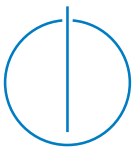
Pooya Salehi

Dependable Publish/Subscribe Systems for Distributed Application Development

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Dependable Publish/Subscribe Systems for Distributed Application Development

Pooya Salehi

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r): Prof. Dr. Georg Carle

Prüfer der Dissertation:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. David Bakken

Die Dissertation wurde am 25.06.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.09.2018 angenommen.

Abstract

The publish/subscribe (pub/sub) communication paradigm provides an asynchronous many-to-many communication substrate between data producers and data consumers. Using pub/sub, components of a distributed application can communicate without direct knowledge of each other which facilitates loose-coupling and scalability. Therefore, providing a managed pub/sub service can reduce the development and operational effort of Internet-scaled distributed applications. In this work, we address four non-functional requirements of distributed content-based pub/sub systems that can facilitate their adoption as the basis of a dependable pub/sub service suitable for distributed application development.

Firstly, we address availability of a pub/sub service during broker failure. Broker failures can cause delivery disruption and therefore, a repair mechanism is required, along with message retransmission to prevent message loss. During repair and recovery, the latency of message delivery can temporarily increase. To address this problem, we present an epidemic protocol to allow a content-based pub/sub system to keep delivering messages with low latency, while failed brokers are recovering. Using a broker similarity metric, which takes into account the content space and the overlay topology, we control and direct gossip messages around failed brokers. Based on our evaluation, our approach is able to provide a higher message delivery ratio than the deterministic alternative at high failure rates or when broker failures follow a non-uniform distribution.

Secondly, we address scalability of the hop-by-hop routing mechanism utilized in such distributed pub/sub systems. In an Internet-scale pub/sub service, this routing scheme allows brokers to correctly forward messages without requiring global knowledge. However, this model causes brokers to forward publications without knowing the volume and distance of matching subscribers, which can result in inefficient resource utilization. In order to raise the scalability of the service, we introduce a popularity-based routing mechanism. We define a utilization metric to measure the impact of forwarding a publication on the overall delivery of the system. Furthermore, we propose a new publication routing algorithm that takes into account broker resources and publication popularity among subscribers. Lastly, we propose three approaches to handle unpopular publications. Based on our evaluation, using

real-world workloads and traces, our proposed approach is able to improve resource efficiency of the brokers and reduce delivery latency under high load.

Thirdly, we address maintainability of the pub/sub overlay which can become inefficient due to the dynamic communication flows between data producers and consumers in a pub/sub service. In such cases, the overlay requires adaptation to the existing load in order to ensure certain quality of service agreements or improve system efficiency. While there exists algorithms to design overlay topologies optimized for a given workload, the problem of generating a plan to incrementally transform the current topology to an optimized one has been largely ignored. To address this problem, we propose an incremental approach based on integer programming which given the current topology and a target topology, generates a transformation plan with a minimal number of steps in order to lessen service disruption. Furthermore, we introduce a plan execution mechanism which ensures correct routing information in the overlay throughout the plan execution. Based on our evaluation, our proposed approach can significantly reduce plan computation time and produce more concise plans with faster execution time compared to existing approaches.

Lastly, we address usability of the pub/sub paradigm for distributed application development. In a pub/sub service based on a distributed overlay of brokers, due to the propagation delay, it takes time for a client's interests to be received by all brokers comprising the overlay. Nonetheless, existing overlay-based systems only guarantee delivery of notifications to subscribers that are already known by all brokers in the overlay. The message propagation delay and unclear delivery guarantees during this time increases the complexity of developing distributed applications based on the pub/sub paradigm. To address this problem, we propose a collection of message processing and delivery guarantees that allows clients to clearly define the set of publications they receive. Based on our evaluation, these delivery guarantees can reduce buffering requirements on clients, prevent missing notifications due to the propagation delay and provide clients with some primitive building blocks that simplifies development of distributed applications.

Zusammenfassung

Das Publish/Subscribe Paradigma stellt ein asynchrones Kommunikationssubstrat zwischen Komponenten einer verteilten Applikation bereit und ermöglicht eine lose Kopplung und Skalierbarkeit. Daher kann die Bereitstellung eines Publish/Subscribe-Services den Entwicklungs- und Betriebsaufwand von verteilten Anwendungen verringern. In dieser Dissertation befassen wir uns mit vier nicht-funktionalen Anforderungen von verteilten content-based Publish/Subscribe-Systemen als Basis für solche Services.

Erstens adressieren wir die Availability eines Publish/Subscribe-Service während eines Brokerfehlers. Brokerfehler können eine Unterbrechung der Nachrichtenzustellung verursachen, und daher ist ein Reparaturmechanismus erforderlich, zusammen mit einer Nachrichtenneuübertragung, um einen Nachrichtenverlust zu verhindern. Um dieses Problem anzugehen, stellen wir ein epidemisches Protokoll vor, mit dem ein content-based Publish/Subscribe-System Nachrichten mit geringer Latenz weiterleiten kann, während sich ausgefallene Broker erholen.

Zweitens adressieren wir die Skalierbarkeit des Hop-by-Hop-Routing-Mechanismus, der in solchen verteilten Publish/Subscribe-Systemen verwendet wird. In einem Internet-Scale-Publish/Subscribe-Service ermöglicht dieses Routing-Schema Brokern, Nachrichten korrekt weiterzuleiten, ohne globale Kenntnisse zu benötigen. Dieses Modell kann jedoch zu einer ineffizienten Ressourcenauslastung führen. Wir führen einen popularity-based Routing-Mechanismus ein und schlagen einen neuen Publication-Routing-Algorithmus vor, der Broker-Ressourcen und Publikations-Popularität berücksichtigt.

Drittens befassen wir uns mit der Maintainability des Publish/Subscribe-Overlays, das aufgrund der dynamischen Kommunikationsverbindungen zwischen Datenproduzenten und Verbrauchern in einem Publish/Subscribe-Service ineffizient werden kann. In solchen Fällen muss das Overlay an die vorhandene Workload angepasst werden. Um dieses Problem anzugehen, schlagen wir eine inkrementelle Integer-Programming-basierte Methode vor, die einen Transformationsplan generiert, um die Zieltopologie von der aktuellen Topologie in einer minimalen Anzahl von Schritten zu erreichen, um die Serviceunterbrechung zu verringern.

Schließlich befassen wir uns mit der Usability des Publish/Subscribe-Paradigmas für die Entwicklung verteilter Anwendungen. Wir bieten eine Sammlung von Nachrichtenverarbeitungs- und Zustellungsgarantien an, die es den Nutzern ermöglichen, die Menge der Nachrichten, an denen sie interessiert sind, klar zu definieren.

Acknowledgments

All the works constituting this dissertation took place in the Middleware Systems Research Group (I13) in the Informatics department of Technical University of Munich under the supervision of Prof. Hans-Arno Jacobsen.

I would like to thank my supervisor, Arno, for his continuous guidance and support during my master's and PhD studies, and giving me the chance to work, study and learn in his research group. Furthermore, I would like to thank the rest of my thesis committee: Prof. David Bakken as the external examiner and Prof. George Carle who chaired the committee.

Many thanks to: Christoph Doblander for his mentorship and feedback on *Gossip-Enhanced Pub/Sub*, Kaiwen Zhang, who inspired *Popularity-Based Routing for Pub/Sub* and helped me finish *Incremental Topology Transformation using Integer Programming*, and Vinod Muthusamy, for his valuable input that helped shape *Delivery Guarantees in Distributed Content-Based Pub/Sub*. It was with their help that I could successfully finish the works in this dissertation. Additionally, thanks to everyone in I13 for the good times we all had together.

I am eternally grateful to my wonderful girlfriend Johanna, without her unconditional love, care and motivation I would not have had the courage to start a PhD, nor the energy to successfully finish it.

Last but not least, I am grateful to my family. To my parents who supported me to study abroad and live so far away from them. To my sisters and my brother who have always been supportive and encouraged me to follow my dreams.

Contents

Chapter 1

Introduction

Over the past few decades with the prevalence of network-connected devices, software systems have evolved from offline and isolated systems to distributed systems consisting of independently-written components communicating via local network or the Internet [1, 2, 3]. Furthermore, recent approaches in software development, such as micro-services and containerization, have resulted in software systems consisting of hundreds of small service instances running on a distributed infrastructure [4, 2]. These systems need to handle a large number of requests via horizontal scaling with each request involving communication among several services. Developing such distributed systems raises many challenges such as communication, reliability, performance, scalability and heterogeneity. The communication substrate of such applications needs to provide one-to-many and many-to-many communication, while maintaining loose-coupling and asynchrony in order to facilitate scalability for the application. Over the past few years, due to its match for the aforementioned requirements, the publish/subscribe (pub/sub) paradigm has been widely adopted as a communication substrate for Internet-scale distributed applications [3, 5]. Pub/sub allows data producers (publishers) and data consumers (subscribers) to communicate in a data-centric manner. Subscribers describe their interests (subscriptions) without direct knowledge of the publishers. The pub/sub substrate makes sure that data produced by the publishers is delivered to subscribers with matching interests.

Despite the wide adoption of the Internet Protocol (IP) suite for point-to-point communication across the Internet, IP multicast has not been successful in providing global multi-point communication [6, 7]. Consequently, application-layer protocols have been used to provide middleware systems (more recently also supporting the pub/sub paradigm) that stand between the distributed heterogeneous infrastructure and the distributed application and provides a communication layer that can simplify distributed application development [8, 9]. Nonetheless, deployment and maintenance of such application-layer solutions can increase the total cost of ownership for software products. Therefore, cloud service providers have been offering messaging and pub/sub services in order to simplify development and administrative tasks required to operate a distributed application [10, 11, 12].

The service provider operates and maintains a pub/sub system that provides application developers with a managed communication substrate. One scalable approach to implement such a pub/sub system is to use a distributed overlay of brokers connected to each other in a predetermined topology and managed by the service provider [13, 14, 10]. This approach allows providing an Internet-scale pub/sub service that offers clients (publishers and subscribers) a pub/sub substrate to communicate across different regions. In this work, we address some challenges that need to be addressed in order to adopt an overlay-based pub/sub system as the underlying infrastructure of a pub/sub service suitable for distributed application development.

1.1 Motivation

Providing a pub/sub service requires addressing many non-functional requirements that impact both the service provider and the clients of the service. The provider needs to make sure that the pub/sub service is available to all of its clients globally with minimal disruption while running efficiently. For the clients that use the pub/sub service as their communication substrate, the pub/sub service must provide a reliable and intuitive building block that simplifies development of distributed applications. In this work, we consider four non-functional requirements of a pub/sub service, namely, availability, scalability, maintainability and usability which can improve

dependability of such systems and consequently reduce the operational costs of a pub/sub service, as well as improve its adoption by application developers. In the following, we provide the motivation for supporting each of these non-functional requirements.

Availability: Availability is considered one of the main objectives of the cloud computing paradigm [15]. A highly-available service can provide its clients with a correct service for a long period of time despite hardware and software failures [16]. Due to the increasing reliance of many businesses on cloud services, availability of such services is an important factor for their success [17]. Over the past years, there has been incidents where due to datacenter outages, cloud services have been unavailable and therefore directly impacting businesses relying on these services [18]. In such cases, failure of a pub/sub service can bring down a distributed application that relies on the service for communication between its components. Therefore, a pub/sub service provider needs to provide measures to increase availability of the service in face of software and hardware failures and datacenter outages.

Scalability: Providing a pub/sub service means that, rather than the clients, the service provider needs to take care of the scalability and resource efficiency of the service. The service provider needs to manage the resources required to keep the service in operation at the quality of service (QoS) level offered to the clients. Over-provisioning resources can lead to high operational costs while under-provisioning can result in poor service quality, such as high latency or unreliable messaging, which in turn can lead to violation of service level agreements (SLAs) [15]. Providing a pub/sub service which is reliable and has low latency requires more resources than a pub/sub service with no delivery or latency guarantees. For example, SDN-based approaches can provide low latency notification delivery at the cost of special hardware [19]. Reliable pub/sub systems use replication and multi-datacenter approaches, redundant network connections [20] or use fault-resilient protocols such as gossiping [21].

On the other hand, different applications have different QoS requirements. While mission critical applications, such as billing services, might require low latency or reliable messaging, applications such as monitoring dashboards can tolerate some

delay and missing a few messages. Therefore, a pub/sub service provider can reduce the resources required to provide the service by combining several approaches to deliver messages. If a client has not requested guaranteed delivery or latency, the pub/sub service provider can deliver messages to that client using approaches that trade reliability or latency for low resource usage.

Maintainability: Maintainability is defined as the ability of the system to undergo repairs and modifications. This definition goes beyond corrective and preventive maintenance, and includes forms of maintenance aimed at adapting or perfecting the system [22]. Maintainability is pivotal to distributed systems since such systems are typically deployed across heterogeneous machines and networks with varying degrees of change. For such systems to be able to maintain correct service and fulfill non-functional requirements, they must react to changes in the workload and infrastructure [23]. Providing a pub/sub service requires the pub/sub overlay to be maintained in different ways, for example, to improve service quality or to scale the system according to the users. Furthermore, a pub/sub service is potentially shared by many clients with different workloads that can change throughout the day [24]. Therefore, if the pub/sub service is able to periodically adjust its resources and topology to the current load, it is possible to provide the service more efficiently and reduce operational cost and effort, as well as improve the quality of the provided service.

Usability: Existing pub/sub systems guarantee publication delivery to interested subscribers in the system. However, this guarantee is provided only for subscribers whose interests have been fully propagated and processed in the overlay [14, 13, 10]. Due to communication and processing delays, the period of time required for a subscription to get fully propagated in the overlay is unknown. Therefore, there is an uncertain period of time between when a subscriber joins the overlay and submits its subscription until it can be sure that it will receive all data matching its interests [25, 26]. However, this waiting period is a function of the overlay size and broker processing power and a client has no way of approximating this time. A pub/sub service, provided using a distributed overlay of brokers connected via the Internet can have different propagation and processing time for messages depending on the existing load and network conditions. Furthermore, if the service provider uses elastic

scaling of resources to accommodate changing workloads, the propagation time can vary. This uncertainty period can hinder applicability of distributed pub/sub systems because applications using pub/sub as their communication substrate need to either make sure all components are setup and data streams between them are established or use distributed synchronization services and out-of-band communication to clarify received data during the initialization of the components. While establishing all connections beforehand may not be possible in all cases, using the latter approach can complicated the application logic where each client should account for propagation times, buffer messages and communicate with other components to make sure the necessary data paths are established.

Providing a set of delivery guarantees that clarify the message propagation period and eliminates any ambiguity in the semantics of the provided delivery guarantees can simplify development of distributed applications, improve usability of a pub/sub service and increase the domain of applications that can benefit from the pub/sub paradigm as their communication substrate.

1.2 Problem Statement

In this work, we focus on four problems relating to the following four non-functional requirements for a managed pub/sub service: availability, scalability (resource efficiency), maintainability and usability of a pub/sub service for distributed application development.

Availability of a pub/sub service in face of broker failures: In many pub/sub systems, the overlay is organized as a tree determining the neighborhood relation of brokers and the links that connect them [27, 28, 14]. The popularity of a tree-based overlay topology is due to routing simplicity and message dissemination efficiency (e.g., there are no issues with routing loops, as shown by Li *et al.* [29].) The existence of a single unique path between broker pairs eliminates the need to unnecessarily manage and detect duplicate messages. On the other hand, tree topologies do not exhibit path redundancy, as, for example, mesh topologies do; hence, even a

single broker failure can disrupt message delivery to parts of the tree. Despite this shortcoming, in scenarios where brokers are more stable and have to handle many clients (*e.g.*, enterprise IT, cloud), tree topologies are used due to their efficiency [28, 14].

Maintaining the dissemination tree of a pub/sub system in face of broker failure is typically assumed to be carried out by human operators [30, 31]. Furthermore, there exists automatic mechanisms that can detect broker failures, trigger broker replacements and rebuild the routing state of the recovering brokers [20, 32]. Regardless of the maintenance approach, during the failure and recovery period, delivery to some subscribers might not be possible if the path between the publisher and the subscriber contains one or more failed brokers. One possible solution to address message loss in this case is to buffer messages at parent brokers for retransmission after recovery [33]. Besides bearing the risk of overwhelming parent brokers under a high message rate, this approach exhibits an increased notification latency under failure, effecting the remaining brokers in the tree branch. This increased latency is a function of the broker failure detection and recovery time in the system.

An alternative to buffering and waiting for broker recovery is to route messages around failed brokers [20, 34, 35]. This results in lower delivery latency since message propagation continues even when there are broker failures. In tree topologies, redundant paths must be created since, unlike mesh topologies, trees have only one path between each broker pair. Where to add and how to maintain these extra links are two important issues to be considered.

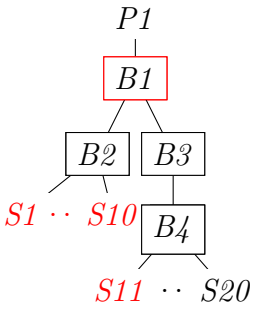


Figure 1.2.1: Example of the binary cost model problem

Inefficient publication routing: In overlay-based pub/sub systems, in order to avoid centralized routing of publications, reverse path forwarding (RPF) is used to establish paths between publishers and subscribers [14]. RPF decouples brokers from the knowledge of all end-to-end paths in the overlay since each broker needs to route messages only to the next hop towards the recipients. This hop-by-hop routing scheme, while scalable since it limits the knowledge required by

each broker to operate, conversely hinders their capacity to apply optimizations that rely on global path knowledge. As a consequence, established pub/sub optimizations are primarily local in nature, such as subscription covering [36].

Hop-by-hop forwarding therefore does not consider the overall system performance. Consider the example in Figure 1.2.1. Publisher *P1*, connected to broker *B1*, is sending a publication destined for subscribers *S1* to *S11*, all outlined in red. In order to reach all matching subscriptions, broker *B1* must forward the publication to both broker *B2* and *B3*. We observe that for the same amount of work, forwarding to *B2* is used for 10 deliveries, while forwarding to *B3* produces only one delivery. In addition, sending to *B3* does not directly reach *S11*; another forwarding to *B4* is needed. Therefore, the utility of sending to *B2* is 20 times greater than sending to *B3*. As pub/sub systems are designed to sustain high publication throughput [37], prioritizing publication forwarding with high utility improves resource utilization in the overlay and the efficiency of the system as a whole.

We identify a problem associated with RPF which we call the *binary cost model problem*. Given an incoming publication to be routed at a broker, this publication must be forwarded to every next hop which contains at least one matching subscription. In other words, the cost to forward a publication to a next hop is the same, no matter the actual number of subscribers downstream.

Reconfiguration of the pub/sub overlay: In an overlay-based pub/sub system, the communication load for each broker is variable, since message flows between publishers and subscribers vary as subscriptions enter and leave the system. Hence, it is possible for some publication to be delivered through a long path of brokers in the current topology, thereby raising the average resource consumption across the brokers and increasing the end-to-end delivery latency. In a large-scale environment such as a cloud pub/sub service, the pub/sub system will experience a diverse range of workloads and thus suffer temporary drops in performance when using a static topology. Therefore, brokers should be dynamically migrated to a new topology which provides efficient delivery paths for the current workload. Furthermore, migration to a new topology might be necessary due to broker and overlay maintenance.

For instance, the Google Pub/Sub System (GooPS), used for the integration of online

services across distributed locations, performs a periodic evaluation of the topology quality [38]. If the current topology is deemed unsuitable under the current conditions, a new topology is generated which is optimized given the actual pub/sub workload.

In this work, we focus on the problem of migrating brokers in an online manner. Modifying the topology while the pub/sub system is still active is a challenging task since the reconfiguration of the broker routing states can induce a temporary partitioning of the overlay network, thereby resulting in a loss of messages in transit during the migration period.

A naive, but reliable approach to migration involves stopping all new operations, allowing all messages currently in transit to be delivered, before applying migration updates and resuming the service. While this does guarantee reliable message delivery, such an approach suffers from high delivery latencies during migration. In a scenario where the overlay is re-evaluated every few minutes, as in the GooPS case, the above approach incurs frequent disruptions of the pub/sub service. There is thus a need to generate an efficient plan which migrates brokers from an initial topology (i.e., the current topology) into a goal topology (i.e., the optimized topology).

Shortcomings of existing delivery guarantees for distributed application development: While pub/sub assumes complete asynchrony and decoupling, and ensures anonymity of clients by providing a data-centric communication paradigm, developing a distributed application only based on such a model can be challenging. A component of a distributed application, deployed on top of a pub/sub substrate, has limited knowledge of other components of the application. Therefore, when any sort of synchronization is required between these components, they need to find each other and resort to out-of-band communication. For example, two components may need to wait and buffer messages until publication streams are established. Note that precise time synchronization protocols such as PTP [39] or distributed coordination services like ZooKeeper and Chubby [40, 41] are useful in single-datacenter applications. However, their application in multi-datacenter scenarios is limited or they suffer from low throughput due to the impact of high latency cross-datacenter communication on their quorum-based mechanisms [42]. Furthermore, in overlay-based pub/sub systems, a message is considered installed in the overlay, when it is fully propagated

in the overlay and processed completely by all brokers. However, it takes time for messages to get installed in the overlay. Existing pub/sub systems provide delivery guarantees for matching publications which reach the overlay after subscriptions are installed in the overlay [14]. These delivery guarantees leave out the installation period (*i.e.* the period of time that the message is being propagated and processed in the overlay) unclear, which can at best be clarified using a probabilistic model [25].

1.3 Approach

In this section, we shortly describe four approaches we propose to address the four problems introduced in Section 1.2. In Chapters 4 to 7, we explain these approaches in detail and present our evaluation results.

Highly available pub/sub via gossiping: Providing redundant paths to be used in case of broker failure for message propagation requires addressing creating and maintaining extra links. Link creation can be decided based on the similarity between brokers' interests (similarity-based) [43, 44, 30], based on the brokers' location in the topology (topology-based) [20, 34] or randomly [35, 45]. The similarity-based approaches create extra links to connect brokers with similar subscriptions. During a failure, an affected broker can keep receiving messages from other brokers which also receive the message due to their similar interests but are not affected by the failure. In contrast, the topology-based approaches create extra links across all of their neighbors to bypass a potential neighbor failure. The topology-based approaches do not need to consider the broker interests and only provide extra links to the closest non-faulty broker in order to keep the tree connected in case of failure. Random approaches can be considered a simplification of the similarity-based approaches.

In multicasting trees or topic-based pub/sub systems, due to the limited type of possible publications and subscriptions (i.e., content space), it is possible to cluster brokers interested in each topic [43]. Brokers belonging to the same cluster can form their own dissemination tree or establish redundant links in the cluster [43, 34, 46]. In content-based pub/sub, however, there is no scalable and straightforward way of

establishing such clusters [44]. Yet, similarity metrics can also be used as a way of grouping brokers based on their set of subscriptions [44, 30].

The second issue is maintaining and updating the extra links in a scalable way which is required in order to react to brokers leaving or joining the overlay, broker failures and recoveries, and subscription changes. In order to avoid scalability issues, link updates must be decentralized and avoid requiring knowledge of all brokers. Topology-based approaches can limit a broker's knowledge of the overlay by forcing a hop-based neighborhood [20, 47], which limits adding or updating links to the defined neighborhood. In contrast, similarity-based or random approaches can potentially choose any two brokers to connect [43, 44]. In such cases, epidemic algorithms provide a decentralized alternative [48], trading global deterministic knowledge for scalability, probabilistic knowledge and fault-resilience. Although choosing where to add extra links is more straightforward for topology-based approaches, our experiments suggest that the maintenance cost of such approaches can overwhelm some parts of the overlay. In comparison, epidemic approach can distribute the link maintenance cost more equally.

Establishing and maintaining redundant links is more challenging in pub/sub systems with high number of broker failures or when several consecutive brokers can fail since these links must cover a larger part of the overlay and be able to bypass multiple broker failures. Therefore, high churn pub/sub systems such as MANETs [49] and pub/sub systems that can experience non-uniformly distributed broker failures such as rack failures or datacenter outages, can benefit from a scalable and low-overhead approach to establish and maintain redundant links.

In this work, we present *Gossip-Enhanced Pub/Sub* (*GEPS*), a similarity-based epidemic approach to increase availability of a content-based pub/sub system organized as a tree of brokers in face of multiple broker failures. Our approach creates redundant paths between brokers based on their interest similarities and utilizes them only when delivery through the original dissemination tree is not possible. GEPS provides the same performance and guarantees as its underlying tree overlay when there is no broker failure. In case of failure, GEPS provides best-effort delivery. GEPS is able to tolerate high rates of broker failure by employing a scalable

epidemic-based (a.k.a. gossip) approach that considers broker similarities to update the extra links. To the best of our knowledge, GEPS is the first epidemic approach for increasing availability that is tailored to advertisement-based content-based pub/sub systems with tree topologies.

Popularity-based publication routing: From the point of view of a broker, forwarding efficiency is dependent upon two factors: distance to reach subscribers and number of subscribers (i.e., popularity). On each broker, we want to prioritize forwarding publications with *high popularity* and *nearby* subscribers. In contrast, subscribers which are located far away with unpopular interests, drag down the efficiency of the whole system by attracting publications down a long path of brokers with low utility. Because it only takes one outlying subscriber to match a publication, addressing the binary cost problem will raise the scalability of the system, even under the presence of a small number of disorderly subscriptions.

The typical solution to raise system efficiency is to construct optimal overlays [50]. While this approach works for topic-based systems, it is not adequate to capture the complex relationships between content-based subscriptions, nor is it designed to create tree topologies, which are the focus of our work.

In this work, we present ***Pop****ularity-Based Publication Routing for Content-based Pub/****Sub*** (*PopSub*). PopSub seeks to address the shortcomings of hop-by-hop forwarding by considering its utility on the end-to-end paths, while retaining the desired property of knowledge decentralization. PopSub measures the impact of publication forwarding in terms of the volume and distance of subscriptions that can be satisfied. Publications are then prioritized based on this popularity-based metric. Publication popularity estimation is performed in a scalable way which does not require any change to the RPF, nor global knowledge of all paths. Furthermore, it is a lightweight mechanism which does not require overlay reconfiguration. We propose three alternatives to the main dissemination tree, namely *Direct Delivery*, *Batching*, and *Gossiping*, to deliver less popular publications to improve system performance.

Incremental topology transformation: In order to migrate the overlay brokers without stopping the pub/sub service, we use an approach called *Incremental Topology Transformation* (ITT) [51]. In this approach, a *transformation plan* is defined as a

sequence of steps that must be applied to incrementally transform the topology into the target form, while guaranteeing reliable message delivery during the execution of the entire plan, and minimizing disruption to the pub/sub service. By leveraging this incremental approach, a pub/sub system can undergo constant topology changes while minimizing the impact of transformations as perceived by the clients (i.e., the publishers and subscribers). Furthermore, execution of an incremental transformation plan can always be partially halted, since every intermediate topology produced by the plan is valid and will guarantee reliable delivery.

Existing works address the problem of optimal overlay design for a particular workload [52, 53], or define primitive operations for reconfiguration of pub/sub topologies (we simply refer to these primitive reconfiguration operations as *operations*) [33, 54, 55]. However, the problem of generating a plan that transforms an initial topology to a goal topology using these operations has been largely ignored. Yoon *et al.* formulate the ITT problem as an automated planning problem[51]. However, they only provide heuristics to calculate a plan which may not be optimal. In other words, the plan is not guaranteed to contain the minimum number of steps needed to perform the transformation.

In this work, we present IPITT: an integer programming-based (IP) approach to the ITT problem. IPITT calculates a sequence of steps that transform an initial topology to a goal topology. Executing each step of the generated plan results in a valid topology with correct routing information. Furthermore, the plan contains the minimum number of steps required to perform the transformation. IPITT uses an integer linear programming formulation of the automated planning problem to solve the ITT problem. An IP-based approach provides a formulation of the problem that is easy to extend in order to add new constraints to the solution, leverages existing highly optimized commercial IP solvers, and generates optimal transformation plans.

Pub/sub delivery guarantees for distributed application development: In order to reduce the complexity of developing distributed applications based on a pub/sub substrate, we introduce a set of delivery guarantees which addresses the message propagation delay in an overlay-based pub/sub system and simplifies the logic of a distributed application by reducing the need for out-of-band communication

and synchronization between different components.

Figure 1.3.1 shows an example inspired by an application based on an industrial workflow management system [56]. Here, a distributed application has one dispatcher component, several workers and clients producing jobs to be executed. Clients advertise about the jobs they will be submitting for execution. Upon receiving this notification, the dispatcher decides to create a new worker or assign the client's jobs to an existing worker. Upon creating a new worker, the dispatcher sends a subscription to the worker in order to for the new worker to subscribe to the jobs. Using existing delivery guarantees, the client and the dispatcher must wait and check until the worker has submitted its subscription and the subscription has been propagated in the overlay. If it were possible for the dispatcher to clearly specify in the newly generated subscription the starting point of receiving jobs by the worker, regardless of overlay size and propagation time, there would be no need to resort to direct out-of-band communication with the worker.

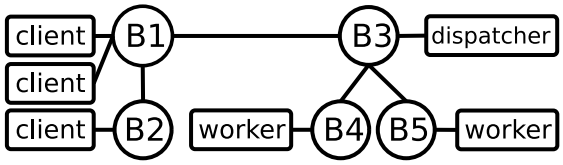


Figure 1.3.1: Example of a distributed application

Existing approaches either study the impact of the message propagation delay in a distributed pub/sub system or extend the pub/sub paradigm to facilitate distributed application development [25, 57, 58]. The latter group of approaches propose a reply message in order to maintain a weak coupling between clients. While the ambiguity of delivery guarantees during message propagation and lack of some form of weak coupling or synchronization have been identified by the existing approaches, there is no comprehensive approach that addresses this issue.

In this work, we propose a set of *Pub/Sub Delivery Guarantees* (PSDG) which isolates clients from the propagation time of messages in the overlay. By introducing these new delivery guarantees, clients can express weak coupling and synchronization requirements that simplify application development using the pub/sub model.

1.4 Contributions

In this section, we describe the contributions of this work with regard to each of the described problems. In order to address availability of a pub/sub service we provide:

1. A similarity metric to measure content similarity between two brokers.
2. A low-overhead, low-latency and scalable, epidemic protocol that complements content-based pub/sub message dissemination by providing improved message delivery during broker failures.

In order to improve resource efficiency of publication routing is a pub/sub service, we provide:

3. A metric which measures the gain of forwarding a publication based on distance and popularity of downstream subscribers. Additionally, we propose a lightweight and scalable way to estimate the popularity of publications among subscribers at each broker without requiring global knowledge and by piggybacking on existing pub/sub traffic.
4. Three alternative dissemination mechanisms for handling low priority publications: *Direct Delivery*, *Batching*, and *Gossiping*.

In order to improve maintainability of the pub/sub service via incremental topology transformation, we provide:

5. An IP-based formulation of the ITT problem, which facilitates integration of custom plan constraints and enables existing IP solvers to generate optimal ITT plans. Additionally, we provide an ITT planner that calculates transformation plans using different operations while minimizing disruptions to the pub/sub system.
6. A mechanism to perform and coordinate plan execution in order to prevent an incorrect routing state or invalid topology.

In order to improve the usability of the pub/sub paradigm for distributed application development, we provide:

7. A clearly defined message installation mechanism that allows client to be notified of when their messages are fully propagated in the overlay. Furthermore, we introduce three delivery guarantees for subscriptions that clarify the set of publications delivered during the subscription propagation period.

8. New routing algorithms for distributed pub/sub systems which realizes the proposed delivery guarantees in a decentralized and scalable fashion.

Parts of the content and contributions of this work have been published in or are submitted to the following venues:

- P. Salehi, C. Doblander, and H.-A. Jacobsen, "Highly-available content-based publish/subscribe via gossiping," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, (New York, NY, USA), pp. 93–104, ACM, 2016 [59]

- 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*, DEBS '17, (New York, NY, USA), pp. 88–99, ACM, 2017 [60]

- P. Salehi, K. Zhang, and H. A. Jacobsen, "Incremental topology transformation for publish/subscribe systems using integer programming," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 80–91, June 2017 [61]

- P. Salehi, K. Zhang, and H. A. Jacobsen, "On delivery guarantees in distributed content-based publish/subscribe systems." Submitted to the 19th ACM/IFIP/USENIX Middleware Conference, 2018 [62]

1.5 Organization

The remainder of this document is organized as follows. In Chapter 2, we provide a short introduction to distributed pub/sub systems and gossip protocols (*a.k.a.* epidemic algorithms). Chapter 3 discusses the related works regarding the four problems introduced in Section 1.2. In Chapter 4, we describe our proposed similarity metric, epidemic protocol for a highly available pub/sub service and present a detailed experimental analysis. Chapter 5 defines the binary cost model for routing and describes our solution, which includes metric collection, publication prioritization, alternative dissemination mechanisms, and traffic handover, and presents the results of our evaluation. Chapter 6 explains the ITT problem, its IP formulation and our proposed approach and evaluation results. In Chapter 7, we clearly define the problem of existing delivery guarantees, introduce our new delivery guarantees, the new routing algorithms to provide these guarantees and their evaluation results. Finally, we draw conclusion in Chapter 8.

Chapter 2

Background

In this chapter, we shortly explain the background information necessary to understand our proposed approaches. First, in Section 2.1, we explain the distributed pub/sub system and its variations. Next, in Section 2.2, we explain gossip protocols since we use them to provide a highly available pub/sub service and briefly introduce the *Lightweight Probabilistic Broadcast* protocol which we use in our popularity-based routing approach. Lastly, in Section 2.3, we explain *automated planning* which is how we formulate our broker migration problem.

2.1 Distributed Pub/Sub

Pub/sub provides a loosely coupled, asynchronous, and selective communication substrate for the dissemination of data between data producers and consumers [13, 28, 14]. The pub/sub system is in charge of matching and forwarding incoming data against registered subscriptions in order to deliver data to interested subscribers. In practice, pub/sub is widely employed for high throughput and low latency data dissemination [38, 63, 64, 65].

There exists three entities in a pub/sub system. A *publisher* is a process producing

and disseminating data, a *subscriber* is a process that is interested in and consumes the data produces by publishers, and a *broker* which is responsible for receiving, matching and forwarding data. Note that, a pub/sub system can be *brokerless* whereby publishers and subscribers connect to each other in an unmanaged peer-to-peer overlay network. In this work, we focus on pub/sub systems with dedicated and managed brokers.

To raise scalability, pub/sub systems are often distributed, where the task of matching and forwarding publications to interested subscribers are divided among a network of pub/sub brokers, collectively called the pub/sub overlay network. In such an overlay-based pub/sub system, brokers are connected together in a topology that determines the neighborhood relationship between brokers. In many systems, the topology of the overlay is organized as a connected tree of brokers [13, 28, 14, 66]. Clients (publishers and subscribers) connect to the brokers in order to publish data or subscribe to data. The broker connecting the client to the overlay is called the *edge broker* (*EB*) of that client. A routing protocol dictates how subscriptions are routed through the overlay and how publications traverse the overlay to reach the intended subscribers. Each broker knows only of its direct neighbors in the overlay and can route subscriptions and publications using its local routing information to the next hops.

There exists three variations of the pub/sub systems depending on how publishers and subscribers describe the data they publish or are interested in: *topic-based*, *typed-based* and *content-based* pub/sub.

In topic-based systems, each publication is associated with a topic name. This topic name simply serves as an ID to identify the set of all publications published under that topic. Each client can subscribe to one or more topics and receive all publications published on these topics. As an example, consider a set of machines that each serve as a sink to collect different measurements such as temperature and humidity from deployed sensors. Each server is responsible for collecting and publishing data for a specific area. The set of all of these servers, the back-end in charge of data management and the rest of the system using this data are all connected using a pub/sub system. In a topic-based system, each measurement is

published under a topic name such as `temperature` or one topic per area such as `munich`.

In type-based systems, publications are categorized depending on their types. Therefore, a publication might publish temperature values of type `TemperatureType` or `PerLocationMeasurements`. Subscribers can then express what types of publications they are interested in. The main difference between topic-based and type-based pub/sub systems is that, the later can enforce type safety, making sure only messages of a specific type or its subtypes are sent to interested subscribers.

In content-based systems, in addition to a type or topic, clients can specify filters defined over the publication contents. For example, a subscriber with the subscription `s = {class=temp, value > 20}` receives only temperature publications that have a value higher than 20. In this example, class is just another attribute of a message and depending on the content-based pub/sub system, `temp` can be a topic, namely merely a tag or ID, or it can be an actual type enforced by the pub/sub system.

In a topic-based or type-based system, subscribers can only express their interest by specifying the type of events or using predetermined topics. However, content-based pub/sub allows more fine-grained subscriptions using additional attributes and filter on the content of the messages, which reduce bandwidth usage. The content-based pub/sub system can be considered a generalization of topic-based and type-based systems which offers clients a higher degree of expressiveness.

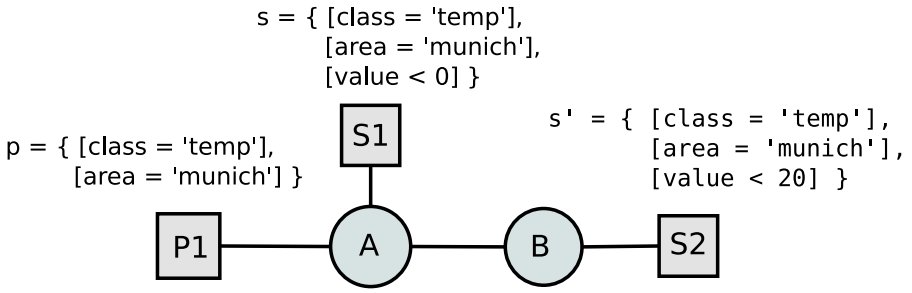


Figure 2.1.1: Content-based Pub/sub

In order to able to forward publications to matching subscribers, brokers need to know all matching subscribers for a publication. There exists two approaches for propagating subscriptions in a content-based pub/sub system: subscription flooding

and advertisement-based routing.

In a system that uses subscription flooding, any subscription received by a broker is simply flooded in the overlay. Upon receiving a subscription, the broker forwards it to other brokers in the overlay. Each broker stores the subscription and the address of the broker sending the subscription in a table called the publication routing table (PRT). Upon receiving a publication, a broker matches the publication against existing subscriptions to determine the next hops that should receive these subscriptions. Table 2.1.1 shows the PRT of Broker A in Figure 2.1.1. For example, upon receiving the publication `p = {[class = 'temp'], [area = 'munich'], [value = 10]}`, Broker A forwards p only to Broker B.

Subscription	NextHop
`[class = 'temp'], [area = 'munich'], [value < 0]`	*S1*
`[class = 'temp'], [area = 'munich'], [value < 20]`	B

Table 2.1.1: PRT of Broker A

In a subscription flooding approach, all brokers record all subscriptions in their routing tables. In scenarios with high number of subscriptions, subscription flooding can result in large PRTs on the overlay brokers. Large PRTs can in turn result in higher memory usage and higher latency as matching incoming publications with a higher number of subscriptions can take longer. Furthermore, in larger overlays, flooding subscriptions can also result in high message overhead.

Note that not all brokers need to know all subscriptions. Only brokers which know of publishers matching a subscription need this information. Advertisement-based routing introduces an advertisement message in order to limit subscription propagation only to matching publishers. In this approach, publishers must first publish an *advertisement* to notify all other brokers about the data they will publish. Similar to a subscription, an advertisement has a *class* and zero or more filters that must be true about all publications send to the overlay by the publisher. In Figure 2.1.1, *P1* sends the advertisement `a = {[class = 'temp'], [area = 'munich']}` to the overlay and any publication published by *P1* must be of class `temp` and the value of the attribute `temp` must be `munich`. Upon receiving an advertisement, brokers store the advertisement and the address of the broker sending the advertisement in a

table called the subscription routing table (SRT). Tables 2.1.2 and 2.1.3 show the SRTs of Broker A and B after flooding the advertisement of $P1$.

