SmartPubSub@IPFS - Extended Abstract

Pedro Agostinho
Instituto Superior Técnico
Lisbon,Portugal
pedro.eduardo@tecnico.ulisboa.pt

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. 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.

1 Introduction

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.

The InterPlanetary File System 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 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.

1.1 Pub-Sub over IPFS

This project can be considered a "marriage" between pub-sub and p2p as they complement themselves. On one side, we have Publish-Subscribe systems, that 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 [1].

A smart pub-sub Currently, IPFS content-addressing system works with Kademlia's DHT and the Bitswap protocol [2]. 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,meaning its raw bits and data type. In the IPFS community, some topic-based systems were created (e.g., Pulsarcast [3]), being GossipSub [4] 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.

So the goal of this work is to provide a dynamic semantic-addressing layer (meaning content-addressing a human can understand) where the in-

formation 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, as in the example ilustrated by Figure 1.

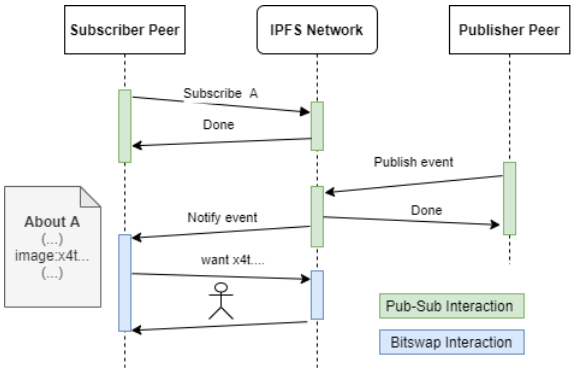

Figure 1: Peers interacting with the current static IPFS layer and a dynamic pub-sub layer

2 Related Work

In this section we present this work's research areas. We present the role and characteristics of **Publish-Subscribe** systems, and the different characteristics of **peer-to-peer** systems, in special content distribution ones.

2.1 Publish-Subscribe

Publish-Subscribe is a message paradigm that provides complete decouplement between data consumers and data producers both in time, space and synchronization. This message paradigm main actors are the publisher (producer of events), the subscriber (consumer of specific events), and the information bus. This last one is the medium where subscriptions made and published events are forwarded and the medium finds a way to notify the interested subscribers of the published events.

Variants There are three types of pub-sub systems regarding the granularity of the subscriptions and events identifiers/predicates:

- **Topic-based**: In a topic approach the events and subscriptions are addressed with a topic. This approach is the simplest and easiest to implement, being a topic a keyword that identifies the event or subscriber interest. An examples of topic-based pub-sub architectures is Rappel [5].
- **Content-based**: Sometimes a topic does not describe the interest of a subscriber, resulting in receiving several unwanted events. A content-based alternative will prove more precise and might even be more efficient regarding bandwidth consumption. In this approach, subscriptions are represented by predicates that not only represent topics and subtopics but also range queries. Examples of content-based pub-sub architectures are Hermes [6], Wormhole [7] and PopSub [8].
- **Type-based**: 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 code. 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 [9].

Architectural Details Today several types of pub-sub systems have been designed and implemented, being one of the main characteristics the centralization degree. Pub-Subs may be implemented using a central server, distributed servers or a completely decentralized approach where nodes have the same role. The more centralized the approach the less efficient is the routing and the structure is less tolerant to failures, but on the other hand, reliability and persistence of the events is easier guaranteed and the algorithms are lighter and easily implemented

The other main characteristics is the strategies used by the pub-sub to forward the events and subscription, which can belong to these main groups:

- **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 by checking a predefined list of rendezvous nodes.

- **Filtering:** this approach can be used in different scenarios, but its goal is to transform a subscription into a filter. This way, once a node receives a filter, it may attempt to merge it with previous ones, reducing the number of filters it needs to check. This approach needs to implement a mechanism of distribution of the filters, redirecting the filters to a rendezvous or event's publishers.

