evaluation stage, our main difficulty was to maintaining an IPFS network stable for a long time, especially with more than 40 nodes. Deploying a network in a cloud cluster should skip our struggle since the problems reside in connection limits inside a VM or single host.

Regarding the obtained results, we may conclude that our pub-sub presents good results in scenarios where not more than 10% of the network is producing information. Reliable variants, in scenarios of high event production, tend to have worst results than unreliable variants. One last positive aspect is that our pub-sub resource usage does not increase with the number of subscriptions on the system.

6 Conclusions

Conclusions

Our work presents a survey with the current state of the art in publish-subscribe, focusing on content-based systems and on p2p content distribution systems to then propose a pub-sub content-based system over IPFS p2p file-sharing system.

We analyzed some of the most relevant pub-sub content-based approaches in the field and some p2p content distribution systems to understand their defining characteristics. We took inspiration from some of Hermes [3] main features and took a closer look at Kademlia's DHT to know how we could take advantage of IPFS's current architecture. With that in mind, we designed a content-based pub-sub architecture that is aimed at enriching IPFS since all its current pub-sub approaches are topic-based.

We tried to fulfil our system requirements regarding scalability and decentralization IPFS demands and maintain the system as efficiently as possible. We implemented a testbed using Testground [14] to evaluate our system performance and resource efficiency.

We conclude after this work that decentralizing web infrastructure can have more benefits, but it is not a perfect solution. With that in mind, we presented FastDelivery to showcase where some centralization can be advantageous. We know that decentralizing a system leads to an increase in the system's algorithmic complexity, and security-wise although that is out of this work's scope. So we produced, to the best of our knowledge, a pub-sub that provides a global content/semantic-addressing layer over IPFS and showcased a simple publisher-centered approach. This one, even being simpler than ScoutSubs, is more efficient and useful for IPFS users when they are interested in both the source and content of the events.

Nevertheless, it is relevant having a global system that is not only physically content-based oriented (as in IPFS static-content-addressed system), but one system that provides a semantic-content-based approach. A system where information is not merely organized by physical content but forwarded through the web depending on its semantic content and users interested in it.

7

Future Work

Future Work

We suggest as future research, four main areas of improvement/enhancement of our content-based publish-subscribe system:

- **Resilience to malicious users:** As mentioned before, our work focuses on a strictly non-byzantine working scenario. To allow our pub-sub to work as part of IPFS and to held relevant responsibility, we need to assure that the system is resistant to any type of attack. Some measures that may be implemented are the use of IPFS keys to validate users, restricting the amount of operations a peer can do in each refreshing cycle to prevent Distributed Denial of Service (DDoS) attacks and have some mechanism of local reputation as in [12] that may de-prioritize misbehaving peers.
- **Attribute mapping:** One way to improve IPFS' performance would be by minimizing the number of attributes in the system. At the same time, it could also be designed a mapping function that could assure attributes are equally spread through IPFS' addressable space.
- **Self-describing events:** Instead of relying on the publisher to assign a predicate to an event, it should be used an event classifier. Classifying text-based events is easy when choosing for example the X most relevant terms by processing it with a simple perceptron vector properly trained for certain events. But to provide a full predicate with range queries it is much more complex and outside of this project's scope. The problem is even bigger regarding images and video content, and is one of IPFS' biggest limitations, since it organizes files by hashing its binary content, without proving a meaningful identifier.
- **Filter matching:** there is always space to improve how we store filters and how we match, and merge them. In our work we rely on simple matching techniques that can be improved.

Bibliography

[1] C. P. Fry and M. K. Reiter, "Really truly trackerless bittorrent," *School of Computer Science, Carnegie Mellon University, Tech. Rep*, pp. 06–148, 2006.

[2] "Ipfs documentation," https://docs.ipfs.io/, Aug 2020.

[3] P. R. Pietzuch and J. M. Bacon, "Hermes: A distributed event-based middleware architecture," in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. IEEE, 2002, pp. 611–618.

[4] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi, "Meghdoot: content-based publish/subscribe over p2p networks," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2004, pp. 254–273.

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[6] A.-M. Kermarrec and P. Triantafillou, "Xl peer-to-peer pub/sub systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 2, pp. 1–45, 2013.

[7] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[8] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[9] A. De la Rocha, D. Dias, and Y. Psaras, "Accelerating content routing with bitswap: A multi-path file transfer protocol in ipfs and filecoin," 2021.