Advertisement	NextHop
`[class = 'temp'], [area = 'munich']`	*P1*

Table 2.1.2: SRT of Broker A

Advertisement	NextHop
`[class = 'temp'], [area = 'munich']`	A

Table 2.1.3: SRT of Broker B

Upon receiving a subscription, brokers match the subscription against existing advertisements recorded in the SRT and determine the next hops that should receive the subscription. Similar to the subscription flooding approach, received subscriptions are recorded in the PRT of the receiving broker. However, since only brokers with matching advertisements receive the subscription, subscriptions are recorded only on the path connecting the subscriber and matching publishers. Therefore, as shown in Table 2.1.4, only *S2*'s subscription is recorded in the PRT of Broker B.

Subscription	NextHop
`[class = 'temp'], [area = 'munich'], [value < 20]`	S2

Table 2.1.4: PRT of Broker B

The rationale behind advertisement-based pub/sub is that usually the number of subscribers in a system is much larger than the number of publishers. Therefore flooding advertisements results in lower message overhead and smaller PRTs which can improve matching time of publications. In this work, we use advertisement-based content-based pub/sub systems due to their efficiency and high expressiveness.

Another optimization that is applied in pub/sub systems is *subscription covering*. Subscription covering [36] is an optimization technique used in distributed pub/sub systems in order to reduce the size of the PRTs on brokers and consequently reduce memory usage and improve publication matching time. In this approach, upon receiving a subscription s on Broker B, if there already exists a subscription s' in the PRT of Broker B such that $s \subseteq s'$, Broker B does not forward s onward

since the set of publications that matches s is a subset of the set publications that matches s'. For example, in Figure 2.1.1, if before receiving s', there already exists a subscription `s" = { [class = 'temp'], [value < 40] }` on Broker B, s' would not be propagated because Broker B receives a superset of publications matching s' due to s''.

2.2 Gossip Protocols

Gossip protocols were first introduced for maintenance of replicated databases [67] and provide a simple approach for disseminating information among distributed processes. In contrast to overlay-based approaches, gossip protocols do not rely on an underlying topology or routing protocol and do not require all the processes to know each other and provide some inherent fault resiliency [68]. Gossip protocols are scalable with the network size and the number of gossiping rounds required to spread a message through out a network is $O(\log(n))$, where n is the size of the network [69]. Simplicity and inherent fault-tolerance of gossip protocols have made them a popular choice for event-dissemination [70, 71], overlay construction and maintenance [72, 73], monitoring and resource allocation [74, 75] and aggregation in distributed systems [76].

Gossip is defined as *repeated probabilistic exchange of information between two members* [69]. This definition emphasizes two main characteristics of gossip protocols. First, processes exchanging information choose each other randomly. Second, spreading information using gossiping is continuous which can result in large message overhead if not controlled [69, 77, 48].

Gossip-based protocols follow the same general set of steps [69] and are only different from each other in the substeps taken for each of the following main steps:

1. *Peer selection* determines the set of processes each process selects from the network to exchange information with.
2. *Data exchange* determines what information is exchanged between the two

communicating processes depending on the application and the context of the problem.

3. *Data processing* determines what a process does with the received information and how spreading the information impacts the process and system state.

A peer sampling service is an important basic building block of gossiping protocols. The aim of this service is to provide every process with other processes to exchange information with [78]. It has been shown [79] that a partial random uniform view of the whole network can replace the full knowledge of all the processes in the system. Therefore, gossiping processes are able to update their partial view of the system using the peer sampling service without knowing about all existing processes. The peer sampling service uses a *membership protocol* to keep track of all existing processes. This membership protocol can be centralized where all processes register themselves with a server or it can be decentralized [48]. An important issue in distributed membership protocols is network partitioning. If the membership protocol does not consider the dynamicity of the overlay, the processes in the network can partition into isolated parts. A consequence of this is incorrect behavior of the peer sampling service since it is no longer possible to provide a uniformly random subset of processes from the whole network.

In the following, we mention some important factors that impact the trade-off between message overhead and efficiency of information dissemination using gossiping [67]:

- *Fanout* determines the number of gossip partners each process contacts in each round. A higher fanout results in a faster dissemination at the cost of a higher message overhead.
- *Cache size* determines how many times each message is gossiped before it is removed from the list of messages that the process is disseminating using gossiping. The longer a message stays in the gossip list, the larger the part of the network receiving it and the more copies of that message exists in the system which results in higher message overhead. Demers, *et al.* provide several methods to decide when to stop gossiping a message and remove it from the cache [67]. A *blind* approach stops gossiping a message based on a fixed counter

value or a $1/k$ probability. A *feedback-based* approach relies on the response of the receiving processes to decide whether the message should be gossiped which means a message is gossiped repeatedly as far as each round of gossiping results in some positive feedback.

- *Data exchange model* determines how information is exchanged between the processes once they have chosen each other. In a *pull* method, on each gossiping round, a process queries selected processes for new information. *Pull* has a low overhead since a message is sent only if it is new and the pulling process is interested, however, it results in a slow dissemination speed. In a *push* method, the gossiping process sends its latest gossips to the selected processes. A message disseminated using this approach reaches the network quickly but at the cost of a large number of redundant messages. *Push-pull* performs both of the previous methods at each gossiping step and *lazy-push* first checks weather a chosen process is interested before sending the information [67, 69].

Lightweight Probabilistic Broadcast (*lpbcast*) is a gossip-based broadcast protocol which provides a scalable approach for large-scale data dissemination [79]. *lpbcast* uses partial views to perform message routing and membership management in a decentralized and scalable fashion. Therefore, processes can avoid global knowledge and rely only on a partial view of the system. This partial view consists of a subset of the processes in the system with a fixed maximum size. On each process b, besides the partial view ($\mathbb{V}_b$), each process maintains the following four sets:

- $\mathbb{M}_b$: set of messages to be gossiped by b
- $\mathbb{D}_b$: a digest of messages gossiped and seen by b
- $\mathbb{S}_b$: set of subscribers (process IDs) that b is aware of
- $\mathbb{U}_b$: set of unsubscribers that b is aware of

Periodically, b creates a gossip message $g = (\mathbb{M}_b, \mathbb{D}_b, \mathbb{S}_b, \mathbb{U}_b)$ and sends it to a random subset of processes in $\mathbb{V}_b$. A receiving process, b', uses $\mathbb{S}_b$, and $\mathbb{U}_b$ to update its partial view. This allows gradual removal of processes not interested in receiving

messages anymore and gradual integration of new processes. Any message in $\mathbb{M}_b$ not previously seen by b' is eligible for delivery and gossiping in the next round. The digest received from b is used to update the knowledge of b' of published messages and retrieval of missing messages. *lpbcast* also supports periodical retrieval of missing messages.

2.3 Automated Planning

Automated planning is a branch of artificial intelligence studying computation of a finite sequence of steps, chosen from a limited set of basic actions, to realize a complex task. A planning problem Π is defined as a tuple $<P, \Sigma, O, s_0, g>$ where: P is a set of predicates used to describe the problem state, Σ is a set of objects, O is a set of operations, s_0 is an initial state and g is a goal state. Each predicate $p \in P$ and each operation $o \in O$ takes one or more objects $\sigma \in \Sigma$ as parameters. A proposition is a predicate $p \in P$ applied to a subset of objects $\Sigma' \subseteq \Sigma$. A state s is a set of true propositions that uniquely defines a problem state. An action a is an operation $o \in O$ applied to a subset of objects $\Sigma' \subset \Sigma$. Applying an action on a state $\gamma(s, a)$ transitions the state s to a new state s'.

Each operation $o \in O$ is defined using three sets of predicates $pre(o), add(o), del(o)$. $pre(o)$ is the set of preconditions of operation o. For an action a to be applicable to a state s, s must satisfy all preconditions of a. $add(o)$ and $del(o)$ are the sets of predicates which are added to or removed from the state as a result of applying an action to s. Therefore, each action a changes the state s by changing the set of propositions defining s.

A solution to the planning problem Π is a sequence of actions $(a_1, a_2, \ldots, a_k)$ that corresponds to a set of state transitions $(s_0, s_1, \ldots, s_k)$ such that $s_1 = \gamma(s_0, a_1), \ldots, s_k = \gamma(s_{k-1}, a_k)$, where s_k is a goal state [80]. The synthesized sequence of actions that transitions the initial state to the goal state is called a plan.

Chapter 3

Related Work

In this chapter, we discuss existing approaches related to availability, maintainability, resource efficiency and usability of pub/sub systems. As these requirements are not entirely orthogonal to each other, many studies target two or more of them at the same time. As an example, approaches improving maintainability of a pub/sub overlay can be applied to improve availability, as well as resource efficiency of the system. Therefore, we categorize the related work based on topics that are related to one or more of these requirements.

3.1 Highly-Available Group Communication

Many tree-based multicasting and broadcasting approaches have been proposed that address high availability [47, 45, 34]. GoCast [47], for example, is a protocol for building and maintaining resilient multicast overlays. GoCast first builds a mesh to connect all brokers where roughly each node has the same number of neighbors. An efficient dissemination tree connecting all brokers is constructed and embedded in the overlay mesh. The remaining links of the mesh are used to optimize and maintain the dissemination tree in case of broker failure. Streamline [45] follows a similar approach for media streaming in overlay networks.

In these scenarios, either all brokers are interested in all messages (broadcast) or there are few different publishers with many similar subscribers. Therefore, the cost of matching interests and routing messages is non-existent or very inexpensive. Consequently, approaches like GoCast or Streamline aim at providing a dissemination tree that can reach as many brokers in the overlay as possible with lowest latency. Although these assumptions fit in the context of group communication, applying them to pub/sub systems where brokers have to manage fine-grained interests can lead to high message overhead. In other words, considering all brokers of the overlay to have the same or similar interests is often not a correct assumption in pub/sub systems.

3.2 Highly-Available Pub/Sub

In topic-based pub/sub systems, each message has a type (or topic ID) and is routed based on this information. Hence, it is feasible to build and maintain one dissemination structure per topic [46, 43, 81]. Providing redundancy inside each of these per-topic-structures can increase availability of the pub/sub system. Vitis [43] and Poldercast [46] are examples of enhancing topic-based pub/sub with epidemic approaches.

Vitis [43] clusters peers with similar topics. The clusters have a maximum size resulting in multiple clusters for the same topic which are connected through intermediary peers. Gossip is used to keep the clusters connected with a minimum number of intermediary peers. Poldercast [46] pursues goals similar to our work such as 100% delivery without node churn and high delivery in presence of node churn. Poldercast uses deterministic dissemination over a DHT-based ring. Besides DHT successors and predecessors, peers use gossip to connect to additional peers with the same topics.

While both of these approaches enhance pub/sub using epidemic protocols and address scalability and peer failure, they exclusively target topic-based pub/sub where the concept of grouping brokers based on topics is applicable. Furthermore,

these approaches are meant to support peer-to-peer systems where typically there is a high churn rate and no administrated infrastructure. With GEPS however, we are not targeting self-organizing peer-to-peer systems but a pub/sub system with a managed overlay, where it might not always be possible to arbitrarily change the overlay topology.

Although, not as simple to realize as for topic-based pub/sub, similarity-based clustering in content-based pub/sub systems have been studied in the context of self-organizing overlays [44, 30]. Here, brokers with similar interests are clustered to lower delivery latency and improve message dissemination efficiency. In Sub-2-Sub [44], for example, the ring structure used per cluster also provides fault-tolerance against broker failure. While we also use a similarity measure, we do not target self-organizing peer-to-peer overlays and aim to provide an approach that does not require change to the underlying dissemination tree. There are scenarios in which the dissemination tree is designed based on an application workload or certain infrastructure constraint [82, 83]. In such cases it is desirable to maintain the original topology since self-organization can be costly. Furthermore, in contrast to self-organizing pub/sub systems [44, 30] where each broker only considers its local interests, we use similarity metrics to measure similarity between two brokers in terms of pub/sub messages that they forward and not just their local interests.

Costa *et al.* [84] study the use of gossiping to improve reliability of content-based pub/sub systems. In their approach, each broker periodically samples its subscription routing table and propagates a digest along the dissemination tree. Any broker receiving the digest and noticing a missing publication can request a retransmission. In contrast, GEPS addresses the intervals when the dissemination tree is disconnected and minimizes the average delivery latency and retransmissions required after recovery. Such retransmission mechanisms can complement GEPS to deliver publications that could not bypass failures.

Two approaches that pursue goals very similar to ours are semi-probabilistic [35] and δ-fault-tolerant [20] pub/sub (δFT). Semi-probabilistic pub/sub targets MANETs where peers have limited resources and high churn rates. Subscription propagation is limited to n hops away to limit memory usage, where n is a configuration parameter.

To further propagate messages when no subscription information is available as a result of limiting subscription propagation, messages are gossiped by randomly selecting neighbors. Although, this approach matches the requirements of MANETs well, in low churn rates and large overlays, random next hop selection can result in high message overhead and low delivery. The reason is that publishers located more than n hops away from the subscribers can only rely on random next hop selection instead of routing based on subscriptions to reach the subscribers. The routing scheme described in this approach does not propagate subscriptions throughout the whole overlay. Since the same routing scheme is used when there are no failures in the overlay, the lack of 100% delivery and high message overhead is present even when there is no churn in the system.

δFT pub/sub targets more stable and managed environments, such as enterprise systems or a pub/sub service providers, and provides a reliable pub/sub service with 100% delivery and per-publisher ordering guarantees. δFT follows a topology-based approach to augment the tree topology using extra links. Extended subscription messages are used to keep track of brokers up to $\delta + 1$ hops along each branch of the dissemination tree. Additional directed links are established and maintained using this information. The established links are used to tolerate up to δ concurrent and consecutive failures should broker failures occur. The reliability of the approach depends on the value of δ, which should not be too high as otherwise scalability issues may ensue and requires that each node maintains more links to cover a bigger neighborhood. Low values of δ make the overlay vulnerable to broker failure, especially when these failures are in close proximity and there is a higher probability for $\delta + 1$ concurrent and consecutive failures. For example, in Figure 3.2.1, with $\delta = 1$, the neighborhood of Broker 2 (*B2*) consists of *B4*, *B5*, *B8*, *B9*, *B10*, *B11*, *B1* and *B3*. The size of the neighborhood grows quickly with the value of δ. An advantage of this approach is that establishing extra links based on the tree branches maintains per-publisher ordering guarantees.

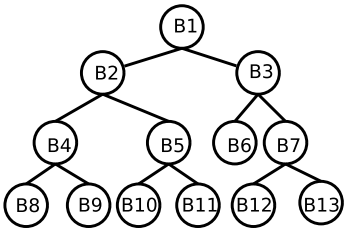


Figure 3.2.1: δFT Overlay example

Compared to the previous two approaches [35, 20], with GEPS, we aim to enhance

content-based pub/sub by using epidemic dissemination controlled by the content-space and tree overlay. Our approach takes advantage of the scalability of epidemic algorithms similar to semi-probabilistic pub/sub, while increasing the publication delivery and lowering the delivery latency during broker failure similar to δFT. In contrast to purely gossip-based pub/sub systems where gossiping is used as the main publication dissemination mechanism [79], GEPS resorts to gossiping only during broker failures. Consequently, GEPS guarantees 100% delivery in case of no broker failure and high delivery in case of failure. GEPS aims to increase delivery and lower latency during broker failure of a content-based pub/sub system but does not directly aim at enhancing reliability of pub/sub. This is different from δFT which provides per-publisher ordering and delivery guarantees which imposes higher overhead and does not target high overlay failure rates. Compared to the deterministic approach of δFT, GEPS aims at providing a probabilistic alternative with low message overhead while maintaining scalability and fault-tolerance inherent to epidemic approaches. Furthermore, our approach does not change the underlying pub/sub routing but rather complements it with a low-overhead protocol to reduce latency and message loss during broker failure.

3.3 Self-Organizing Overlays and Overlay Reconfiguration

In managed overlays, the topology is designed for efficient publication routing and high throughput. In contrast, self-organizing overlays rely on gradually constructing and periodically reconfiguring the overlay to achieve these properties. Several existing works use self-organizing overlays to avoid costly publication forwarding by identifying middle steps which are pure forwarders (brokers with no local interest) [85, 86, 87]. Scribe is a DHT-based peer-to-peer multicasting system which can route publications via interested peers and thereby avoid unnecessary publication forwards. Spidercast [86] relies on its own protocol to construct an efficient overlay for topic-based dissemination. These works only target multicast or topic-based systems with self-organizing overlays, whereas PopSub addresses content-based pub/sub using managed overlays.

Some studies suggest overlay reconfiguration to improve publication delivery efficiency in content-based systems with managed overlays [55, 88, 89]. In these approaches, brokers monitor and compare their subscriptions; brokers with similar interests are then connected together to reduce number of publication forwarding operations required to deliver the publications.

PopSub differs from self-organizing and reconfiguration works in three ways. First, performing these reconfigurations is costly as they require to buffer publications and stop routing while paths are updated [51]. Therefore, overlay reconfiguration temporarily increases publication delivery latency [55] which may not be tolerable in all use cases. Furthermore, overlay construction and transformation may also not be achievable when the network is geographically-aware [90], or when the subscription churn is high [91].

Secondly, overlay reconfiguration is optimized for the benefits of large clusters of similar subscriptions without considering isolated subscriptions. For example, in Figure 3.3.1, since majority of subscriptions on *B9* are black, clustering brokers and reconfiguring the overlay accordingly is more likely to benefit black subscriptions. Consequently, assuming a uniform distribution of subscriptions on brokers, there will always be isolated subscriptions. PopSub on the other hand, can break down subscriptions on each broker and use different dissemination mechanisms for a subset of a broker's subscriptions. In other words, while self-organization and reconfiguration can benefit subscribers of popular contents, PopSub is still applicable in such systems to identify unpopular content and improve the overall publication routing cost even more.

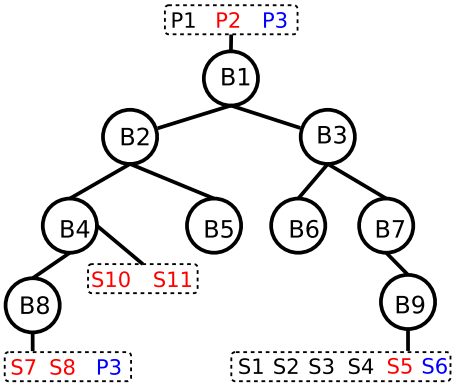


Figure 3.3.1: Reconfiguration example

Lastly, existing works can only improve routing efficiency by avoiding pure forwarding brokers. In scenarios where middle forwarding brokers have only one subscription matching the routing publication, they cannot improve the routing efficiency and

therefore suffer from the *binary cost model problem* (Section 5.1).

In comparison to IPITT which focuses on topology transformation of managed overlays, overlay reconfiguration techniques focus on reducing delivery latency and minimize the routing cost in self-organizing overlays which are more common in peer-to-peer systems. Furthermore, the combination of different overlay design algorithms and IPITT provides a general framework that allows topology maintenance of pub/sub systems subject to a wide range of constraints.

Changing the overlay topology of a running system requires clearly defined reconfiguration operations in order to prevent message loss, invalid routing state, and service disruption. Yoon *et al.* propose a set of operations used to transform an overlay while maintaining delivery guarantees [33]. Nitto *et al.* also propose operations for self-adaptive overlays [54]. While the proposed set of operations between these two works are similar, they provide different delivery guarantees and can handle different failure models. IPITT can produce plans using any operation (including those in the above sets) that can be defined in terms of preconditions and effects on the overlay. For example, the *swapLink*[54] and *shift*[33] operations are equivalent but with different delivery guarantees. Therefore, depending on the delivery guarantees that the plan must maintain, the corresponding operations can be used in the IPITT planner.

3.4 Overlay Topology Design

A minimum cost topology minimizes the overall cost of routing messages in the overlay. Minimum cost topology construction is a NP-hard problem [92, 93]. Nonetheless, there exists many algorithms which construct overlay topologies given different constraints while minimizing routing cost [92, 52, 94, 93]. Elshqeirat *et al.* provide different heuristics to design minimum cost overlay topologies subject to reliability constraints [92]. Chen *et al.* present algorithms for designing topic-connected overlays with fast implementations [52]. Weighted overlay design incorporates underlay information and is used to design overlay topologies that maintain their performance

in the presence of underlay changes [94]. These works are orthogonal to IPITT and can be used to generate goal topologies which are then provided to IPITT for generating efficient transformation plans while minimizing disruptions.

3.5 Overlay Topology Transformation

Several approaches address performing overlay reconfiguration on an online system using a continuous and incremental approach. ZUpdate provides a plan-based approach to migrate the topology of SDN-based datacenter networks (DCN) [95]. Similar to IPITT, ZUpdate uses integer programming to find a transition plan, consisting of *zUpdate* operations, to migrate the topology of DCNs. However, the proposed operations and plans are specific to OpenFlow switches in DCNs. Furthermore, rather than migrating to a new topology, the focus of the work is updating the switches without overloading existing links when specific switches are down.

Yoon *et al.* address incremental topology transformation in pub/sub systems [51]. They formulate the ITT problem as an automated planning problem [80] and use existing planners [96, 97] to find a solution. However, as existing planners are not scalable, the authors propose heuristics for the ITT problem. Similarly, IPITT uses an automated planning formulation, but uses integer programming to solve the automated planning problem. The IP-based approach of IPITT leverages commercial solvers such as Gurobi [98] and CPLEX [99] to solve the problem. These solvers are heavily optimized and can efficiently exploit multicore and cluster architectures [100]. Furthermore, IPITT can customize the plan search by changing the objective function of the IP model or adding constraints to the plan search. Section 6.5 presents experiments which compares our solution (IPITT) to the state of the art heuristic proposed by the aforementioned approach.

GooPS [38] is an internal pub/sub system used for online service integration at Google. GooPS uses a controller that periodically recomputes a cost-minimal tree based on the existing workload, network underlay, and utilization information that

it collects. GooPS uses a central routing approach which relies on Chubby to maintain all the subscriptions and routing information. Routing updates resulting from overlay changes are addressed with versioning of the routing information on Chubby. Unlike GooPS, IPITT focuses on overlay-based pub/sub systems where the routing information is distributed across brokers.

3.6 Efficient Publication Routing in Pub/Sub

Opportunistic multipath forwarding (OMF) [101] improves publication routing efficiency in content-based pub/sub systems by building and maintaining extra links on top of the managed overlay topology. These links are used to bypass pure forwarding brokers and improve delivery latency and throughput. For example, in Figure 3.3.1, using OMF, *B1* should forward publications directly to brokers *B4*, *B8* and *B9*, since *B2*, *B3* and *B7* have no local black, red or blue subscriptions. However, this increases number of publication forwards that *B1* needs to perform in order for all subscribers to receive their matching publications. In topologies with a higher fanout degree and for more popular publications, this can result in a high number of forwarders for *B1*. To avoid overwhelming forwarding brokers, OMF performs these opportunistic bypasses considering the free capacity available at the forwarding broker. For brokers which are utilized more than a certain threshold, OMF reverts back to forwarding publications only via the tree, assuming the managed overlay topology is designed to tolerate such loads. In comparison, PopSub considers the overall performance using the publication gain metric and can improve publication delivery even under high loads.

Atmosphere [102] identifies situations where clients suffer from several intermediate forwarding hops due to the present overlay topology. For each publisher, Atmosphere identifies the most relevant subscribers to be directly served by the publisher, forming *überlays*. Therefore, Atmosphere allows faster publication delivery to subscriptions served by direct links. In contrast, PopSub does not involve changing client connections and does not rely on client resources to improve delivery latency. While Atmosphere provides faster than overlay deliveries, PopSub prevents delayed publications due to

congestion and improves overall resource utilization.

Despite the different scenarios addressed by OMF and Atmosphere, the idea of directly delivering publications to improve resource utilization and publication delivery latency is applicable in our popularity-based routing approach as well. Therefore, one of the proposed methods to handle unpopular publications, *Direct Delivery*, is based on these two works. We evaluate and compare the performance of such a bypassing mechanism on our proposed popularity-based routing scheme.

QoS-aware pub/sub systems, similar to PopSub, aim to augment the knowledge of brokers about existing subscribers without relying on global knowledge and without degrading scalability [103, 104]. Nonetheless, the aim in these systems is to satisfy precise requirements received from clients (QoS guarantees) by estimating and provisioning the resources required. PopSub on the other hand, aims to improve the overall resource utilization of the pub/sub system without involving clients.

There exists a number of works which handle overload situations in highly congested pub/sub systems. These works either throttle the publication rate [105] or use admission control to accept incoming subscriptions [106]. In contrast, PopSub does not assume control over the publisher and subscriber clients. Our work processes all incoming publications and subscriptions. In overload situations, PopSub employs a novel popularity metric to prioritize the appropriate data flows.

3.7 Pub/Sub for Distributed Application Development

The complete decoupling of clients in pub/sub and consequently its limitation on distributed application development has been pointed out in some studies [58, 57]. These works argue that while pub/sub provides a more flexible and scalable alternative to the traditional request/reply paradigm, the reply mechanism is a natural and useful component missing in pub/sub. Furthermore, adding a reply mechanism increases the domain of applications that can benefit from the pub/sub paradigm.

Existing works on pub/sub-with-reply propose a reply to publication messages and evaluate the alternative algorithms that can be used to provide such a mechanism [58, 57]. Reply management in a scalable fashion and without requiring global knowledge of the overlay has been studied in works addressing pub/sub-with-reply and aggregation in pub/sub [57, 58, 107]. Cugola *et al.* propose two out-of-band and two in-band protocols to collect and deliver reply messages [57]. In one of these protocols, *KER*, upon receiving a publication p, a forwarding broker B looks up the neighbors that lead to a subscriber matching p and waits for one reply from each neighbor before aggregating these replies into one reply and sending it to the last hop that p came from. However, these works only address reply to publication messages and do not address the propagation time for subscriptions and its impact on the provided delivery guarantees.

Baldoni *et al.* identify the message installation problem in distributed pub/sub systems and provide a probabilistic model that can be used to approximate the percentage of notifications that are guaranteed to be delivered during the propagation phase [25]. Nonetheless, they do not provide a solution to clarify the provided delivery guarantees in a distributed pub/sub system.

Kazemzadeh *et al.* also identify the *subscription stabilization delay* in distributed pub/sub systems and propose a clear definition for when the processing of a new subscription is finished [26]. This *point of registration* is defined as the time that all confirmations regarding processing of the new subscription is received by the subscriber. However, this confirmation-based approach is used to ensure that a subscription is successfully installed in the overlay and does not clarify the set of publications delivered to the subscriber while it is propagating in the overlay.

Transactional message processing in pub/sub is required in some use cases such as workflow management [108, 56]. A transaction in pub/sub is a demarcated group of pub/sub messages that provides an all-or-nothing execution semantic and guarantees a consistent routing table at all brokers. Some studies have formulated transactions in pub/sub, define a consistency model and provide ACID properties for such transactions [108, 109]. In comparison, in this work, we address the propagation delay and installation time for single massages in distributed pub/sub systems. In

this context, brokers located on the path between a publisher and a subscriber, do not have inconsistent routing tables but rather it takes some time until they all converge to the same state. Therefore, we suggest providing clients with delivery guarantees that decouples them from the propagation delay in the overlay.

To address scenarios where a publisher needs to buffer messages and forward them to late-joining subscribers, the Data Dissemination Service (DDS) standard provides a durability QoS policy [110] based on a publisher-side history cache which can temporarily hold produced publications. In a distributed pub/sub, such late-joins can happen as a result of the propagation delay. In these cases, before publishing, a component might have to wait for all subscriptions (other components of the system) to get installed in the overlay. However, in DDS, buffering publications and forwarding them to subscriptions that join the overlay later is provided as a client-side functionality which can overload the publishers.

View-oriented group communication systems (VGCS) provide a building block for distributed application development and facilitate many-to-many communication by allowing processes to be organized in logical groups [111]. However, these systems provide a basic multicasting service and lack the selectivity and fine-grained communication that is provided by content-based pub/sub systems. Furthermore, the logical groups defined in the system, are used for routing purposes as well. In contrast, in our content-based pub/sub system, this information is used to simply provide some weak coupling between different processes and the routing is still performed based on filters installed by clients on the brokers.

Lastly, while relying on clock synchronization in Internet-scale distributed systems has been considered unreliable (*e.g.* NTP) or not cost-efficient (*e.g.* GPS clock synchronization), more recently cloud providers offer highly accurate reference clocks (using atomic clocks and satellite-based synchronization in each region) as a free service to their users which can facilitate distributed application development [112, 113].

Chapter 4

Gossip-Enhanced Publish/Subscribe

In this chapter, we present our *Gossip-Enhanced Pub/Sub* (GEPS) approach to provide a highly-available content-based pub/sub service. We first explain a similarity metric that GEPS uses for establishing extra links in the topology. Next, in Sections 4.2 to 4.4 we describe the gossip protocol that is used to maintain the established links and how GEPS uses it to bypass broker failures. In Section 4.5, we explain how the presented gossip protocol can be extended to bypass consecutive broker failures. Lastly, we present the result of out experimental evaluation in Section 4.6.

4.1 Broker Similarity

GEPS complements the advertisement-based content-based pub/sub protocol [14] described in Chapter 2. Figure 4.1.1 gives an example of such an advertisement-based pub/sub overlay. Client 1 (*C1*) intends to publish temperature values about two cities and sends advertisements about them to the overlay. As a result of advertisement flooding, the SRT of all brokers is built. Clients *C2*, *C3* and *C4* issue their subscriptions which are propagated towards *C1* and create PRT entries on

brokers located on the path between the subscriber and the advertiser. GEPS requires each broker to maintain a counter for each advertisement. This counter represents the number of subscriptions the broker has successfully matched with that advertisement. For example, after *C2* sends subscription S_1, the subscription reaches *C1* via *B8*, *B4*, *B2*, and *B1*. As a result, the counter for A_1 (the advertisement matching S_1) at these brokers is incremented. In Figure 4.1.1, the number to the left of each advertisement in the SRT represents this counter. The figure represents the routing tables of each broker after all advertisements and subscriptions are propagated.

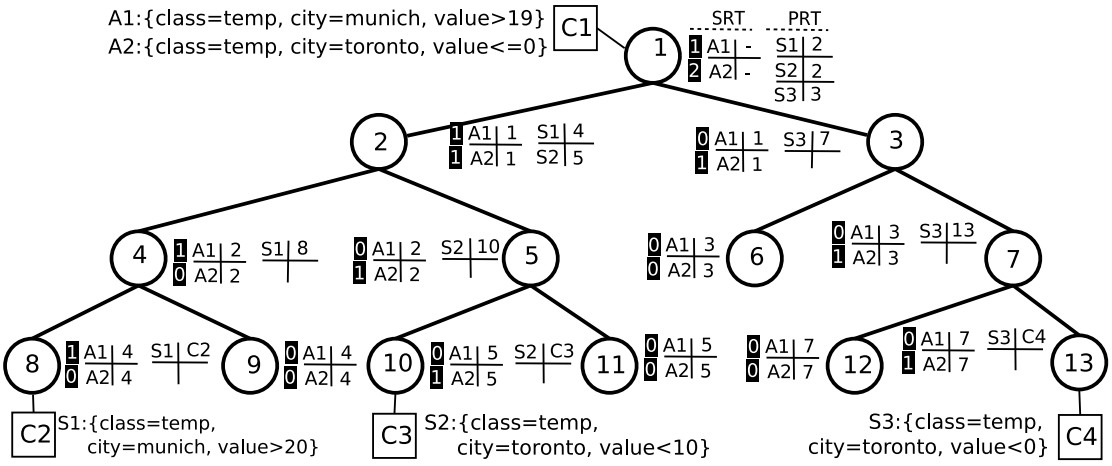


Figure 4.1.1: Similarity metric among brokers of a content-based pub/sub system

GEPS augments the dissemination overlay tree by establishing additional directed links between brokers. These links are created based on two factors: similarity between brokers and their position in the tree.

The similarity between two brokers quantifies the commonality of the content that two brokers route. We define the similarity function as $S : (V_b, V_{b'}) \rightarrow \mathbb{N}$. It takes as input two vectors of the same size and its output is a positive integer. V_n is called the similarity vector and represents the counter values associated with the advertisements at Broker n (B_n). Since advertisements are flooded, all brokers have the same advertisement set. In order to address situations where an advertisement is not yet received by all brokers or an unadvertisement has been issued, this vector can be extended to include the advertisement ID and its associated counter. $V_n[i]$ is initially 0 and is incremented every time B_n forwards a subscription matching A_i. The greater the output of $S(V_b, V_{b'})$, the more similar the two brokers b and b' are

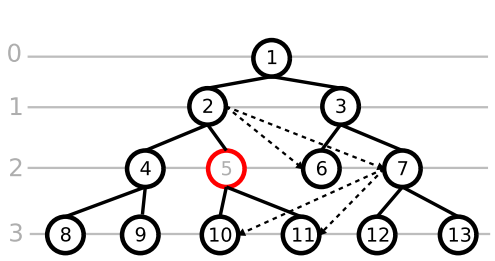
in terms of the subscriptions they have routed. S creates a new vector V_c where $V_c[i] = min\{V_b[i], V_{b'}[i]\}$ to calculate the similarity. V_c represents the commonality between the two input vectors. The output of S is the sum of the elements of V_c. For example, consider brokers *B4*, *B5*, and *B7* in Figure 4.1.1, where $V_4 = \{1, 0\}$, $V_5 = \{0, 1\}$ and $V_7 = \{0, 1\}$. With $S(V_5, V_4) = 0$ and $S(V_5, V_7) = 1$, *B7* is more similar to *B5* than to *B4*.

4.2 Broker Partial View

In this section, we explain how the position of the broker in the tree impacts the creation of the directed links. The subset of the overlay that the broker is aware of through the dissemination tree and established extra links is called the broker's partial view of the overlay. While partial view is a concept used mostly in epidemic approaches [79, 48], topology-based approaches like δFT use a similar concept. For example in δFT, the δ-neighborhood is the broker's partial view of the overlay comprising knowledge of the next δ hops across each neighbor. Although the concepts are similar, how a partial view is created, updated and used is the main difference. Besides broker similarity, the creation of additional links in GEPS is determined by the position of a broker in the primary dissemination tree. We divide brokers into sibling groups based on their depth in the tree.

Figure 4.2.1 illustrates the four different sibling groups extracted from Figure 4.1.1. Brokers having the same depth in the dissemination tree belong to the same group. Therefore, the four sibling groups are {1}, {2, 3}, {4, 5, 6, 7} and {8, 9, 10, 11, 12, 13}. GEPS requires each broker to gather information about other members in its sibling group. This information includes the similarity between the broker and other members of the sibling group and whether they are available or have failed. The similarity is measured by exchanging similarity vectors and using the defined similarity metric. The availability among siblings is determined by exchanging heartbeat messages.

Although in small sibling groups brokers can maintain the view of their group by



PV(2)

sibling	3
parent	-
child	6, 7

PV(5)

sibling	7,6
parent	3
child	13, 9

PV(7)

sibling	5, 4
parent	2
child	10, 11

PV(10)

sibling	13,9
parent	6,7
child	-

Figure 4.2.1: Broker partial view in GEPS

interacting with all other siblings, this approach is not scalable to large groups. In order to overcome this problem, each broker gathers and maintains information only about a subset of its siblings providing each broker with a partial view of its sibling group. Partial views are used extensively in epidemic protocols [79, 48] as a way of providing a decentralized and scalable method to gather and update a process's knowledge of a large group. The information included in these partial views, how they are maintained and what global state they converge to, is application-specific. One of the contributions of this work is a similarity-based overlay partial view construction and maintenance protocol tailored to an advertisement-based content-based pub/sub system with a tree topology.

We define B_n's partial view of sibling group d as $PV_r(n)_d$, where r is the configurable maximum size of the view. Each member of $PV_r(n)_d$ contains the ID of a broker, a timestamp indicating its last received heartbeat, and a similarity vector. The size of the partial view can be between 0 and r. Each B_n, located at depth d in the tree, keeps three partial views of the overlay, which are $PV_r(n)_{sibling} = PV_r(n)_d$, $PV_r(n)_{parent} = PV_r(n)_{d-1}$, and $PV_r(n)_{child} = PV_r(n)_{d+1}$. Initially, all three partial views contain up to r random brokers which have the same depth in the topology tree as B_n. We assume there exists a topology manager entity that can initialize the partial views of each new broker. A broker can ask the topology manager for r brokers on depth d of the topology. The child partial view of brokers located on depth d_{max} and the parent partial view of the tree root remain empty. Each broker maintains directed links to the brokers located in its parent and child partial views.

4.3 Partial View Maintenance

In this section, we explain our gossiping algorithm utilized by each GEPS broker to maintains its partial view of the overlay. Among the three partial views that each broker maintains, only the sibling view is updated by the broker. Partial view updates are performed in order to check availability of view members and discover more similar siblings. The view maintenance is done periodically by each broker every T seconds, during which brokers send their sibling views to each other. The value of T is configurable and is the same for all brokers in the overlay.

In order to keep track of broker failures and recoveries, each broker maintains two sets, F is the set of sibling failures and R the set of sibling recoveries the broker knows about. Each member of these sets is a tuple (n, t) and represents a failure or recovery incident of Broker n discovered in view maintenance cycle t. We refer to the maintenance cycle number as timestamp of the failure or recovery incident.

In each view maintenance cycle, Broker n heartbeats the siblings in its sibling partial view. Brokers that do not respond to heartbeat messages are perceived as failed, removed from the partial view and added to the set of failed siblings, along with the current maintenance cycle number. The broker then sends its sibling view and set of recovered and failed siblings it is aware of to each member of its sibling view. The ID of the receiver is excluded from the copy of the sibling view that is sent to it in order to prevent the receiver from recording itself as a sibling. The failures set (F) helps brokers identify recently failed siblings and remove them from their view, even if the failure is observed by another broker. The recoveries set (R) serves the purpose of keeping the failures set updated by removing newly recovered brokers from the failures set.

The regular heartbeat and exchange of failures and recoveries provide a distributed fault detection mechanism inside each sibling group. Along with the similarity metric, this fault detection mechanism affects which brokers should be replaced in a partial view. Due to the presence of asynchronous links, this fault detection mechanism may provide false positive or false negative results. While these false detections do not impact the main dissemination tree, we argue that the accuracy and completeness of

the fault detection can only have short-term and local impact on the sibling views. A Broker b perceived falsely as failed, for example, as a result of a slow link or process, is removed from the partial view of siblings that have b as a sibling view member. However, b can invalidate this failure report in the next view exchange round by sending view exchange requests to its sibling members. Due to the similarity metric, b's view includes all or part of the brokers that acted according to the false failure detection. The new view exchange request of b propagates in the sibling group the same way as the false failure report. However, the more recent timestamp (maintenance cycle) of the view exchange invalidates the false failure report. Existing false negatives can only impact gossip propagation by reducing the number of effective members of the parent or child view. Nonetheless, more accurate failure detection approaches can be utilized to improve partial view maintenance [114, 75].

Along with each view member, the timestamp of its last heartbeat and similarity vector is sent. The timestamp here refers to the maintenance cycle number that the heartbeat was checked. The timestamp only provides a way to check whether a failure/recovery report precedes another report and does not require clock synchronization across brokers.

The received view is processed using Algorithm 1. Broker b first merges the received failures set with its own. In Lines 2 to 6, for each failure (n', t') received from b', if Broker b does not have any failure record for n', the new failure is added to F. If there is already a failure record for n' with timestamp t at b, the timestamp of the existing failure record is updated to $max(t, t')$. Lines 7 to 9 check if the sender of the view, Broker b', is in the failures set of b. If so, b' is removed from the failures set of b and a recovery incident for b' is recorded with the current timestamp.