- **Gossip:** one alternative approach is to diffuse events via gossiping. Here subscribers are grouped by interest, and publishers somehow forward their events to some of the interested subscribers. These subscribers then guarantee the event's forwarding between the remaining interested ones, relying on the ability of the subscriber to find the best neighboring peers based on its interests.

- **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 easiest to implement but wastes bandwidth by forwarding unwanted events and does not scale well.

2.2 Peer-to-Peer

What characterizes a p2p system is commonly seen as its decentralized sharing of computational resources on a particular network, maintaining it a properly functioning even in the presence of node failures, connectivity problems, and churn.

P2P networks are highly scalable and fault tolerante because every node acts as both a server and client, and so, the number of servers grows linearly with the number of clients, preventing server bottlenecks.

There are several p2p application types, but we are mainly interested in **content distribution systems**, which is the category that incorporates IPFS. These are designed for the sharing of digital media and other data between users. This systems range from 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.

During our research we came with two main characteristics of this content distribution systems, being the first the **level of decentralization** of these content distribution overlays:

- **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 [10], and CAN [11].

- **Partially Centralized:** is similar to purely decentralized systems, although some nodes assume a more important roles, like acting as local indexes for files shared by neighboring peers. Since these super-nodes are dynamically assigned by the overlay and they are not single points of failure. One example of this is Kazaa [12].

- **Hybrid Decentralized:** in these systems, there is a central server facilitating the interaction between peers by maintaining directories of metadata or describing the shared files stored by the peer nodes. Although the end-to-end interactions take place directly between two peer nodes, the central servers facilitate these interactions by performing lookups and identifying the nodes storing the files. One example of this is Publius [13].

The second is a **Structure** criteria referring how the overlay organizes. In more structured approaches content is placed in specific locations of the overlay, making data fetching faster, more efficient, and scalable. On the other hand, the placement of content is unrelated to the overlay's topology in unstructured approaches, making this approach better in environments with transient node population.

3 Architecture

From the start, we tried to design one optimal solution using the existing IPFS routing overlay (Kadem-

lia DHT). Because Kademlia [10] 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.

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.

- **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 [6] 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. 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.

Our system's network stack is comprised of three main layers illustrated in Figure 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

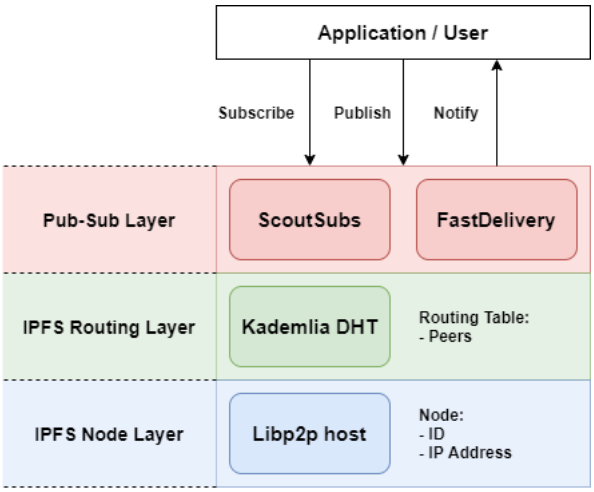

Figure 2: Pub-Sub Protocol Stack

application the possibility of publishing or subscribing to events in a content/meaningful way.

3.1 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.

3.2 ScoutSubs Protocol

As mentioned before, the ScoutSubs protocol provides a pub-sub middleware over IPFS' content routing overlay. The base design of this system was inspired by Hermes [6], 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 to ours, and then we 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 nodes of its event attributes. Figure 3 illustrates that.

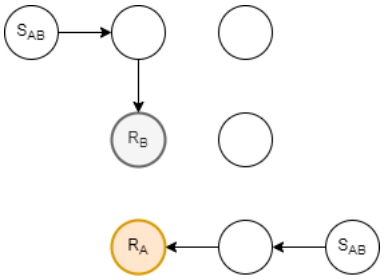

Figure 3: Subscription Forwarding

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 4 shows.

All the filtering information is kept at a filtertable which initially is a replica of the kademlia routing table [10]. 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. Filters upon received can be merged or ignored if the result is the same in the forwarding process, to minimize the size of the filtertables.