[10] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni *et al.*, "Wormhole: Reliable pub-sub to support geo-replicated internet services," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 351–366.

[11] J. Antunes, D. Dias, and L. Veiga, "Pulsarcast: Scalable, reliable pub-sub over P2P nets," in *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, Z. Yan, G. Tyson, and D. Koutsonikolas, Eds. IEEE, 2021, pp. 1–6. [Online]. Available: https://doi.org/10.23919/IFIPNetworking52078.2021.9472799

[12] D. Vyzovitis, Y. Napora, D. McCormick, D. Dias, and Y. Psaras, "Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks," *arXiv preprint arXiv:2007.02754*, 2020.

[13] P. Agostinho, "A golang implementation of a content-based pubsub middleware over ipfs content routing," https://github.com/pedroaston/contentpubsub, Oct 2021.

[14] "Testground," https://docs.testground.ai/, Aug 2021.

[15] P. Agostinho, "Testground's plan for testing my content-based pub-sub middleware over ipfs' dht," https://github.com/pedroaston/contentps-test, Oct 2021.

[16] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[17] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, 2007, pp. 14–25.

[18] J. A. Patel, É. Rivière, I. Gupta, and A.-M. Kermarrec, "Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems," *Computer Networks*, vol. 53, no. 13, pp. 2304–2320, 2009.

[19] S. Voulgaris, E. Riviere, A.-M. Kermarrec, and M. Van Steen, "Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks," 2005.

[20] I. Aekaterinidis and P. Triantafillou, "Pastrystrings: A comprehensive content-based publish/subscribe dht network," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 2006, pp. 23–23.

[21] P. Salehi, K. Zhang, and H.-A. Jacobsen, "Popsub: Improving resource utilization in distributed content-based publish/subscribe systems," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 88–99.

[22] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics,"

in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 246–255.

[23] N. Cassavia, S. Flesca, M. Ianni, E. Masciari, and C. Pulice, “Distributed computing by leveraging and rewarding idling user resources from p2p networks,” *Journal of Parallel and Distributed Computing*, vol. 122, pp. 81–94, 2018.

[24] D. Dias and L. Veiga, “browsercloud.js: a distributed computing fabric powered by a P2P overlay network on top of the web platform,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, H. M. Haddad, R. L. Wainwright, and R. Chbeir, Eds. ACM, 2018, pp. 2175–2184. [Online]. Available: https://doi.org/10.1145/3167132.3167366

[25] M. Selimi, L. Cerdà Alabern, F. Freitag, L. Veiga, A. Sathiaseelan, and J. Crowcroft, “A lightweight service placement approach for community network micro-clouds,” *Journal of Grid Computing*, vol. 17, no. 1, pp. 169–189, 2019.

[26] M. Fazlali, S. M. Eftekhar, M. M. Dehshibi, H. T. Malazi, and M. Nosrati, “Raft consensus algorithm: an effective substitute for paxos in high throughput p2p-based systems,” *arXiv preprint arXiv:1911.01231*, 2019.

[27] M. Stoyanova, Y. Nikoloudakis, S. Panagiotakis, E. Pallis, and E. K. Markakis, “A survey on the internet of things (iot) forensics: challenges, approaches, and open issues,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1191–1221, 2020.

[28] A. E. Gencer, S. Basu, I. Eyal, R. Van Renesse, and E. G. Sirer, “Decentralization in bitcoin and ethereum networks,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 439–457.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001, pp. 161–172.

[30] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[31] J. Liang, R. Kumar, and K. W. Ross, “Understanding kazaa,” 2004.

[32] M. Waldman, A. D. Rubin, and L. F. Cranor, “Publius: A robust, tamper-evident censorship-resistant web publishing system,” in *9th USENIX Security Symposium*, 2000, pp. 59–72.

[33] C. G. Plaxton, R. Rajaraman, and A. W. Richa, “Accessing nearby copies of replicated objects in a distributed environment,” *Theory of computing systems*, vol. 32, no. 3, pp. 241–280, 1999.

[34] S. Saroiu, P. K. Gummadi, and S. D. Gribble, “Measurement study of peer-to-peer file sharing systems,” in *Multimedia Computing and Networking 2002*, vol. 4673. International Society for Optics and Photonics, 2001, pp. 156–170.

[35] P. Labs, “A modular network stack,” https://libp2p.io/, Aug 2021.

[36] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, “Filtering algorithms and implementation for very fast publish/subscribe systems,” in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 2001, pp. 115–126.