Next, Broker b uses the received recovery set to update its failures set. Lines 10 to 12 remove failure records for any broker that has a recovery incident in the received recoveries set with a more recent timestamp. In order to propagate this recovery information, it is added to the recoveries set of Broker b. Since b' heartbeats its view members and sends to b only available brokers, Broker b uses this list to update its recoveries set. Lines 13 to 16 check if there is a failure incident recorded for any broker in the received sibling view that is older than the heartbeat value of

Algorithm 1: Processing received sibling view at Broker b

```
1  Function ProcessSiblingView(PV_r(b')_sibling, F', R')
2      for (n', t') ∈ F' do // merge fail lists
3          if ∃ (n, t) ∈ F such that n = n' and t' > t then
4              t ← t' // update to latest failure report for n
5          else if ∄ (n, t) ∈ F such that n = n' then
6              F ← F ∪ (n', t') // add new entry for n
7      if b' ∈ F then // check if sender recovered
8          F ← F \ {∀(n, t) ∈ F such that n = b'}
9          R ← R ∪ {(b', curCycle)}
10     for (n', t') ∈ R' do // process received recovery list
11         F ← F \ {∀(n, t) ∈ F such that n = n' and t ≤ t'}
12         R ← R ∪ {(n', t')}
13     for (n', t') ∈ PV_r(b')_sibling do
14         if ∃(n, t) ∈ F such that n = n' and t' > t then
15             F ← F \ {∀(n, t) ∈ F such that n = n'}
16             R ← R ∪ {(n', t')}
17     all ← PV_r(b')_sibling ∪ PV_r(b)_sibling ∪ {(b', curCycle)}
18     all ← all \ F
19     for n ∈ all do
20         n.similarity ← S(V_b, V_n)
21     sort(all) // sort by decreasing similarity
22     PV_r(b)_sibling ← all[1...r] // trim list
```

the received view member. If so, a recovery record is added and a failure record is removed.

The difference between the two loops at Lines 10 and 13 is that the received recovery set is always reflected in the receiver's recovery set. However, the received partial view updates this set only if there is a previously recorded failure for any member of the received view.

Broker b merges its sibling view, received sibling view and the sender of the sibling view after updating and merging its information about sibling failures and recoveries. After removing any broker in this set that is not available based on the current failures set, the resulting set represents the available siblings that b knows about. In order to choose the most similar brokers among this subset of siblings, the similarity

function is applied to the similarity vector of b and all other brokers in the sibling subset. The top r similar brokers are selected as the new sibling partial view of b.

In Algorithm 1, the size of F and R depends on the failure and recovery rates in the overlay. However, since a broker only keeps this information about its siblings and only one entry is kept per sibling, the maximum size of these sets is the size of the sibling group that the broker belongs to and therefore the size of both sets is limited. In each maintenance cycle, the similarity function is calculated for at most $2r$ pair of brokers (existing r members plus r members of the received view), where r is the partial view size. Therefore, the complexity of the view maintenance algorithm is $\mathcal{O}(n)$.

So far, we have only explained the sibling view maintenance. However, each broker also holds a parent and child partial view which are not maintained by the broker itself. Each broker asks for the sibling view of its parent and children and uses these views as its parent and child view. Therefore, the parent partial view of Broker b is: $PV_r(b)_{parent} = PV_r(b')_{sibling}$, where b' is the parent of b in the dissemination tree. The child partial view of Broker b is: $PV_r(b)_{child} = \cup PV_r(b'_i)_{sibling}$, where b'_i is a child of b in the dissemination tree. The calculated partial view, $PV_r(b)_{child}$, is truncated to the r most similar members. In other words, each broker maintains a partial view of the available and most similar siblings and provides them to its parent and children as its temporary replacement in case of failure.

Updating parent and child partial view can be disrupted in three cases: (1) unbalanced tree topologies where leaf brokers can be located not on on the last sibling group (e.g., *B6* in Figure 4.2.1), parent failure and failure of all of the broker's children. In these cases, the broker can directly contact its child/parent partial view and request their sibling partial views to update its own child/parent partial views. Note that these partial views start up containing up to r random brokers and are not empty. This means the highest view maintenance cost is $r + 2r'$ where r is the sibling view size and r' is the parent/child view size.

A joining (or recovered) broker only requires to receive some random siblings and the current maintenance cycle number to start its partial view. Similar to overlay initialization, we assume there exists a topology manager entity that can be used for

this purpose. Using the partial view maintenance algorithm and sharing information about siblings, the joining broker is able to find more similar brokers. In order to address unadvertisements and new advertisements, the similarity vector can be extended to include the advertisement IDs along with each counter. This makes it possible to calculate the similarity between two brokers which do not have the exact same set of advertisement. Unsubscriptions can be accommodated by simply decrementing the counter of their matching advertisement.

While we address service availability during broker failure and recovery, in our evaluation, we assume that the failure period of a broker has an upper bound and the failed broker is replaced or recovered. Topology repair and broker recovery are orthogonal to this work and there exist different approaches that address these problems [20, 33, 32].

4.4 Publication Propagation

The protocol presented in the previous section helps brokers discover a set of additional brokers which are available and, based on the similarity metric, have a better chance to act as a temporary replacement for a broker's failed parent or children. While the views are maintained regardless of failures in the system, view members are only used for propagating publications as a fail-over mechanism in case of parent or child failures.

Publications are propagated using the primary dissemination tree whenever the next broker that is selected based on the message routing is available. Whenever a broker cannot send a publication to its next hop, it resorts to employing its partial view of the overlay to further propagate the publication. Therefore, the forwarding broker gossips the publication to all members of its child view when the unavailable next hop is from among its children. In case of parent failure, the gossip messages are sent to all members of the parent partial view. We call the publication messages that are forwarded using the additional directed links in the overlay, *gossip messages*. Therefore, GEPS provides the same delivery guarantees as its underlying pub/sub

system when there is no broker failure and best-effort delivery in case of broker failure.

In order to be able to forward the publication towards the brokers that might be affected by the failed broker, brokers that receive a gossip message, forward it one more time to their partial view members. For example, say *B5* in Figure 4.2.1 crashes, and, then, if *B2* needs to send a message towards *B10*, it resorts to its partial view to gossip the publication, since dissemination through the primary tree is not possible. *B2* gossips the publication to its child partial view, *B6* and *B7*, respectively, each one of them further gossiping the publication one more hop using their child partial view. In order to take advantage of the possibility of shortcutting through the tree, gossip recipients route the publication received through gossip as well.

The two-hop propagation of gossip messages is done to allow the similarity-based partial views to propagate the publications towards children or parent of the failed broker. In the example, the content similarity that causes *B7* to be in the partial view of *B5*, is likely to cause the children of *B7* to know about the children of *B5*. This is a consequence of basing the similarity on the pub/sub traffic that goes through each broker and not just their local interests.

Small sibling groups can disrupt gossip propagation when all of the group members fail. For example, in Figure 4.2.1, if *B2* and *B3* both fail, *B1* cannot use its partial view to propagate messages and *B4*, *B5*, *B6* and *B7* cannot reach *B1*. To address such cases, a minimal group size can be used to merge small sibling groups. As an example, by merging the first two sibling groups, we have three sibling groups which are {1, 2, 3}, {4, 5, 6, 7} and {8, 9, 10, 11, 12, 13}. Establishing a minimum size for the sibling groups is merely to prevent gossip disruption under low failure rates which is inevitable under high failure rates. As in the previous example, the number of siblings that should fail in order to disrupt gossiping is now twice as much. In all of our experiments, we consider the first three levels as one sibling group.

In order to prevent duplicate publication delivery and gossip messages looping in the overlay, brokers need to keep track of seen publications and gossip messages. Since gossip messages carry publications, it suffices to keep track of seen publication IDs. However, brokers do not need to remember all seen publications and only have to keep

track of recently received publications. The period of time that should be covered by this recent publication list in practice is not unlimited. This period does not need to cover the failure period of neighboring brokers since we do not buffer publications and do not wait for broker recovery. However, this period should be long enough to prevent gossip messages from looping in the overlay. This can happen when gossip messages are routed back using the tree towards their original sender. The upper bound of this time depends on the overlay size and link latencies. Therefore the minimum period required for publication ID retention is the maximum time required for a publication to reach from one edge broker to another (the diameter of the overlay).

GEPS is also compatible with subscription covering [14] with the only requirement that subscriptions should get fully propagated to update the counters. Although there is no subscription saving in this case, the main benefit of subscription covering, namely reduction of routing table size, can be preserved.

4.5 Multi-Level Partial View

The two-hop propagation explained in the previous sections can bypass a broker failure when the child of the failed broker is available (for example Figure 4.2.1). This allows the publication to reach the sub-tree of the failed broker. Nonetheless, in case of two consecutive failures, the sub-tree cannot receive the publication through gossip. In Figure 4.2.1 let's assume *B10* has children and they have subscriptions matching *B1*'s advertisements. When *B5* and *B10* are both down, even if the similarity metric allows the publication to bypass *B5*, *B10*'s sub-tree does not receive the publication since *B10* is down. To address such cases, we make some modifications to the view maintenance protocol. In each maintenance cycle, each broker sends along with its sibling partial view, its own ID and the set of failed siblings (F) discovered in that cycle. Furthermore, a broker also sends the IDs, the merged sibling partial views and failure set of its children level in the same message to its parent. This multi-level partial view contains information about the last N levels. For example, in the previous example, *B5* receives sibling partial views and failure set of its children,

merges them and uses them as its own children partial view. Merging of the received partial views is done by sorting based on similarity and keeping the top r (maximum children view size) members. Received failure sets are merged by taking their union and keeping the most recent record if there are duplicates. In turn, *B5* sends its own sibling partial view, failure set and its merged children partial view and failure set to *B2*. Consequently, *B2* can periodically update its partial view, which covers the next N levels, without any extra view maintenance cost.

In order to use this multi-level partial view to propagate publications, a broker includes the intended recipient sub-tree in the gossip. Each broker knows the members of its sub-tree via the propagated partial views. The maximum depth of this sub-tree is $N - 1$. For example, *B2* includes the sub-tree of *B5* as intended recipients of its gossip, namely {(10, 11), (children of 10 and 11)}. This means *B2* gossips this publication to bypass a failed child (*B5*) and reach the sub-tree of *B5*. *B6* and *B7* are members of $B2$'s child partial view and receive this gossip. Next, $B7$ (and similarly $B6$) starts from the first entry of the recipient sub-tree and if it does not find a matching failure for the ids recorded in the first entry of the recipient sub-tree, in the first entry of its multi-level partial view, it forwards the gossip to the sibling list of that entry. Otherwise, it moves to the next level. If all levels have a matching failure record, the first level is used. Each broker keeps one multi-level partial view for upstream propagation and one for downstream propagation. The sibling partial view is the same as explained in Section 4.2.

4.6 Experimental Evaluation

In this section, we evaluate our approach using simulation. We implemented GEPS and the related approaches as part of a content-based pub/sub system simulator written in Java.

We compare our work with three other approaches, δ-fault-tolerant pub/sub (δFT) [20], a simplified version of GEPS which does not use any similarity metric and instead uses random gossip (RGEPS), and a content-based pub/sub system without any feature to increase availability as baseline. We include the baseline to show the improved publication delivery ratio and introduced message overhead of each approach. Semi-probabilistic pub/sub [35] performs based on a subscription-flooding pub/sub scheme and cyclic topologies. In the subscription-flooding scheme, there is no advertisement. Subscriptions are flooded in order to build publication routing tables. Nonetheless, we have implemented Semi-probabilistic pub/sub using the simulator. Due to the very low performance results in an advertisement-based pub/sub with acyclic topology, we have not included its results.

δFT is targeted at enterprise systems and therefore not designed to tolerate high failure rates but to provide 100% delivery and per-publisher ordering guarantees. Nevertheless, we include it in our evaluation to study the effectiveness of its broker bypassing mechanism in high failure rates as a deterministic alternative to our epidemic approach. In our experiments δFT is configured with $\delta = 3$.

In order to study the effectiveness of our similarity metric, we have also implemented another version of GEPS, where partial view members are chosen randomly from all brokers of the overlay. This view is updated periodically by random members and all brokers have an equal chance of being in another broker's partial view. Similar to GEPS, the view is used for gossip when propagation through the tree is not possible. However, RGEPS uses a three-hop propagation in order to provide results comparable to GEPS. GEPS uses a sibling view size of 10 and a multi-level parent and children partial view size of 6. The number of levels covered by the multi-level partial views is also 6. RGEPS has a view size of 6. In all three approaches the partial view maintenance is performed every 2 seconds.

4.6.1 Workload

The generated tree topologies used for the evaluation have a maximum node degree of 4, 5, 6 or 7 and each have two balanced and unbalanced variations. The degree of each broker is selected based on a random uniform distribution and the maximum degree of the tree. Each topology has four variations generated with different seeds. Overall we run our experiments using 32 topologies covering a wide variety of tree topologies.

The workload used for evaluation is automatically generated based on a Zipf distribution among 100 different publication classes each having one to four attributes. Each edge broker (tree leaf) has one client with one advertisement and 2 subscriptions and publishes 4 publications per second. The sibling groups are determined once, based on the generated tree topology with an arbitrary broker as root and is the same for all. The dissemination trees, however, is an induced dissemination tree per publisher rooted at the publisher edge broker. The publication period is one minute. Increasing the number of advertisements, publications and subscriptions only increases the overall message overhead of all approaches and does not change the results. The experiments are conducted using generated broker failures and traces from a Google production cluster [115]. In the synthesized failure set, Each broker failure has a limited length since we assume a system with a centralized or decentralized failure detection and recovery mechanism. Unless stated otherwise, all following experiments use the same workload.

The generated broker failures follow a uniform random distribution. The overall number of failed brokers in the overlay is measured as the maximum total percentage of the overlay brokers that are simultaneously down. Therefore, an overlay with 20% broker failure has at some point of the simulation time 20% of its overlay brokers down at the same time. Each simulation starts without any broker failure, between the 20th and 40th seconds of the publication time, F% of the overlay becomes unavailable, followed by the final 20 seconds, where all failed brokers have recovered. In the simulations that use the Google cluster data, however, broker failures can happen at any time during the simulation. We also run the experiments using generated broker failures that follow a non-uniform distribution to study the effect

of clustered broker failures. These failures are clustered in sub-trees of size 4 to 5. Each experiment has been run with 32 different topologies and 4 seeds (128 runs) and the values are averaged.

4.6.2 Metrics

The metrics we study in our evaluation are the following:

Message overhead is defined as the total number of publication and gossip messages in the system during the experiment. In other words, the total number of messages that an approach requires to result in maximum publication delivery.

Delivery ratio is defined as the total number of successful publication deliveries divided by the total number of publications that should be delivered to achieve 100% delivery. Therefore, if the total number of matching subscribers for publication p_i is $|S_{p_i}|$, then the total number of publications to be delivered is $\Sigma|S_{p_i}|$ for all published publications. For example, assume in an overlay, publication p_1 has two subscribers and p_2 has three subscriber. If after publishing p_1 and p_2, two subscribers receive p_1 and one subscriber receives p_2, the delivery ratio is 60%.

Delivery latency is defined as the number of hops that it takes to deliver a publication. The 99^{th} percentile of delivery latency of all successfully delivered publications is defined as the delivery latency of an approach.

The measurements are done on an overlay where brokers have already propagated their advertisements and subscriptions, and therefore does not include advertisement and subscription propagation overhead. This overhead is equal in all approaches since they all use the same tree topology.

4.6.3 Experiments

Effect of partial view size: In this experiment, we study the impact of the partial

view size on our metrics. We measure delivery ratio, message overhead and delivery latency in an overlay of size 400, with an overlay failure rate of 10%, variable parent and children partial view size between 1 and 10, and a fixed sibling view size of 10.

Figure 4.6.1a shows the increasing message overhead as a result of growing view size. The message overhead grows linearly with the partial view size. As it can be seen in Figure 4.6.1b and 4.6.1d, increasing the partial view size improves the delivery ratio without impacting delivery latency. The reason is that each time a broker uses its partial view to bypass failed brokers, larger number of brokers receive the gossiped publication. Each of these recipients route this publication using their publication routing table. Higher number of these recipients results in more branches of the tree receiving the publication. Consequently this means higher overhead and higher delivery ratio. Since all gossiped publication follow the same two-hop propagation, the delivery latency is not affected.

The increase in the delivery ratio is not linear and in larger partial views a small improvement in delivery ratio has a big impact on message overhead. Nonetheless, due to the linear growth of message overhead, larger partial views can still be used to achieve higher delivery ratios. Hence, the partial view size provides a configuration mechanism to tune the trade-off between message overhead and delivery ratio without degrading delivery latency. Another important parameter impacting the delivery ratio, is the effectiveness of the similarity metric in terms of identifying brokers that can increase delivery ratio. Although in this paper we use a simple similarity metric, we believe more complicated similarity metrics can possibly improve the delivery ratio with lower increase in message overhead.

Figure 4.6.1c shows the impact of an increasing sibling view size when the parent and children view size are fixed. Increasing sibling partial view size can improve the delivery ratio, since a larger partial view can collect more information about its sibling group. The sibling view size can also impact the time required for a broker's partial view to stabilize. Figure 4.6.2 shows the number of view maintenance cycles required to achieve a stable sibling partial view in an overlay with 800 brokers and 10% failure rate. We compute the convergence of the sibling partial view by comparing it with a view calculated based on total knowledge of all siblings. The

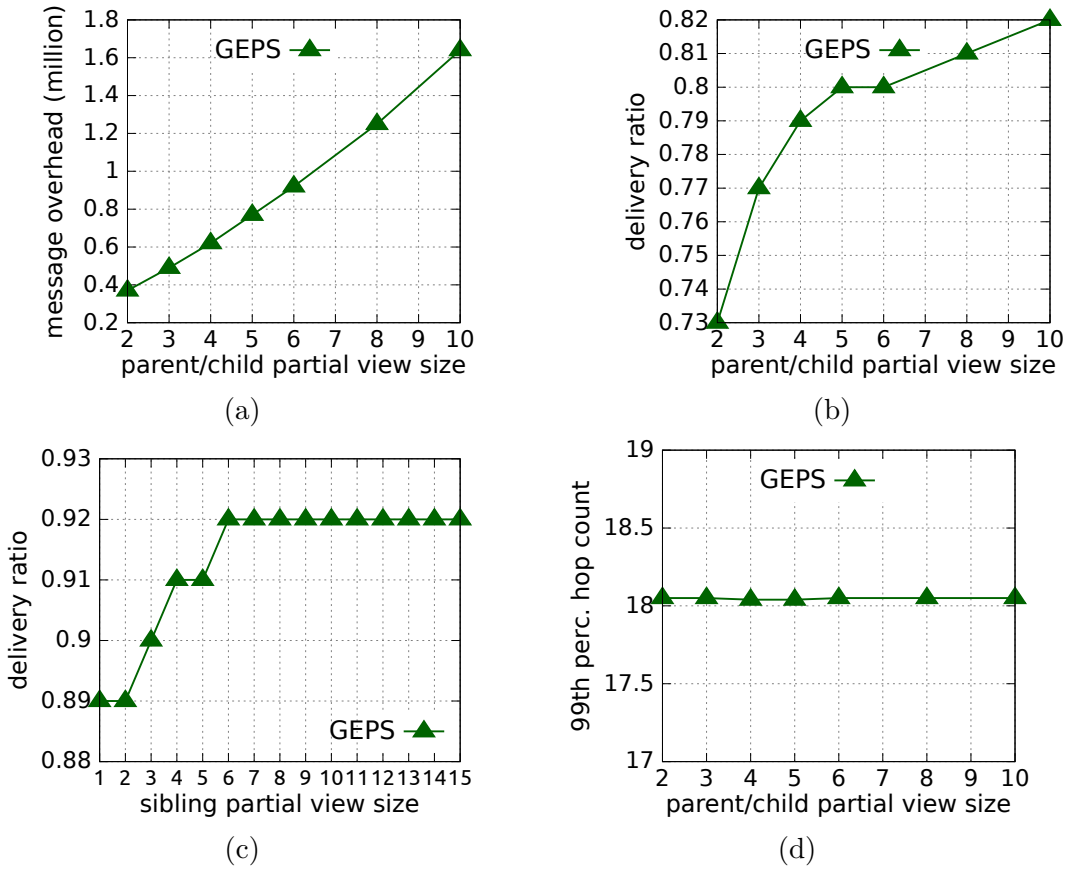


Figure 4.6.1: Impact of view size on GEPS metrics

number of siblings discovered using partial knowledge of the sibling group that are part of the optimal partial view divided by the size of the optimal view is the convergence ratio of the partial

The convergence ratio is essentially the fraction of the optimal view that exists in the partial view. The optimal view of Broker b is the set of brokers b' such that $S(V_b, V_{b'}) > 0$. Such an optimal view can potentially be as large as the sibling group. Therefore, the convergence ratio depends also on the size difference of the optimal and partial view. Figure 4.6.2 suggests that increasing the sibling view size improves the convergence of the partial view to the optimal view. Furthermore, regardless of the convergence ratio achievable by the sibling partial view, a broker's partial view starting with random members can quickly stabilize after two maintenance cycles with a sibling view size of 20. A sibling partial view with 10 members takes twice as

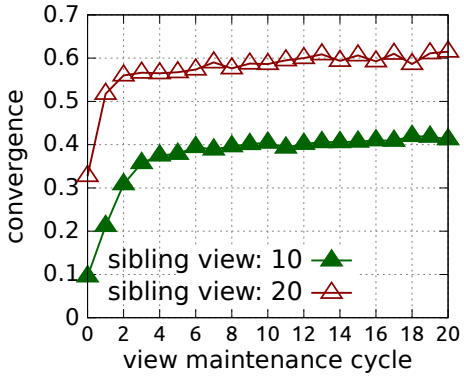
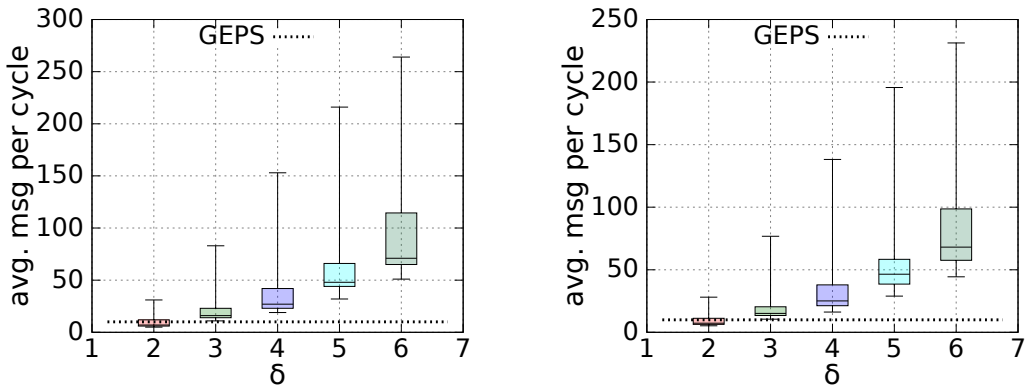


Figure 4.6.2: Partial view convergence time in GEPS

long to stabilize.

An important difference between GEPS and δFT is the possibility of dynamically changing the size of the partial views in GEPS by monitoring the rate of failure records added to a broker's failure set. This can be perceived as an indication of increasing failure rate in the overlay, and the broker can increase its sibling view size to gather more information on its sibling group or increase its parent/children view size. In δFT, changing the value of δ requires retransmission of subscriptions that were not covered by the smaller value of δ. Choosing large δ values from the beginning can cause scalability issues (Figure 4.6.3a).



(a) (b)

Figure 4.6.3: Impact of neighborhood size (δ) on effective view size

Figure 4.6.3a shows the effect of δ on the view size of brokers in an overlay of 400

brokers with 10% failure rate. Increasing δ from 3 to 4 requires more than 90% of the brokers to manage an overlay partial view between 22 and 153 brokers. To maintain the topology-based partial view, brokers do not need to check all of their neighborhood members in each maintenance cycle since finding the first non-faulty broker across each sub-tree will suffice. Nonetheless, the view maintenance overhead grows quickly for a large portion of the overlay with higher δ. Figure 4.6.3b shows the number of brokers that need to be monitored for view maintenance. Similar to the previous figure, a δ value of 4 can results in 72% of brokers contacting 22 to 138 brokers on each maintenance cycle in order to correctly update the topology partial view. Furthermore, this overhead increases with the average broker degree of the topology and failure rate of the overlay since in higher failure rates, the broker must check larger portions of each sub-tree in order to find a non-faulty next hop. In comparison, the average view maintenance cost of GEPS is 10 messages per cycle with an upper bound of 22 messages. In an overlay of 400 brokers, δFT results in 41% higher view maintenance cost compared to GEPS. Furthermore, GEPS's view maintenance protocol exhibits a linear increase by increasing the view size and distributes the view maintenance overhead equally across all brokers. Figures 4.6.3a and 4.6.3b suggest that it is not feasible to start the topology with large values of δ due to the excessive overhead of view maintenance and requires some brokers to keep a close to global view of the topology. It has been shown that centralized fault-detection mechanisms suffer from increasing false positive rates or scalability issues with an increase of the set of nodes to monitor [75]. Furthermore, dynamically adjusting the value of δ requires retransmissions of the subscriptions.

This experiment confirms that GEPS provides a lightweight and scalable view maintenance protocol in comparison to topo-logy-based approaches such as δFT.

Effect of overlay size: In this experiment, we study the effect of the overlay size on the defined metrics in order to evaluate and compare the scalability of GEPS against the other approaches. We run the experiment using an availability trace from a Google production cluster [115]. The trace covers a period of 29 days for a cluster of 11k machines and includes information about machine availability and failure events. We have extracted the machine failures from this trace and scaled it to the overlay sizes. Therefore, broker failures can happen at any time during the simulation and

the distribution of the failures or any correlations between them follows the cluster trace.

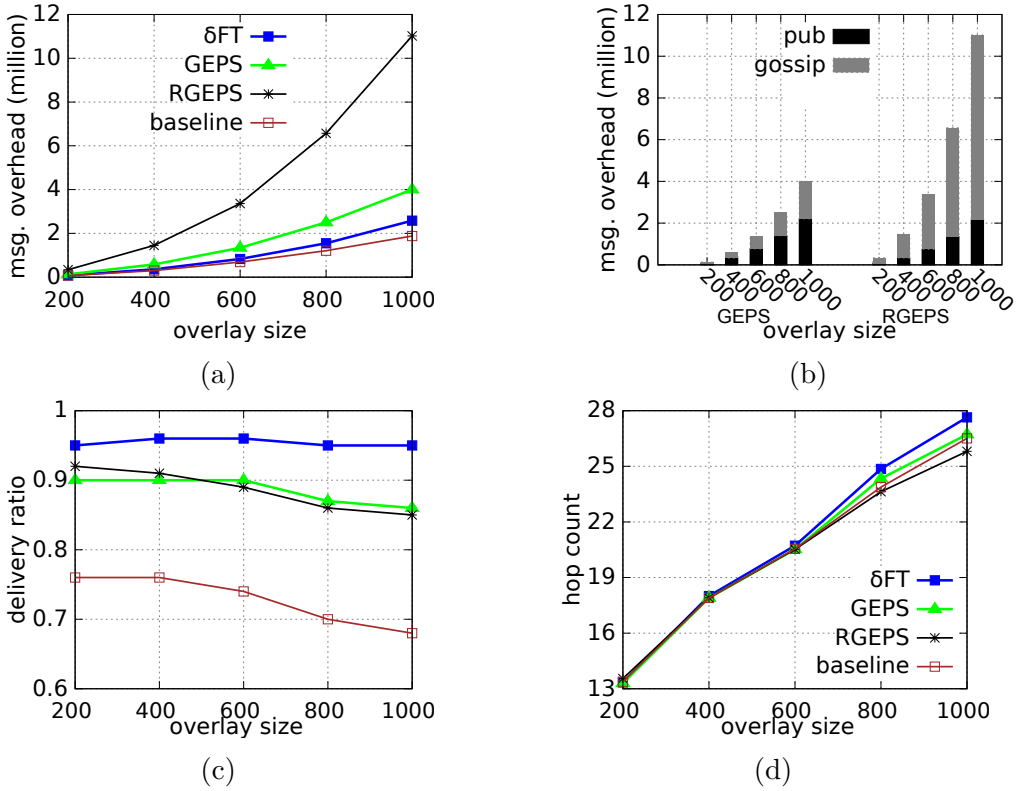


Figure 4.6.4: Impact of overlay size on GEPS metrics

Since all edge brokers have publications and subscriptions and each of them can potentially fail, the delivery ratio cannot be 100%. The reason is that we calculate the total number of subscriptions to be satisfied in order to achieve 100% delivery based on the generated workload. Therefore, failed publishers and subscribers prevent the delivery ratio from reaching 100%. A publication might be correctly routed to the subscriber but can not be delivered since the subscriber broker failed, or a publication might not be published in the first place due to failure of the publishing broker. This effects all approaches equally since they use the same workload, topology and failures.

Figure 4.6.4a shows the message overhead of each approach as the overlay size increases. Figure 4.6.4b shows the ratio of gossip and publication messages that make up the overhead of GEPS and RGEPS. While RGEPS results in up to 320%

higher overhead than δFT, the message overhead of GEPS is at most 62% higher than δFT. GEPS reduces the gossiping overhead of RGEPS by 65% while providing the same delivery ratio. Figure 4.6.4b shows that a large portion of RGEPS's overhead comes from gossip messages. This is due to the three-hop publication propagation of RGEPS which also increases its delivery ratio. Nonetheless, random target selection reduces effectiveness of gossiping in RGEPS. The random gossip recipients most likely do not have matching subscriptions in their PRT to route the gossiped publication in the tree. The baseline approach incurs the least overhead, since all publications that have a failed next hop are simply dropped.

The higher message overhead of GEPS and RGEPS in comparison to δFT is inherent to epidemic approaches which trade off message overhead for a more equally distributed load on all brokers (the view maintenance cost) and inherent fault tolerance. As we see later, this inherent message redundancy also contributes to resiliency of epidemic approaches in higher failure rates.

Figure 4.6.4c shows the percentage of successful deliveries. Here, δFT is able to provide a constant 95% delivery ratio in all overlay sizes. GEPS on the other hand, provides a delivery ratio of 90% for overlays up to 600 brokers and 87% and 86% for overlays of 800 and 100 brokers, respectively. The slightly decreasing delivery ratio of GEPS in larger overlays can be related to the fixed sibling view size. Although the specified view size can result in quicker convergence of the sibling view in smaller overlays, larger overlays may require an increased view size to cope with the larger sibling groups. Nonetheless, GEPS provides up to 22% improvement to the delivery ratio of the baseline with a comparable overhead to a deterministic approach such as δFT. Providing the same delivery ratio as RGEPS despite the much lower message overhead, suggests the effectiveness of the similarity metric in propagating messages past broker failures using gossiping. As shown in Figure 4.6.4d, since gossiping in GEPS is directed upstream or downstream and gossip messages do not randomly loop in the overlay to propagate publications, there is no increase in delivery latency. In all approaches, the 99^{th} percentile of delivery hop count increases as a result of a larger overlay. While the delivery latency of all approaches are the same in overlays up to 800, the increase in δFT's latency can be attributed to its higher delivery ratio in larger overlays. More deliveries in larger overlays means higher number of

deliveries over longer distances which can increase the 99th percentile.

This experiment shows the scalability of GEPS in terms of providing high delivery ratio with low message overhead. GEPS improves the delivery ratio of its underlying pub/sub tree up to 22% with no impact on publication delivery latency. Compared to random gossiping, the similarity metric utilized in GEPS significantly decreases the message overhead.

Effect of broker failure rate: In this experiment, we study the impact of the number of simultaneous broker failures on our metrics. Here, we use an overlay of 400 brokers and increase the broker failures from 1% to 40%. The generated broker failures follow a random uniform distribution.

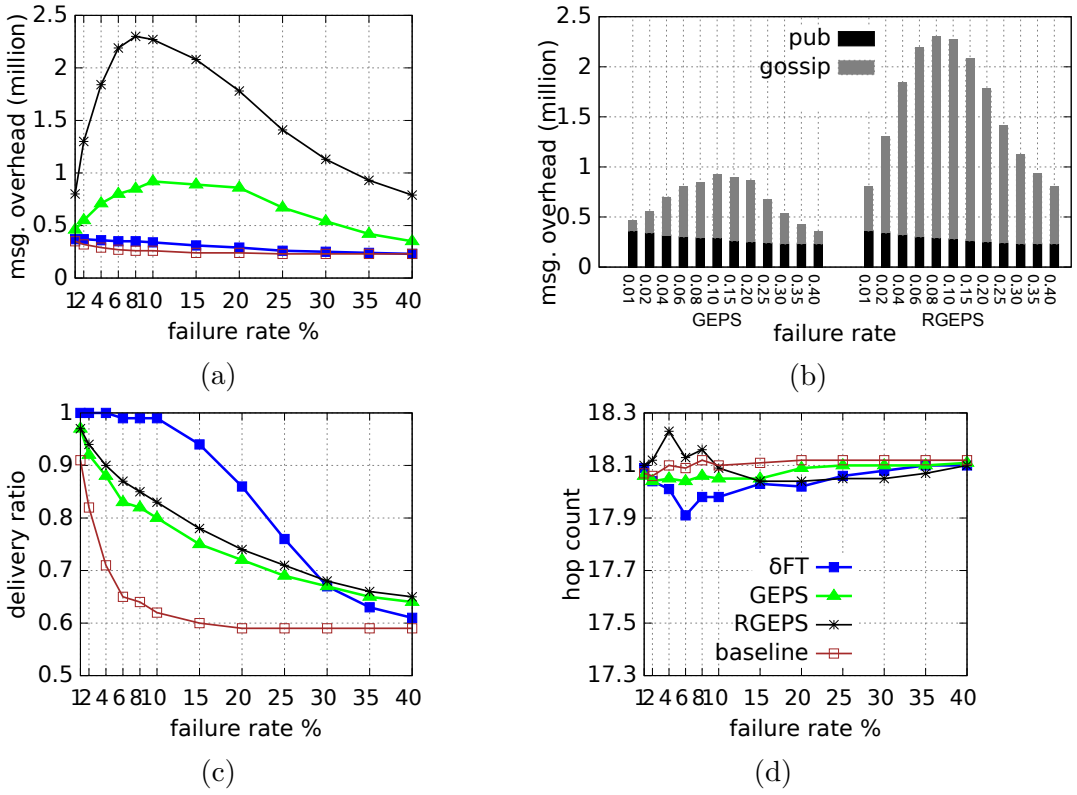


Figure 4.6.5: Impact of failure rate on GEPS metrics

Figure 4.6.5a shows the message overhead of each approach. While GEPS first shows an increase in the message overhead, the message overhead is decreasing for 10% or higher failure rates. This is because, due to the larger number of failures,

a higher number of publications are dropped, as next hops are unavailable and bypassing these failures becomes unfeasible. RGEPS exhibits a similar trend for the same reason. Eventually, the message overhead, delivery ratio and delivery latency of all approaches converge to the same value in very high failure rates. This is caused by decreasing number of deliveries over longer paths. GEPS, however, reacts to increasing failure rate by higher gossip ratio in its message overhead. As depicted in Figure 4.6.5b, when the failure rate is 20% or more, gossip messages decrease. The reason for this is that brokers cannot find enough similar and available siblings at high failure rates and as a result the number of effective gossip partners that they can resort to decreases.

Figure 4.6.5c shows the effect of an increasing number of broker failures on publication delivery ratio. Similar to the previous experiment, GEPS provides almost the same delivery ratio as RGEPS (with 2 to 3% difference) despite the much lower message overhead. While δFT is able to tolerate up to 10% failure rate in the overlay, after that the delivery ratio declines at a faster pace than GEPS and RGEPS, which is an indication of the graceful performance degradation of epidemic approaches, an important inherent property of such approaches [116]. Here, GEPS is able to provide a higher delivery ratio in an overlay with 30% or more failed brokers compared to a topology-based approach like δFT.

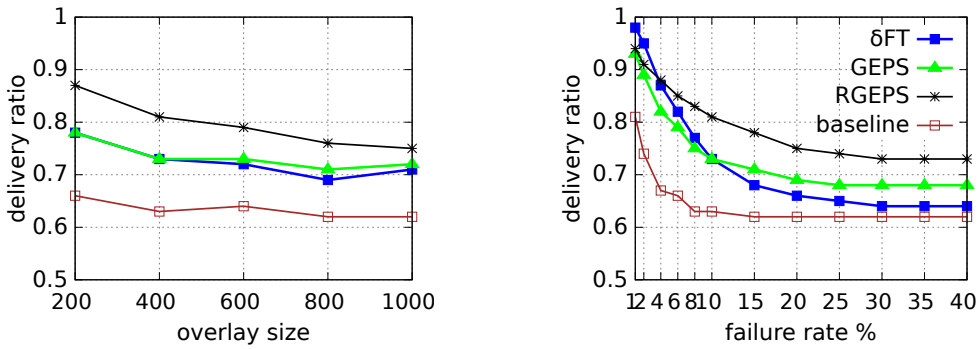
Figure 4.6.5d shows that all three approaches result in a very similar delivery latency to the baseline. This means the increased number of deliveries experience no change in delivery latency. At smaller failure rates, slight differences of each approach can be seen. δFT bypasses failures by directly sending to next available brokers. Hence, it can result is lower latency. RGEPS, on the other hand, might gossip a publication up to 3 times to bypass a failure which can increase the latency. This means that although higher number of gossiping hops can result in higher delivery and message overhead, they can also increase the delivery latency. In contrast, GEPS uses a two-hop propagation which stands in between δFT and RGEPS. At higher failure rates this difference does not exists because at higher failure rates, the effectiveness of the failure bypassing approach decreases. Consequently, delivery over longer paths, meaning bypassing multiple failures, happens less and the overlay is partitioned. A partitioned overlay limits the upper bound of the 99^{th} percentile of the delivery

latency.

This experiment shows that GEPS is able to retain the fault-resilience and graceful degradation properties of epidemic algorithms. These characteristics, while inherent to these approaches, introduce redundancy and latency. GEPS is able to successfully avoid high latency in publication delivery and reduce the message overhead of gossiping up to 65% compared to random gossiping while improving the delivery ratio of the underlying pub/sub tree up to 29%. This overhead reduction is done without affecting the inherent fault tolerance of gossiping which can trade off message redundancy for higher failure tolerance.

Effect of non-uniform failure: As shown above, maintaining a partial view based on the topology is practical when using small values of δ to limit the view maintenance overhead. Such small values of δ can make the overlay vulnerable to specific non-uniform failure patterns, for example, rack failures, datacenter outages or other correlated outages. The reason is that if more than δ consecutive brokers fail, the rest of that branch will not be reachable. Figure 4.6.6a and 4.6.6b show the effect of variable overlay size and fault rate on publication delivery rate when broker failures are non-uniform. In these experiments, all broker failures happen in cluster sizes of 4 or 5 which can not be covered by δ. In comparison to Figure 4.6.4c and 4.6.5c, δFT experiences a larger decrease in delivery ratio even with a 2% failure rate. RGEPS is able to provide a better delivery ratio than all approaches since its random target selection is least affected by non-uniform failures. However, this is achieved at the cost of very high message overhead. The message overhead and delivery latency of the approaches are very similar to the previous two experiments with GEPS reducing the gossiping overhead by up to 70%.

RGEPS is able to provide a high delivery ratio due to the fact that it can use any broker in the overlay as a partial view member and there is a smaller chance that a broker cannot find any view member. GEPS on the other hand, can bypass non-uniform failures using multi-level partial views which allow knowledge of broker failures to propagate to further brokers. Note that this is different from δFT, since increasing the number of levels in a multi-level partial view does not increase the view maintenance cost. GEPS maintains the same delivery ratio with a difference of



(a) (b)

Figure 4.6.6: Evaluation of GEPS using non-uniform failures

up to 6% regardless of the distribution of the failures while δFT is not able to bypass non-uniform failures. As a result, the delivery ratio of δFT decreases up to 26%.

Our experiments confirm that GEPS can combine epidemic similarity-based and topology-based approaches. By controlling the overhead of gossiping, GEPS maintains the distributed nature of epidemic approaches while providing a high delivery ratio. Furthermore, GEPS is able to tolerate clustered broker failures without any additional view maintenance cost.