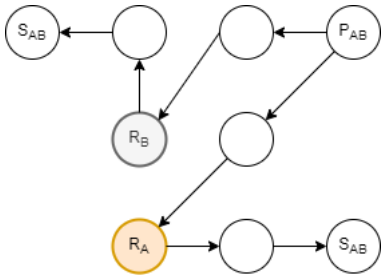

Figure 4: Event Forwarding

3.2.1 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 5. 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.

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

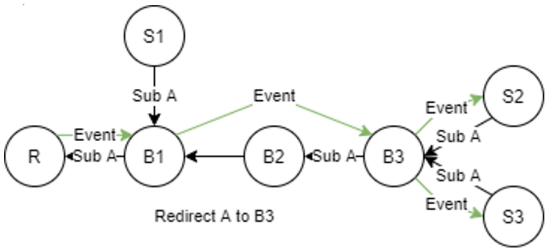

Figure 5: Forwarding using shortcut

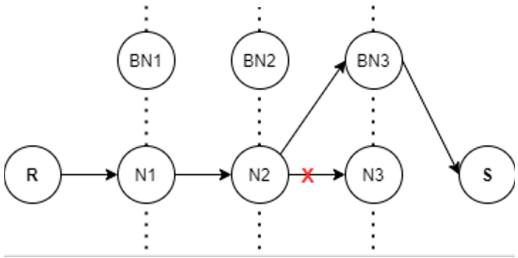

Figure 6: Forwarding event using backup

on unpopular events, meaning that searching a filter table with lots of filters to then have only one hit avoids wasting CPU unnecessarily.

3.2.2 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. 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 independently of a peer's Id, our system can withstand several node failures.

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 6.

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. This working time requirement is necessary since a new node entering the network and closer to the key than its previous rendezvous node will become the new rendezvous node for that attribute. 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.

3.2.3 Achieving Reliability

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 resembling an acknowledgement

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. 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, and 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.

3.2.4 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.

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.

3.3 FastDelivery Protocol

As previously mentioned, FastDelivery's main objective is to disseminate events as fast as possible. 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 of premium subs, to which it sends events directly or in 2 geographically oriented hops (overlay hops). Premium subscribers need to provide the publisher their endpoint, location (Region/Country), and resources (network and CPU-wise).

Motivation The reason of 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.

3.3.1 Design

In this protocol, we decided that a publisher manages 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 (apple/france/price[0,1]).

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

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. If the multicast group's predicate has several range attributes,

he would need the same number of range trees and intercept their query results.

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.

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. A publisher may also prefer to keep its endpoint address private.

4 Implementation

We implemented our pub-sub in golang, and kept several variants with and without the tracking and redirect mechanisms [14]. We developed a testing environment using testground [15] and implemented several several testing scenarios to test our pubsub [16].

5 Evaluation

The following results were achieved using an Ubuntu 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 [17].

5.1 Variant's testing

Here we will present the results of each variant tested through different scenarios. The goal is to analyze if the redirect and reliable mechanisms are working and are not inefficient. 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.

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, resource consumption was slightly lower than the other variants. Because our experience used a 60 node network, and each node had around 30-40 peers, communication is mostly direct (2-4 hops) and redundant paths are common.

Event Latency The results from a normal scenario were an average event latency for the used network composition and configuration of 200-250 ms. In a scenario subscription being made at the time of publishing, the results of the average event latency were of 200-250 ms. In a scenario with 10 times more events being published than normally, the average event latency was of 1780-2500 ms. In a scenario with 2 failing peers, the average event latency was of 200-270 ms. These results reflect the higher the event production is, the higher the event latency will be.

Resource Consumption Looking at the memory and CPU consumption of our pub-sub during our test runs, an higher memory usage is followed by a higher CPU time usage. 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 1, we can see that the reliable variants, especially in the event burst scenario, have a substantially higher CPU and memory usage. The reasons for this are the extra relation between tracker and rendezvous node and the management of the acknowledge chains. The impact of the redirect mechanism is not palpable with a small network, and so we cannot comment on its performance.

