LANGUAGE EXPRESSIVENESS AND QUALITY OF SERVICE FOR CONTENT-BASED PUBLISH/SUBSCRIBE SYSTEMS

by

Kaiwen Zhang

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2016 by Kaiwen Zhang
Abstract

Language Expressiveness and Quality of Service for Content-based Publish/Subscribe Systems

Kaiwen Zhang

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2016

The publish/subscribe paradigm is known for its loosely coupled interactions and event filtering capabilities. Traditional applications using pub/sub systems require large-scale deployment and high event throughput. Thus, pub/sub has always put the emphasis on scalability and performance, to the detriment of filtering expressiveness and quality of service. The matching language is usually limited to topic-based or content-based event filtering and does not allow complex stream-based subscriptions to be expressed. Messages are delivered on a best-effort basis without any ordering or reliability guarantees. Recently, modern pub/sub applications such as online games, social networks, and sensor networks, have specifications which extend beyond the basic semantics provided by standard systems. Installing additional services and event processing systems at the endpoints can overcome these limitations. However, this thesis argues that such solutions are inefficient and put an avoidable strain on the pub/sub layer itself. Therefore, the focus of this thesis is to develop integrated solutions to extend pub/sub language expressiveness and quality of service, as well as demonstrate that this approach results in better performance from a holistic perspective. The different pub/sub extensions described are ranked data dissemination, fair subscription filtering, and total order. Each section first describes the application
use cases which justify the support for the developed feature in pub/sub. Those requirements are then extracted from our use cases and a general solution is developed within the pub/sub layer. A theoretical analysis is also conducted to demonstrate the correctness of every approach. Finally, experiments are employed to compare the performance of our solution to baselines which rely on end-to-end services and holistic evaluations are performed to assess the impact of our work.
Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Hans-Arno Jacobsen, for his guidance throughout this journey. He has provided me with the tools to quickly get started and feel comfortable delving into the research area. He provided me with numerous opportunities to connect with other peers and find motivation to pursue topics of interest. He has inspired me to continue working in academia.

The people at the Middleware Systems Research Labs have all helped me in one form or another. I’ve had the privilege to meet many interesting people that have enriched my life, academically and otherwise: Tilmann, Prasanth, Kianoosh, Chen, Mo, Reza, Young, Vinod, to name a few. MSRG is a tight-knit group, which makes it an enjoyable place to spend time daily.

Thank you to all the people I’ve met during my studies. Prof. Roman Vitenberg leads a fantastic research group at UiO which I had a pleasure to collaborate with. Thanks in particular to Navneet for being a good friend and collaborator. Thanks to Prof. Bettina Kemme and Prof. Jörg Kienzle for being wonderful persons to work with. May we continue working together for many years to come. Thanks to César for his effort in our joint work and just being a fun person to talk to. Thanks to Prof. Hein Meling for his kindness and advices. Thanks to Prof. Songlin Hu who welcomed me into his group during my stay in Beijing.

Lastly, thanks to my family for supporting me during this process. My parents always encouraged me to pursue my goals, and my wife Yin gave me the energy needed to finish the thesis.
Contents

1 Introduction ........................................ 1
   1.1 Motivation ...................................... 2
   1.2 Problem statement .............................. 4
      1.2.1 Ranked data dissemination ............... 5
      1.2.2 Top-k subscription filtering ............. 6
      1.2.3 Total order ................................ 7
   1.3 Contributions .................................. 7
   1.4 Thesis organization ............................ 9

2 Background ......................................... 11
   2.1 Introduction .................................... 11
      2.1.1 General model ............................. 12
      2.1.2 Applications .............................. 13
   2.2 Interest expression and message filtering .... 15
      2.2.1 Topic-based ............................... 16
      2.2.2 Content-based ............................. 17
      2.2.3 Single publication predicates .......... 17
      2.2.4 Composite subscriptions ............... 18
      2.2.5 Parametrization ......................... 19
4.6.1 Performance and sensitivity analysis ........................................ 58
4.6.2 Performance comparison ..................................................... 60
4.6.3 Sensitivity analysis ............................................................ 63
4.6.4 Online social networks use case ............................................ 71