Chapter 5

Popularity-Based Routing for Publish/Subscribe

In this chapter, we present our *Popularity-Based Publication Routing for Content-based Pub/Sub* (PopSub) approach to improve resource efficiency of publication routing in content-based pub/sub systems. In Section 5.1, we first explain the binary cost model for publication routing and the associated weakness inherent to existing distributed pub/sub systems. After that we present the various aspects which compose PopSub. This includes a metric to measure publication popularity in a pub/sub system, along with a popularity-based cost model for publication routing (Section 5.2) and how PopSub evaluates publication popularity (Section 5.3). In Section 5.4 we discuss different alternatives to handle unpopular publications. Lastly, we present the result of our experimental evaluation in Section 5.5.

In this chapter, we use *publisher* and *subscriber* to refer to edge brokers which are responsible for processing and propagating publications and subscriptions of their clients. Therefore, the source and destination of all messages are brokers that are connected to clients which are the actual publishers and subscribers. If necessary, clients are explicitly refereed to as *publisher client* or *subscriber client*. Furthermore, we refer to the data dissemination acyclic graph connecting all brokers in the overlay as *tree*.

5.1 Binary Cost Model Problem

The goal of publication routing in a pub/sub system is to correctly deliver publications while minimizing routing cost and avoiding unnecessary communication [117]. In an overlay-based pub/sub system, a publisher $\mathcal{P}$ and a subscriber $\mathcal{S}$ are connected by a tree $T_{\mathcal{P}\leftrightarrow\mathcal{S}} = <\mathbb{B}, \mathbb{L}>$ consisting of brokers $\mathbb{B}$ and overlay links $\mathbb{L}$. At each hop, the cost of routing a publication from $\mathcal{P}$ to $\mathcal{S}$ consists of two parts: matching the publication on Broker $B_i \in \mathbb{B}$ to find the next hop leading to $\mathcal{S}$, and forwarding the publication across the link $L_i \in \mathbb{L}$ to the next hop. Therefore, the cost of delivering a publication from $\mathcal{P}$ to $\mathcal{S}$ is $C_{\mathcal{P}\leftrightarrow\mathcal{S}} = \sum M(B_i) + \sum F(L_i)$ where $M(B_i)$ is the cost of matching publication $\mathcal{P}$ against existing subscriptions on Broker B_i and $F(L_i)$ is the cost of forwarding the publication on link L_i towards $\mathcal{S}$.

This cost model can be generalized to scenarios where a set of subscribers $\mathbb{S}$ are interested in publisher $\mathcal{P}$. The cost of routing a publication via $T_{\mathcal{P}\leftrightarrow\mathbb{S}}$ is $C_{\mathcal{P}\leftrightarrow\mathbb{S}} = \sum M(\mathbb{B}_i) + \sum F(\mathbb{L}_i)$, where $\mathbb{B}_i$ is the set of brokers receiving the publication i hops away from the publisher and $\mathbb{L}_i$ is the set of links that the publication traverses in step i of delivery to $\mathbb{S}$.

Since publication routing is performed in a hop-by-hop fashion, the cost of routing a publication from $\mathcal{P}$ to $\mathbb{S}$ is the sum of the costs on each hop. Therefore:

$$C_{\mathcal{P}\leftrightarrow\mathbb{S}} = C^1_{\mathcal{P}\leftrightarrow\mathbb{S}} + ... + C^n_{\mathcal{P}\leftrightarrow\mathbb{S}}$$

where $C^i_{\mathcal{P}\leftrightarrow\mathbb{S}} = M(\mathbb{B}_i) + F(\mathbb{L}_i)$ and n is the maximum distance between $\mathcal{P}$ and any $\mathcal{S} \in \mathbb{S}$. In existing routing algorithms, in step i of routing a publication, Broker B_i's decision to incur the cost $C^{i+1}_{\mathcal{P}\leftrightarrow\mathbb{S}}$ solely depends on whether there exists at least one subscriber in $\mathbb{S}_{n-i}$, the subset of $\mathbb{S}$ reachable from BrokerB_i in the next $n-i$ hops. Therefore:

$$C_{\mathcal{P}\leftrightarrow\mathbb{S}} = k_1 C^1_{\mathcal{P}\leftrightarrow\mathbb{S}} + \ldots + k_n C^n_{\mathcal{P}\leftrightarrow\mathbb{S}},\ k_i = \begin{cases} 0, & \text{if } \mathbb{S}_{n-i} = \emptyset. \\ 1, & \text{otherwise.} \end{cases}$$

This cost model is oblivious to the cardinality of $\mathbb{S}$, which results in all publications being considered equal and incurring the same cost regardless of the number of subscriptions that they match. We call this cost model the *binary cost model* for publication routing.

The *binary cost model* is essential for guaranteeing correct publication delivery. However, routing publications based on this cost model can result in inefficient use of resources and consequently increasing delivery latency or reducing the overall publication delivery of a pub/sub system. Furthermore, under high load, publications that have many subscribers can experience the same high latency as publications with lower popularity. Therefore, it is beneficial to prioritize publications based on their impact on the overall publication delivery in order to improve resource utilization and to mitigate the impact of delayed or dropped publications when the system is under high load. In the next section, we propose a metric to estimate the impact that routing a publication can have on the total publication delivery.

5.2 Popularity-Based Cost Model

In order to address the routing problem associated with the *binary cost model*, we need to estimate the resources required by a publication and its impact on the overall publication delivery. Using these two factors, we can extend the *binary cost model* and give priority to publications that result in the highest publication delivery with the lowest resource requirement.

The amount of resources required to route a publication from $\mathcal{P}$ to all matching subscribers $\mathbb{S}$ is proportional to the number of hops that it takes to reach all $\mathcal{S} \in \mathbb{S}$. Consequently, longer paths result in more brokers and overlay links involved in publication routing. However, two publications traversing the same path and incurring the same routing cost can have a different impact on the total publication delivery. For example, in Figure 5.2.1, delivering the blue and gray publications from *B1* to *B9* result in the same routing cost. Nonetheless, in this case, routing the gray publication can result in four times more publication delivery.

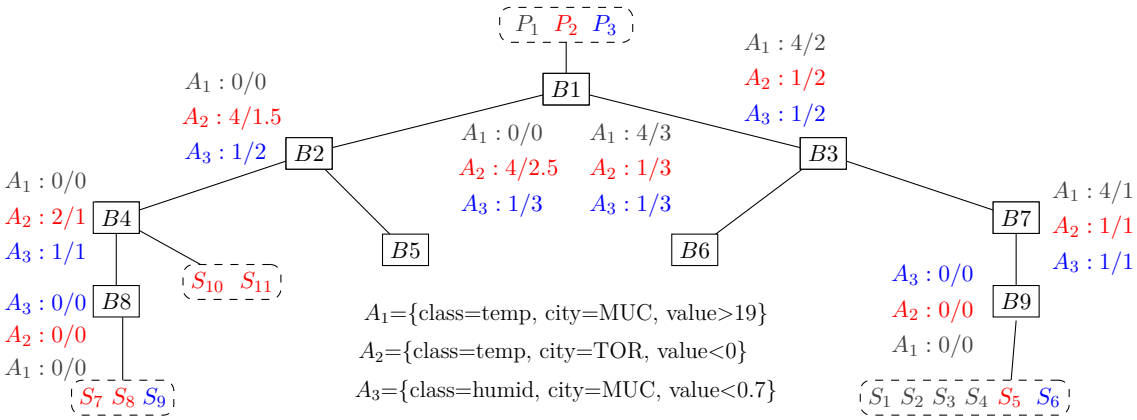


Figure 5.2.1: Publication gain ratio

In order to prioritize publications with the highest number of matching subscribers and lowest resources required for routing, we introduce a gain ratio that determines for each publication p the benefit of routing that publication one hop on the overall publication delivery of the system. We define the gain ratio of routing publication p as:

$$\mathcal{G}_p = \frac{|sub(\mathbb{S})|}{\overline{|T_{\mathcal{P}\leftrightarrow\mathbb{S}}|}}$$

where $|sub(\mathbb{S})|$ is the number of matching subscriptions in $\mathbb{S}$ and $\overline{|T_{\mathcal{P}\leftrightarrow\mathbb{S}}|}$ is the average path length from publisher of p to $\mathbb{S}$. $\mathcal{G}_p$ estimates the maximum number of deliveries that can result from routing publication p one hop further. Since $\mathcal{G}_p$ depends on the number of matching subscriptions and average path length from the current broker, each routing broker on the path from publisher to subscribers has a different estimate for $\mathcal{G}_p$. Therefore, on Broker B_i, $\mathcal{G}_p$ is the local estimate of B_i for routing p one hop further.

For example, in Figure 5.2.1, *B1* publishes 3 different sets of publication, blue (A_3), red (A_2) and gray (A_1). On *B4* there are two matching subscriptions, S_{10} and S_{11}, matching A_2. On *B8* there are three subscriptions, S_7 and S_8 matching A_2, and S_9 matching A_3. Therefore, on *B1*:

$$\mathcal{G}_p(red) = \frac{4}{(2+2+3+3)/4} = \frac{4}{2.5} \quad , \qquad \mathcal{G}_p(blue) = \frac{1}{3}$$

This means on *B1* routing a red and blue publication one hop towards *B2* results in increasing the overall publication delivery by $\frac{4}{2.5}$ and $\frac{1}{3}$, respectively. Note that these numbers are just local estimates calculated by the broker and only provide a normalized metric to compare the gain of routing different publications, as fraction of a delivery does not exist and in this example routing both publications one hop would not result in any delivery. Furthermore, only distant subscriptions are taken into account for calculating publication gain. For example, while routing a red publication on *B4*, $\mathcal{G}_p(red) = \frac{2}{1}$ rather than $\frac{4}{1}$, as the local subscriptions can be delivered to, without any further forwarding.

The estimated $\mathcal{G}_p$ depends on the location of the broker routing p and popularity of p, *i.e.*, the number of matching subscriptions reachable from that broker. Therefore, a broker might have different popularity estimates for a publication across each outgoing overlay link since each link leads to a different subset of subscribers. Furthermore, popularity of publications belonging to different advertisements are unrelated to each other and must be estimated separately. On each broker, PopSub estimates the publication gain ratio per advertisement per link. For example, in Figure 5.2.1, *B1* maintains a list of publication gain estimates for each overlay link going to *B2* and *B3*. The list of publication gain estimates for each link includes one estimate per advertisement.

Each broker B_i maintains a table of publication gain estimates, $\mathcal{G}_{B_i}$ where $\mathcal{G}_{B_i}[\ell, \mathfrak{a}]$ records the gain of routing a publication p across link ℓ for each p matching advertisement $\mathfrak{a}$. Since advertisements are flooded in the overlay and all brokers know about all advertisements, the number of publication gain estimates Broker B_i needs to maintain is $|\mathbb{L}_{B_i}| \times |\mathbb{A}|$ where $\mathbb{L}_{B_i}$ is the set of overlay links of B_i and $\mathbb{A}$ is the set of all advertisements in the system.

PopSub estimates publication gains on each broker during subscription propagation. Initially, all entries of $\mathcal{G}_{B_i}$ are set to $\frac{0}{0}$. A subscription s traversing towards its matching advertisement $\mathfrak{a}$ is required to record the number of hops it has passed. While going from Broker B_i to Broker B_j, $\mathcal{G}_{B_i}[\ell_{ij},\mathfrak{a}]$ is updated by incrementing the number of matching subscriptions and updating the average path length using $s.hopcount$. Consequently, all brokers on the path between the publisher and the

subscriber update their publication gain estimate. Figure 5.2.1 shows the updated publication gain tables of all brokers after propagation of all subscriptions. The missing tables all have three entries, each initialized to $\frac{0}{0}$, and are omitted to improve readability. Unsubscriptions are processed similarly and are also required to record the number of hops they traverse.

Performing the publication gain estimation during subscription propagation has the benefit of not requiring global knowledge about all subscribers and not hindering scalability of the routing algorithm since the only additional requirement is recording hop count of subscriptions.

The *binary cost model* can be extended to use the local estimation of $\mathcal{G}_p$ to reflect the publication gain in the cost. Therefore, the cost of routing publication p on Broker B_i to the next hop is $\frac{1}{\mathcal{G}_p} C^{i+1}_{\mathcal{P}\leftrightarrow\mathbb{S}}$. In other words, the cost of routing p to the next hop is inversely proportional to the estimated gain of p.

PopSub is also compatible with subscription covering [14] with the only requirement that subscriptions should get fully propagated to update the publication gain tables. Although there is no communication saving in this case, the main benefit of subscription covering, namely reduction of routing table size, can be preserved. In the next section, we explain how PopSub routes publications based on the popularity-based cost model.

5.3 Publication Popularity Evaluation

On each broker B_i, publications are categorized into popular and unpopular publications. Similar to the publication gain table, this categorization is performed for each outgoing link of B_i. Therefore, for each entry $\mathcal{G}_{B_i}[\ell, \alpha]$ in the publication gain table, a flag `pop` is stored indicating whether publications matching advertisement α are popular or not in the subset of the overlay reachable via link ℓ. Initially all entries have their `pop` flags set to true. Furthermore, each broker calculates and updates the message rate for each advertisement on each link. This message rate is also stored for each entry of the publication gain table in a `rate` field. $\mathcal{G}_{B_i}[\ell, \alpha]$.`rate` records the

average number of publications matching advertisement α that has been forwarded via link ℓ per second. Initially, all `rate` fields are set to 0. Broker B_i updates this message rate periodically based on the publications that it routes.

Periodically, Broker B_i evaluates the popularity of all table entries (ℓ, α). This is achieved by filling the broker capacity with the publications that have the highest gain ratio. We consider the broker's capacity as its throughput in terms of number of publications it can forward per second (τ_i). Therefore, regardless of whether τ_i is limited by broker's CPU, memory or network resources, the aim is to achieve the highest number of deliveries using the available capacity. This problem is similar to the knapsack problem where the weight of each item is equal to its message rate and the value of each item is equal to its gain ratio. Table entries that are chosen to fill up the broker capacity are deemed popular, the rest of the entries are considered unpopular. Algorithm 2 is the algorithm performed by each broker periodically to re-evaluate publication popularity.

Algorithm 2: Evaluating publication popularity on B_i

```
1 Function EvaluatePopularity(G_{B_i})
2     sort G_{B_i} // by descending gain ratio
3     filled ← 0
4     for g ∈ G_{B_i} do
5         if filled < τ_i × φ then // φ ∈ (0, 1]
6             g.pop ← true
7             filled += g.rate
8         else
9             g.pop ← false
```

In Algorithm 2, the entries of the $\mathcal{G}_{B_i}$ table are first sorted by decreasing order of their gain ratio. Next, starting from the entry with the highest $\mathcal{G}_p$, entries are marked as popular until we find an entry g such that the sum of the message rates so far plus g's message rate exceeds the broker capacity to be filled. All entries with a $\mathcal{G}_p$ equal to or lower than g's publication gain is marked as unpopular. Since the average number of links per each broker is bounded and usually small, the complexity of Algorithm 2 on each broker is $\mathcal{O}(n)$, where n is the number of advertisements in the system.

Algorithm 2 solves a simplified version of the knapsack problem since we do not require to minimize the remaining capacity. Parameter φ allows changing the threshold for popularity evaluation. Higher values of φ means that more entries in $\mathcal{G}_{B_i}$ are marked as popular since a higher share of the broker capacity is dedicated to popular publications. The value of φ must be chosen in such a way that reflects the popularity distribution of the workload. For example, in a Zipfian workload, typically most subscribers are interested in only 20% of the publishers [118, 119]. Therefore, the value of φ should correspond to 0.2 or less. φ should be chosen using the average system utilization and the CDF of the workload distribution. For example, in a workload with a normal distribution and average system utilization of 50%, $\varphi \approx 0.5 \times 0.3 = 0.15$, as in a normal distribution more than 68% of values lie within the first standard deviation (σ) and almost all values lie within 3σ. While workloads with uniform distribution are very rare [120, 118, 119], in these cases, φ simply defines the percentage of table entries that are considered popular. Lastly, In order to avoid overloading the broker, it is preferred to always leave some headroom during capacity planning ($\varphi < 0.8$).

While in this work we assume that all publications are of equal priority, in some scenarios there can be single publications with few subscribers that have a high priority. To accommodate these cases, the capacity filing algorithm can be modified to first fill up the broker capacity with high priority messages, and after that, divide the remaining capacity among popular and unpopular content.

Any publication that is not planned to be routed through the tree is considered unpopular. In the next three sections, we discuss three alternative approaches to disseminate unpopular publications, namely, *Direct Delivery*, *Batching* and, *Gossiping*.

5.4 Handling Unpopular Publications

In this section, we present three alternative approaches to disseminate unpopular publications.

5.4.1 Direct Delivery

In this approach, inspired by Atmosphere [102] and OMF [101], a publisher delivers unpopular publications directly to the subscribers without forwarding the publication through the overlay. For example, in Figure 5.2.1, if blue publications are unpopular on the link between *B1* and *B2*, *B1* directly delivers blue publications to *B8*. This requires subscriptions to record the broker they originated from. Therefore, each subscriber records its own address in the subscription before propagating it in the overlay. Note that, since the identity of the actual client is not used and the broker ID is only used within the overlay brokers, the anonymity of the client is preserved.

Direct delivery, while effective and simple, is susceptible to scalability issues. The reason is that, in the worst case, a publisher must forward a publication to all other brokers in the overlay. OMF and Atmosphere address this problem by limiting number of out-of-overlay forwards considering available resources on the publishing broker. In PopSub, however, direct delivery is performed only for unpopular publications. In a skewed workload, typical for real-world scenarios, unpopularity of a publication results in limited number of interested subscribers. This prevents *Direct Delivery* to encounter scalability issues. We evaluate scalability of *Direct Delivery* in Section 5.5.

5.4.2 Batching

Batching messages to improve resource utilization is a common practice and supported in many existing message passing and pub/sub systems [64, 121]. Next, we explain a batching approach to handle unpopular publications in a content-based pub/sub system.

In this approach, upon receiving a publication p to publish (Algorithm 3), the broker checks the popularity of p's advertisement, α, on each outgoing link ℓ. If the entry (ℓ, α) is marked unpopular, the publication gets buffered rather than forwarded. The set of buffered publications $\mathbb{B}$, are batched in one publication and forwarded via the tree whenever one of the following conditions is satisfied:

- $|\mathbb{B}| \geqslant \lceil\ \mathcal{G}_{B_i}.g_{min}\ /\ \overline{G}\ \rceil$, where $G = \{gain(p) | \forall p \in \mathbb{B}\}$ and $\overline{G}$ is the average gain of buffered publications and $\mathcal{G}_{B_i}.g_{min}$ is the minimum gain among popular entries.

- T seconds has passed since the first publication was buffered.

The first condition ensures that enough publications are batched to maintain the minimum publication gain on the broker and the timeout condition provides an option to prevent starvation of subscribers to unpopular content. Note that, publications are only buffered at the publisher and once published as a batch publication, they are not buffered again by any forwarding broker downstream.

Algorithm 3: Publishing publications on B_i (*Batching*)

```
1  Function Publish(p)
2      α ← MatchAdvs(p) // find matching advertisement
3      L^p_{B_i} ← MatchSubs(p) // find links to forward p
4      for ℓ ∈ L^p_{B_i} do
5          if G_{B_i}[ℓ, α].pop = true then
6              send(p, ℓ)
7          else // p is unpopular
8              B_{(ℓ, α)}.add(p) // add to buffer for (ℓ, α)
9              if |B_{(ℓ, α)}| = 1 then
10                 timer.run (T, SendBatch(B, ℓ) )
11             else if |B| ⩾ ⌈ G_{B_i}.g_min / Ḡ ⌉ then
12                 timer.cancel()
13                 SendBatch(B, ℓ)
```

Algorithm 4: Receiving publications on B_i (*Batching*)

```
1  Function ReceiveBatchPub(bp)
2      α← MatchAdvs( bp)  // find advertisement (same for all)
3      find matching local subscriptions and deliver
4      for p ∈ bp do
5          L^p_{B_i} ← MatchSubs( p)
6          for  ℓ ∈ L^p_{B_i} do
7              add p to the batch publication created for ℓ
8      send created batch publications across each ℓ
```

Publications are buffered separately for each (ℓ, α). In order to avoid buffering multiple copies of the same publications, the buffer can be implemented as a list of pointers to a pool of buffered publications. Upon receiving a batch publication (Algorithm 4), a forwarding broker goes through the publications in $\mathbb{B}$ and creates new batch publications for each of its outgoing link and sends them across the link to the next hop.

Alternatively, publications can be batched per link. However, after forwarding and rebatching $\mathbb{B}$ in the next hop or next few hops, this can result in batch publications containing only one or few unpopular publications, which does not improve the publication gain compared to per entry buffering. In other words, buffering publications per each table entry can result in batch publications with larger sizes to reach downstream.

Unlike *Direct Delivery*, which can be used without scalability concerns only in workloads with a skewed interest, *Batching* can be used in all types of workloads. *Batching* trades off publication delivery latency for efficiency of batch processing and forwarding of publications. However, due to the popularity-based prioritization, the increased latency, only affects unpopular publications.

5.4.3 Gossiping

Gossiping (*a.k.a.* epidemic protocols) provides a scalable dissemination mechanism without requiring building and maintaining an overlay. PopSub utilizes an existing gossiping protocol, namely *lpbcast* [79] introduced in Section 2.2, to disseminate unpopular publications via gossiping.

A straightforward approach to use *lpbcast* for dissemination of unpopular publications is to run one *lpbcast* instance on each broker. Each broker B_i maintains a partial view of all brokers in the system and participates in gossiping publications of all types. Upon routing a publication, popular publications are routed through the tree based on the matching results and unpopular publications are added to $\mathbb{M}_{B_i}$ to be gossiped along with other pending unpopular publication in the next gossip round.

Utilizing *lpbcast* in this way can result in unnecessary gossip overhead on brokers. Since *lpbcast* is a broadcast protocol, all brokers receive all unpopular publications, even if a broker is not a publisher, subscriber or forwarding broker (located on the path between a publisher and subscriber).

In order to reduce the gossip overhead, PopSub creates broadcast gossip groups for each advertisement on demand. Furthermore, each group includes as members only brokers that send or receive publications belonging to that advertisement. Therefore, *Gossiping* provides an alternative which combines *Direct Delivery* and *Batching*. Similar to *Direct Delivery*, unpopular publications are not processed by uninterested middle brokers. Furthermore, batching publications in one gossip and periodically disseminating them allows brokers to benefit from batch processing and forwarding. Furthermore, any received gossip only needs to be matched against local interests of the broker and unlike *Batching*, *Gossiping* does not require the forwarding broker to match the publications with all subscriptions known to the broker. Lastly, *Gossiping* provides a tunable dissemination mechanism. For example, by limiting history of events to gossip, it prevents overloading and drops messages if necessary. *Gossiping* provides a probabilistic delivery guarantee and increased delivery latency of unpopular publications. These are tradeoffs to achieve higher scalability and performance.

Next, we explain our proposed on demand gossip group creation, and, routing and propagation of unpopular publications.

Upon receiving publication p, publisher B_i uses Algorithm 5 to route p. B_i first finds the matching advertisement (α) and the set of links ($\mathbb{L}^{p}_{B_i}$) leading to the next hop. Next, if p is popular on ℓ (Line 5), B_i forwards p to ℓ (Line 8). If not, p needs to be routed via gossip.

In order to inform subscribers that this publication is unpopular, B_i sets a `switchToGossip` flag in p to allow subscribers to prepare for receiving the rest of the unpopular publications belonging to the same advertisement via gossip. Therefore, if such a flagged publication has not been sent on ℓ for an unpopular advertisement α (Line 9), B_i sets this flag (Lines 10-11) and starts a new gossip group for α if necessary (Line 13). Next, the broker includes itself as a partial view of the gossip

Algorithm 5: Publishing publications on B_i (*Gossiping*)

```
1  Function Publish(p, groups_{B_i}, flags_{B_i})
2      α ← MatchAdvs(p) // find matching advertisement
3      L^p_{B_i} ← MatchSubs(p) // find links to forward p
4      for ℓ ∈ L^p_{B_i} do
5          if G_{B_i[ℓ, α]}.pop is true then
6              if flags_{B_i}[ℓ, α] is true then
7                  p.swithToTree ← true
8              send(p ,ℓ)
9          else if flags_{B_i}[ℓ, α] is false then
10             flags_{B_i}[ℓ, α] ← true
11             p.switchToGossip ← true
12             if group_α ∉ groups_{B_i} then
13                 group_α = new lpbcast(α)
14             p.view ← {B_i} ∪ group_α.𝕍
15             send(p, ℓ)
16         else // p is unpopular and flags_{B_i}[ℓ, α] is set
17             group_α.𝕄 = { p } ∪ group_α.𝕄
```

group in p (Line 14) and forwards p on ℓ. This ensures that the receiving subscriber knows of at least one member of the gossip group for α in case $group_\alpha$ was just created and is empty. This is crucial to make sure that no gossip group experiences partitioning, since, in the worst case, all members at least know the publisher. Note that $group_\alpha$ can already exist if for example, α was already unpopular on another link $\ell' \in \mathbb{L}^p_{B_i}$. In that case instead of the publisher, its partial view is included.

An unpopular advertisement can become popular due to the periodical re-evaluation of publication popularity (Algorithm 2). B_i controls this by checking for publications that are popular but B_i has already sent a flagged publication indicating their unpopularity. In this case a flag indicating switching delivery to the tree is set in p before sending it on ℓ.

After sending the first flagged publications, future unpopular publications belonging to advertisement α on link ℓ are simply added to the set of messages of the *lpbcast* process to be gossiped on the next round (Line 17). Using the gossip broadcast, unpopular publications are batched together and sent as one message representing the new

unpopular publications since the last gossip round. All subscribers with a subscription matching α, receive all unpopular publications without any filtering since the group uses a broadcast protocol.

Subscribers process publications based on Algorithm 6. If p includes a flag to switch to gossip dissemination and a gossip group for α does not exist, B_i starts an *lpbcast* process and joins the broadcast group via the partial view included in p. Processing a publication with a flag to switch back to the tree is simply gossiping an unsubscription and stopping the *lpbcast* instance on B_i. A subscriber receiving a publication p via gossip has to match p against its local subscriptions before delivery.

Any broker B_i receiving publication p for routing, which is not a publisher or subscriber of p, simply forwards p on all $\ell \in \mathbb{L}^p_{B_i}$ without considering the popularity of p. This means that gossip groups consist only of publisher and subscriber brokers.

Algorithm 6: Receiving publications on B_i (*Gossiping*)

```
1  Function Receive(p, groups_{B_i})
2      α ← MatchAdvs(p) // find matching advertisement
3      if p.switchToGossip is true then
4          if group_α ∉ groups_{B_i} then
5              group_α ← new lpbcast(α)
6              group_α.𝕍 ← p.view
7      else if p.switchToTree is true then
8          𝕌 = {B_i} ∪ 𝕌 // Unsubscribe from gossip group
9          send gossip // Required to propagate unsubscription
10         remove group_α
11     find matching local subscriptions and deliver
```

The frequency of switching a stream of publications between gossiping and the tree is a function of the frequency of running Algorithm 2. For publications which are the last few popular ones, it is possible to switch between gossip and overlay upon every re-evaluation, for example due to changes in message rate of a more popular publication. We avoid this thrashing effect by allowing a 10% threshold while planning broker capacity. When there is no change in the gain ratio table, PopSub maintains the same capacity planning as the last round, as long as the current filled capacity is within 10% of the capacity to fill. Changes in subscriptions

and gain ratios are effected in the next re-evaluation.

While switching the dissemination mechanism, it is possible for a subscriber to miss some publications or receive duplicates. In order to guarantee complete delivery, each B_i needs to maintain a cache of the publications it publishes and ID of publications it has seen. These two sets are used for retransmitting missing publications and detecting duplicates. The size of these caches must be large enough to cover the period of time it takes for a flagged publication to reach its subscribers. For example, in Figure 5.2.1, after *B1* sends a flagged publication to *B2*, it does not forward any more publication over link ℓ_{12} but rather keeps them for gossiping. Therefore, a broker must cache publications published from the time the flagged publication is sent until the subscriber receives the publication and subsequently creates and joins the gossip group. This time period can be assumed in the worst case as long as the longest path from a publisher to a subscriber (*i.e.*, the diameter of the overlay). Furthermore, publication sets that have a subset of them delivered via gossip might not maintain the publisher order. This is also the case for *Direct Delivery*. Therefore, PopSub provides per-publisher ordering if a publication set is all delivered via the main dissemination tree (i.e., is popular or uses *Batching*). Per-publisher unique IDs for publications and reordering buffers can be used to provide per-publisher ordering guarantees for unpopular publications as well.

5.5 Experimental Evaluation

In this section, we evaluate our approach via simulation using real-world workloads, latency values, and traces. We implemented our approach and a baseline, as part of a content-based pub/sub system simulator written in Java. We evaluate PopSub using the three proposed dissemination mechanisms for unpopular publications, namely *Direct Delivery* (based on Atmosphere and OMF), *Batching*, and *Gossiping*. Furthermore, our baseline is a popularity-agnostic pub/sub system which handles all publications using the tree.

5.5.1 Workload

We evaluate PopSub using hierarchical topologies which are used for high-throughput pub/sub systems[14, 3]. We use real-world traces which provide latency values across 1715 machines, connected via the Internet and estimated using the *King* method [122, 123]. We generate overlay topologies of different sizes from this graph as follows. We select one random node in the graph and perform a BFS traversal on the graph, with the selected node as root, until we have collected a subgraph with the same size as the desired topology size. Next, we calculate the shortest-path tree from the selected root to all other nodes in the subgraph and use this tree as the overlay topology. Furthermore, any out-of-overlay connection uses the latencies provided in the subgraph. Using this process, we create five different sets of topologies using different seeds.

Since the popularity of publishers is an important factor that PopSub can take advantage of, we use a workload based on Twitter traces [124]. These traces provide the follower relationship among 81306 users where each user has at least one follower (subscriber). We generate our workload based on the popularity distribution extracted from this trace. We sort users based on the number of followers they have and calculate each user's normalized popularity as the percentage of followers from the total user count. In each experiment, there are 200 publisher clients and N subscriptions. The number of subscriptions per publisher client is selected based on the Twitter follower distribution.

Publisher clients are connected to the top of the tree (root and its children) and each publish for one minute at a rate of 10 *pub/sec*. Subscriber clients each have one subscription and are distributed among all brokers of the tree topology based on a random uniform distribution. The number of advertisements is 200 (one per publisher client) and in total there are 20 different classes of advertisements. The number of advertisements per class is based on a random uniform distribution. Each message has two attributes and the range of each attribute is 0 to 1 000. Publication and subscription attribute values are chosen based on a random uniform distribution over the content space. Each experiment is run with 5 different topologies and 5 different workloads (25 runs per results) and the values are averaged. In order to

study the impact of publication popularity on our metrics, we also use a synthesized workload where the popularity of the advertisement classes among the subscribers follows a Zipf distribution with different Zipf exponent values.

Another important factor for our evaluation is broker throughput and performance gains achieved by batch processing and forwarding publications (*batch gain*). Rather than using random values, we use an existing content-based pub/sub system to run benchmarks, collect throughput values and measure batch gains. We use two brokers on a Dell PowerEdge R430 server. Each broker runs in a VM with 4 CPU cores and 10GB RAM. The publishing broker publishes 3 000 publications. On each run the publisher batches publications in different sizes from 1 to 256. Each benchmark is repeated 5 times with different seeds, and the results are averaged. The publication throughput per second for each publisher is calculated, and throughput values of different batch sizes is normalized with respect to batch size 1. The collected set of throughput measurements for batch size 1, is used as the throughput value of brokers in the simulation. In the simulations, each time a broker forwards a batch publication, bp, of size n, the number of throughput units required to forward bp is looked up in the batch factor table.

Each PopSub broker re-evaluates its publication popularity metrics every 2 seconds based on the message rates collected in the last interval. *lpbcast* is configured with a view size of 10 and a fanout of 2. This means in each gossip round, each gossip is sent to 2 brokers randomly selected from the 10 members of the partial view. The gossip interval is every 10 seconds. Furthermore, *Batching* timeout is set to 3 seconds.

5.5.2 Metrics

Publication Gain ratio is the average gain ratio of publications that are routed through the tree. This measures the resource efficiency of the pub/sub system. A higher gain ratio means a higher number of successful deliveries to resource utilization. Each broker that evaluates the publication popularity metrics, records the gain ratio of publications that the broker decides to route via the overlay. The values collected by all brokers is averaged. While for baseline, *Batching*, and *Direct Delivery* all

publications are routed via the overlay, for *Gossiping* this metric only represents popular publications.

Pure forwards is the total number of publications forwarded by the overlay brokers where the forwarding broker has no local subscriber matching the publication. Reducing number of pure forwards improves the routing efficiency.

Unpopular deliveries is the total number of publications delivered via *Direct Delivery*, *Batching*, or *Gossiping*. This metric is always zero in the baseline. *Publication delivery latency* (PDL) is the time that it takes to deliver a publication to a subscriber. The 99^{th} percentile of delivery latency of all successfully delivered publications is defined as the PDL. We report PDL values for popular and unpopular publications separately. *popular PDL* represents latency of publications delivered via the tree and *unpopular PDL* represents latency of deliveries via *Direct Delivery*, *Batching*, or *Gossiping*.

Gossip false positive ratio represents the percentage of publications that were received via gossip but did not match any local subscription and therefore did not result in a publication delivery. Furthermore, *Match per gossip* is the average number of publications delivered per each received gossip. These two metrics show the effectiveness of gossiping.

In order to study the scalability of *Direct Delivery*, we count *number of forwards per publication* on each publisher, each time an unpopular publication is published. This represents the number of direct publication forwards required to deliver unpopular publications. The presented value is the 99^{th} percentile of all the collected values. *Number of gossip groups* shows the number of publishers that have unpopular publications in all or a subset of the overlay.

Lastly, *average queue size* is the average length of the queue of publications buffered due to overloaded brokers. The size of this queue directly influences the publication delivery latency. The measurement is performed as follows: each time there is a publication arriving at a broker to be forwarded, and due to overload the publication has to be queued, the length of the queue is recorded. All collected queue lengths are averaged to represent the average queue size of an approach.

5.5.3 Experiments

In this section, we evaluate and discuss the impact of four variables namely, filled broker capacity (φ), overlay size, subscription size and workload skewness on the metrics defined in Section 5.5.2.

Impact of φ: In this experiment, we study the impact of φ on our metrics. φ determines the percentage of broker capacity allocated to handling publications via the tree. Increasing φ results in more publications being considered popular. The overlay consists of 200 brokers and the number of subscriptions (clients) is 22 000.

Figure 5.5.1a shows that for our Twitter-based workload, determining popularity threshold by filling 5-10% of publisher broker capacity, results in 29-62% higher gain ratio compared to baseline. This means that for *Batching* and *Gossiping*, each publication forward in the overlay, results in up to 4.3 deliveries on average. This value is 2.65 for the baseline.

Direct Delivery results in gain ratios similar to the baseline because all unpopular deliveries have a gain ratio of 1, *i.e.* one forward results in one delivery, and this brings down the total average. While the total average gain ratio of the three approaches is different, the average gain ratio of popular publications is similar in all three approaches and is the same as *Gossiping*. By increasing the threshold for popularity evaluation, all approaches result in a similar gain ratio since all publications are considered popular. This is also evident from Figure 5.5.1b, where increasing φ decreases number of publications delivered via *Direct Delivery*, *Batching*, or *Gossiping*.

While *Direct Delivery* shows a gain ratio similar to the baseline, all three approaches benefit from the popularity evaluation algorithm as shown in Figure 5.5.1c. Here, identifying unpopular publications and delivering them via one of the three proposed approaches, can reduce number of pure forwards by up to 59% compared to the baseline. As mentioned previously, φ can be used to tune the popularity evaluation algorithm to match the popularity of the workload. Since the Twitter-based workload has a Zipfian distribution, smaller values of φ result in better performance. The local

gain ratio estimation and relaxed capacity filling threshold (to prevent thrashing) results in different number of unpopular deliveries and consequently different pure forward counts for the three approaches.

This experiment shows that choosing φ according to the popularity of the workload can result in up to 62% improvement in gain ratio and up to 59% lower number of pure forwards.

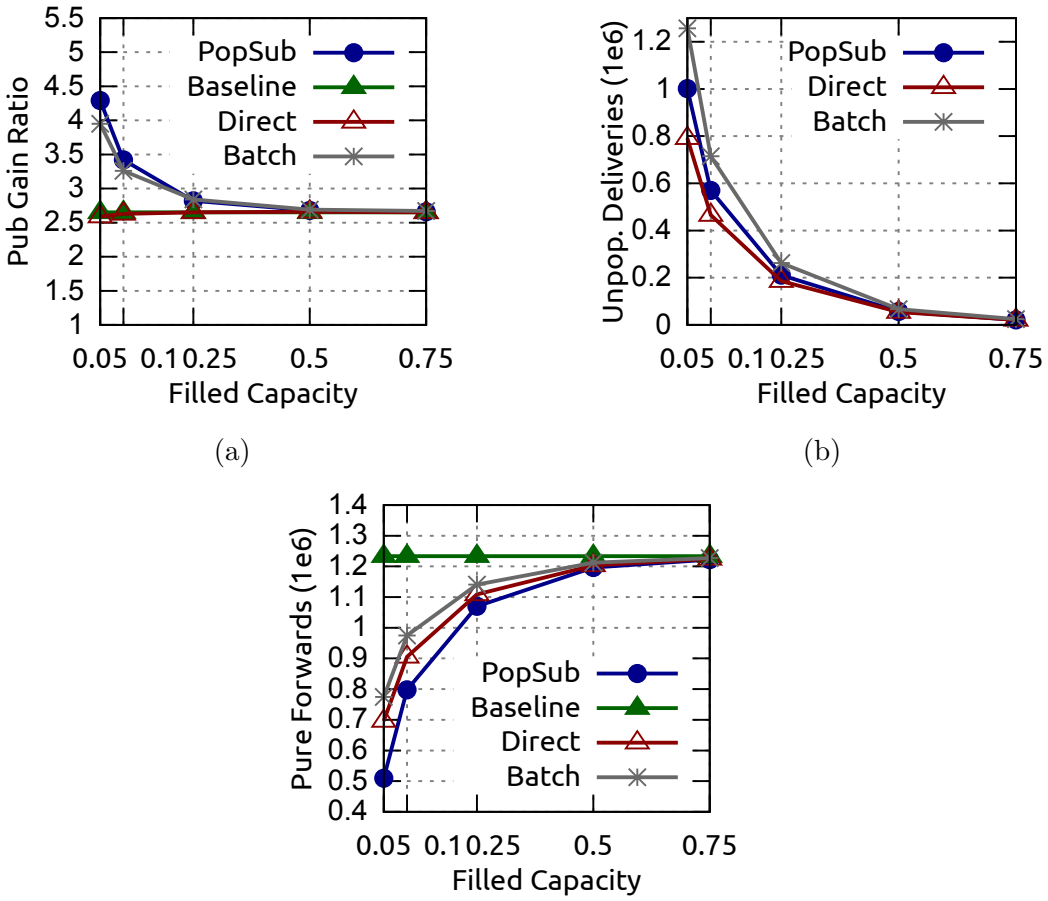


Figure 5.5.1: Impact of filled capacity on PopSub metrics

Impact of overlay size: In this experiment, we study the impact of the overlay size on our metrics while the number of subscriptions is fixed at 22 000 and $\varphi = 0.05$. Figure 5.5.2a shows that increasing number of brokers while the number of clients is the same, reduces the gain ratio. This is inevitable since larger overlays result

in longer paths. As number of subscriptions is the same, the gain ratio decreases. Consequently, increasing number of brokers with the same number of clients, decreases resource utilization of the system. The increasing number of pure forwards (Figure 5.5.2b) in all approaches also confirms longer paths between publisher and subscribers with no interested forwarding brokers.