5.2 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. We vary the pub-sub's replication from 1, 2, ,3 and 5 in a 75 node network.

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	116.9 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	108.3 MB 8.31 s	37.0 MB 3.61 s	268 MB 21.38 s	21.5 MB 3.84 s

Table 1: Average Memory and CPU user-time used per node

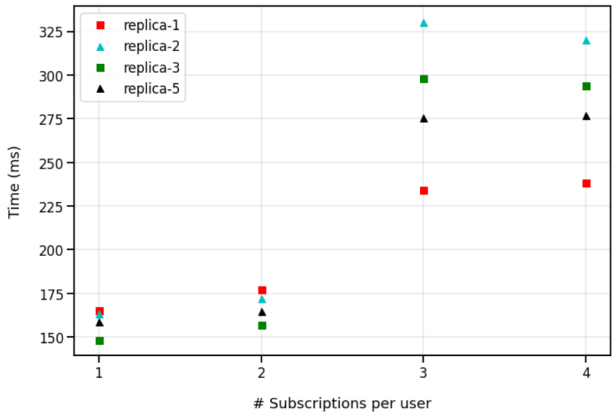

Figure 7: Average Event Latency per replica factor at each stage

Event and Subscription latency We can start looking at the Figures 7 and 8 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, 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).

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 filters. In an event 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. Event latency is dependent more on the order of the filters in the at each filtertable entry, because if the first filter matches the events, that node will not have to check the other ones.

CPU and memory usage Memory and CPU 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, and the extra CPU work is almost none.

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 increase with 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 slight change, although far from affecting the user perception of the event speed delivery.

6 Conclusions

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

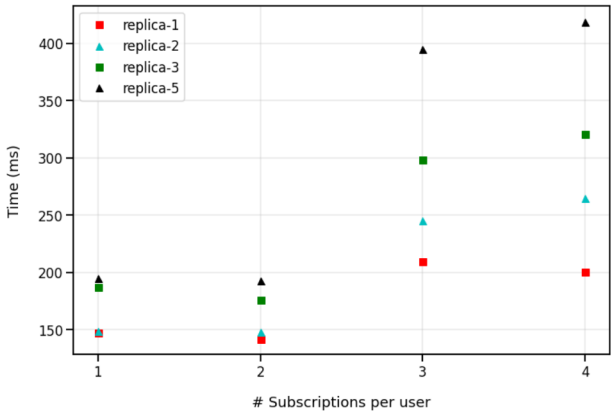

Figure 8: Average Subscription Latency per replica factor at each stage

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.

References

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

[2] Alfonso De la Rocha, David Dias, and Yiannis Psaras. Accelerating content routing with bitswap: A multi-path file transfer protocol in ipfs and filecoin. 2021.

[3] João Antunes, David Dias, and Luís Veiga. Pulsarcast: Scalable, reliable pub-sub over P2P nets. In Zheng Yan, Gareth Tyson, and Dimitrios Koutsonikolas, editors, *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, pages 1–6. IEEE, 2021.

[4] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks. *arXiv preprint arXiv:2007.02754*, 2020.

[5] Jay A Patel, Étienne Rivière, Indranil Gupta, and Anne-Marie Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 53(13):2304–2320, 2009.

[6] Peter R Pietzuch and Jean M Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618. IEEE, 2002.

[7] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin 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)*, pages 351–366, 2015.

[8] Pooya Salehi, Kaiwen Zhang, and Hans-Arno 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*, pages 88–99, 2017.

[9] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert 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*, pages 246–255. IEEE, 2014.

[10] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[11] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

[12] Jian Liang, Rakesh Kumar, and Keith W Ross. Understanding kazaa, 2004.

[13] Marc Waldman, Aviel D Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident censorship-resistant web publishing system. In *9th USENIX Security Symposium*, pages 59–72, 2000.

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

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

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

[17] Pedro Agostinho. A content-based pub-sub middleware over ipfs content routing. https://github.com/pedroaston/smartpubsub-ipfs, Oct 2021.