A Study on Publish-Subscribe Systems over Overlay Networks

António Duarte[*]
an.duarte@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Diogo Almeida[†]
daro.almeida@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Diogo Fona[‡]
d.fona@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

A peer-to-peer topic based publish-subscribe system is one in which participants can exchange information in a way where processes that share information do not need to know who are its receivers, and processes that receive information do not need to know who are the senders. This can be achieved by having processes that can be subscribers, receiving messages of topics they subscribe to, and publishers, that disseminate messages tagged with a topic. This work proposes two publish-subscribe systems, one operating on top of an unstructured overlay network, using an epidemic broadcast protocol, and the other using a pubsub protocol on top of a structured one. We evaluate the performance of these proposals, explaining their trade-offs and possible applications.

ACM Reference Format:
António Duarte, Diogo Almeida, and Diogo Fona. 2022. A Study on Publish-Subscribe Systems over Overlay Networks. In *The Projects of ASD - first delivery, 2021, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Publish-Subscribe systems are a message exchange pattern between processes in a system. Processes can subscribe to topics (thus being subscribers to a certain topic), or publish messages to topics (thus being publishers). It is important to note that processes can be both publishers and subscribers at the same time, to one or more topics. The subscribers of a topic should receive all the messages that were published to that topic.

Although the concept may seem simple, there are challenges related to the implementation of such systems. A naive approach would assume a global known membership, making the system easy to reason about and implement: Whenever a node subscribes to a topic, it broadcasts the subscription to all nodes in the system. Each node would keep track of all the topics subscribed by each node, and whenever it publishes a message to a topic, it broadcasts the messages to all the known nodes subscribed to that topic.

[*]Student number 58278.
[†]Student number 58369.
[‡]Student number 57940.

ASD22/23, 2022, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The challenge arises when we decide to tackle the issue of scalability, given that the bookkeeping necessary to maintain a system consisting of tens of thousands of nodes would be excessively expensive.

To solve that issue, we need to implement solutions that rely on membership protocols built to be scalable, this is done by guaranteeing that each node only has a partial view of the system. These systems then require some form of dissemination protocol, to (generally) probabilistically disseminate a message through all the nodes in the system. These solutions are typically called overlay protocols which can be structured, where the nodes organize into specific topologies according to characteristics of the nodes such as a logical identifier, or unstructured, where nodes organize in a non-predictable manner.

The above-mentioned implementations, however, impose challenges. Imagining a distributed system in which we represent processes as vertices of a graph, and connections between those processes as edges, the first challenge would be to guarantee that the resulting graph is strongly connected, specially in the case of churn or node failure. Another important detail is the network load imposed by the system, given that under specific solutions the redundancy (an undesired or repeated message being sent to a process in the system) of the messages being propagated throughout the system could be excessively high.

In this work, we evaluate two possible solutions to implement a Publish-Subscribe system: The first on top of a Structured Overlay (Kademlia), using GossipSup as the Publish-Subscribe protocol that is responsible for disseminating messages to subscribers of a topic. The second on top of an Unstructured Overlay (HyparView), using PlumTree as the dissemination protocol, which as we'll explain later, helps reduce redundancy in the message dissemination process.

The remainder of this document is organized as follows: In the following Section we will be going over the Related Work, approaching in detail some already introduced methodologies, as well as some other possibilities that we could've chosen; In Section 3 we will go over the Implementation details of both our chosen protocols, and the way we decided to use them in conjunction with each other. In Section 4 we will present the results of our experimental evaluation, testing both of the systems against different loads, both in terms of the rate in which messages are propagated and their size. In Section 5 we will summarize the applicability of our solutions given the results obtained.

2 RELATED WORK

2.1 Membership Protocols

Membership protocol are the ones that are used to ensure the scalability of the system, by ensuring that each node doesn't need to maintain a global knowledge of all the nodes that join the system.

2.1.1 HyParView. HyParView is a Membership Management protocol that forms an Unstructured Overlay network amongst the peers that are part of it. This protocol is highly resistant to the failure of even a high percentage of nodes in the system. As already stated, in a Membership Protocol it's important for a node to not be dependent on a Global View of the system, since the book-keeping would be excessively expensive. HyParView keeps two sets of nodes, called views, on each node. The active view, which keeps open TCP connections with the nodes in it, should have a size of $ln(n)$, where n is the number of nodes in the system, and is relatively stable, only changing when a connection is dropped. The passive view, larger than the active view, is shuffled regularly with nodes selected through random walks through the network, and is used to replace the nodes in the active view in case of their failure. The protocol also ensures that the active views are always symmetric, which is essential for the usage of some dissemination protocols.

2.1.2 Kademlia.

2.2 Dissemination Protocols

An essential part of building a pub-sub system, is to think about in which way we will propagate messages throughout the underlying logical network (the membership protocol), to achieve this, we studied two of the possible dissemination protocols:

2.2.1 Flood. A simple way to propagate messages in a system composed of many interconnected nodes, would be to flood the links between them. In a flood protocol, when a node decides to broadcast a message, it sends the message to all of its neighbors (the nodes that it is connected to). When a node receives a message that is being flooded throughout the system for the first time, it forwards the message to all of its neighboring nodes, except the one that it received the message from. If we're using a membership protocol that ensures a strongly connected graph, with an adequate fanout, this dissemination strategy guarantees with probabilistic certainty that a message will reach every node in the system. This protocol, however, causes a large strain on the network, given that many nodes in the system will receive the same message multiple times. Although some level of redundancy is what ensures that a message will reach every node in the system, specially in the event of node failure, lower levels of redundancy can be achieved by employing other dissemination strategies.