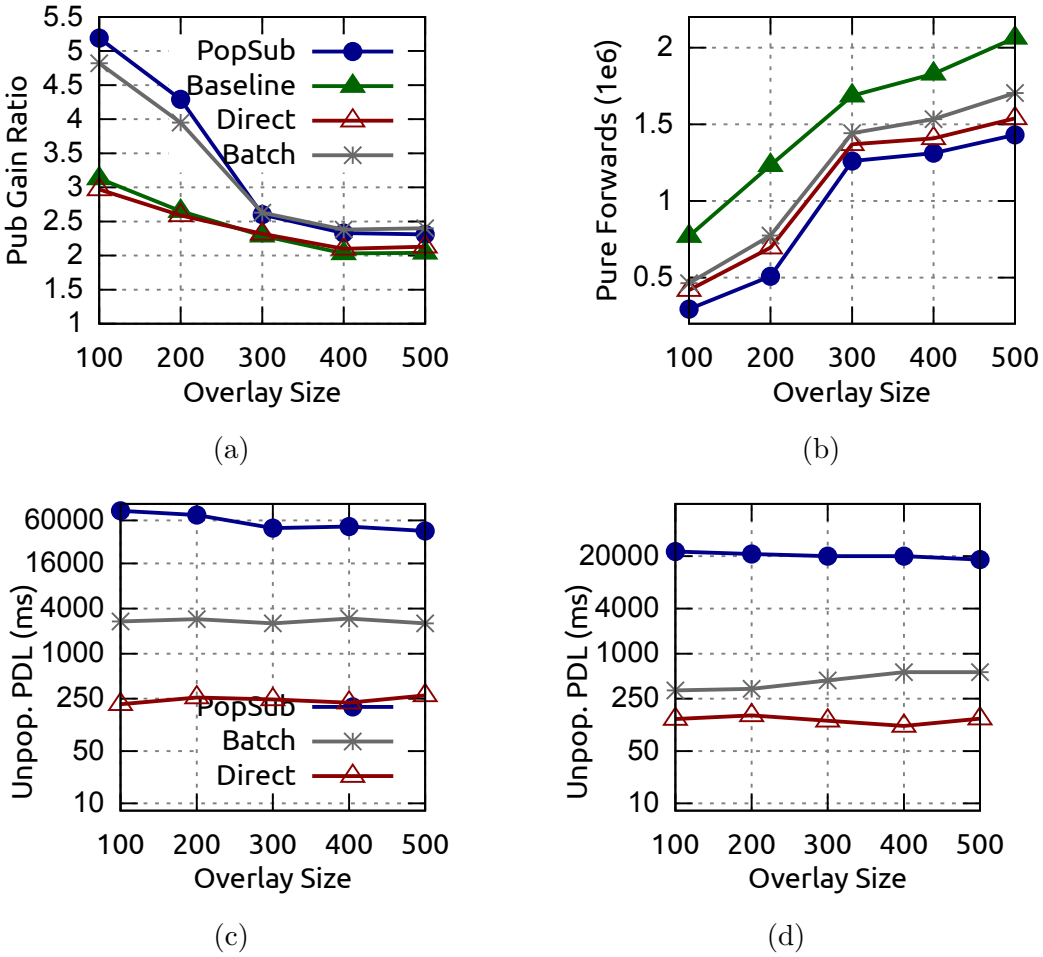


Figure 5.5.2: Impact of overlay size on PopSub metrics

The three proposed approaches to handle unpopular publications, provide different tradeoffs. While *Direct Delivery* does not improve the average gain ratio, it reduces number of pure forwards (Figure 5.5.2b) and provides an unpopular PDL similar to that of the popular publications (The popular PDL of all approaches is around 135ms). On the other hand, *Batching* and *Gossiping* improve the average gain ratio

and reduce pure forwards at the cost of higher unpopular PDL.

Figure 5.5.2c shows that, compared to *Direct Delivery*, unpopular deliveries via *Batching* and *Gossiping* result in PDL of up to 10 times and 200 times slower, respectively. The large difference however only applies to a small portion of deliveries since compared to the 99^{th} percentile, the 90^{th} percentile of PDL is much smaller.

Figure 5.5.2d shows that 90^{th} percentile of unpopular PDL for *Batching* and *Gossiping* is up to 2 times and 70 times higher, respectively. *Gossiping* trades off the increased latency for higher scalability in comparison to *Batching*. This tradeoff is studied in the last experiment with an overloaded pub/sub system. Note that, upopular PDL of *Direct Delivery* and *Gossiping* does not increase despite larger overlay sizes and longer paths, since these approaches do not use the tree.

This experiment shows that PopSub provides better performance in systems which are not underutilized. Furthermore, we showed the tradeoffs provided by the three proposed approaches.

Impact of number of subscriptions: In this experiment, we study the impact of number of subscriptions on our metrics. We use an overlay of size 200, $\varphi = 0.05$ and change number of subscriptions from 20 000 to 25 000.

Figure 5.5.3a shows that increasing subscriptions increases gain ratio in all approaches. The reason is that higher number of subscriptions means a higher number of matching subscriptions on each forwarding broker on average. Since the overlay size and hence the average path length is fixed, the gain ratio increases.

Similar to previous experiments, Figure 5.5.3a shows that *Batching* and *Gossiping* can improve average gain ratio by up to 62% compared to the baseline. Note that *Gossiping* can slightly outperform *Batching*, because batched publications still have to go through the tree and potentially pass through uninterested brokers. This reduces the average gain ratio compared to *Gossiping* which gossips unpopular publications only among brokers with a matching local subscription. The lower number of pure forwards for *Gossiping* compared to *Batching* in Figure 5.5.3b, confirms this. Compared to the baseline, *Gossiping*, *Batching* and *Direct Delivery*

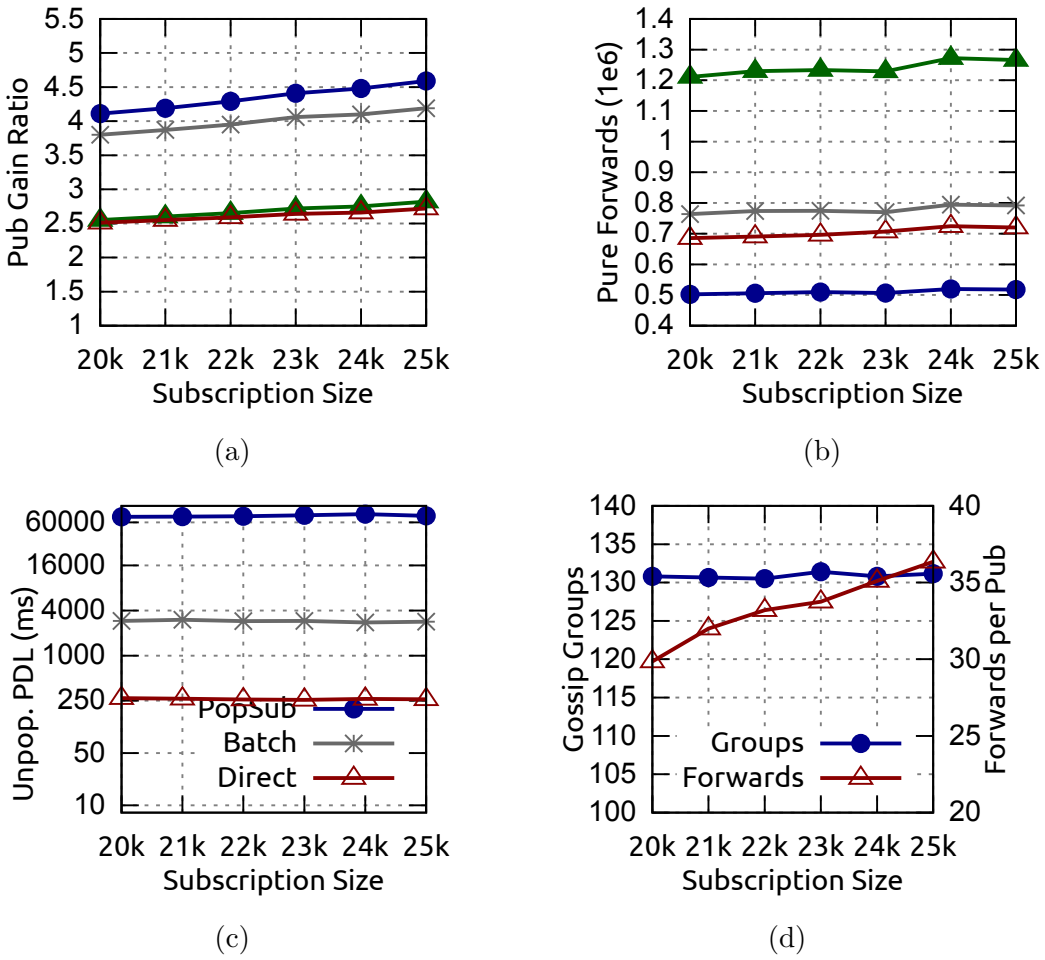


Figure 5.5.3: Impact of subscription size on PopSub metrics

reduce pure forwards by up to 56%, 37%, and 42%, respectively.

As shown in Figure 5.5.3c, increasing number of subscriptions does not impact unpopular PDL. The reason is that since the overlay size and hence number of subscribing brokers is the same, only number of subscriptions per broker changes. Consequently, number of unpopular deliveries increases. However, as the system is not overloaded, all approaches are able to scale to larger number of subscriptions. The impact of increasing subscriptions on an overloaded system is studied in the last experiment.

As discussed previously, *Direct Delivery* is scalable in skewed workloads since it only

forwards unpopular publications. Figure 5.5.3d shows the 99^{th} percentile of number of publication forwards required to deliver unpopular publications. *Direct Delivery* requires up to 29 to 36 publication forwards to deliver unpopular publications since a publisher must send the publication directly to subscribers with one or more matching subscriptions. Since this number is much higher for popular publications, using *Direct Delivery* for popular publications is not scalable. Furthermore, increasing number of subscriptions results in a sub-linear increase in number of direct forwards per publisher. The reason is that increasing number of subscriptions increases the number of matching subscribers for all publishers. Consequently, unpopular publications also have a higher number of subscribers and the publisher needs to forward each unpopular publication to a larger number of subscribers. In contrast, number of groups in *Gossiping* is not affected by number of subscribers since the number of publishers with unpopular publications is related to the workload and client distribution. Furthermore, unlike *Direct Delivery*, a publisher publishes unpopular publications by only gossiping the publication to two members of the gossip groups, regardless of its number of subscribers.

This experiment shows that while PopSub can improve the gain ratio of the pub/sub system, the three proposed approaches greatly reduce number of pure forwards in the overlay. Furthermore, we showed that *Direct Delivery* is scalable for scenarios where the workload is skewed and subscribers do not follow publishers based on a uniform distribution.

Impact of workload skewness: In this experiment, we study the impact of workload skewness on our metrics using a synthetic workload that follows a Zipf distribution. We change workload skewness by increasing the Zipf exponent (s) which results in more subscriptions to popular advertisements and reduction of the number of subscriptions to unpopular advertisements. While $s = 1$ results in a skewed workload, were almost 80% of subscriptions are interested in a small set of popular advertisements, an exponent value of 0.05 results in a uniform-like distribution of subscriptions among advertisements. The overlay size and number of subscriptions are fixed at 200 and 22 000, respectively and $\varphi = 0.05$.

Figure 5.5.4a shows the impact of workload skewness on each approach's gain ratio.

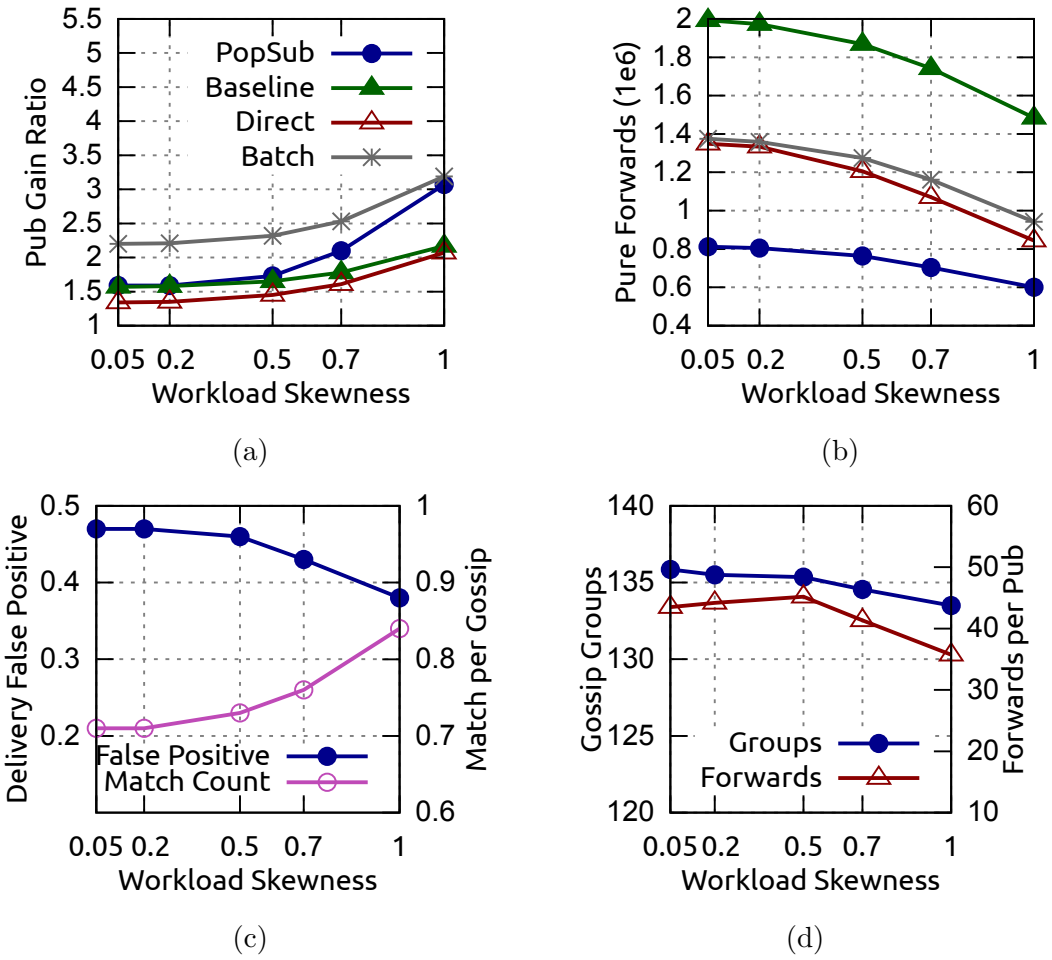


Figure 5.5.4: Impact of workload skewness on PopSub metrics

All approaches result in a higher gain ratio for workloads where distribution of subscriptions matching publishers is more skewed (increasing s). The reason is that in such non-uniform workloads, there actually are popular and unpopular publications and PopSub is able to take advantage of this to improve gain ratio. This is also evident from Figure 5.5.4b where the higher popularity of some publications in more skewed workloads, results in reduced pure forwards for all approaches.

Two important observations in Figure 5.5.4a are the following. First, *Batching* can improve gain ratio by up to 40% regardless of the workload skewness. The reason is that while *Direct Delivery* and *Gossiping* move unpopular publications out of the tree to improve gain ratio of the tree, *Batching* always uses the tree. Therefore, by

combining several unpopular publications, *Batching* can always improve the gain ratio. This is the inherent performance gain of batching. However, prioritizing based on the introduced metric allows identifying slightly more popular publications and limiting batching latency to a predictable subset of the publications which are slightly less popular. Therefore, the more skewed the workload, the less number of clients are affected by this batching latency.

Secondly, *Gossiping* improves gain ratio in skewed workloads ($s > 0.5$). The reason is that in such workloads, the popularity evaluation algorithm can identify popular publications and by keeping unpopular publications out of the tree, improve the gain ratio. In uniform workloads, the popularity evaluation algorithm merely chooses a subset of the publishers which may only slightly be more popular, since there is no real popular publisher in a uniform workload. While *Direct Delivery* shows gain ratios similar to baseline due to direct deliveries with a gain ratio of 1, the gain ratio of *Direct Delivery* for only popular publications is similar to *Gossiping*.

Figure 5.5.4c shows that besides a gain ratio improvement in skewed workloads, *Gossiping* performance also improves. Gossip false positives decrease by up to 19% and publication match per gossip improves by up to 18%. The reason is that in a skewed workload, number of subscription to unpopular publishers decreases as majority of subscribers are interested in popular publishers. Therefore, in a smaller gossip group the likelihood of receiving publications that do not match the local interest of the broker decreases. This is also evident from the increasing publication match per each received gossip. Figure 5.5.4d shows that due to the decreased number of subscriptions to unpopular advertisements, the number of gossip groups slightly decreases in skewed workloads. Furthermore, the decrease in the number of publication forwards required for direct delivery shows that in skewed workloads *Direct Delivery* is scalable and does not overwhelm publishers.

This experiment shows that PopSub improves gain ratio and reduces pure forwards regardless of the distribution of the subscriptions among publishers. While *Direct Delivery* should be used only in skewed workloads to avoid scalability issues, *Batching* and *Gossiping* do not have this limitation. Furthermore, *Batching* can improve gain ratio even in uniform workloads.

PopSub in an overloaded system: In this experiment, we study the performance of PopSub in an overloaded pub/sub system. Here, in an overlay of 300 brokers, 200 publishers publish at a rate of 15 *pub/sec* and we increase number of subscriptions from 30 000 to 50 000.

Figure 5.5.5a shows that similar to previous experiments, PopSub can improve gain ratio and increasing number of subscriptions increases the gain ratio. Figure 5.5.5b shows the average queue size of the overlay brokers. As evident from the baseline, the pub/sub system is overloaded which results in high number of messages queued to be processed. However, *Batching*, *Direct Delivery* and *Gossiping* reduce the average queue size by up to 33%, 60% and 98%, respectively. *Direct Delivery* results in smaller queue sizes because unpopular publications do not go through the overlay. In contrast, *Batching* routes all publications through the overlay and therefore results in a higher load on forwarding brokers. In comparison to *Direct Delivery* which puts higher load on already overloaded publishing brokers, *Gossiping* avoids this by distributing the dissemination of unpopular publications evenly among all interested subscribers, an inherent property of gossip protocols.

Figure 5.5.5c and 5.5.5d show the 99^{th} percentile of PDL for popular and unpopular publications. *Direct Delivery* and *Gossiping* reduce popular PDL by up to 40% and 57%, respectively. Similar to Figure 5.5.5b, *Gossiping* outperforms *Direct Delivery* by avoiding overloading publishing brokers. Note that, while due to the overloaded brokers, in all approaches, popular and unpopular publications are affected, using *Gossiping*, popular publications have 17% to 33% lower PDLs compared to unpopular publications. This is due to the fact that by involving only subscribers to unpopular advertisements and avoiding overwhelming publishers, *Gossiping* can reduce the impact of unpopular publications on the deliveries made through the overlay. Lastly, The higher PDL of *Batching* is due to routing all publications through the overlay, which in comparison to the other two approaches results in a higher load on the brokers. This experiment shows that prioritizing publications based on their gain ratio can reduce load on overlay brokers. Furthermore, handling unpopular publications via *Direct Delivery* and *Gossiping* improves publication delivery latency.

Based on our experiments, *Direct Delivery* is suitable for systems which are not

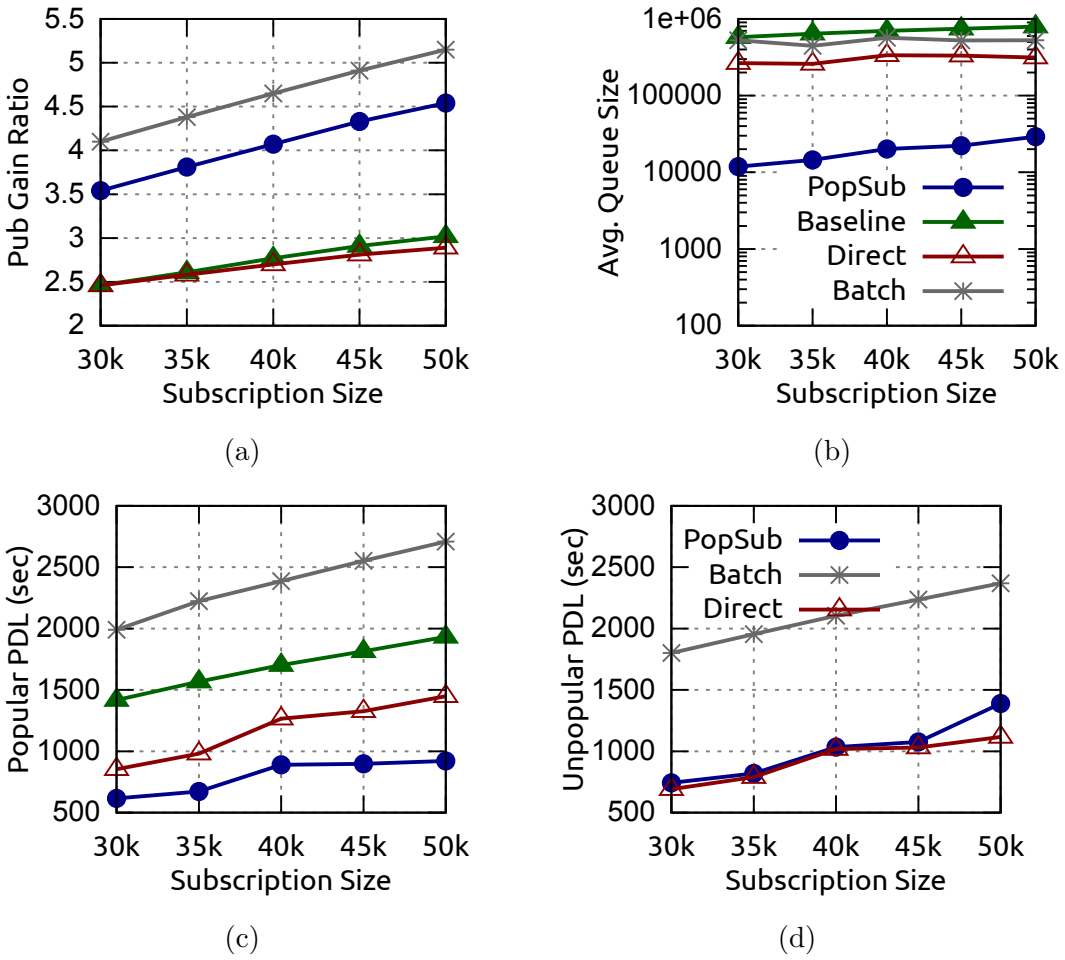


Figure 5.5.5: Evaluation of PopSub in an overloaded scenario

over-utilized and cannot tolerate PDLs higher than the overlay average. Although *Batching* and *Gossiping* can always improve scalability and resource utilization, *Batching* is more suitable for scenarios where the subscription distribution among publishers is not skewed and *Gossiping* can be used in scenarios where the system may be overloaded.

Chapter 6

Incremental Topology Transformation using Integer Programming

In this chapter, we present our *Integer-Programming-based Incremental Topology Transformation* (IPITT) approach to update the overlay topology of a pub/sub system without stopping the pub/sub service. In sections 6.1 and 6.2, we first explain how the ITT problem can be formulated as an automated planning problem and how integer programming can be used to solve automated planning problems. Next, in sections 6.3 and 6.4, we explain the different components of our approach and how IPITT can be used as a framework for incremental topology transformation that can be further customized depending on available overlay operations and broker migration requirements. Lastly, we present and discuss the result of our experimental evaluation in Section 6.5.

6.1 ITT as Automated Planning

In order to define the ITT problem as a planning problem, we need to define the set of predicates, objects, and operations of an ITT problem. In this work, we adopt the same ITT formulation as presented by Yoon *et al.* [51].

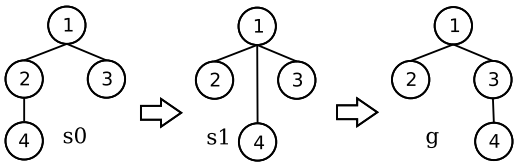


Figure 6.1.1: Initial and goal topology

The set of objects, Σ, is the set of brokers comprising the overlay (e.g., in Figure 6.1.1, $\Sigma = \{1, 2, 3, 4\}$). We use the following predicates to uniquely describe any ITT problem state: $P = \{conn(i, j), rem(i, j), eq(i, j)\}$. $conn(i, j)$ is a predicate indicating whether there is a direct link between brokers i and j in the topology. In order to define all existing links in the topology, we use *conn* propositions. In Figure 6.1.1, $C = \{conn(1, 2), conn(1, 3), conn(2, 4)\}$. In an overlay with undirected links, $conn(i, j) = T \Longrightarrow conn(j, i) = T$.

$rem(i, j)$ indicates whether the link between i and j can be removed (i.e., (i, j) is a removable link). During the topology transformation, any link which is not part of the goal topology is removable. In Figure 6.1.1, the link $(2, 4)$ is not part of the goal topology, therefore $rem(2, 4) = T$. Lastly, $eq(i, j)$ indicates whether brokers i and j are identical. There exists one $eq(i, j)$ proposition for each broker in the overlay. Therefore, using the defined P and Σ, we can describe the initial and goal state of Figure 6.1.1 as the following set of true propositions: $s_0 = \{conn(1, 2), conn(1, 3), conn(2, 4), rem(2, 4), eq(1, 1), eq(2, 2), eq(3, 3), eq(4, 4)\}$ and $g = \{conn(1, 2), conn(1, 3), conn(3, 4), eq(1, 1), eq(2, 2), eq(3, 3), eq(4, 4)\}$.

In order to transform any initial topology to any given goal topology, we require the following operations: *append(i, j)*, *detach(i, j)*, and *shift(i, j, k)*. These operations have been shown to safely enable any arbitrary transformation [33, 125]. *append* creates a new broker i and attaches it to the broker j. *detach* removes broker i from the overlay by disconnecting it from its neighbor j. *shift(i, j, k)* removes the link

between i and j and establishes a link between i and k. These three operations are defined as follows [51]:

$$append(i,j): \begin{cases} pre(O) = \{\neg eq(i,j), \neg conn(i,j)\} \\ add(O) = \{eq(i,i), conn(i,j)\} \\ del(O) = \{\} \end{cases}$$

$$detach(i,j): \begin{cases} pre(O) = \{\neg eq(i,j), conn(i,j), rem(i,j)\} \\ add(O) = \{\} \\ del(O) = \{eq(i,i), conn(i,j), rem(i,j)\} \end{cases}$$

$$shift(i,j,k): \begin{cases} pre(O) = \{conn(i,j), conn(j,k), rem(i,j), \\ \qquad\qquad \neg eq(i,j), \neg eq(j,k)\} \\ add(O) = \{conn(i,k), \\ \qquad\qquad rem(i,k) \text{ if } (i,k) \notin g\} \\ del(O) = \{conn(i,j), rem(i,j)\} \end{cases}$$

For example, in Figure 6.1.1, *shift(4, 2, 1)* is applicable to the initial topology because there is a link between $(4,2)$ ($conn(4,2)$) and $(2,1)$ ($conn(2,1)$) and the link $(4,2)$ is removable ($rem(4,2)$). Applying this action results in:

$$s_1 = \{conn(1,2), conn(1,3), conn(1,4), rem(1,4), eq(1,1), eq(2,2), eq(3,3), eq(4,4)\}$$

The link $(1,4)$ is required, because only after transitioning the overlay to the state s_1, can we apply *shift(4, 1, 3)* to reach the goal topology.

Note that *shift* requires three distinct brokers. Therefore, we need *eq(i, j)* in order to represent the existence of a broker in the overlay and prevent the generation of invalid *shift* actions such as *shift*(1, 1, 2).

In order to prevent message loss, we use the *synchronous shift protocol* [33]. In this protocol, i, j and k first start buffering publications and then perform changes on

their routing tables. Once all participants of a *shift* have finished updating their routing tables, processing of publications (including buffered publications) is then resumed.

6.2 Integer Programming for Automated Planning

There exists general planners that can be used to solve any planning problem encoded using the Planning Domain Definition Language (PDDL) [96, 97, 126]. Yoon *et al.* have studied the applicability of such general planners to the ITT problem [51]. However, these planners fail to scale to overlays of 50 or more brokers. This is due to the fact that, given the PDDL encoding of the ITT problem, the general planners consider all possible actions. This can result in a search space of $O(n^3)$ for all combinations of *shift(i, j, k)*.

In this work, we adopt an integer programming approach to planning for the following reasons:

- An IP-based approach specifies the set of actions (A) that the planner should consider. Consequently, we can reduce the search space by providing a smaller set of actions to the planner rather than all possible actions.
- It has been shown that IP is more general than propositional logic and allows for a more concise representation of constraints [127]. This results in a rich modeling formalism which enables customization of the plan search [128].
- An IP-based approach lends itself to being used with existing commercial solvers. Due the wide applicability of IP, such solvers are highly optimized and can even be easily configured to utilize computer clusters [100, 129].

We use the general IP approach for solving planning problems as presented in *OptiPlan*, and customize it by defining ITT-specific predicates and operations to create an ITT planner [130, 128]. To formulate the planning problem of transforming

an initial state to a goal state in T steps, we define the binary variable $x_{a,t} = 1$ if action a is executed in step t and 0 otherwise.

Using the following objective function, the IP formulation finds a plan that minimizes the number of actions required to perform the transformation:

$$\min \sum_{a \in A} \sum_{t \in \{1,\ldots,T\}} x_{a,t}$$

Furthermore, the formulation defines other binary variables and constraints that link actions and propositions in each planning step, guaranteeing mutual exclusion of the chosen actions, restricting parallel state changes, and ensuring backward chaining.

In the following, we present a more detailed explanation of the IP-based planning formulation.

The IP formulation of automated planning defines the following sets for each proposition $f \in F$:

- pre_f: set of actions that have f as precondition
- add_f: set of actions that have f as add effect
- del_f: set of actions that have f as delete effect

Furthermore, five more binary variables besides $x_{a,t}$ are defined. These state change variables are used to model transitions between different states.

- $y_{f,t}^{maintain} = 1$ if the value of proposition f (true/false) is propagated to step t.
- $y_{f,t}^{preadd} = 1$ if an action is executed in step t that requires proposition f and does not delete it.
- $y_{f,t}^{predel} = 1$ if an action is executed in step t that requires proposition f and deletes it.

- $y_{f,t}^{add} = 1$ if an action is executed in step t that does not require proposition f but adds it to the state.
- $y_{f,t}^{del} = 1$ if an action is executed in step t that does not require proposition f but removes it from the state.

The complete formulation is as follows:

$$\min \sum_{a \in A} \sum_{t \in \{1,\ldots,T\}} x_{a,t}$$

$$y_{f,0}^{add} = 1 \qquad \forall\ f \in I \tag{1}$$

$$y_{f,0}^{add} + y_{f,0}^{maintain} + y_{f,0}^{preadd} = 0 \qquad \forall\ f \notin I \tag{2}$$

$$y_{f,T}^{add} + y_{f,T}^{maintain} + y_{f,T}^{preadd} \geq 1 \qquad \forall\ f \in G \tag{3}$$

$$\sum_{a \in add_f \setminus pre_f} x_{a,t} \geq y_{f,t}^{add} \tag{4}$$

$$x_{a,t} \leq y_{f,t}^{add} \qquad \forall\ a \in add_f \setminus pre_f \tag{5}$$

$$\sum_{a \in del_f \setminus pre_f} x_{a,t} \geq y_{f,t}^{del} \tag{6}$$

$$x_{a,t} \leq y_{f,t}^{del} \qquad \forall\ a \in del_f \setminus pre_f \tag{7}$$

$$\sum_{a \in pre_f \setminus del_f} x_{a,t} \geq y_{f,t}^{preadd} \tag{8}$$

$$x_{a,t} \leq y_{f,t}^{preadd} \qquad \forall\ a \in pre_f \setminus del_f \tag{9}$$

$$\sum_{a \in pre_f \wedge del_f} x_{a,t} \ = \ y_{f,t}^{predel} \tag{10}$$

$$y_{f,t}^{add} + y_{f,t}^{maintain} + y_{f,t}^{del} + y_{f,t}^{predel} \leq 1 \tag{11}$$

$$y_{f,t}^{preadd} + y_{f,t}^{maintain} + y_{f,t}^{del} + y_{f,t}^{predel} \leq 1 \tag{12}$$

$$y_{f,t}^{preadd} + y_{f,t}^{maintain} + y_{f,t}^{predel} \leq y_{f,t-1}^{preadd} + y_{f,t-1}^{add} + y_{f,t-1}^{maintain} \qquad \forall\ f \in F, t \in \{1,\ldots,T\} \tag{13}$$

$$x_{a,t}, y_{f,t}^{preadd}, y_{f,t}^{predel}, y_{f,t}^{add}, y_{f,t}^{del}, y_{f,t}^{maintain} \in \{0,1\} \tag{14}$$

Constraints 1 and 2 add the initial state propositions. Constraint 3 ensures that any proposition comprising the goal state is either propagated to the last step or added in the last step. Constraints 4 to 10 link state change variables to actions ensuring propositions are added to the state or removed from it only as a result of action execution. Constraints 11 and 12 act as mutexes between state change variables, preventing parallel changes that can result in an inconsistent state. This means that during each step of the transformation, a proposition f can be either added, propagated, or removed from the state. Constraint 13 ensures backward chaining requirements, which means that if in step $t-1$ a proposition f is added or maintained, then f can be added or removed in step t, or it can be propagated to step t. Finally, the last constraint ensures that all defined decision variables have a valid value in each step.

6.3 ITT using Integer Programming

In this section, we first introduce the different components of the IPITT system. Next, we explain how our ITT planner functions and describe its IP model. Lastly, we explain how plans are deployed and executed.

6.3.1 System Overview

Figure 6.3.1 presents the different components of IPITT. We assume there exists a topology manager which knows the identity of all brokers in the overlay. The topology manager does not need to know the routing table nor subscriptions located on each broker, only the overlay topology. The four main components of IPITT are the overlay evaluator, ITT planner, plan deployer, and ITT agent.

The overlay evaluator is responsible for periodically evaluating the overlay performance and designing a new topology if necessary. The topology manager periodically provides the overlay evaluator with the current topology and performance metrics. If the current topology is deemed inefficient by the overlay evaluator, it calculates a new

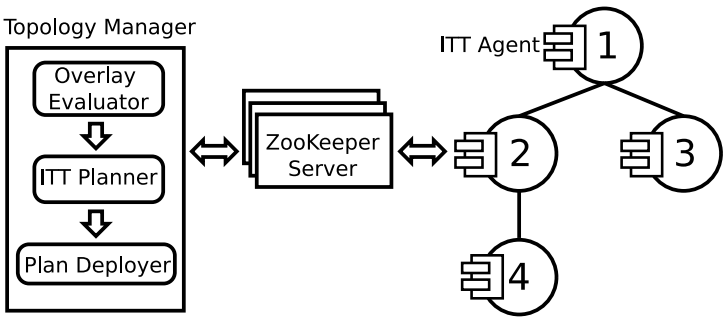


Figure 6.3.1: IPITT components

topology. Note that the focus of this paper is on the generation of a transformation plan given initial and goal topologies. The strategy taken to evaluate the current topology and design a new one is considered out of the scope of this work. Works on designing the overlay and evaluating the efficiency of pub/sub and overlay-based systems are orthogonal to our work, and their techniques can be integrated in our system [53, 94, 38, 131].

If a new topology is returned by the overlay evaluator, IPITT calls the ITT planner with the current topology and the new topology. The ITT planner is responsible for finding a plan to perform the transformation. The plan calculation takes into consideration time requirements. In other words, a new plan must be returned before the next topology evaluation. The calculated plan is passed to the plan deployer which is responsible for executing the transformation on the overlay. The deployer ensures each step of the plan is performed successfully before moving to the next step, since each step may depend on intermediate links previously established in the plan. Furthermore, in order to prevent any message loss or an invalid topology, the steps and operations within each step need to be synchronized. To do so, the deployer communicates with the ITT agent located on each broker.

6.3.2 ITT Planner

The planner is responsible for finding a plan that transforms the initial topology to the goal topology using a minimum number of steps. Algorithm 7 shows the

operation of the ITT planner. First, it generates the set of all actions to be considered for the plan. The generated set A does not describe any order between the actions but simply provides a superset of actions that can form a viable plan. In order to create the IP model of the ITT problem, the planner needs to determine the length T of the plan. This is required because the IP formulation of the planning problem requires the plan length to create a decision variable for each action in each step ($x_{a,t}$). In order to ensure a plan with a minimum number of steps, the planner starts with $T = 1$. In our implementation, we speed up the calculation of the algorithm by starting with a higher value of T.

The minimum number of steps required to transform a topology is proportional to the topology size and the number of brokers facing a link change. Therefore, the result of previous plan calculations can be stored in a table where each entry records the starting value of T for a given topology size and percentage of overlay links that are changed in the goal topology.

Algorithm 7: ITT planner

```
1  Function Planner(s_0, g)
2      A ← GenerateActions(s_0, g)
3      T ← 1
4      do
5          M ← CreateModel(s_0, g, A, T)
6          M.optimize() // Pass to solver
7          if M is infeasible then
8              T ← T + 1
9      while M is infeasible
10     plan ← ExtractSteps(M)
11     return plan
```

Given the initial and goal topologies, the action set, and plan length, the planner creates the IP model of the ITT problem instance. This is done by first creating the set of all propositions $F = \{f \in pre(a) \mid a \in A\} \cup \{f \in add(a) \mid a \in A\} \cup \{f \in del(a) \mid a \in A\}$. The IP model consists of a set of binary variables and constraints. Besides creating a variable for each action on each step, we create five variables for each proposition for each step. These variables link the propositions to the actions in each step of the plan.

After adding constraints over the defined variables, the IP model is given to the IP solver. If it is not possible to find a plan with the input T, the solver determines that the model is infeasible. In this case, we increment T, create a new model, and pass it to the solver. Once the solver returns the optimized model, the planner extracts the plan based on the values of the $x_{a,t}$ variables. The result is a sequence of steps, where each step contains one or more actions. For example, the plan for the ITT problem in Figure 6.1.1 is the following:

```
1:[shift(4,2,1)], 2:[shift(4,1,3)]
```

In this case, a plan in one step is not possible since an intermediate state with the link $(1, 4)$ is required to achieve the goal topology.

The size of the IP model, and consequently the time required to solve it, is related to the number of variables and constraints defined in the model. Since the defined number of variables and constraints for each proposition and action is fixed, decreasing the number of actions is the only way to reduce the total number of variables and shorten the time required to find a plan.

Since *append* and *detach* can only add or remove leaf brokers [54, 55, 33], finding the set of *append*s and *detach*es is trivial. First, all brokers which will be removed from the overlay are added as leaf brokers to the goal topology. Next, any broker which will be added to the overlay, are added as leaves to the initial topology. Consequently, both the initial and goal topologies have the same set of brokers. Once a plan is calculated, the set of *append*s is added to the beginning of the plan and the set of *detach*es are added to the end of the plan.

We separate calculation of the set of *append* and *detach* operations in order to reduce the IP model size which does not impact the correctness of the planner. Therefore, we consider only initial and goal topologies which have the same set of brokers and focus on generating a set of *shift* actions. Due to this property, there is no need to use the $eq(i, j)$ predicate, since we consider only valid *shift* actions (where i, j, and k are distinct brokers). Excluding *append* and *detach* from the IP formulation reduces the model size and hence the time required to find the plan. Note that

migrating clients connected to a removed broker is out of the scope of this paper and is addressed in client mobility studies [132, 133].

A naive approach to generate the set A is to consider all possible subsets of size 3 of the brokers. This results in $P_{n,3} = n!/(n-3)! \approx O(n^3)$ actions, where n is the overlay size. In medium to large overlay sizes (*e.g.*, more than 40 brokers), this results in a very large action set. Although generating all possible actions guarantees that the optimal solution can be found, the resulting set can include *shift* actions which involve brokers facing no change in the transformation. Based on this observation, we propose two heuristics which can considerably reduce the size of the generated action set.