[37] Y. K. Dalal and R. M. Metcalfe, “Reverse path forwarding of broadcast packets,” *Communications of the ACM*, vol. 21, no. 12, pp. 1040–1048, 1978.

[38] J. L. Martins and S. Duarte, “Routing algorithms for content-based publish/subscribe systems,” *IEEE Communications Surveys & Tutorials*, vol. 12, no. 1, pp. 39–58, 2010.

[39] libp2p, “A kademlia dht implementation on go-libp2p,” https://github.com/libp2p/go-libp2p-kad-dht, Aug 2021.

[40] “Gopsutil: Psutil for golang,” https://github.com/shirou/gopsutil, Aug 2021.

[41] P. Agostinho, “A content-based pubsub middleware over ipfs content routing,” https://github.com/pedroaston/smartpubsub-ipfs, Oct 2021.

A

Appendix - Post Delivery Update

Refactoring and improving codebase After the dissertation's delivery, we continued the work and proceeded to further clean, refactor and debug our code to make it available and let new developers and IPFS [2] users explore it. The main reason behind this appendix was to update our work before the dissertation's presentation.

During our effort the past weeks, we managed to detect a non-deterministic behavior in the code, namely in how the data structure that saves IPFS addresses is laid out (fields not necessarily stored always in the same order). Due to this ill specification, occasionally, other parts of the code end up accessing other peers' UDP endpoint (once in 1000 address fetching) instead of the TCP one. When this happens, complete connectivity across the overlay is not not possible and the experiment must be restarted.

Addressing this issue allowed us to consistently extend the size of our testbed from 75 nodes to 250. Still, we believe that further optimization can still be achieved, and test runs may support 500 nodes in more capable machines.

Scaled Up Results

The following graphics represent the new results with similar settings to the ones used during the variants testing (Section 5.3.1) but in a 250 node network, executing this time in a more capable machine: Intel Xeon CPU with 32 hardware threads and 128 GB of RAM.

Reliability The reliability in a normal scenario is around 65-75% because the dissemination chains of each attribute are not centered at a fixed rendezvous node due to ill attribute mapping or a badly bootstrapped network. The first cause is the most probable one because in the event burst scenario (reliability is around 45-55%) events come from more different sources than in a normal scenario, resulting in a higher probability of peers choosing different rendezvous nodes for the same attribute.

Once an event hits a dissemination chain rendezvous, the reliability in this chain is 100%. A trivial fix would be to assign the rendezvous role to a predefined peer Id to each attribute, which we will try in the future.

Event Latency Even with no reliability, in the normal event scenario, around 160 were delivered, and in the event burst scenario around 3000 events, resulting in the data shown in Figures A.1 and A.2.

From the obtained results, we come to realize that our dissemination chains are scaling well compared with the results and traffic of a 60 node network of Section 5.3.1. This supports the affirmation that with a larger network, nodes will start to manage fewer nodes directly in comparison to the total size of the network.