2.2.2 Plumtree. In order to solve the excessive redundancy that comes from employing a flooding strategy to broadcast messages, a possible solution is the usage of Epidemic Broadcast Trees. In order to understand the functioning of this protocol, we need to first go over two of the possible mechanisms for two nodes to exchange a message, which we refer to as gossip mechanisms: Eager push gossip, where a node sends a message directly to another node, and lazy push gossip, where a node first sends an identifier of the message, and the node on the receiving end can then ask the sending node for the full message. A process in Plumtree keeps two sets, eager push peers, and lazy push peers. In Plumtree, at an initial moment, all the neighbors of a node are considered eager push peers. When a node receives a message from another node, it forwards it to its eager push peers using the eager push gossip strategy, and sends an identifier of the message to all of its lazy push peers. When it receives the same message twice, it moves the node that it received it from to its lazy push peers set. It also sends a Prune message to the node it received the message from. Upon receiving a Prune message, a process moves the node it received the message from to its lazy push peers set. When a node receives a message identifier from one of its lazy push peers, it starts a timer. The timer is canceled upon receiving the full message that corresponds to the identifier of the message. If the timer runs out, the process assumes that one of its existing eager push links failed. In order to recover from the failure, and receive the message that is now missing, it sends a Graft message to the peer that it received the identifier from, and moves it to its eager push set. Upon receiving the graft message, the node forwards the message to the node it received it from, and places the node on its eager push peers set. This strategy guarantees that in the end of the first broadcast, which works as a Flood, all the nodes in the system that are used to eagerly disseminate the following messages, form a tree structure, which diminishes the stress on the network by reducing the redundancy. The fact that it uses both eager push and lazy push strategies, in conjunction with prune and graft messages, also makes this strategy resilient to node failures.

2.3 Pubsub Protocols

2.3.1 GossipSub. Gossip-based pubsub protocols have been introduced in the past as a way to limit the number of messages propagated between processes in pub/sub systems. In most of these approaches, processes forward metadata related to messages they have previously received without forwarding the messages themselves, a method generally called *pull*. GossibSub, a gossip-based pubsub protocol, was designed to deal with both fast and resilient message propagation in *permissionless* networks, e.g., open blockchain networks such as Ethereum [1]. GossipSub assures its fast message propagation with its *Mesh Construction*, where a connected mesh is created for each topic. A mesh of a topic in this context is a sub-network of peers, not necessarily forming a clique, that are subscribed to that topic. Each node is connected to a limited number of other peers, forming its local (view of the) mesh. Mesh-connected nodes directly share messages with one another, realizing an *eager push* communication model. Nodes join and leave the mesh as they subscribe and unsubscribe to topics. Those nodes that are not part of the mesh communicate with mesh-connected nodes through gossip (*lazy push*). The resilient message propagation is accomplished with GossipSub's *Score Function*, where every node participating in the network is observed by every other one in a reputation system where its actions are evaluated, and *Mitigation Strategies*, such as mesh maintenance using the Score Function or the isolation of malicious nodes for example.

3 IMPLEMENTATION

This section presents the implemented variants of our Publish-Subscribe systems. These are assumed to execute above asynchronous systems using the crash fault model.

3.1 Unstructured Variant

3.1.1 Plumtree + HyParView. Our Publish-Subscribe system for Unstructured Overlays uses a filtering based solution, using HyParView as the Membership protocol and Plumtree to disseminate messages in the network.

Subscribe operations are completely local to each node, the Publish-Subscribe protocol keeps track of all the topics that the node is subscribed to. Whenever a publish is made, the message and its topic are disseminated to the entire system, using Plumtree as previously explained. When a node receives the published message, it simply filters the message, only delivering it to the Application Layer if the message's topic is contained in the set of currently subscribed topics.

Algorithm 1: Unstructured Publish-Subscribe

```
Interface:
    Requests:
        Subscribe(topic);
        Unsubscribe(topic);
        Publish(msg, topic);
    Indications:
        PubsubDeliver(msg);
        SubscriptionReply(topic);
        UnsubscriptionReply(topic);

State:
    $seenMessages$;
    $subscribedTopics$;

Upon PlumtreeDeliver(sender, {GOSSIP, msg, topic}) do
    If $topic \in subscribedTopics \wedge msg \notin seenMessages$ do
        $seenMessages \leftarrow seenMessages \cup \{message\}$;
        Trigger PubsubDeliver(msg);

Upon Subscribe(topic) do
    $subscribedTopics \leftarrow subscribedTopics \cup \{topic\}$;
    Trigger SubscriptionReply(topic);

Upon Unsubscribe(topic) do
    $subscribedTopics \leftarrow subscribedTopics \setminus \{topic\}$;
    Trigger UnsubscriptionReply(topic);

Upon Publish(message, topic) do
    Trigger PlumtreeBroadcast(message, topic);
    If $topic \in subscribedTopics$ do
        Trigger PubsubDeliver(msg);
```

3.2 Structured Variant

3.2.1 GossipSub + Kademlia. For the structured variant of the Publish-Subscribe system, we use the Kademlia DHT in conjuntion with GossipSub pubsub protocol.

Since we are only addressing systems with the crash fault model in this work, our implementation of GossipSub is modified, stripped away of all its resilient inducing components, namely the *Score Function* and *Mitigation Strategies*, leaving only the *Mesh Construction* details.

3.2.2 KadPubSub + Kademlia.

4 EXPERIMENTAL EVALUATION

4.1 Reliability

4.2 Redundancy

4.3 Latency

5 CONCLUSIONS

ACKNOWLEDGMENTS

To João Leitão, for teaching us how to implement protocols and write good reports.

REFERENCES

[1] Chris Dannen. 2017. *Introducing Ethereum and solidity.* Vol. 1. Springer.