The first heuristic, shown in Algorithm 8, starts by identifying links that must be established in the goal topology. For example, in Figure 6.3.2, $(1, 3)$ and $(6, 9)$ are removable links and $(4, 7)$ and $(8, 9)$ are goal links. Next, for each goal link, the path between its source and destination in the initial topology is calculated. We call these *transformation paths*, highlighted in red in Figure 6.3.2. The intuition behind this heuristic is that any *shift* action that can contribute to the transformation of the initial topology to the goal topology must involve brokers located on these paths. Finally, at Line 7, we generate all possible actions that can happen among the set of brokers located on a transformation path.

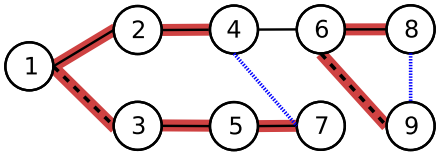


Figure 6.3.2: Transformation path

The size of the action set generated by this heuristic (Algorithm 8) is no longer a function of the overlay size, but rather a function of the number of brokers located on a transformation path. Therefore, this algorithm takes into account the size of the set of brokers which are affected by the transformation. Furthermore, in an ITT problem where brokers affected by the transformation are located close to each other, the generated set of actions is smaller. Algorithm 9 further reduces the search space by considering each transformation path separately. Therefore, rather than first collecting all affected brokers, the algorithm computes all possible *shift* actions

Algorithm 8: Generate actions for all paths

```
1 Function GenerateActionsForAllPaths(s_0, g)
2     goalLinks ← { ℓ ∈ g | ℓ ∉ s_0 }
3     brokers ← {}
4     for ℓ ∈ goalLinks do
5         path ← GetPath(ℓ_src, ℓ_dest, s_0)
6         brokers ← brokers ∪ path
7     actions ← perm(brokers, 3)
8     return actions
```

for each path.

The main difference between Algorithm 8 and 9 is that the former considers any *shift* action that can happen among different transformation paths. For example, in Figure 6.3.2, the set of actions generated by Algorithm 9 does not include any *shift* among brokers 4, 6 and 7 because these three brokers are located on two different transformation paths. In contrast, Algorithm 8 generates the *shift*s among these three brokers as well. This may result in shorter sequence of actions, if they exist. Nonetheless, in scenarios where two different parts of the overlay undergo transformation (*e.g.*, Figure 6.3.2), this can generate unnecessary actions. Algorithm 9 ensures that only *shift* actions among brokers located on at least one transformation path are considered. Since there exists always at least one sequence of *shift* actions to transform a removable link located on the transformation path of a goal link, Algorithm 9 generates an action set which contains enough actions to create at least one viable plan.

Algorithm 9: Generate actions for each path

```
1 Function GenerateActionsForEachPath(s_0, g)
2     goalLinks ← { ℓ ∈ g | ℓ ∉ s_0 }
3     actions ← {}
4     for ℓ ∈ goalLinks do
5         path ← GetPath(ℓ_src, ℓ_dest, s_0)
6         actions ← actions ∪ perm(path, 3)
7     return actions
```

6.3.3 Plan Deployment

The task of the plan deployer is to execute a given plan on the topology. Furthermore, the plan deployment must be synchronized and performed step-by-step, in order to prevent invalid routing states and an incorrect topology. Each broker has an ITT agent which is listening for *shift* actions, performing them on its host broker, and reporting back. The plan deployer uses ZooKeeper to communicate with the ITT agents, to deploy a plan, and to coordinate its execution [40]. ZooKeeper allows complete decoupling of the plan deployer from the pub/sub system and is scalable to large overlays. Furthermore, it facilitates reactions to plan execution failures. Figure 6.3.3 shows the ZooKeeper tree used by IPITT.

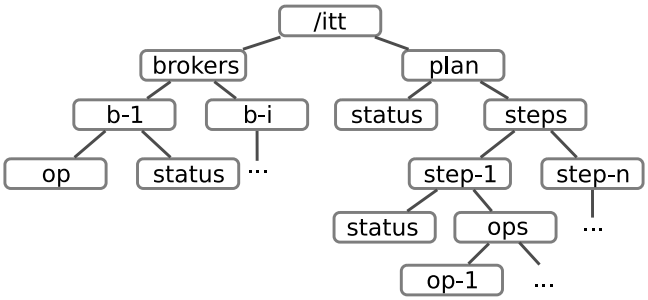


Figure 6.3.3: IPITT ZooKeeper tree

Upon receiving a plan to execute, the deployer writes the plan to */itt/plan/steps*. Each step in the plan is mapped to a node in ZooKeeper and each action is mapped to a child node of its step. Furthermore, each broker is mapped to a node. Upon broker start, each ITT agent establishes a watcher on */itt/brokers/brokerID*, waiting for *shift* actions to be issued by the deployer. In order to execute a plan step, for each step action, the deployer writes the *shift* action to the data node of each involved broker and creates watchers on the *status* node of each broker. Upon receiving a *shift* command, the ITT agents trigger a shift by directly contacting other involved brokers and exchanging routing table entries if necessary and changing overlay links. After successful completion of the *shift* command, each ITT agent updates the status of the received *shift* action on ZooKeeper as *successful*.

In case of failure or timeout on any broker, the *shift* action is aborted and the status of the corresponding broker is set to *error*. Once all ITT agents have updated the

status of the assigned action, the deployer updates the status of the *shift* action in the ZooKeeper tree. The status of a step is updated to successful only if all actions for that step have completed successfully. The deployer moves on to the next step for execution only if the previous step is successful. If a step fails and cannot be finished within a timeout, the plan execution is aborted. Aborting the plan is necessary in order to prevent invalid routing information or an invalid topology from occurring if further steps are taken. However, since *shift* actions of each step are independent from each other, a step can be partially successful without resulting in an incorrect state. Furthermore, by keeping the plan state on ZooKeeper, plan execution can recover from the failures of the deployer or the topology manager. Using ZooKeeper allows the ITT agents to use the ZooKeeper tree to send back statistics about the broker. This information can be used as input to evaluate whether it is necessary to create a new topology. This allows IPITT to create a feedback loop between the overlay and the topology manager.

While the deployer ensures fault tolerance during plan execution, it relies on the reliability of the operations to ensure fault tolerance during the execution of each action. IPITT requires each action to either succeed or fail without leaving the local routing table and links invalid. Consequently, any intermediate state of the topology during plan execution is valid and capable of correct routing of publications. Existing operations assume different fault models [54, 33, 55] which are applicable in different scenarios.

6.4 IPITT as a Framework

IPITT can be used as a framework for the incremental topology transformation of pub/sub systems. The ITT planner is able to find a plan consisting of operations that can be defined in terms of precondition, add, and delete effects on the overlay state. The choice of transformation operations to use in the plan depends on the guarantees that the transformation must maintain. While *SwapLink* [54] (or *shift*) performs small changes which only involve three brokers, there exists composite operations that perform larger transformations in one step. *Move* [33] and *LinkExchange* [55]

are two examples of operations which directly replace a removable link with a goal link. Compared to *shift*, these operations require synchronization among a larger number of brokers. Thus, they require a longer time to execute and have a larger impact on the clients. Nonetheless, in scenarios where the larger delay is tolerable, they can be used for building a plan.

The IP formulation of ITT allows changing the objective function to define a new cost model for planning. In this paper, we assume all actions have an equal cost. Therefore, we minimize the number of actions to improve plan quality. However, it is possible to assign each action a cost depending on the involved brokers and find a plan which minimizes the total cost of the plan. If each action a has a cost C_a, the objective function minimizing planning cost is:

$$\min \sum_{a \in A} \sum_{t \in \{1,\ldots,T\}} C_a \times x_{a,t}$$

It is also possible to define new constraints for the plan. For example, if we want to limit the maximum number of links of each broker during the transformation, we can define a variable $D_{b,t}$ which is the degree of Broker b in step t of the plan. The maximum degree can be defined using the set of constraints $D_{b,t} \leq D_{max}$, $\forall\ b \in B$, where B is the set of overlay brokers. Furthermore, the *add* and *delete* effect of the operations must be extended to consider $D_{b,t}$. Lastly, the ITT planner and deployer can be used in combination with any overlay-based pub/sub system, as long as each broker implements an ITT agent which can access the ZooKeeper tree of IPITT and can execute actions on the host broker.

6.5 Evaluation

In this section, we evaluate IPITT operating on an existing pub/sub system. We implemented our ITT planner using Python and the Gurobi Optimizer[98], and implemented the *shift* operation[33] and ITT agent in Java on the PADRES pub/sub system[13].

We use a cluster of 20 Dell PowerEdge R430 servers for our evaluation.

Note that PADRES uses the content-based subscription model, which allows fine-grained filters to be expressed. Although IPITT is not tied to any subscription model, we choose to evaluate our approach using a content-based system since it generates larger routing tables. Consequently, broker migration incurs more overhead and is therefore more challenging for ITT, which performs frequent routing table updates.

Furthermore, we compare our approach with the best-first search (*BFS*) heuristic to solve ITT, presented by Yoon *et al.*[51]. *BFS* uses an algorithm similar to those underlying state of the art planners, utilizes domain specific heuristics, and avoids generating all actions upfront. *BFS* models the planning problem as a graph with each node representing a different state of the problem (*i.e.*, a set of propositions) and each edge representing an action. Given a time limit and depth limit, *BFS* searches for a plan and returns the best result (lowest number of actions). The depth limit stops the heuristic from following a branch of the graph for an excessive period of time. Upon reaching the depth limit, *BFS* starts a new search on a different part of the graph. This is repeated until the time limit is reached.

6.5.1 Workload

We evaluate IPITT using acyclic hierarchical topologies which are common in high-throughput pub/sub systems[14, 3]. The generated tree topologies used for the evaluation have a maximum node degree of 5, 6 or 7 and each have two balanced and unbalanced variations. The degree of each broker is selected based on a random uniform distribution and the maximum degree of the tree. Each topology has two variations generated with different seeds. The link latencies are between 10 milliseconds and 50 milliseconds and selected based on an uniform distribution. We run our experiments using 12 topologies covering a wide variety of trees.

We evaluate IPITT and *BFS* using a synthesized workload with the following parameters: We use 20 publishers all connected to the root of the tree topology each publishing 5 publications per second. The content space has 10 different class of

publications with two attributes each between 0 and 1000. Each publisher generates publications based on an uniform distribution from the content space. Each subscriber subscribes to one class of publication with the popularity of classes based on a Zipf distribution with $a = 1.1$ [119] and the attribute range of each subscription based on a random uniform distribution. Subscribers choose their edge brokers based on a random uniform distribution among the leaf brokers of the overlay. Clients are located on different VMs than the brokers, in groups of at most 80 clients per VM. Publishers publish messages for 10 minutes. Starting from the 120^{th} second of each experiment, an incremental topology transformation is performed on the overlay. Each experiment is run with 12 different topologies and 2 different workloads (24 runs) and the metric values are averaged.

We evaluate the impact of three variables on our metrics. *Overlay size* is the number of brokers in the overlay and *change ratio* is the percentage of links changed in the goal topology. These variables influence the IP model size and the size of the generated plan in terms of number of actions and steps. Furthermore, we change the number of *subscribers* to study the scalability of plan execution in terms of client size.

In order to be able to control the change ratio between the goal and initial topologies, we do not employ an overlay design algorithm. The goal topology is calculated by changing $N\%$ of the links in the initial topology. While a randomly generated topology may degrade the performance of the overlay after the transformation, our purpose is to study the transformation period itself. Optimal overlay construction is orthogonal to our work and is not evaluated here.

6.5.2 Metrics

Planning time is the time required for the ITT planner to find a plan to transform the initial topology to the goal topology. This includes the time it takes to generate the model and solve it. It is important that the planning time is limited, as the objective is to perform transformations periodically. Systems such as GooPS perform re-evaluation and transformations frequently, requiring a planning time in order of

minutes [38].

Plan quality is defined in terms of the number of steps and number of actions in the plan. A high quality plan minimizes disruption to the pub/sub system. Since the number of link changes directly influences service disruption in the overlay[131], we consider a plan with a minimal number of actions as high quality. Furthermore, a higher number of steps can result in a longer plan execution time which can impact service disruption.

Publication delivery latency is the time it takes to deliver a publication to a matching subscriber. We collect all delivery latencies from subscribers and calculate the 99^{th} percentile of publication delivery latencies (PDL) of all publications which are in transit during the topology transformation and are therefore affected by it. This metric represents the effect of the transformation on the clients of the pub/sub system.

Broker queue size is the growth of the message queue size of brokers during transformation. Performing each *shift* action requires the buffering of messages until the *shift* is finished. This metric represents the buffering load and therefore the impact of the transformation on the overlay brokers. We measure queue size of each broker at the beginning and end of each *shift*, calculate the difference and take the 99^{th} percentile of the collected values.

Plan execution time is the time it takes to completely execute a transformation plan computed by the planner on the overlay. This metric represents the duration of service disruption and its effect on the clients.

6.5.3 Experiments

Impact of overlay size on planning: In this experiment, we study the impact of overlay size on the planning time and plan quality. We change the *overlay size* from 20 to 150 brokers, which is in line with the overlay size of systems such as GooPS [38], with a change ratio of 20%. This means each goal topology is generated by randomly changing 20% of the links in the initial topology. We evaluate IPITT using

the first heuristic which generates actions for all paths (IPITT$_A$), the second heuristic which generates actions for each path separately (IPITT$_E$), naive IPITT (IPITT$_N$) which generates all possible actions, and *BFS* with a time limit of 10 minutes.

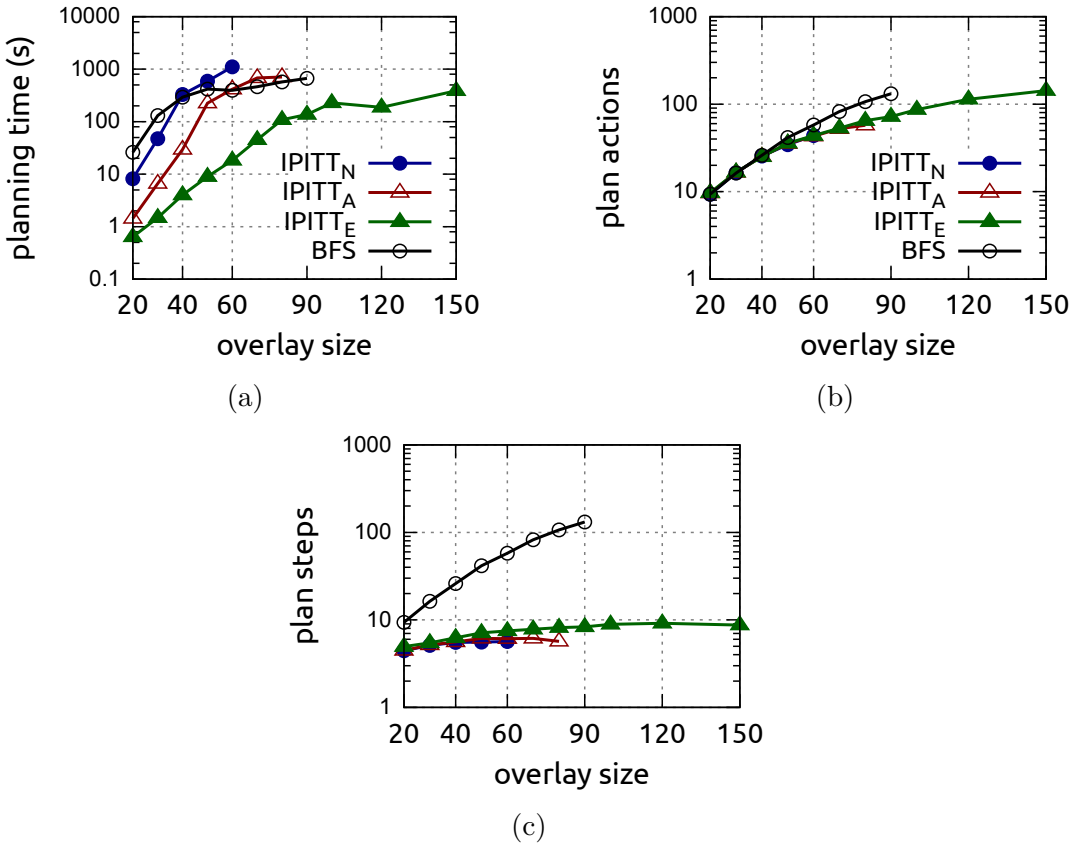


Figure 6.5.1: Impact of overlay size on IPITT planner

Figure 6.5.1a shows that increasing the number of brokers in the overlay results in a longer planning time. This is inevitable as a larger number of brokers results in a larger IP model for IPITT and a larger state space graph for *BFS*. Nonetheless, IPITT$_E$ results in the lowest planning time for all overlay sizes. IPITT$_N$ is able to find a plan only for overlays of size 50 or less and it cannot find a plan for overlays with more than 60 brokers. The reason is that the model size and consequently the planning time exhibits a polynomial growth (n^3). IPITT$_A$ shows a similar result but with a slower growth rate. However, since increasing the overlay size results in a larger number of brokers located on a transformation path and IPITT$_A$ generates all actions for all such brokers, the planning time increases polynomially. IPITT$_A$ can

find a valid plan for overlays of size 60 or smaller in less than 10 minutes and is not able to find a plan for overlays of size larger than 80.

BFS performs similarly to IPITT_A and outperforms IPITT_A for overlay sizes of 60 or more. Furthermore, *BFS* can find a plan for overlays of up to 90 brokers but not beyond that. The reason can be attributed to how it searches the state space graph. *BFS* identifies the next action based on the estimated cost and follows this branch until it reaches the specified limit. Upon reaching the end, it starts over from the second best option. In larger overlays, the probability that multiple candidates for the next action have the same cost estimate increases, with such ties broken randomly. Therefore, in larger overlays, despite a larger state space size, *BFS* still relies on random selection of the next best option which results in a decreasing chance of success.

IPITT_E is the only approach that is able find a plan for overlays of 100 or more brokers and is able to find such a plan in under 45 seconds for overlays of 70 or less, and under 400 seconds for overlays of size 150. The reason for the scalability of IPITT_E is that the action set generation does not grow polynomially with the overlay size since IPITT_E generates actions for each path separately. Therefore, compared to *BFS*, IPITT_E reduces the planning time for overlays of size 50 or less by a factor of 61 and for overlays of size 60 or more by a factor of 10. Compared to IPITT_N, IPITT_E reduces the planning time by a factor of up to 82 times. However, our current heuristic is not able to find a plan for overlays of size 200 or more in under 10 minutes. While we have not performed any optimization for our evaluation, there exists techniques such as warm starting of the IP solver that can improve the overall performance of the IP solver [134]. Furthermore, IPITT_E uses a simple heuristic to limit the search space which can be further extended to provide a smaller action set.

Table 6.5.1 shows the memory footprint and IP model size of the different approaches for an overlay size of 60 and change ratio of 20%. Since the IP formulation creates several variables and constraints for each action, reducing the number of shift actions greatly reduces the IP model size. Consequently, the solver can find a solution faster and requires less memory.

Figure 6.5.1b and 6.5.1c show the quality of the plan calculated by each approach

Approach	Mem. (MB)	Actions	Variables	Constraints
IPITT$_N$	11 352	205 320	1 355 820	3 810 515
IPITT$_A$	2 750	35 904	392 928	919 799
IPITT$_E$	405	1 860	154 620	211 973
BFS	230	-	-	-

Table 6.5.1: IPITT planner memory footprint and model size

in terms of number of actions and number of steps. For overlays of size 40 or more, IPITT$_E$ can find a plan with up to 45% fewer actions compared to *BFS*. Furthermore, compared to IPITT$_N$ and IPITT$_A$ which can find an optimal or near optimal plan due to their larger search space, IPITT$_E$ calculates a plan with only up to 5% and 12% more actions, respectively. *BFS* does not parallelize the plan actions: Therefore, the number of steps is equal to the number of actions in the plan. In contrast, the IP approach produces a plan with the lowest possible number of steps. However, due to the use of heuristics to reduce the search space size, only IPITT$_N$ guarantees that no plan with a lower number of steps is possible. In comparison to the optimal plan, IPITT$_E$ produces plans with up to 32% higher number of steps. The increase in IPITT$_A$ is at most 11%. This means that generating an action set which includes *shift*s across different transformation paths can result in more efficient plans.

Impact of change ratio on planning: In this experiment, we study the impact of the *change ratio* on planning time and plan quality. We increase the change ratio from 5% to 30% in an overlay of 60 brokers. With frequent evaluation of the overlay, high change ratios are in practice unlikely. Figure 6.5.2a shows the planning time of each approach. In all cases, the planning time increases because a higher change ratio results in a larger IP model or space state graph. A higher change ratio requires a higher number of actions and a larger plan, which increases T in the IP model and graph depth in *BFS*. Due to the larger problem size, IPITT$_N$ and IPITT$_A$ are not able to find a plan for change ratios of 25% or higher. *BFS* can find a plan for 30% change ratio in 546 seconds and IPITT$_E$ in 117 seconds. Although *BFS* does not return a plan until the time limit is reached, in this experiment, we have recorded the time that *BFS* finds the best plan. Nonetheless, IPITT$_E$ can reduce planning time by up to 95% compared to that of *BFS*. Regarding plan quality, similar to previous

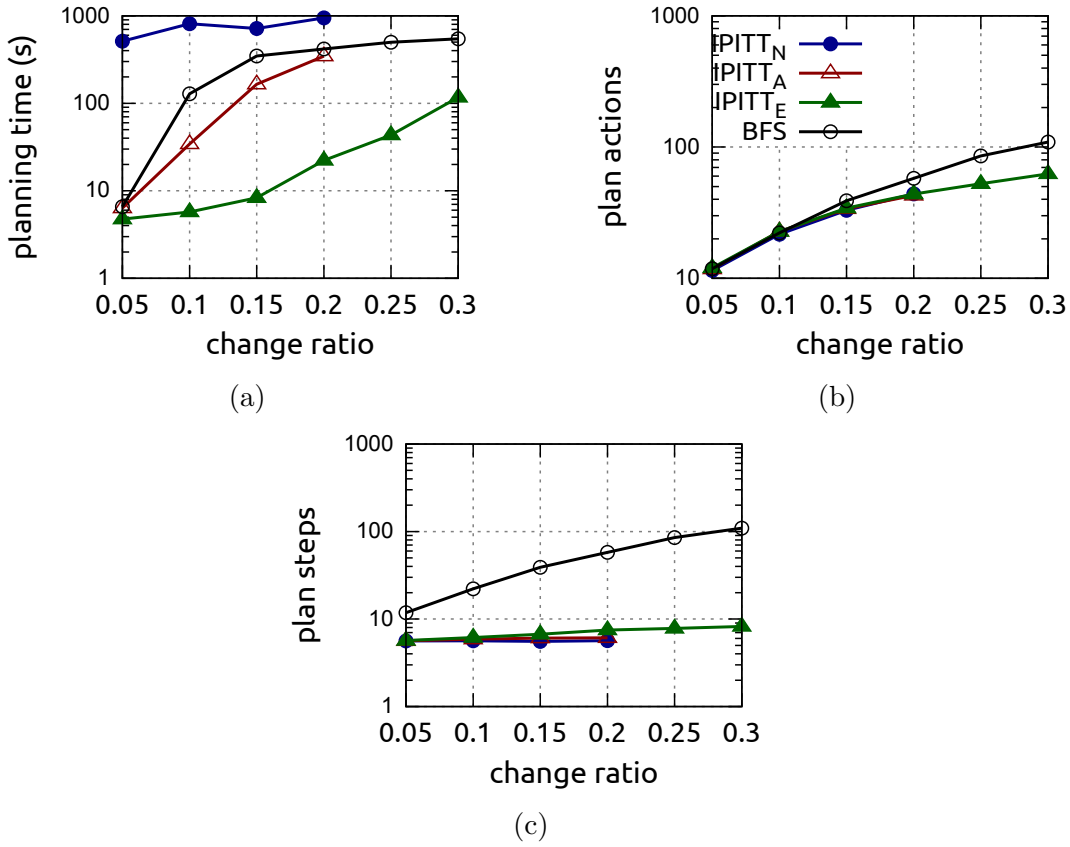


Figure 6.5.2: Impact of overlay change ratio on IPITT planner

experiment, IP-based approaches can outperform *BFS*. While for change ratios of 5% and 10%, IPITT_E and *BFS* produce plans with the same number of actions, in larger change ratios, IPITT_E results in plans with up to 42% less actions. In comparison to the optimal plans calculated by IPITT_N, IPITT_E produces plans with up to 5% more actions and up to 32% more steps.

These two experiments confirm the effectiveness of our two proposed heuristics in reducing the search space size of the ITT problem and scaling the ITT planner to overlays of up to 150 brokers. Furthermore, we have shown the different levels of trade-off that the three IP-based approaches can provide.

In the next two experiments, we use the calculated plans by *BFS* and IPITT_E which we simply refer to as IPITT.

Impact of ITT on clients: In this experiment, we study the impact of ITT on the subscribers of the pub/sub system. Figure 6.5.3a shows the impact of increasing the overlay size on the PDL with 1000 subscribers and a change ratio of 20%. Since each action forces temporary publication buffering by the involved brokers, subscribers receiving publications which traverse brokers involved in a *shift* operation experience a higher delivery latency. IPITT and *BFS* show similar affected latencies which are up to 2 seconds more than the PDL for publications not affected by the transformation. The impact of transformation on latency is similar in both approaches since it primarily depends on the type of operation used, which is the same (*shift*). In all cases, increasing the overlay size results in a higher latency since larger overlays have a longer path between publishers and subscribers.

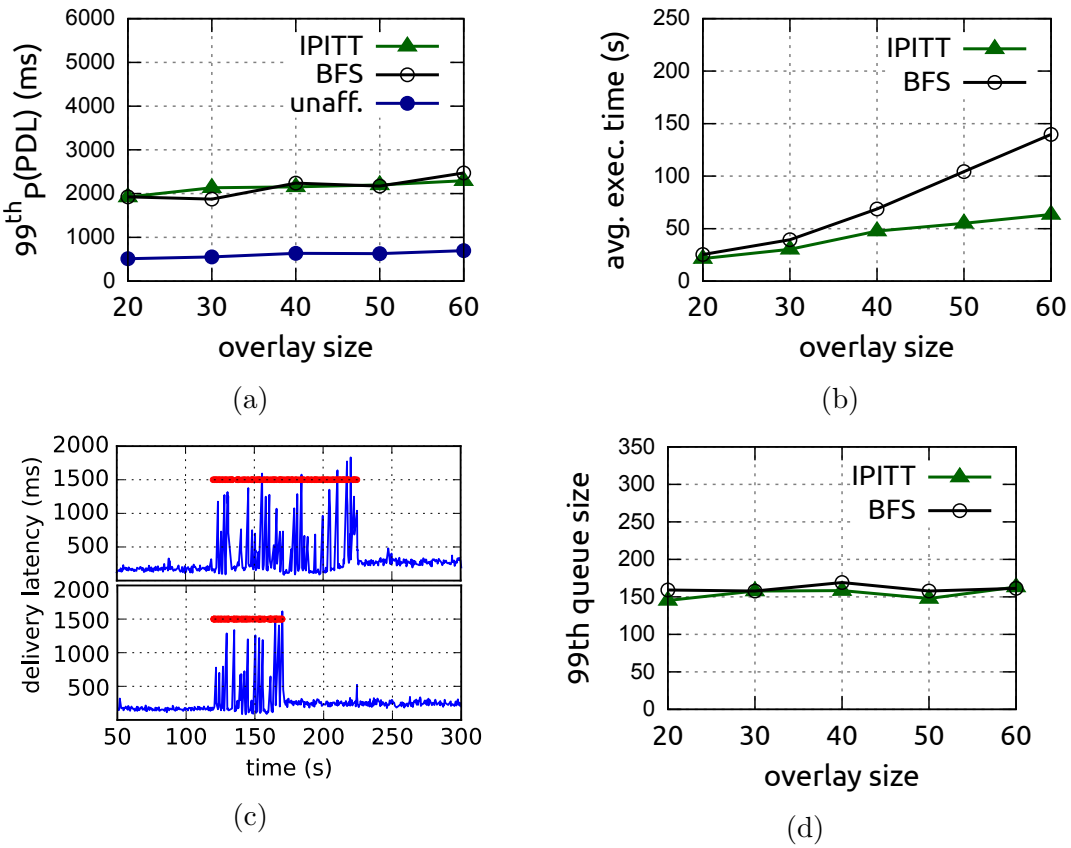


Figure 6.5.3: Impact of overlay size on IPITT plan execution metrics

Figure 6.5.3b shows the average plan execution time (M_{PET}) of each approach. Since IPITT produces plans with fewer steps, and actions in each step are executed in

parallel, the M_{PET} of IPITT is up to 55% shorter than that of *BFS*. While the execution time does not affect the PDL (this is affected by how long the operation buffers publications), it does influence the number of messages that have a higher delivery latency. Figure 6.5.3c shows the delivery latency of delivered publications to the same client in two different scenarios. In the lower section, the transformation uses the plan generated by IPITT and the upper section is the delivery latencies during the execution of a *BFS* plan. The red dots on top of each plot show the time taken by *shift* actions executed on the overlay. While in both approaches the plan execution period results in an increase in delivery latency, higher number of publications face this increase in *BFS*. The difference in the number of affected messages is proportional to the plan execution time (Figure 6.5.3b). This experiment confirms that IPITT is able to minimize disruption to publication delivery by finding a plan with a minimal number of steps.

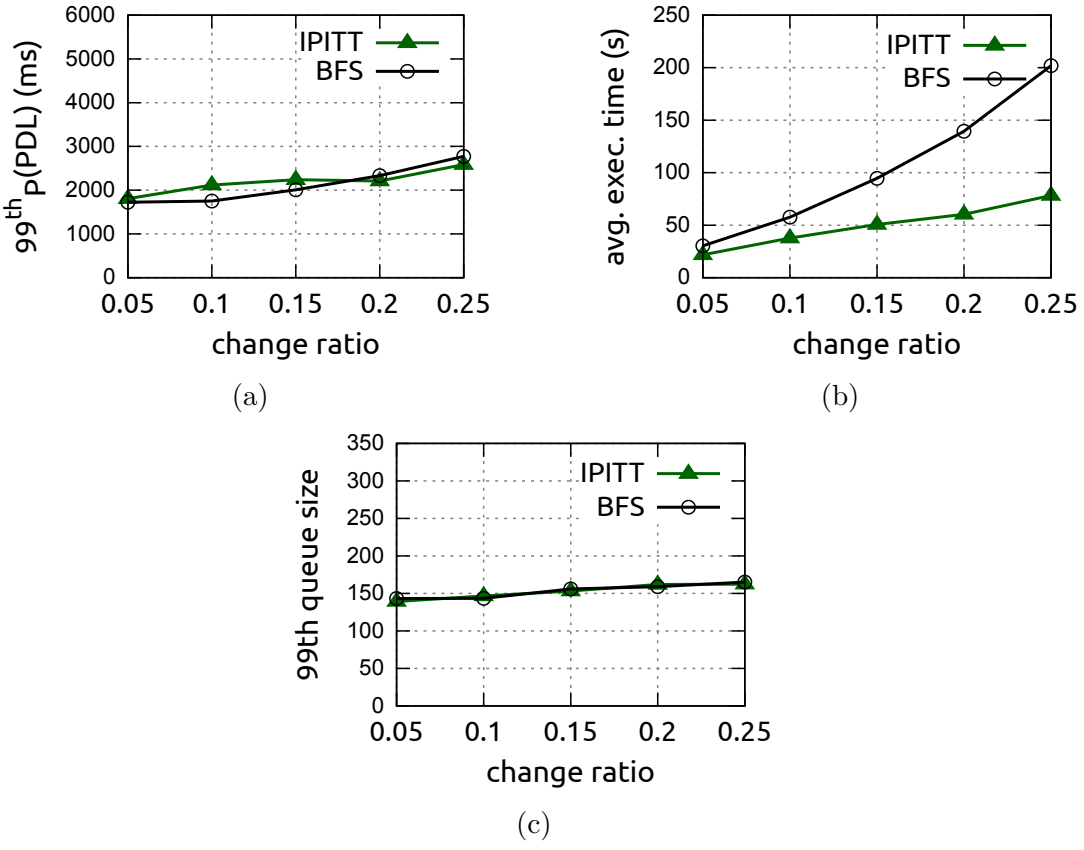


Figure 6.5.4: Impact of overlay change ratio on IPITT plan execution metrics

Figure 6.5.4a shows the impact of increasing the change ratio on the PDL in an overlay of 60 brokers with 1000 subscribers. A higher change ratio requires a plan with a larger number of actions and steps. Consequently, publications affected by the transformation are more likely to encounter more than one action execution and be buffered more than once while being routed. Therefore, both approaches show an increase in PDL of up to 2.2 seconds. However, similar to the previous experiment, since the M_{PET} of IPITT is up to 60% shorter than that of *BFS* (Figure 6.5.4b), a smaller number of publications are affected by this disruption.

Figure 6.5.5a shows that increasing the number of subscribers in an overlay of 60 brokers with 20% change ratio results in a higher PDL. The reason is that a larger volume of subscriptions in the system leads to a larger routing table size at the brokers. Since *shift* actions perform changes on the routing table of each involved broker, greater table sizes result in longer latencies. For the same reason, the M_{PET} of each approach grows with an increase in subscribers. However, both PDL and M_{PET} grow sublinearly and therefore the incremental transformation is scalable with respect to the number of subscribers. This increase depends on the type and implementation of the operation. In this case, it affects both approaches similarly.

These experiments confirm that IPITT provides a scalable approach to incremental topology transformation while minimizing disruption to clients.

Impact of ITT on brokers: In this experiment, we study the impact of ITT and plan quality on the overlay brokers. Since transformations without message loss require temporary publication buffering, it is important that the generated plans do not result in buffering requirements that are beyond the capacity of individual brokers. An incremental transformation approach requires less buffering since each action involves only a small group of brokers and finishes quickly. Figure 6.5.3d and 6.5.4c show the growth of the queue size of brokers during topology transformation. Increasing the overlay size and change ratio does not impact any of the approaches. This is due to the inherently low buffering requirement of incremental transformation. However, IPITT is able to execute plans faster without any increase in queue size. The reason is that the plan calculation and execution of IPITT parallelize actions which are independent from each other and do not involve the same broker. Therefore, a

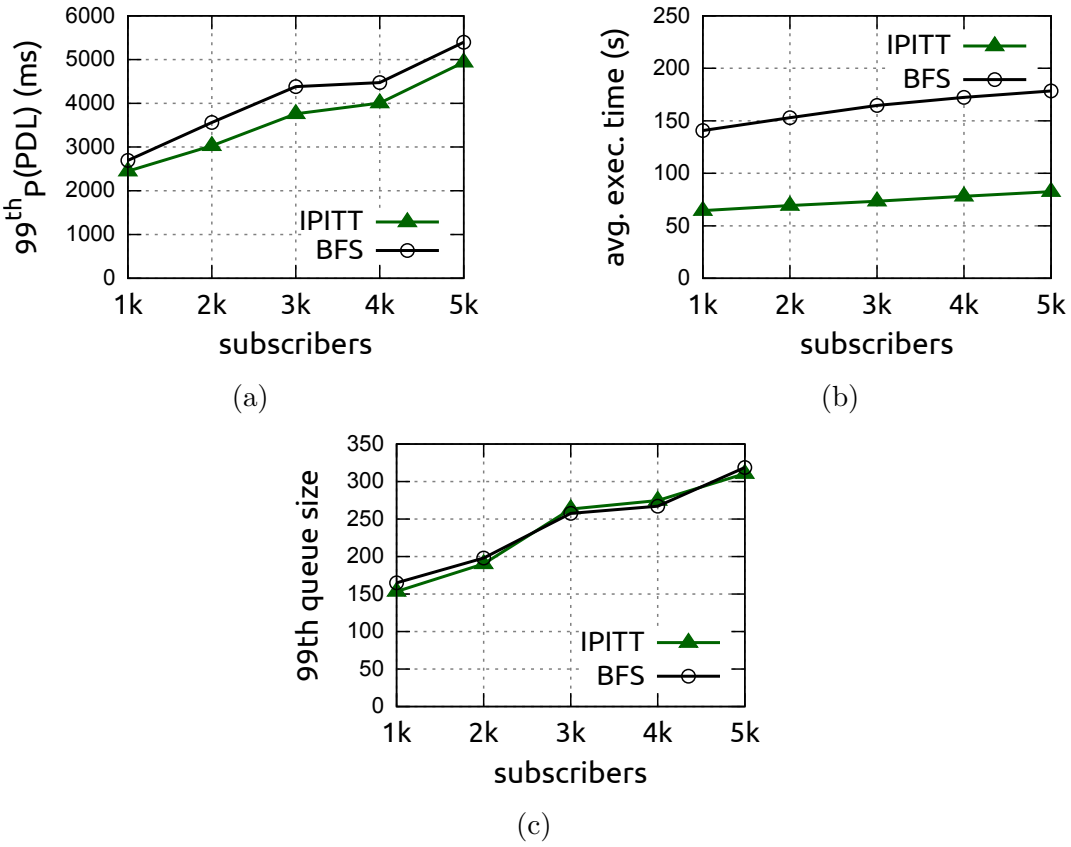


Figure 6.5.5: Impact of number of subscribers on IPITT plan execution metrics

higher number of actions per step does not influence the queue size of individual brokers.

Figure 6.5.5c shows that increasing subscribers results in a sublinear growth of the queue size of brokers by up to 110% when the number of subscribers increases by a factor of 5. The reason is that for the same workload, a higher number of subscribers downstream (at leaf brokers) results in a higher number of publications routed downstream. Therefore, the number of publications in transit is higher at any given time when compared to measurements using the same overlay and workload but with a lower number of subscriptions. In fact, any parameter that increases the number of publications in the overlay, such as publication rate and match ratio, will have the same effect on the brokers during topology transformation. This experiment shows that plans generated by IPITT can perform incremental transformations on the overlay with minimal load increase on brokers.

Chapter 7

Delivery Guarantees in Distributed Content-Based Pub/Sub

In this chapter, we present a set of *Pub/Sub Delivery Guarantees* (PSDG) which addresses the shortcoming of existing delivery guarantees and simplifies development of distributed applications based on a pub/sub service. We explain the shortcomings of the delivery guarantees which do not consider the message installation period in Section 7.2. In Section 7.3 and 7.4, we introduce a set of delivery guarantees for overlay-based pub/sub systems and corresponding routing algorithms to provide these guarantees. Lastly, we present and discuss the result of our experimental evaluation in Section 7.5.

7.1 Message Installation in Distributed Pub/Sub

Figure 7.1.1 shows an example of a content-based pub/sub system. The overlay consists of five brokers and brokers *B1*, *B3*, *B4* and *B5* are the *EB*s of *S1*, *P1*, *S2* and *P2*, respectively. Clients can submit different types of messages to their

*EB*s, where a message $m \in \{a, s, p, ua, us\}$, namely, advertisement, subscription, publication, unadvertisement and unsubscription. In such a pub/sub system, message m is considered *installed* on a broker B, when m is received and processed by B. Processing m on B includes updating the routing tables if necessary and sending out new messages in order to route m. Message m is considered installed in the overlay when it has been installed on all brokers receiving m.

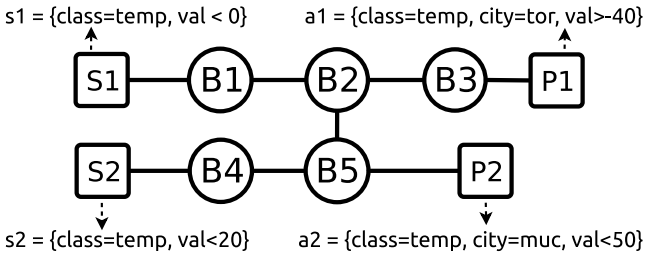


Figure 7.1.1: Content-based publish/subscribe

In the life cycle of each pub/sub message m, we consider three stages identified by timestamps m_g, m_r and m_p (Table 7.1.1). For example, in Figure 7.1.1, $s1_r$ is the time that *B1* receives $s1$. Since $s1$ matches both $a1$ and $a2$, $s1_p$ is the time that $s1$ is processed by the brokers on $T_{EB(S1)\leftrightarrow EB(P1)}$ (the path connecting *EB*s of *S1* and *P1*) and $T_{EB(S1)\leftrightarrow EB(P2)}$, namely, *B1*, *B2*, *B3* and *B5*. Note that due to link latency and message processing time the following is always true: $m_g < m_r < m_p$. Furthermore, we define the installation time of message m as $m_i = m_p - m_r$ (Figure 7.1.2).