4.7 Conclusions ............................................................................ 77

5 Top-k subscription filtering ......................................................... 78

5.1 Introduction ............................................................................. 78
5.2 Semantics and naive solution .................................................. 83
  5.2.1 Top-k model and ranking criteria ........................................ 83
  5.2.2 Semantics propagation ....................................................... 84
  5.2.3 Naive solution .................................................................... 85
  5.2.4 Problem with naive solution .............................................. 86
5.3 Top-k forwarding algorithms .................................................... 87
  5.3.1 Top-k-aware subscription covering ..................................... 87
  5.3.2 Rank-cover ........................................................................ 89
  5.3.3 Ancestor counting optimization (ACO) ............................... 91
  5.3.4 Pruned rank-cover .............................................................. 92
  5.3.5 Fairness ............................................................................. 93
  5.3.6 Discussion .......................................................................... 95
5.4 Evaluation ................................................................................ 96
  5.4.1 Setup ................................................................................ 96
  5.4.2 Metrics .............................................................................. 98
  5.4.3 Covering performance ....................................................... 99
  5.4.4 Fairness evaluation ........................................................... 102
  5.4.5 Processing overhead ......................................................... 105
7.2.1 Extending event processing systems . . . . . . . . . . . . . . . 152
7.2.2 Integrating new technologies . . . . . . . . . . . . . . . . . . . 154

Bibliography 155
List of Figures

1.1 Pub/Sub and client-side processing ........................................ 3
1.2 Interest granularity .......................................................... 3

2.1 Publish/Subscribe mechanism .............................................. 13
2.2 Pub/Sub and CEP interaction .............................................. 15
2.3 Multi-stage event processing .............................................. 16
2.4 Scribe subscription mechanism and multicast tree creation .......... 21
2.5 Advertisement-based forwarding ......................................... 21
2.6 Examples of covering relationship ....................................... 23
2.7 Type-based rendezvous forwarding ....................................... 24

4.1 Naïve top-k counter-example .............................................. 38
4.2 Chunk structure and inter-chunk overlaps .............................. 43
4.3 Broker pipeline for top-k processing of count-based windows ...... 47
4.4 Broker overlay topology .................................................. 58
4.5 Performance comparison .................................................. 60
4.6 Chunk guards offset ....................................................... 62
4.7 $\delta$/$W$ ratio analysis ..................................................... 63
4.8 Chunk size analysis ........................................................ 65
4.9 Outgoing traffic vs. publication delay .................................... 67
Chapter 1

Introduction

Publish/subscribe is a simple communication paradigm which allows loosely coupled entities to disseminate data in an event-based manner. Data producers publish data to a broker which forwards the publications through the broker overlay network to the interested data consumers. Interest in a publication is defined by consumers using subscriptions, to which each publication is matched against to determine the set of consumers to deliver to. Additionally, some pub/sub systems require the data producers to advertise the space they are publishing to initialize routing paths in the overlay network. Applications which employ the pub/sub abstraction include business process execution [110], workflow management [31], business activity monitoring [44], stock-market monitoring [116], selective information dissemination and Rich Site Summary (RSS) filtering [106], complex event processing for algorithmic trading [71], and network monitoring and management [44].

For the most part, these applications only require simple matching semantics. For instance, RSS filtering requires only topic-based matching, since each subscription is attached to a feed and all the publications generated within that channel are delivered to the matching subscriptions. Others can benefit from more advanced filtering
capabilities which allow for predicates to filter within a topic (known as content-based matching). In the case of stock market monitoring, a client could define interest in a certain stock only if it satisfies certain conditions (e.g., the value of the stock quote exceeds a certain threshold).