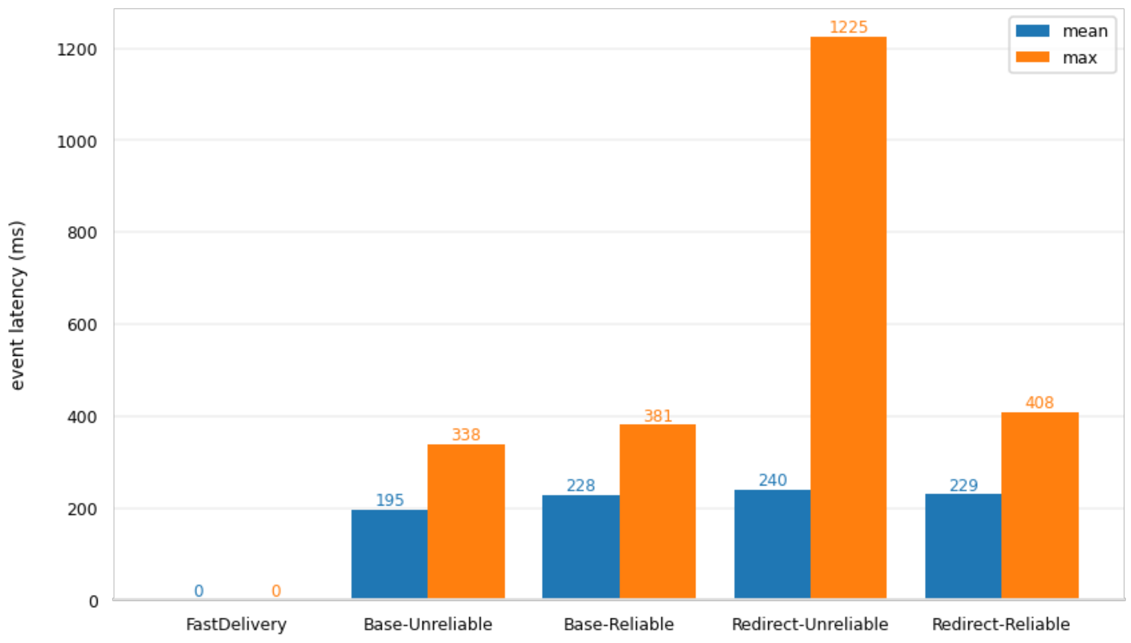


Figure A.1: New values for event latency in a normal scenario

We have approximately the same results of event latency but in a network four times larger and with twice the number of events delivered. The main conclusions taken in our evaluation Section are still valid. One example of a confirmed conclusion is those unreliable variants are much better performance-wise in burst scenarios compared with the reliable ones.

Existing Limitations Although our testing environment grew, new limitations of our pub-sub have risen. Some concurrent writes and deadlocks were corrected, but new challenges appeared.

With an increase in the number of peers and hops to reach a peer/key, our attribute mapping and bootstrapping mechanism are occasionally failing to connect the dissemination chains (that have earlier been confirmed to be 100% reliable before executing the experiment).

These intricate details are complex, and parts of them were actually inherited from existing IPFS code. The bootstrapping mechanism was taken from a libp2p [35] testplan, and the attribute mapping was made using simple hashing functions of the Kademlia module [39]. They have surfaced in execution, in part, because we are running the code of hundreds of IPFS nodes in a single machine, where in the real world, each machine would run just a small number of threads of IPFS code.

These aspects will be shortly reviewed and the implementation optimized since our protocols fulfill something that is lacking to IPFS and the Web.

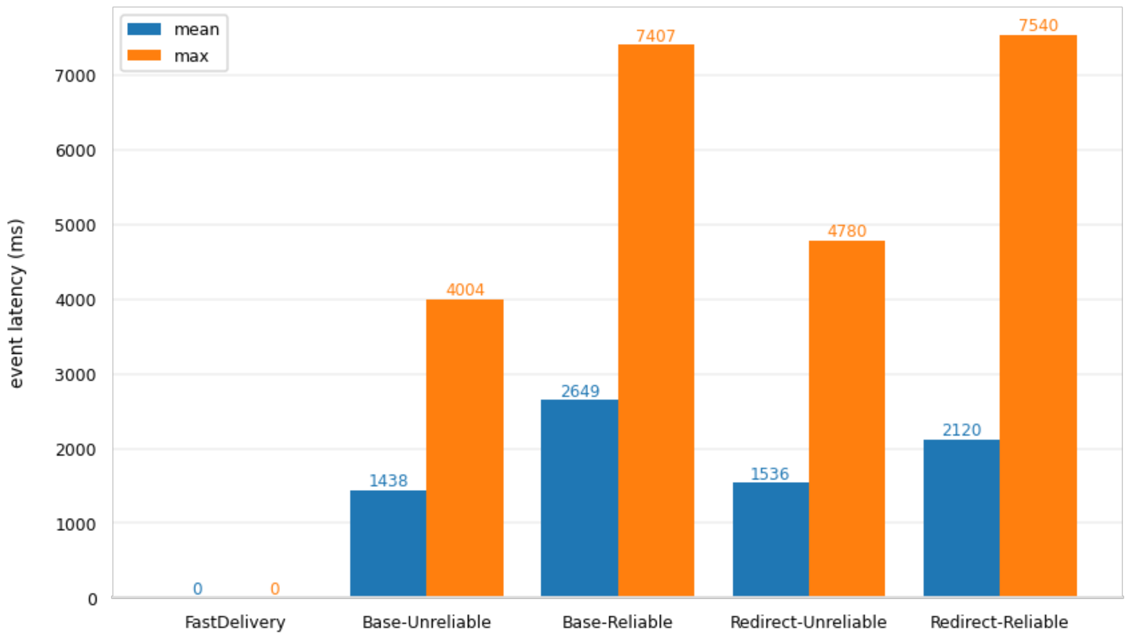


Figure A.2: New values for event latency in a event burst scenario