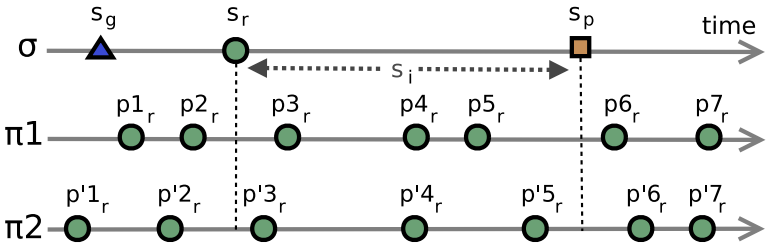


Figure 7.1.2: Installation time for subscription s

For the remainder of this chapter, we make the following assumptions on the underlying pub/sub system:

1. Brokers are connected to each other in an acyclic topology. Messages between

two neighboring brokers are transmitted without message loss or reordering. This can be provided using reliable TCP connections.

2. Brokers process messages in the same order that they are received. In combination with the previous assumption, this means publication delivery in the overlay maintains per publisher ordering.

3. Overlay brokers have clocks synchronized up to an error bound. The clock synchronization can be NTP or more accurate methods such as GPS-based clock synchronization.

4. Lastly, each client publishes publications with unique increasing IDs. This can be achieved by giving publishers a unique ID and making sure each publication contains the publisher ID and an incrementing publication ID.

$\mathbb{A}$, $\mathbb{UA}$	set of all advertisements and unadvertisements in the system
$\mathbb{S}$, $\mathbb{US}$	set of all subscriptions and unsubscriptions in the system
$\mathbb{P}$	set of all publications in the system
Π	set of all publishers in the system
Σ	set of all subscribers in the overlay
$\mathbb{B}$	set of all brokers in the system
$m \models m'$	message m matches message m'
$T_{B \leftrightarrow B'} \subseteq \mathbb{B}$	subtree connecting brokers B and B'
$C(m)$	client generating message m
$EB(m)$	edge broker of $C(m)$
m_g	the time that message m is generated on $C(m)$
m_r	the time that message m is received by $EB(m)$
m_p	the time that message m is installed in the overlay
m_p^B	the time that message m is installed in $T_{EB(m) \leftrightarrow B}$
$m_i = m_p - m_r$	the installation time of m in the overlay
$m_i^B = m_p^B - m_r$	the installation time of m in $T_{EB(m) \leftrightarrow B}$
$\mathbb{S}(p)$	set of subscriptions matching p $\in \mathbb{P}$
$\mathbb{P}(s)$	set of publications received by $C(s), s \in \mathbb{S}$
$\mathbb{P}_\pi$	set of publications published by publisher $\pi \in \Pi$
$\mathbb{P}_\pi(s)$	set of publications published by π and received by $C(s)$
$\mathbb{P}^i(s)$	set of publications received by $C(s)$ during s_i
$\mathbb{P}^i_\pi(s)$	set of publications published by π, received by $C(s)$ during s_i

Table 7.1.1: PSDG Notations

7.2 Shortcomings of Existing Delivery Guarantees

Existing pub/sub systems guarantee publication delivery to all matching subscriptions in the system. However, this guarantee is only provided for subscriptions that are installed in the overlay [14, 26]. This means that the subscription has been forwarded according to the SRT on each broker and has updated the PRT of each broker. Therefore, based on existing guarantees, the set of publications received by subscription s is $\mathbb{P}(s) = \{\bigcup \mathbb{P}_\pi(s) \mid \forall \pi \in \Pi\}$. For each publisher π, $\mathbb{P}_\pi(s) = \{p \in \mathbb{P}_\pi \mid p_r \geq s_p^B \wedge p \models s\}$, where B is the *EB* of π. For example, in Figure 7.1.2, where susbcriber σ matches publishers $\pi 1$ and $\pi 2$, the set of publications guaranteed to be delivered to σ is any publication in $\{p6, p7, p'6, p'7\}$ that matches s.

The installation time of message m, m_i, is a function of the overlay size, link latency and processing power of the overlay brokers. During the installation time, the delivery guarantees of the distributed pub/sub system is non-deterministic [25] as the overlay routing state takes some time to stabilize after a new message is submitted to the overlay [26]. Furthermore, m_i can change if for example the overlay is elastic and adapts to the existing load. In the following, we explain the consequences of an unknown m_i in a distributed pub/sub system.

Unknown s_i – For a subscription s, $s_i = max(\{s_i^{EB(a)} | \forall a \in \mathbb{A}, s \models a\})$, *i.e.* when s has reached the *EB* of all matching advertisements. For each publisher with advertisement a, $\pi = C(a)$, the set of publications that $C(s)$ should receive from π during s_i, $\mathbb{P}_\pi^{\mathrm{i}}(s) = \{p \in \mathbb{P}_\pi \mid s_r \leq p_r \leq s_p^B \wedge p \models s\}$, *i.e.* publications published while s is propagating towards the publisher, where B is the *EB* of publisher π. Therefore, $\mathbb{P}^i(s) = \bigcup \mathbb{P}_{\mathrm{P}}^{\mathrm{i}}(s)$, *i.e.* any publication that arrives at its *EB* after its matching subscription s is received by its *EB* but before s is installed on $T_{EB(s) \leftrightarrow EB(p)}$. The set of publications received by s from publisher π consists of publications published during s_i and publications published after s_p. Furthermore, $\mathbb{P}_\pi^{\mathrm{i}}(s) \subset \mathbb{P}_\pi(s)$. For example, in Figure 7.1.2, $\mathbb{P}_{\pi 1}^{\mathrm{i}}(s) = \{p3, p4, p5\}$ and $\mathbb{P}_{\pi 2}^{\mathrm{i}}(s) = \{p'3, p'4, p'5\}$. However, for two identical subscriptions s and s', submitted at the same time to the overlay ($s_r = s'_r$), connected to different edge brokers, $EB(s) \neq EB(s')$, $\mathbb{P}_\pi^{\mathrm{i}}(s)$ and $\mathbb{P}_\pi^{\mathrm{i}}(s')$ can be different and consequently $\mathbb{P}_\pi(s) \neq \mathbb{P}_\pi(s')$. The reason is that there exists no delivery guarantees clarifying $\mathbb{P}^i(s)$ and this set depends on the overlay and existing

subscriptions.

Let $T_{EB(p)\leftrightarrow EB(s)}$ be the subtree that connects publisher $\pi = C(p)$ and subscriber $\sigma = C(s)$. For each $p \in \mathbb{P}^i(s)$, p is delivered to s if one of the following two cases occur.

1. $\exists \;\; s' \in \mathbb{S}$ such that $\; T_{EB(p)\leftrightarrow EB(s)} \subseteq T_{EB(p)\leftrightarrow EB(s')}$ and $p \models s'$, *i.e.* an existing subscription s' results in p being propagated towards s even though s is not fully propagated.

2. $\exists \;\; s' \in \mathbb{S} \quad$ such that $\quad T_{EB(p)\leftrightarrow EB(s)} \not\subset T_{EB(p)\leftrightarrow EB(s')} \quad$ and $T_{EB(p)\leftrightarrow EB(s)} \cap T_{EB(p)\leftrightarrow EB(s)} \neq \varnothing$, *i.e.* a existing subscription s' covers part of the path that p must traverse towards s, and s reaches the broker that disjoints the two paths before p.

Otherwise, s does not receive p. As an example of case 2, let's assume in Figure 7.1.1, $s1_r > s2_p$, namely, $s1$ arrives at *B1* after $s2$ is installed in the overlay. Since $s1 \subseteq s2$ and the two paths connecting $S1$ and $S2$ to $P1$ have overlaps but are not exactly the same, during $s1_i$, any p that matches both $s1$ and $s2$, is delivered to $s1$ only if p arrives at *B2* after $s1$ is installed on *B2*.

Unknown p_i –An unknown installation time for publication p means that a publisher does not know when its publication p has reached the *EB* of all matching subscribers. If for example, a publisher wants to wait for the delivery of a publication before leaving the overlay, it has to resort to estimating p_i or use distributed coordination services. Similarly, as a result of an unknown us_i and ua_i, clients do not know when an unsubscription or unadvertisement has been completely processed by the overlay.

The unknown installation time of a message, as described above, applies to distributed overlay-based pub/sub systems that utilize a multi-hop message forwarding scheme. *SIENA* addresses the message installation time by providing guaranteed delivery only to subscription that are installed in the overlay [14]. *Google Pub/Sub* provides a similar guarantee, however publishers and subscribers are connected via only two brokers and the system makes sure that the path between each publisher and

subscriber *EB* is the shortest path via the Google datacenters [10], thereby minimizing the installation time.

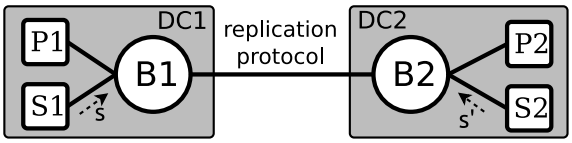


Figure 7.2.1: Cross Datacenter pub/sub

Some pub/sub systems, such as Kafka [64], Pulsar [63] and BlueDove [135], use a one-hop communication in the overlay whereby a publisher and subscriber are connected via only one broker. This one-hop message forwarding scheme (*a.k.a.* rendezvous-based routing), while limiting the propagation time of a message, can still suffer from unknown installation time. These systems partition the publication space and assign each topic (or attribute space in case of content-based pub/sub) to a specific broker. While in single-datacenter scenarios, these systems can perform well, in multi-datacenter scenarios, the unknown installation time of a message can become an issue. In multi-datacenter scenarios, these systems resort to one cluster (overlay) per DC and use replication to address cross datacenter partitioning of the publication space. For example, in Figure 7.2.1, *P1* and *S1* are located in one datacenter (DC1) and *P2* and *S2* are located in DC2. Since both subscribers are interested in both publishers, due to the potentially high-latency connection between *B2* and *B1*, s'_i is unknown and $\mathbb{P}^{\mathrm{i}}_{\mathrm{P1}}(s')$ can be different than $\mathbb{P}^{\mathrm{i}}_{\mathrm{P1}}(s)$. Even if all publications of *P1* published via *B1* are replicated to *B2*, without clearly specifying $\mathbb{P}^i(s)$ and $\mathbb{P}^i(s')$, $\mathbb{P}(s)$ can be different from $\mathbb{P}(s')$. Topic-based pub/sub systems that use a log (*a.k.a.* ledger) abstraction, such as Kafka and Pulsar, provide the concept of cursors (*i.e.* a log offset) which can be used to refer to a specific message in the log. While log offsets are specific to log-based systems, we use a generalization of this idea as the basis for one of the delivery guarantees that we propose. Some pub/sub systems such as *Wormhole* [65], omit the broker and have publishers directly serve subscribers. While reducing message installation to possible minimum, these systems have very limited scalability in terms of number of subscribers that a publisher can serve.

The unknown installation time of messages in the overlay also affects the brokers. For example, Listing 7.2.1 shows a snippet from an existing pub/sub system resorting

to estimating the time required for queued messages to get installed in the overlay, before the broker can shutdown. Allowing the broker to receive a notification when the last queued message is installed in the overlay, would provide a more reliable alternative.

```
public void shutdownBroker() {
  // prepare broker for shutdown and update neighbors
  ...
  // Allow some time for the message to get through
  // using some statistical data
  long t = //estimated installation time for queued messages
  sleep(t);
  inQueue.add(new ShutdownMessage());
  brokerCore.shutdown();
}
```

Listing 7.2.1: Estimating message installation time

7.3 Message Installation and Delivery Guarantees

In this section, we present acknowledgment-based (*ack-based*) message installation for pub/sub and three new delivery guarantees that can be utilized by subscribers to clearly specify $\mathbb{P}^i(s)$ and consequently $\mathbb{P}(s)$.

Ack-based message installation: Any client C can request an acknowledgment (ack) for any message m that it sends to its *EB*. C receives the ack, ack_m, when m is installed in the overlay and ack_m is propagated back to $EB(m)$.

Using the ack-based installation, a publisher can make sure that its advertisement a is installed in the overlay by waiting for ack_a. Similarly, to make sure a publication p has been received by the *EB* of all existing subscribers, a publisher can request an ack and wait for ack_p. This *ack-based* installation is a generalization of the reply message in pub/sub-with-reply [57, 58], where a reply message is proposed only for publications.

Due to the decentralized routing information and limited overlay knowledge of each broker, a broker is not aware of all the other brokers that need to process m in order for ack_m to be sent to the client. Therefore, in order to maintain the scalability and anonymity provided by the pub/sub paradigm, ack messages should also be forwarded in a hop-by-hop fashion.

In order to collect and aggregate ack messages, we use a generalization of the *KER* approach proposed by Cugola *et al.* [57]. We generalize this approach by determining the number of ack messages to collect for advertisements (and unadvertisements) based on $\mathbb{N}_B$ (the set of neighboring brokers of B) and for subscriptions (and unsubscriptions) based on matching advertisements in SRT. In case of advertisements, each broker B waits for $|\mathbb{N}_B| - 1$ acks.

In this work, we distinguish between two states that a subscription can have: *installed* and *active*. Upon processing s at broker B and adding s to its PRT, s is considered *installed* at B. However, an installed subscription is initially inactive and is not considered for matching upon processing a publication. In order for s to become *active* and be matched with arriving publications, s must be explicitly activated. Note that in existing pub/sub systems, a subscription is active upon being installed in the PRT.

Next, we explain three delivery guarantees that a subscriber can specify, *ack-based*, *time-based* and *ID-based*.

Ack-based delivery guarantee: This guarantee is based on ack-based message installation for subscriptions. A subscriber requesting an acknowledgment upon submitting its subscription s to its EB is notified when s is installed in the overlay. Furthermore, no publication is delivered to s before it is installed in the overlay. The set of publications received by s is $\mathbb{P}(s) = \{p \in \mathbb{P} \mid p \models s \wedge p_r \geq s_p\}$.

Note that a subscription might receive publications before it is completely installed in the overlay, depending on which broker it is connected to and existing subscriptions in the overlay, as described in Section 7.2.

While the *ack-based* delivery guarantee defines $\mathbb{P}^i(s)$ based on the condition that s is

installed in the overlay, the next two delivery guarantees, which we collectively refer to as *ref-based* delivery guarantees, allow a subscriber to define $\mathbb{P}^i(s)$ with regards to a given timestamp or publication ID.

Time-based delivery guarantee: A subscriber requesting a *time-based* delivery guarantee for subscription s can specify $\mathbb{P}^i(s)$ by providing a timestamp that serves as s_p on all matching publishers. The set of publications received by s is $\mathbb{P}(s) = \{p \in \mathbb{P} | p \models s \wedge p_r \geq s.timestamp\}$. In other words, a subscriber can specify a point in time where s is considered active in the overlay.

When a *time-based* delivery guarantee is requested without specifying a timestamp, a subscriber is considered active as soon as it is received by its *EB* ($s.timestamp = s_r$). Therefore, regardless of overlay size and propagation time the subscriber receives any matching publication that enters the overlay at the same time or after s. By explicitly defining $s.timestamp$, the *time-based* delivery guarantee allows a subscriber to receive publications published in the past or to subscribe to future publications.

As the *time-based* guarantee relies on timestamps generated on *EB* of publisher and subscribers, the accuracy of the provided guarantee depends on the clock synchronization accuracy and has an error range of 2δ where δ is the accuracy of the clocks on *EB*s.

The *time-based* guarantee can be used by a group of subscribers located on different parts of the overlay to receive the same set of publications. However, since subscriptions timestamped by different *EB*s will not have the exact same s_r, for each subscription s, $\mathbb{P}^i(s)$ can be different. In order to resolve this problem, we introduce the notion of subscription groups (*sgroup*) into our pub/sub model. An *sgroup* is a set of subscriptions identified with the same group ID. The timestamp of the first subscription group member that is received by the publisher's *EB* is considered as the s_r for all group members on that *EB*. This is necessary to make sure that other group members arriving after s, receive the same set of publications that was sent to $C(s)$. Therefore, for a subscription s which is a member of the *sgroup* G, the set of publications received from publisher π is $\mathbb{P}_\pi(s) = \{p \in \mathbb{P} : p \models s \wedge p_r \geq G_r\}$ where G_r is s_r of the first subscription belonging to G that reaches the *EB* of publisher π. Note that *sgroup* is a different concept than consumer group in Kafka [136].

Consumer groups provide a way to distribute publications on a topic among the group members, for example, in a round robin fashion.

As an example, consider an application consisting of a data producer and several workers that subscribe to the producer and are required to receive the same set of publications once they join the overlay. Even if all workers submit their subscriptions at the same time to the overlay, they might see a different publication set for $\mathbb{P}^i(s)$. In order for the subscribers (i.e. workers) to see the same set of publications, they can all be assigned the same group ID with $s.timestamp = s_r$.

ID-based delivery guarantee: A subscriber requesting an *ID-based* delivery guarantee for subscription s specifies $\mathbb{P}^i(s)$ by providing a publication ID pid_P, for each matching publisher π. The set of publications received from π is $\mathbb{P}_\pi(s) = \{p \in \mathbb{P}_\pi : p \models s \wedge p.Id > pid_P\}$. Therefore, the subscriber includes a map $\mathcal{M}$ which maps a publisher π to pid_P. If $\mathcal{M}$ does not specify pid_P for a publisher P, $\mathbb{P}_\pi(s) = \{p \in \mathbb{P}_\pi : p \models s \wedge p_r \geq s_p^B\}$ where B is the *EB* of publisher π. Alternatively, we can use a time-based guarantee as default and consider s_r as the activation time of s *w.r.t.* a publisher that has no entry in the map.

The ID-based guarantee provides a similar mechanism to the log offsets in Kafka [136], where any topic is considered an append-only sequence of publications ordered by time. As an example, in Figure 1.3.1, if the dispatcher decides to create a new worker in order to handle some part of the tasks published from the client, the new worker can submit a subscription to receive tasks from a client starting with a specific number.

Some pub/sub systems support advertisement covering in order to reduce the size of the SRT, a similar approach to subscription covering [36]. In this work, we do not support advertisement covering. We require each advertisement to be known by all brokers in the overlay and each advertisement can be identified by a unique ID on each broker. Furthermore, in order to be able to distinguish among two publication streams coming from two publishers with covering advertisements, each publication must include its matching advertisement ID. Alternatively, we can use advertisement covering and assume all matching publishers of a superset advertisement as one stream of publications.

7.4 Routing Algorithms

In this section, we present algorithms for routing messages in a distributed content-based pub/sub system that supports *ack-based* message installation and the delivery guarantees presented in Section 7.3. These algorithms are an extension of the reverse-path-forwarding-based routing algorithms for content-based pub/sub [14]. In our routing algorithms, we also provide support for subscription covering.

SRT	subscription routing table
PRT	publication routing table
$\mathbb{N}_B$	overlay neighbors of B
ACKS	table of received acks per msg and link
Groups	groups known to B
IsActive	subscription activation status per advertisement
StoredPubs	stored publications per publisher

Table 7.4.1: Data structures on broker B

Table 7.4.1 shows the data structures that each broker B must maintain. Besides the routing tables PRT and SRT, and the neighborhood $\mathbb{N}_B$, brokers maintain four additional data structures. ACKS is the table of received acknowledgments per link for each message. For example, in Figure 7.4.1, while an advertisement a with a requested ack coming from *P1* is being flooded in the overlay, *B2* maintains an entry for a in the ACKS table with two pending ack entries, one per each outgoing link, *i.e.*, *B2–B4* and *B2–B5*. In order to keep track of the existing subscriber groups in the overlay, brokers gather group information from subscriptions they process. For each group $g \in$ Groups, broker B keeps one timestamp. Therefore, each publisher *EB* knows of only one timestamp (*i.e.* the first one to reach the broker) and uses that for all subscriptions of the same group.

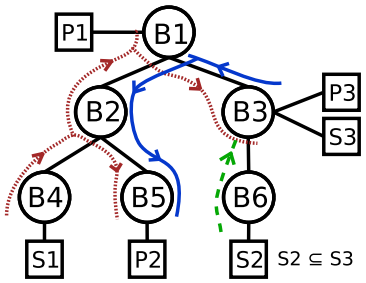


Figure 7.4.1: Subscription installation paths with subscription covering

While subscriptions are being installed in the overlay, we must prevent unwanted publication deliveries that can happen due to overlapping paths of two similar

subscriptions. Therefore, for each subscription s, the EB of s, maintains a flag whether s is installed in the overlay. An *ack-based* subscription s is considered active when it is installed with respect to all of its matching advertisements. However, a *ref-based* subscription s' is activated independently for each matching advertisement. The reason is that an *ack-based* delivery guarantee delivers publications to a subscription s (*i.e.* considers s active) only when it is installed in the overlay *w.r.t.* all matching advertisements. Therefore, IsActive contains one flag for each ack-based subscription and a list of flags (one per matching advertisement) for each *ref-based* subscription. In the latter case, each flag indicates whether s is installed *w.r.t.* a specific publisher identified by its advertisement a. IsActive is maintained only on EB of s. In order to improve readability of the algorithms, reading subscription status at broker $B \neq EB(s)$ always returns true and updating the subscription status of s at $B \neq EB(s)$ is ignored. Furthermore, EB of publisher π maintains a buffer of the recent publications of π.

7.4.1 Handling Advertisements

In Algorithm 10, upon receiving advertisement a, after updating SRT and forwarding a, if an ack is requested for a, broker B sets up a timer waiting for acknowledgments. If B is an edge broker receiving a, Line 5, it sends back an acknowledgment. In order to reduce the number of ack messages in the overlay, rather than sending all acknowledgments for a message m back to $EB(m)$, we aggregate acknowledgments on each intermediate broker, as shown in Algorithm 11. Upon receiving all acknowledgments for an ack-based subscription s, EB(s) marks s as active. In case an acknowledgment does not arrive on time, a timeout occurs and a negative acknowledgment is sent back to inform the subscriber. As an acknowledgment is routed similar to other message types in the pub/sub system according to $\mathbb{N}_B$, SRT or PRT, failure can be addressed using existing works on reliable message delivery [20, 59] which is not in the scope of this work.

Algorithm 10: Handling advertisements on Broker B

```
1  Function ReceiveAdv(a)
2      SRT [a] ← a.lastHop
3      nextHops ← 𝕹_B − a.lastHop
4      if a has ack request then
5          if nextHops = ∅ then // reached an edge broker
6              send(ack_a, a.lastHop)
7          else
8              ACKS [a][∀ l ∈ nextHops] = false
9              StartAckTimer(a)
10     send(a, nextHops) // flood to next hops
```

Algorithm 11: Handling ACKs on Broker B

```
1  Function ReceiveAck(ack)
2      if waiting for ack has not timed out then
3          ACKS [ack.msg][ack.lastHop] ← true
4          if received all pending acks for ACKS [ack.msg] then
5              if ack is for an ack-based s and B = EB(s) then
6                  IsActive [s] ← true
7              send(ack, ack.msg.lastHop)
```

7.4.2 Handling Subscriptions

Subscriptions are routed based on Algorithm 12. Upon receiving subscription s, $EB(s)$ timestamps s in order to record the time that s entered the overlay. This timestamp is used for time-based subscriptions which do not explicitly define $s.timestamp$. Next, after updating the PRT and finding next hops to route s to, Broker B searches for a covering subscription s' for s. If subscription s is *ack-based*, it is marked *not active* until s is installed in the overlay and an ack message is received. On any given subtree T rooted at Broker B, the number of acks that B must wait for before aggregating the acks for the subtree T is determined based on the SRT. Similar to Algorithm 10, starting with Line 14, Broker B sets up pending ack flags and timers and forwards s. On the subtree T, s is considered installed when either of the following two conditions are true (Line 10):

Algorithm 12: Handling subscriptions on Broker B

```
1  Function ReceiveSub(s)
2      timestamp s // only if B = EB(s)
3      advs ← SRT .findMatch(s)
4      nextHops ← {a.lastHop : ∀ a ∈ advs}
5      coverSub ← s′ ∈ PRT : s ⊆ s′ // find covering sub
6      update PRT with s
7      switch guarantee type of s do
8          case ack-based do
9              IsActive [s] ← false // only if B = EB(s)
10             if found active coverSub or nextHops = ∅ then
11                 IsActive [s] ← true // only if B = EB(s)
12                 send(ack_s, s.lastHop)
13             else
14                 ACKS [s][∀ s ∈ nextHops] ← false
15                 StartAckTimer(s)
16                 send(s, nextHops)
17         case ID-based or time-based do
18             UpdateGroups(s) // see Algorithm 13
19             groupTS ← getTimestamp( s)
20             IsActive [s][a ∈ advs] ← false // only if B = EB(s)
21             for a ∈ advs do
22                 if B = EB(a) then
23                     pubs ← GetMatchingPubs( s, groupTS, a)
24                     bp = new BatchPub(pubs, a, s)
25                     send( bp, s.lastHop) // ReceiveBatchPub(bp) if B = EB(s)
26             send(s, nextHops)
```

1. There are no more brokers left in T to install s.

2. There exists a covering subscription s' on B which is active.

In the second case, when s reaches B, existence of an active s' on B guarantees delivery of any matching publication published in the subtree T to B and therefore s can be safely considered active on T. For example, in Figure 7.4.1, upon receiving *S2*, *B3* can issue ack_{S2} without further forwarding *S2*.

If s is *ref-based*, after updating group information known to B, the subscription is

Algorithm 13: Handling buffered publications and groups

```
1 Function GetMatchingPubs(s, timestamp, a)
      // return set of matching publications for s from
      // StoredPubs [a.publisherId], filtering based on
      // reference or timestamp if necessary
2 Function UpdateGroups(s)
3     if s.groupId is set and s.groupId ∉ Groups then
4         Groups [s.groupId] ← new Group(s.timestamp)
5 Function getTimestamp(s)
6     if s has no groupId then
7         return s.timestamp
8     return Groups [s.groupId].timestamp
```

marked inactive with respect to each matching advertisement a. This is only done on the *EB* of s. Subscription s is considered active *w.r.t.* a when *EB* of s receives an acknowledgment from *EB* of a. This acknowledgment is of type BatchPub, a special publication addressed only to s. BatchPub.*pubs* contains $\mathbb{P}^{\mathrm{i}}_{\pi}(s)$, a subset of matching publications that *EB*(a) has received during s_i, filtered based on $s.timestamp$ or publication IDs recorded in $\mathcal{M}[\pi]$, where $\pi = C(a)$.

Unlike *ack-based* subscriptions, *ref-based* subscriptions must be routed towards *EB* of any matching publisher, even if there exists an active covering subscription on the subtree leading to the publisher. This is necessary in order to fetch any matching publications that was published before the subscription arrives at the *EB* of the publisher. Without complete propagation of the subscriptions, in order to serve any subscription s arriving at broker B where a covering subscription s' already exists, B must buffer any publications matching s'. While this leads to lower cost for delivering $\mathbb{P}^{\mathrm{i}}_{\pi}(s)$, all brokers in the overlay must provide the same buffering capacity as the *EB* of the publisher, in order to be able to serve any potential subscription that might be covered by an existing active subscription on the broker. Therefore, in order to buffer publications only on *EB* of the publisher, we require subscriptions to be fully propagated in the overlay even in subtrees where there already exists a covering subscription. Although this prevents the covering relation to save the message overhead of subscription forwarding, it does not hinder the main advantage of subscription covering, namely smaller PRTs on brokers. The reason is that when

Algorithm 14: Handling batch publications on broker B

```
1 Function ReceiveBatchPub(bp)
2     if B = EB(bp.s) then
3         IsActive [bp.s][bp.a] ← true
4         for p ∈ bp.pubs do
5             send(p, PRT [bp.s].lastHop)
6     else if found coverSub then // s' ∈ PRT : s ⊆ s'
7         send(bp, coverSub.lastHop)
8     else
9         send(bp, PRT [bp.s].lastHop)
```

a covering subscription exists on the broker, a new PRT entry is not created.

BatchPub messages are processed according to Algorithm 14. The BatchPub message bp is routed back towards the subscription s that resulted in bp. Upon receiving bp by the $EB(s)$ (Line 3), s is marked as active *w.r.t.* the advertisement a matching bp. Furthermore, the included publications in bp are sent one by one to the subscriber.

While it is possible for EB of a matching publisher to directly send bp to the EB of s, doing so can violate the per publisher ordering guarantee. Therefore, we simply forward bp back to s through the overlay. Only in cases where $bp.pubs$ is empty, it is safe to send bp directly to the EB of s. Note that, in any case, bp must be received and processed by EB of s in order for s to be activated *w.r.t.* a. Furthermore, unlike *ack-based* subscriptions, we do not aggregate BatchPub messages.

7.4.3 Handling Publications

Publications are routed according to Algorithm 15. Since the EB of each publisher keeps a buffer of the recent publications published by the publisher, upon receiving a publication p by EB of publisher π, p is added to this buffer. Next, matching subscriptions are looked up from the PRT. For each matching subscription s, if s is *ack-based*, B forwards p to s if s is active. Since this status is maintained only on $EB(s)$, on other brokers, Line 9 is always true and therefore, p is forwarded towards s regardless of whether s is active or not. Similarly, if s is *ref-based*, only EB of s checks

Algorithm 15: Handling publications on broker B

```
1  Function ReceivePub(p)
2      if B = EB(p) then
3          update StoredPubs [p.publisherId] with p
4      subs ← PRT .findMatch(p)
5      nextHops ← {}
6      for s ∈ subs do
7          switch guarantee type of s do
8              case ack-based do
9                  if IsActive [s] = true then
10                     nextHops ← nextHops ∪ {s.lastHop}
11             case time-based or ID-based do
12                 if IsActive [s][p.a]
13                 and p satisfies timestamp/pid_P of s then
14                     nextHops ← nextHops ∪ {s.lastHop}
15     send(p, nextHops)
16     if ack requested for p then
17         if nextHops ∩ N_B = ∅ then // reached an edge broker
18             send(ack_p, p.lastHop)
19         else
20             ACKS [p][∀ l ∈ nextHops : l ∈ N_B ] ← false
21             StartAckTimer(p)
```

whether s is active *w.r.t.* a and whether p satisfies $s.timestamp$ or the publication ID mentioned in $\mathcal{M}$. If so, p gets forwarded towards s.

As a consequence of maintaining subscription status only on the *EB* of the subscription, matching publications for a subscription s are received by the *EB* of s and dropped if IsActive $[s]$ or IsActive $[s][a]$ is false. While this can increase the message overhead, it simplifies the approach as otherwise each broker located on the path between s and any matching publisher must maintain the status of s. These dropped messages are incurred only when s is covered by another subscription s' which attracts publications while s is being propagated towards the publisher, as otherwise, s has no PRT entry on the publisher's *EB*. For example, in Figure 7.4.1, after *S2* is processed by *B3*, until bp_{S2} activates *S2*, all matching publications that are sent to *B6* because of *S3* (which covers *S2*), are also sent to *B6* and dropped there. Therefore, for a pub/sub system with reasonable subscriber churn, this message

overhead is small and can be ignored. Alternatively, we can maintain status of s on each broker on the path and prevent forwarding of messages at the covering broker.

The number of publications that gets buffered on EB of the publishers is a configuration parameter. This *publication history size* parameter defines how long a publication p is buffered in StoredPubs on $EB(p)$. While providing an *ack-based* delivery guarantee does not require buffering publications, in order to provide *ref-based* guarantees where $s.timestamp = s_r$, the publication history size must be large enough to buffer a publication for at least 2τ, where τ is the time that it takes for a message m to traverse the diameter of the overlay. *Publication history size* is similar to the log size of systems such as Kafka and can be configured depending on the application deployed on top of the pub/sub system.

Unadvertisement and unsubscription messages which have an ack requested are handled similar to the previous algorithms by maintaining and aggregating acknowledgment flags. Furthermore, any unadvertisement ua and unsubscription us is effective as soon as it reached the EB of the issuing client. For an unsubscription us, this means any publication arriving at $EB(us)$ after us_r is not delivered to $C(us)$. For an unadvertisement ua, any publication p arriving at $EB(p)$, such that $p_r > ua_r$ and $C(p) = C(ua)$, is ignored.

7.4.4 Avoiding Duplicate/Missed Publications

In the *ack-based* delivery guarantee, the EB of the subscriber s makes sure that no publication is delivered to s before receiving ack_s. After receiving ack_s, s is installed and active and the pub/sub system delivers any publication that reaches $EB(s)$ after the ack_s to $C(s)$. However, for *ref-based* subscriptions, there can be scenarios which leads to duplicate or missing publications. For example, in Figure 7.4.2, there are three publishers matching s (*ref-based*) on B. The highlighted publications of each publisher, are the ones already processed and buffered on B. The first publication not highlighted, is the next publication to be processed from the publisher, namely $p1.3$, $p2.2$ and $p3.3$ for *P1*, *P2* and *P3* respectively. If B processes messages it receives without any specific order, one of the following two cases can happen:

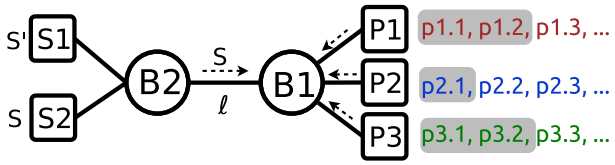


Figure 7.4.2: Processing order on broker $B1$

While s is being processed on $B1$ and before bp_s is sent back, it is possible that a publication is sent on ℓ after adding s to the PRT but before bp_s is sent. Since bp_s is not sent back and s is not active, this can lead to missing publications on $C(s)$.

Alternatively, to avoid missing publications, we can mark publications received by B while s is processed and sent them to s after bp_s. However, if meanwhile they have been forwarded to another matching subscription s' that covers s (similar to the example in Figure 7.4.2), resending them can result in duplicate delivery to $C(s')$.

In order to guarantee that none of the above happens, each broker must provide the following processing order guarantee regarding the order according to which incoming messages are processed on the broker:

Processing Order Guarantee – Upon receiving subscription s from link ℓ on Broker B, B guarantees that for each matching publisher π connected to B, no publication matching s is sent out on ℓ before bp_s is sent out on ℓ.

Providing the proposed processing order guarantee on each broker B in the overlay guarantees that for each subscriprion s and matching publisher π, $\mathbb{P}_\pi(s)$ has no miss or duplicate publications.

Proof. The proof relies on the per publisher ordering guarantee provided by the pub/sub system. Upon receiving s, Broker B guarantees sending out bp_s to $C(s)$ before processing any publication from publisher π (with advertisement a) matching s. Any publication p where $p_r \leq s_p^B$ and $p \models s$ is included in bp_s. Furthermore, any publication p' such that $p' \models s$ and $p'_r > s_p^B$, is not processed before bp_s is sent. Therefore, B sends first bp_s (containing matching publications received by B before s) and then any publication arriving at B after s. Due to the per publisher

ordering guarantee provided by the system, all brokers first receive bp_s and following publications from π in the same order sent by B. On each broker, bp_s activates s *w.r.t.* a and therefore following publications are routed towards $C(s)$. $EB(s)$ forwards the publications in $bp_s.pubs$ and the publications following bp_s to $C(s)$ and therefore, $C(s)$ receives the matching publications from π without any misses or duplicates. □

Relying on the processing order guarantee simplifies the algorithms and makes it unnecessary to keep publication IDs on brokers in order to prevent duplicates. One simple approach to provide the processing order guarantee is to provide mutual exclusion between Algorithm 12 and Algorithm 15. Alternatively, in order to reduce contention, more fine-grained locks (e.g. lock per entry in SRT) can be used.

7.5 Evaluation

In this section, we evaluate our approach (PSDG) which we have implemented in Java on the PADRES pub/sub system [13]. We use a cluster of 20 Dell PowerEdge R430 servers for our evaluation where each broker is located on one VM and the latency between brokers is enforced using the Linux *tc* command to configure the Linux kernel packet scheduler.

As a baseline, we use an existing pub/sub system that does not specify the delivery guarantees during message installation. Instead, clients (*i.e.* publishers and subscribers) are themselves responsible for specifying what happens while their messages are propagated. Therefore, in the baseline, the ack-based installation and delivery guarantees provided by PSDG are approximated by clients as follows:

- Ack-based installation and ack-based guarantee for subscriptions are emulated on clients by using timeouts. As an example, after advertising, a publisher waits for t seconds and after this time it assumes the advertisement has been completely installed in the overlay. However, there is on guarantee that the installation has been completed successfully.

- Time-based and ID-based guarantees for subscriptions are achieved on clients by simply discarding publications that do not satisfy the ID or timestamp constraints of the subscription.

Note that the baseline cannot guarantee publication delivery if subscription s requests a ref-based delivery guarantee that matches publication p such that $p_p < s_r$. Therefore, in these cases the baseline provides only the publications that are available in the overlay.

7.5.1 Workload

Figure 7.5.1 shows the topologies that we use in the evaluation. Each topology is represented with a dashed rectangle. The three overlays, *ov1, ov2, ov3* consist of 3, 6 and 10 brokers, respectively. Furthermore, the round-trip latencies between each two neighboring brokers are given in milliseconds. We have selected the nodes based on the available regions in Microsoft Azure [137] and the latencies are based on ping results available from WonderNetwork [138].

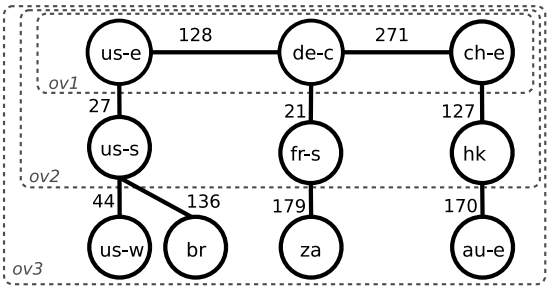


Figure 7.5.1: Evaluation topologies

We evaluate PSDG and the baseline using a workload with the following parameters: We use 10 publishers connected to one of the three brokers *us-e*, *de-c* and *ch-e*. In other words, publishers can publish from one of these three regions. Each publisher publishes 10 publications per second for 5 minutes. The content space consists of 5 different classes of publications, each with two attributes. Each attribute can have a value between 0 and 1000. Each publisher generates publications based on uniform

distribution from the content space. Subscribers are distributed among all brokers in the overlay based on a random uniform distribution. Each subscriber subscribes to one class of publications with the popularity of classes based on a Zipf distribution with $a = 1$ and the attribute range of each subscription based on a random uniform distribution. In each experiment, initially, we have 20 subscribers in the overlay. The rest of the subscribers join the overlay at a rate of 5 *sub/sec* and submit their subscriptions. All subscribers join the overlay over a period of 30 seconds; after this period no new subscriber joins the overlay. We study the impact of two variables on our metrics. Overlay size is the number of brokers in the overlay (3, 6 and 10 brokers). Furthermore, we study the impact of the number of subscribers that join the overlay each second (5, 10, and 15 *sub/sec*). The evaluation consists of three experiments. In the following, for each experiment, we explain the purpose of the experiment, the metrics we collect and the results.

7.5.2 Ack-based Delivery Guarantee

In order to evaluate the impact of the installation time on clients, we use a workload where all subscriptions are ack-based and all advertisements and 10% of the publications request an acknowledgment. Publication history size in PSDG is set to zero and each broker waits at most 10 minutes for an acknowledgment. We control the installation time by increasing the size of the overlay while measuring the following metrics:

Ack-time is the time it takes for a publication p to be completely propagated in the overlay and its acknowledgment, ack_p to be delivered to the publisher of p. We report the average ack-time across all publishers in the overlay.

While a publication is being propagated in the overlay, the publisher P must temporarily buffer the publication in order to be able to retransmit the publication if necessary. Only after receiving ack_p, P is able to dismiss the publication p. *Publications in memory* shows the number of publications that a publisher has to keep in memory while they are being propagated in the overlay. For each publisher P, we measure the number of un-acked publications that P keeps track of each

second. We take average of all collected values for all of the publishers in the system. For the baseline, where a timeout is used instead of an acknowledgment, we set the timeout based on the maximum acknowledgment time measured by PSDG for the same overlay rounded up to the next second. For example, if the maximum timeout for an overlay of size 3 is 800 milliseconds, we set the timeout for the baseline to 1 second.

False positives are the publications that match a subscription s but are received by s before s is fully installed in the overlay, $\mathbb{P}_{fpos}(s) = \{p \in \mathbb{P} \mid p_p^{EB(s)} < s_p\}$. While these false positives can happen in the baseline, PSDG prevents such deliveries. We record the number of false positive publication deliveries for each client and report the average.

Figure 7.5.2a shows the average time it takes for a publication p to be propagated in the overlay, and the acknowledgments of the *EB* of matching subscribers to be propagated back, aggregated and delivered to the publisher. As the overlay size grows, the average path length connecting the publishers and subscribers increases which results in the ack-time to increase by up to 54% when we increase the overlay size from 3 to 10. Accordingly, as the average ack-time increases, the number of publications that the publisher must keep track of increases (Figure 7.5.2b). In comparison, a publisher using timeouts to keep track of delivered publications, needs to buffer up to 10 times more publications. Note that, the buffering requirement of the timeout approach depends on the maximum ack-time and publication rate of the publisher which can result in keeping a large number of publications in comparison to an ack-based approach that removes a publication as soon as the ack is received. Furthermore, while here we have used the maximum ack-time measured in the overlay, in practice, there is no upper bound on the maximum ack-time. Therefore, the actual timeout used needs to be much higher in order to account for worst case scenarios. Consequently, this results in higher buffering requirements in the timeout approach.

PSDG prevents publication delivery to ack-based subscriptions before they are fully installed. This means for an ack-based subscription s, $\mathbb{P}^i(s) = \varnothing$. However, in the baseline, $\mathbb{P}^i(s)$ depends on existing subscriptions and the overlay. Figure 7.5.2c shows that as the overlay size increases, up to 40% of the clients receive false positives.

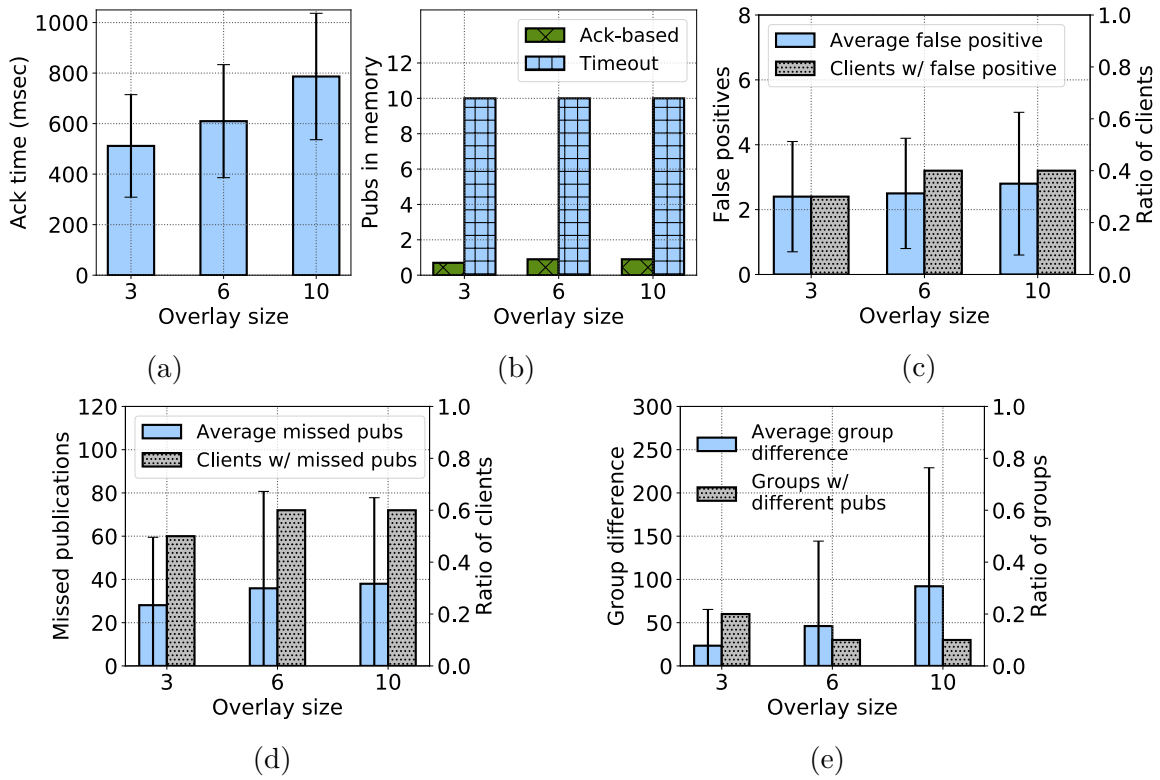


Figure 7.5.2: Impact of overlay size on PSDG metrics

Furthermore, the average number of false positives received by each client also increases up to 17%. This is due to the longer paths that connect publishers and subscribers as the overlay size grows. These longer paths result in a larger probability for a propagating publication to cross a subscription path while it is being installed.

This experiment shows that providing an ack-based message installation can reduce the buffering requirements of propagating publications by up to 10 times. Furthermore, the ack-based delivery guarantee can prevent any publication delivery before the subscription is installed in the overlay.

7.5.3 Ref-Based Delivery Guarantee

In this experiment, we evaluate the impact of the installation period on subscribers in terms of matching publications they miss. Furthermore, we show that even for

identical subscriptions connected to different *EB*s in the overlay, the set of received publications during installation time can be different. All subscriptions are of the ref-based type and are chosen based on a random uniform distribution among time-based and ID-based delivery guarantees. For time-based subscription s, $s.timestamp = s_r$, which means s is considered active as soon as it is received by its *EB*. For each ID-based subscription s' we have: $\mathcal{M}[P] = s'_g * P_{pub\_rate}, \forall a \in \mathbb{A}, s' \models a$ where P is the publisher with advertisement a, and s'_g is the time that s' is generated on the client. In other words, s' chooses a pid_P that happens in the same second as s' is being sent to its *EB*. Furthermore, each subscription s has an identical copy s', where s and s' belong to the same *sgroup* if they are time-based, or use the same $\mathcal{M}$ if they are ID-based. However, the *EB*s of s and s' are chosen randomly and therefore they may end up on different parts of the overlay. Publication history size in PSDG is 2 minutes and each broker waits at most 10 minutes for an acknowledgment. Similar to the previous experiment, we control the installation time by increasing the size of the overlay while measuring the following metrics:

Missed publications are the set of publications that match subscription s, however, due to the propagation delay of s, the subscriber misses these publications in the baseline, while the time-based and ID-based delivery guarantees allow s to clearly define the set of publications it receives during propagation. We count the number of publications missed by a client in the baseline and report the average of all the collected values.

Group difference is the number of publications that are different between two identical subscriptions s and s' submitted to the overlay at the same time. Therefore, if $A = \mathbb{P}^i(s)$ and $B = \mathbb{P}^i(s')$, the group difference is defined as $(A - B) \cup (B - A)$. This metric shows the difference

Figure 7.5.2d shows that in an overlay of 3 brokers, up to 50% of the subscribers do not receive matching publications that are published before their subscriptions are fully installed in the overlay. The set of publications missed by subscription s is a subset of $\{p \in \mathbb{P} \mid s_r < p_r < s_p^{EB(p)}\}$. Furthermore, due to the increase in the installation time in larger overlays, increasing the overlay size can cause up to 60% of subscribers missing matching publications. The average number of missed

publications also increases from 28 to 38 publications as the overlay size grows. Note that, increasing the number of publishers and the publication rate of publishers can increase the average number of publications missed by clients.

Figure 7.5.2e shows that 10% to 20% of the subscription groups receive different sets of publications and the group difference increases by up to 3 times, from 23 to 92, when increasing the size of the overlay from 3 to 10. The reason is that as the average path size grows, the installation time and consequently the size of $\mathbb{P}^i(s)$ and $\mathbb{P}^i(s')$ also increase. This can lead to larger group differences for subscriptions located on different *EB*s.

This experiment shows that in the baseline, even in small overlays, the installation time influences the set of publications received by subscribers while PSDG allows subscribers to clearly define the set of publications they receive during their installation time. Furthermore, while PSDG makes sure ref-based subscriptions belonging to the same *sgroup* receive the same set of publications, in the baseline, the group difference can be on average up to 92 publications.

7.5.4 Performance Evaluation

In this experiment, we compare PSDG and the baseline in terms of scalability and resources required to route messages and deliver publications. Furthermore, we study the impact of the new routing algorithms on the brokers and the clients. We use a mixed workload where 50% of advertisements and 10% of publications request an ack-based installation and each subscription chooses one of the three delivery guarantees based on a random uniform distribution. Ref-based subscriptions use a timestamp or publication ID that is chosen randomly from 10 seconds before to 10 seconds after subscription time s_g. Publication history size in PSDG is 2 minutes and each broker waits at most 10 minutes for an acknowledgment. In order to increase the load on the overlay, we keep the overlay size fixed at 6 but increase the subscription rate from 5 to 15 subscriptions per second. We measure the following metrics:

CPU load is the system load on each broker obtained from the *top* utility of Linux.

We normalize the value by dividing the output of the 1 minute system load by the number of cores. Therefore, a CPU load of 1 means that all cores are utilized 100% and no process is waiting for CPU. For each broker, we measure the average CPU load and report the average CPU load of all brokers in the overlay.

Memory usage is calculated by measuring the maximum memory usage on each broker during the experiment, and taking the average of all maximum memory usages collected.

Sent messages is the total number of messages a broker sends during the experiment. We report the average across all brokers in the overlay.

Throughput is the the number of publications processed by a broker per second. On each broker, we measure the number of publications processed per second. We calculate the average throughput for each broker and report the average throughput across all brokers in the overlay. This metric helps us study whether providing the delivery guarantees impacts broker performance or not.

In order to study the impact that providing the processing order guarantee on brokers might have on clients, we use the 99^{th} percentile of *Publication delivery latency* (PDL) at each client. We calculate the 99^{th} percentile of PDL at each client and take the average of this value across all clients.

As seen in Figure 7.5.3a and 7.5.3b, PSDG and baseline have similar CPU load and memory usage. The memory usage increases for both approaches as the number of subscriptions increases. This is due to the higher number of publications that need to be routed when there are more subscribers in the overlay.

Figure 7.5.3c shows the total number of sent messages required to deliver all publications in each approach. Although 50% of advertisements, 10% of publications and 33% of subscriptions request an ack-based installation, the number of messages sent by PSDG is only 2 to 3% higher than the baseline. This is due to aggregating acknowledgments in PSDG which reduces the number of messages required to deliver acknowledgments. Increasing the number of subscribers decreases the message overhead of PSDG in comparison to the baseline. This is due to the increase in the

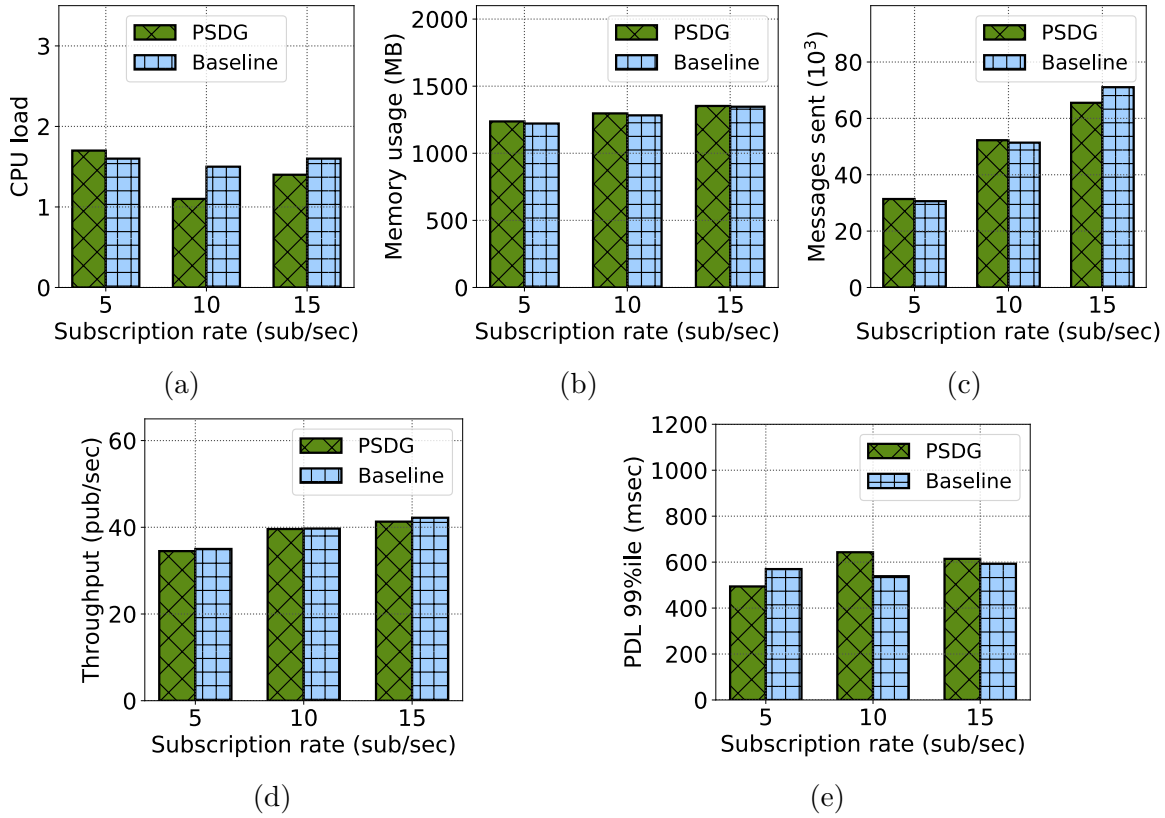


Figure 7.5.3: Impact of subscription rate on PSDG metrics

number of subscriptions that receive false positives which is avoided in PSDG.

Figure 7.5.3d shows that the routing algorithms in PSDG do not hinder the throughput of the brokers. The throughput for both approaches increases as the number of subscribers and consequently the number of publications propagating in the overlay increases. Furthermore, clients experience similar PDLs in both baseline and PSDG (Figure 7.5.3e).

This experiment shows that PSDG is able to provide the delivery guarantees while keeping a similar CPU, memory and network requirement to the baseline and without impacting broker throughput or PDLs experienced by clients.

Chapter 8

Conclusions

8.1 Summary

In this work, we presented four approaches to address four non-functional requirements of a distributed content-based pub/sub system in order to facilitate adoption of such systems for providing a dependable pub/sub service suitable for distributed application development.

Firstly, we presented GEPS, a low-overhead gossip-based mechanism that increases publication delivery ratio in case of broker failure without increasing delivery latency. With this approach, we aimed at maintaining fault-resilience and scalability of epidemic algorithms while minimizing their inherent redundancy. To this end, we used a similarity metric and partial view of the overlay to forward publications past failed brokers. GEPS provides a configurable approach to tune the message overhead and fault tolerance trade-off. Using GEPS, a pub/sub service can reduce the impact of broker failures on publication delivery and tolerate failure of large subsets of the overlay.

The results of our evaluation based on synthesized and real-world traces confirms that GEPS improves the delivery ratio of the underlying dissemination tree up to

29%. Furthermore, we have shown that GEPS can provide higher delivery ratio than the evaluated topology-based approach when failure rate is high (30% or more) or the failures follow a non-uniform distribution. By comparing similarity-based and random gossip target selection, we confirm the effectiveness of the similarity metric in reducing the gossiping message overhead up to 70%. Compared to a topology-based approach, GEPS results in 29% lower view maintenance cost.

Secondly, we presented PopSub, our approach to increase resource utilization of a pub/sub system by prioritizing publications based on their popularity and handling less popular publications with approaches that require fewer resources. The three proposed approaches for handling unpopular publications provide different tradeoffs and are suitable for different scenarios. Furthermore, by identifying unpopular publications and using the proposed alternative dissemination approaches only for such publications, PopSub is able to maintain the same publication delivery latency for a majority of publications while increasing resource utilization of the system. Using PopSub's popularity-based routing scheme, a pub/sub service can improve its resource utilization, and consequently, reduce its operational costs.

The result of our evaluations, using real-world workloads and traces, confirms that PopSub is able to improve resource efficiency of the system by up to 62%, reduce unnecessary publication forwards by up to 59%, and reduce popular publication delivery latencies by up to 57% in an overloaded pub/sub system. Additionally, these improvements can be achieved with workloads following any distribution of subscriptions on the advertisements.

Thirdly, we presented IPITT, an integer-programming-based approach to incremental topology transformation of pub/sub systems. Using an IP-based formulation, IPITT is able to find transformation plans with a minimal number of steps and minimize service disruption. Furthermore, IPITT facilitates integration of new constraints and cost models to customize the generated plans. Using IPITT, a pub/sub service can continuously evaluate and adapt its overlay to the existing usage patterns and minimize the impact of performing overlay changes on the service users.

Compared to existing solutions, IPITT, with our proposed optimization heuristics, reduces planning time by a factor of 10 while producing plans that have up to 45%

fewer actions and can be executed up to 55% faster. Furthermore, we showed that IPITT minimizes disruption to clients by producing plans with higher quality.

Lastly, we presented PSDG, a collection of message installation and delivery guarantees for a distributed content-based pub/sub system. Using PSDG, a client can request an acknowledgment for any message it submits to the overlay in order to be notified when its message is installed. Furthermore, the proposed *ack-based*, *time-based* and *ID-based* delivery guarantees allow subscribers to clearly define the set of publications they receive during and after the propagation of their subscription in the overlay. Using PSDG, a pub/sub service can ensure clear delivery guarantees to its clients regardless of the size and latency of the overlay.

Our evaluation suggests that, without the proposed set of delivery guarantees, an overlay-based pub/sub system can result in missing or false positive publications due to the message propagation delay. Furthermore, by introducing the notion of subscription groups, a group of subscribers can request to receive the exact same set of publications and thereby avoid out-of-band communication or synchronization.

8.2 Future Work

The GEPS protocol as explained in Chapter 4 provides a best-effort delivery guarantee. However, the protocol can be extended to incorporate message retransmission. While we use push messages to propagate publications during failures, a combination of publication digest exchange and pull messages can be used to achieve 100% delivery ratio. Another direction for extending GEPS is providing more sophisticated similarity metrics to improve delivery ratio and lower message overhead.

IPITT can produce topology transformation plans consisting of different overlay reconfiguration operators. As the current planner aims to reduce the number of actions required to perform the transformation, it prefers larger modifications such as *move* over *shift*. This means in conditions where for example, one *move* has the same effect as three *shift*s, the planner will choose *move*. However, depending on link latency and overlay load, performing the reconfiguration using a higher number of *shift*s might result in lower service disruption. Future work can consider finding run-time performance metrics that can be used to decide between *move* and *shift* actions. Furthermore, the planner can be extended to allows customization of the plan search based on these run-time metrics.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] T. Schlossnagle, "Distributed systems, like it or not," (Dublin), USENIX Association, 2017.

[2] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 318–325, IEEE, Dec 2016.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[4] C. Ueland, "A 360 degree view of the entire netflix stack." `http://highscalability.com/blog/2015/11/9/a-360-degree-view-of-the-entire-netflix-stack.html`. Accessed: 2018-04-26.

[5] Y. Liu and B. Plale, "Survey of publish subscribe event systems," Tech. Rep. TR574, Computer Science Dept., Indiana University, Bloomington, IN 47405-7104, 2003. ftp://www.cs.indiana.edu/pub/techreports/TR574.pdf.

[6] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, "Revisiting IP multicast," in *ACM SIGCOMM Computer Communication Review*, vol. 36, pp. 15–26, ACM, 2006.

[7] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the IP multicast service and architecture," *IEEE Network*, vol. 14, pp. 78–88, Jan 2000.

[8] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, *Scalable Application Layer Multicast*. SIGCOMM '02, New York, NY, USA: ACM, 2002.

[9] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas, "A survey of application-layer multicast protocols," *Commun. Surveys Tuts.*, vol. 9, pp. 58–74, July 2007.

[10] Google LLC, "Cloud pub/sub: A Google-scale messaging service." `https://cloud.google.com/pubsub/architecture`, 2018. Accessed: 2018-05-01.

[11] Amazon.com, Inc., "Amazon simple notification service (SNS)." `https://aws.amazon.com/sns/`, 2018. Accessed: 2018-05-19.

[12] Microsoft Corporation, "Azure event grid." `https://docs.microsoft.com/en-us/azure/event-grid/`, 2018. Accessed: 2018-05-19.

[13] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski, "The PADRES distributed publish/subscribe system," in *FIW*, pp. 12–30, 2005.

[14] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Trans. Comput. Syst.*, vol. 19, pp. 332–383, Aug. 2001.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[16] "What high availability for cloud services means in the real world." `https://www.techrepublic.com/blog/the-enterprise-cloud/what-high-availability-for-cloud-services-means-in-the-real-world/`, 2011. Accessed: 2018-03-18.

[17] L. Columbus, "Roundup of cloud computing forecasts." `https://www.forbes.com/sites/louiscolumbus/2017/04/29/`

roundup-of-cloud-computing-forecasts-2017/, 2017. Accessed: 2018-03-18.

[18] M. Gagnaire, F. Diaz, C. Coti, C. Cerin, K. Shiozaki, Y. Xu, P. Delort, J.-P. Smets, J. Le Lous, S. Lubiarz, and P. Leclerc, "Downtime statistics of current cloud solutions," tech. rep., International Working Group on Cloud Computing Resiliency, 2014.

[19] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel, "PLEROMA: A SDN-based high performance publish/subscribe middleware," in *Proceedings of the 15th International Middleware Conference*, Middleware '14, (New York, NY, USA), pp. 217–228, ACM, 2014.

[20] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, (Washington, DC, USA), pp. 41–50, IEEE Computer Society, 2009.

[21] C. Esposito, D. Cotroneo, and A. Gokhale, "Reliable publish/subscribe middleware for time-sensitive internet-scale applications," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, (New York, NY, USA), pp. 16:1–16:12, ACM, 2009.

[22] A. Avizienis, J.-C. Laprie, and B. Randell, *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science Newcastle upon Tyne, UK, 2001.

[23] I. Ben-Shaul, H. Gazit, O. Holder, and B. Lavva, "Dynamic self adaptation in distributed systems," in *Self-Adaptive Software*, pp. 134–142, Springer Berlin Heidelberg, 2001.

[24] C. Kilcioglu, J. M. Rao, A. Kannan, and R. P. McAfee, "Usage patterns and the economics of the public cloud," in *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pp. 83–91, International World Wide Web Conferences Steering Committee, 2017.

[25] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito, "Modeling publish/subscribe communication systems: towards a formal approach," in

Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003)., pp. 304–311, Jan 2003.

[26] R. S. Kazemzadeh, *Overlay neighborhoods for distributed publish/subscribe systems*. PhD thesis, University of Toronto, 2012.

[27] G. Li and H.-A. Jacobsen, "Composite subscriptions in content-based publish/subscribe systems," in *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, (New York, NY, USA), pp. 249–269, Springer-Verlag New York, Inc., 2005.

[28] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827–850, Sept. 2001.

[29] G. Li, V. Muthusamy, and H.-A. Jacobsen, "Adaptive content-based routing in general overlay topologies," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, (New York, NY, USA), pp. 1–21, Springer-Verlag New York, Inc., 2008.

[30] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito, "Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA," *The Computer Journal*, vol. 50, pp. 444–459, July 2007.

[31] S. Bhola, "Topology changes in a reliable publish/subscribe system," Tech. Rep. RC23354, IBM Research, 2004.

[32] B. Porter, F. Taiani, and G. Coulson, "Generalised repair for overlay networks," in *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pp. 132–142, Oct 2006.

[33] Y. Yoon, V. Muthusamy, and H. A. Jacobsen, "Foundations for highly available content-based publish/subscribe overlays," in *2011 31st International Conference on Distributed Computing Systems*, pp. 800–811, June 2011.

[34] J. Liu and M. Zhou, "Tree-assisted gossiping for overlay video distribution," *Multimedia Tools and Applications*, vol. 29, pp. 211–232, Jun 2006.

[35] P. Costa and G. P. Picco, "Semi-probabilistic content-based publish-subscribe," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 575–585, June 2005.

[36] G. Li, S. Hou, and H. Jacobsen, "A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 447–457, June 2005.

[37] P. R. Pietzuch and S. Bhola, "Congestion control in a reliable scalable message-oriented middleware," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, (New York, NY, USA), pp. 202–221, Springer-Verlag New York, Inc., 2003.

[38] J. Reumann, "GooPS: Pub/sub at Google," *Lecture & Personal Communications at EuroSys & CANOE Summer School*, 2009.

[39] J. Eidson and K. Lee, "IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems," in *2nd ISA/IEEE Sensors for Industry Conference*, pp. 98–105, Nov 2002.

[40] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[41] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 335–350, USENIX Association, 2006.

[42] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, (Berkeley, CA, USA), pp. 251–264, USENIX Association, 2016.

[43] F. Rahimian, S. Girdzijauskas, A. H. Payberah, and S. Haridi, "Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous

routing in unstructured overlay networks," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 746–757, May 2011.

[44] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. V. Steen, "Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks," in *In IPTPS'06: the fifth International Workshop on Peer-to-Peer Systems*, 2006.

[45] A. Malekpour, F. Pedone, M. Allani, and B. Garbinato, "Streamline: An architecture for overlay multicast," in *2009 Eighth IEEE International Symposium on Network Computing and Applications*, pp. 44–51, July 2009.

[46] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris, "Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub," in *Middleware 2012: ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, pp. 271–291, Springer Berlin Heidelberg, 2012.

[47] C. Tang, R. N. Chang, and C. Ward, "GoCast: gossip-enhanced overlay multicast for fast and dependable group communication," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pp. 140–149, June 2005.

[48] A. M. Kermarrec, L. Massoulie, and A. J. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 248–258, March 2003.

[49] M. Petrovic, V. Muthusamy, and H. A. Jacobsen, "Content-based routing in mobile ad hoc networks," in *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pp. 45–55, July 2005.

[50] C. Chen, H. A. Jacobsen, and R. Vitenberg, "Algorithms based on divide and conquer for topic-based publish/subscribe overlay design," *IEEE/ACM Transactions on Networking*, vol. 24, pp. 422–436, Feb 2016.

[51] Y. Yoon, N. Robinson, V. Muthusamy, S. McIlraith, and H.-A. Jacobsen, "Planning the transformation of overlays," in *Proceedings of the 31st Annual*

ACM Symposium on Applied Computing, SAC '16, (New York, NY, USA), pp. 500–507, ACM, 2016.

[52] C. Chen, R. Vitenberg, and H.-A. Jacobsen, *A Generalized Algorithm for Publish/Subscribe Overlay Design and Its Fast Implementation*, pp. 76–90. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[53] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Constructing scalable overlays for pub-sub with many topics," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, (New York, NY, USA), pp. 109–118, ACM, 2007.

[54] E. D. Nitto, D. J. Dubois, and A. Margara, "Reconfiguration primitives for self-adapting overlays in distributed publish-subscribe systems," in *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 99–108, Sept 2012.

[55] H. Parzyjegla, G. G. Muhl, and M. A. Jaeger, "Reconfiguring publish/subscribe overlay topologies," in *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, pp. 29–29, July 2006.

[56] M. Kim, A. Mohindra, V. Muthusamy, R. Ranchal, V. Salapura, A. Slominski, and R. Khalaf, "Building scalable, secure, multi-tenant cloud services on IBM Bluemix," *IBM Journal of Research and Development*, vol. 60, pp. 8:1–8:12, March 2016.

[57] G. Cugola, M. Migliavacca, and A. Monguzzi, "On adding replies to publish-subscribe," in *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, (New York, NY, USA), pp. 128–138, ACM, 2007.

[58] J. C. Hill, J. C. Knight, A. M. Crickenberger, and R. Honhard, "Publish and subscribe with reply," Tech. Rep. CS-2002-32, University of Virginia, Charlottesville, VA, USA, 2002. http://www.dtic.mil/get-tr-doc/pdf?AD=ADA479195.

[59] P. Salehi, C. Doblander, and H.-A. Jacobsen, "Highly-available content-based publish/subscribe via gossiping," in *Proceedings of the 10th ACM International*

Conference on Distributed and Event-based Systems, DEBS '16, (New York, NY, USA), pp. 93–104, ACM, 2016.

[60] 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*, DEBS '17, (New York, NY, USA), pp. 88–99, ACM, 2017.

[61] P. Salehi, K. Zhang, and H. A. Jacobsen, "Incremental topology transformation for publish/subscribe systems using integer programming," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 80–91, June 2017.

[62] P. Salehi, K. Zhang, and H. A. Jacobsen, "On delivery guarantees in distributed content-based publish/subscribe systems." Submitted to the 19th ACM/IFIP/USENIX Middleware Conference, 2018.

[63] "Yahoo Pulsar pub/sub messaging platform." `https://github.com/yahoo/pulsar`. Accessed: 2016-12-05.

[64] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, pp. 1–7, 2011.

[65] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni, S. Kumar, H. Li, J. Li, E. Makeev, K. Prakasam, R. V. Renesse, S. Roy, P. Seth, Y. J. Song, B. Wester, K. Veeraraghavan, and P. Xie, "Wormhole: Reliable pub-sub to support geo-replicated internet services," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 351–366, USENIX Association, 2015.

[66] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "A peer-to-peer approach to content-based publish/subscribe," in *Proceedings of the 2Nd International Workshop on Distributed Event-based Systems*, DEBS '03, (New York, NY, USA), pp. 1–8, ACM, 2003.

[67] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, (New York, NY, USA), pp. 1–12, ACM, 1987.

[68] S. Birrer and F. E. Bustamante, "A comparison of resilient overlay multicast approaches," *IEEE Journal on Selected Areas in Communications*, vol. 25, pp. 1695–1705, Dec. 2007.

[69] A.-M. Kermarrec and M. van Steen, "Gossiping in distributed systems," *SIGOPS Operating Systems Review*, vol. 41, pp. 2–7, Oct. 2007.

[70] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, pp. 41–88, May 1999.

[71] M. Allani, J. Leitao, B. Garbinato, and L. Rodrigues, "RASM: A reliable algorithm for scalable multicast," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pp. 137–144, Feb 2010.

[72] M. Jelasity, A. Montresor, and O. Babaoglu, "T-Man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, pp. 2321–2339, Aug. 2009.

[73] J. Leitao, J. Marques, J. Pereira, and L. Rodrigues, "X-BOT: A protocol for resilient optimization of unstructured overlay networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, pp. 2175–2188, Nov 2012.

[74] R. Subramaniyan, P. Raman, A. D. George, and M. Radlinski, "GEMS: Gossip-enabled monitoring service for scalable heterogeneous distributed systems," *Cluster Computing*, vol. 9, pp. 101–120, Jan. 2006.

[75] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, (London, UK, UK), pp. 55–70, Springer-Verlag, 1998.

[76] M. Jelasity and A. M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, pp. 117–124, Sept 2006.

[77] I. Gupta, A. M. Kermarrec, and A. J. Ganesh, "Efficient and adaptive epidemic-style protocols for reliable and scalable multicast," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 593–605, July 2006.

[78] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, (New York, NY, USA), pp. 79–98, Springer-Verlag New York, Inc., 2004.

[79] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," *ACM Transactions on Computer Systems*, vol. 21, pp. 341–374, Nov. 2003.

[80] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[81] M. Matos, A. Nunes, R. Oliveira, and J. Pereira, "StAN: exploiting shared interests without disclosing them in gossip-based publish/subscribe," in *IPTPS'10 Proceedings of the 9th international conference on Peer-to-peer systems*, 2010.

[82] N. Tajuddin, B. Maniymaran, and H.-A. Jacobsen, "Minimal broker overlay design for content-based publish/subscribe systems," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, (Riverton, NJ, USA), pp. 25–39, IBM Corp., 2013.

[83] M. Onus and A. W. Richa, "Minimum maximum degree publish-subscribe overlay network design," in *IEEE INFOCOM 2009*, pp. 882–890, April 2009.

[84] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola, "Epidemic algorithms for reliable content-based publish-subscribe: an evaluation," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pp. 552–561, 2004.

[85] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, pp. 1489–1499, Oct 2002.

[86] 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*, DEBS '07, (New York, NY, USA), pp. 14–25, ACM, 2007.

[87] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "TERA: Topic-based event routing for peer-to-peer architectures," in *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, (New York, NY, USA), pp. 2–13, ACM, 2007.

[88] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based publish-subscribe over structured overlay networks," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 437–446, June 2005.

[89] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann, "Self-organizing broker topologies for publish/subscribe systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, (New York, NY, USA), pp. 543–550, ACM, 2007.

[90] K. Kim, Y. Zhao, and N. Venkatasubramanian, "GSFord: Towards a reliable geo-social notification system," in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 267–272, Oct 2012.

[91] K. R. Jayaram, C. Jayalath, and P. Eugster, "Parametric subscriptions for content-based publish/subscribe networks," in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, (Berlin, Heidelberg), pp. 128–147, Springer-Verlag, 2010.

[92] B. Elshqeirat, S. Soh, S. Rai, and M. Lazarescu, "Topology design with minimal cost subject to network reliability constraint," *IEEE Transactions on Reliability*, vol. 64, pp. 118–131, March 2015.

[93] Y. Zhao and J. Wu, "On the construction of the minimum cost content-based publish/subscribe overlays," in *2011 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pp. 476–484, June 2011.

[94] C. Chen, Y. Tock, H. A. Jacobsen, and R. Vitenberg, "Weighted overlay design for topic-based publish/subscribe systems on geo-distributed data centers," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, pp. 474–485, June 2015.

[95] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating data center networks with zero loss," in *Proceedings of the ACM SIGCOMM Conference*, (New York, NY, USA), pp. 411–422, ACM, 2013.

[96] S. Richter and M. Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks," *Journal Of Artificial Intelligence Research*, vol. 39, pp. 127–177, Sept. 2010.

[97] N. Lipovetzky and H. Geffner, "Searching for plans with carefully designed probes," in *Proceedings of the Twenty-First International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'11, pp. 154–161, AAAI Press, 2011.

[98] "Gurobi optimizer." `http://www.gurobi.com/products/gurobi-optimizer`. Accessed: 2016-11-08.

[99] "CPLEX optimizer." `http://www-01.ibm.com/software/commerce/opti\mization/cplex-optimizer/`. Accessed: 2016-11-08.

[100] "Gurobi compute server." `http://www.gurobi.com/products/gurobi-compute-server/gurobi-compute-server`. Accessed: 2016-11-08.

[101] R. S. Kazemzadeh and H.-A. Jacobsen, "Opportunistic multipath forwarding in content-based publish/subscribe overlays," in *Proceedings of the 13th International Middleware Conference*, Middleware '12, (New York, NY, USA), pp. 249–270, Springer-Verlag New York, Inc., 2012.

[102] C. Jayalath, J. J. Stephen, and P. Eugster, *Atmosphere: A Universal Cross-Cloud Communication Infrastructure*, pp. 163–182. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[103] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish-subscribe systems," *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014.

[104] N. Carvalho, F. Araujo, and L. Rodrigues, "Scalable QoS-based event routing in publish-subscribe systems," in *Fourth IEEE International Symposium on Network Computing and Applications*, pp. 101–108, July 2005.

[105] Z. Jerzak and C. Fetzer, "Handling overload in publish/subscribe systems," in *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, pp. 32–32, July 2006.

[106] X. Guo, H. Zhong, J. Wei, and D. Han, "A new approach for overload management in content-based publish/subscribe," in *International Conference on Software Engineering Advances (ICSEA 2007)*, pp. 32–32, Aug 2007.

[107] S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch, "Aggregation for implicit invocations," in *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, (New York, NY, USA), pp. 109–120, ACM, 2013.

[108] M. Jergler, K. Zhang, and H.-A. Jacobsen, "Multi-client Transactions in Distributed Publish/Subscribe Systems," tech. rep., Technical University of Munich, 2017.

[109] L. Vargas, L. I. W. Pesonen, E. Gudes, and J. Bacon, "Transactions in content-based publish/subscribe middleware," in *ICDCS Workshops '07*, 2007.

[110] G. Pardo-Castellote, "OMG data-distribution service: architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pp. 200–206, May 2003.

[111] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, pp. 427–469, Dec. 2001.

[112] R. Hunt, "Keeping time with Amazon Time Sync service." `https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service`, 2017. Accessed: 2018-03-05.

[113] Google LLC, "Google public NTP." `https://developers.google.com/time/`, 2018. Accessed: 2018-03-05.

[114] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, (New York, NY, USA), pp. 170–179, ACM, 2001.

[115] J. Wilkes, "More Google cluster data." `https://research.googleblog.com/2011/11/more-google-cluster-data.html`. Accessed: 2018-04-21.

[116] M.-J. Lin, K. Marzullo, and S. Masini, "Gossip versus deterministically constrained flooding on small networks," in *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, (London, UK, UK), pp. 253–267, Springer-Verlag, 2000.

[117] J. L. Martins and S. Duarte, "Routing algorithms for content-based publish/subscribe systems," *IEEE Communications Surveys Tutorials*, vol. 12, pp. 39–58, First 2010.

[118] M. E. Newman and J. Park, "Why social networks are different from other types of networks," *Physical Review E*, vol. 68, no. 3, 2003.

[119] H. Liu, V. Ramasubramanian, and E. G. Sirer, "Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2005.

[120] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), pp. 591–600, ACM, 2010.

[121] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[122] C. Lumezanu, D. Levin, and N. Spring, "PeerWise discovery and negotiation of faster paths," in *HotNets*, 2007.

[123] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: Estimating latency between arbitrary internet end hosts," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, (New York, NY, USA), pp. 5–18, ACM, 2002.

[124] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 539–547, Curran Associates Inc., 2012.

[125] Y. Yoon, *Adaptation Techniques for Publish/Subscribe Overlays*. PhD thesis, University of Toronto, 2013.

[126] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL - the planning domain definition language," 1998.

[127] J. N. Hooker, "A quantitative approach to logical inference," *Decision Support Systems*, vol. 4, pp. 45–69, Mar. 1988.

[128] M. H. L. Van den Briel, *Integer programming approaches for automated planning*. PhD thesis, Arizona State University, 2008.

[129] "IBM ILOG CPLEX enterprise server." `https://www-01.ibm.com/software/commerce/optimization/cplex-enterprise-server/`. Accessed: 2016-11-21.

[130] M. van den Briel and S. Kambhampati, "Optiplan: Unifying IP-based and graph-based planning," *Journal Of Artificial Intelligence Research*, vol. 24, pp. 919–931, 01 2005.

[131] J. Fan and M. H. Ammar, "Dynamic topology configuration in service overlay networks: A study of reconfiguration policies," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pp. 1–12, April 2006.

[132] V. Muthusamy, M. Petrovic, D. Gao, and H. A. Jacobsen, "Publisher mobility in distributed publish/subscribe systems," in *25th IEEE International Conference on Distributed Computing Systems Workshops*, pp. 421–427, June 2005.

[133] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler, "Supporting mobility in content-based publish/subscribe middleware," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, (New York, NY, USA), pp. 103–122, Springer-Verlag New York, Inc., 2003.

[134] J. Gondzio, "Warm start of the primal-dual method applied in the cutting-plane scheme," *Mathematical Programming*, vol. 83, pp. 125–143, Jan 1998.

[135] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei, "A scalable and elastic publish/subscribe service," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 1254–1265, May 2011.

[136] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide*. O'Reilly Media, Inc, 2016.

[137] Microsoft Corporation, "Azure regions." `https://azure.microsoft.com/en-us/regions/`, 2018. Accessed: 2018-01-15.

[138] WonderNetwork, "Global ping statistics." `https://wondernetwork.com/pings`, 2018. Accessed: 2018-01-15.