In order to satisfy the specifications of these applications, publish/subscribe systems have traditionally put the emphasis on high performance and scalability of a set of communication primitives with expressiveness limited to topic or content-based matching [39]. Recently, we have discovered applications which constitute a good fit for pub/sub due to its loosely coupled, many-to-many communication pattern, but require semantics which are not supported by current systems.

This thesis is about investigating these modern applications employing publish/subscribe technology, eliciting requirements not supported by current systems, and developing solutions to better fit the specifications of the studied use cases.

1.1 Motivation

Publish/subscribe is at its core a fundamentally simple delivery semantic. This allows systems to be designed to perform only simple message passing communication, and thus cannot perform more advanced (stateful) tasks. For instance, current pub/sub systems are not capable of expressing interest computed over a window of publications. However, we have determined that the ability to rank publications, and select the top-k best publications within each window, is necessary for applications such as social networks, where large volumes of data are summarized and disseminated to the application layer [119]. Thus, it is required that the systems handling those applications be capable of providing top-k filtering.

These limitations can be alleviated by attaching a CEP (Complex Event Pro-
Chapter 1. Introduction

Figure 1.1: Pub/Sub and client-side processing

Figure 1.2: Interest granularity

Processing) system at the subscribers (see Figure 1.1) [82]. Higher level subscriptions submitted to the CEP system at the client can be translated into pub/sub subscriptions: the interest of the original subscription can then be reconstructed by the CEP system using the publications received at the pub/sub layer. Figure 1.2 illustrates this approach. Coarse-grained subscriptions are thus submitted to the pub/sub system and matched against publications residing in the publication space. The resulting stream of publications is then processed at the subscriber’s side to produce the final result, which is the application’s interest.

High level expressions are usually not supported due to the overhead it puts on the pub/sub system. However, increasing the expressiveness of the subscription language
can actually raise the scalability of the system. For instance, it is already known that content-based filtering reduces publication traffic by improving the selectivity of the subscriptions [24]. We therefore argue that bridging the gap between the subscription space and the desired application interest at the endpoints is an inefficient solution and that we should strive to increase the expressiveness of pub/sub to reduce the amount of data sent for post-processing at the subscriber’s end.

We provide an illustrative use case in the social networks domain. Users obtain feeds generated from their friends’ updates, videos, application notifications, location-based events, offers, and coupons. This amount of information generated is often overwhelming, as the user is often only interested in the most relevant subset of data. Yet, the publish/subscribe must carry this entire stream to the user, who then locally processes the information for relevancy. There is therefore a clear gap between the subscription space and the application space. Supporting top-k filtering in the pub/sub system to perform additional relevance-based filtering reduces the size of the event stream and improves the user experience.

1.2 Problem statement

The goal of this thesis is to extend the subscription language for the purpose of improving the performance of the system. We argue that providing higher level capabilities in the publish/subscribe layer can result in a more efficient system as a whole. The premise of this thesis challenges the established notion that the pub/sub system should only involve stateless match-and-forward interactions between publications and subscriptions [41, 19]. Instead, we argue that the publish/subscribe layer is the most suitable location to perform certain advanced features. The work described in this thesis allows subscribers to specify their interest at a granularity closely matching
Chapter 1. Introduction

their true application interests, thus reducing the amount of traffic to be transported by the system.

A second objective is to investigate quality of service (QoS) for pub/sub systems. While there is a body of work concerning fault-tolerance [97, 68], little work exist on other delivery guarantees such as ordering [130], security, privacy, and latency. Again, QoS guarantees can be enforced via an external service deployed at the endpoints, but more efficient solutions can be achieved within the pub/sub system.

Concretely, this thesis studies the above research questions in three areas: ranked data dissemination (paper published in ICDCS [129]), top-k subscription filtering (under submission, available as a technical report [128]), and total order (paper published in ICDCS [127]).

1.2.1 Ranked data dissemination

In the context of applications built for online social networks, the amount of content served can overwhelm their users, who must sift through the data for relevant information. To assist users, we develop and implement dissemination of ranked data in content-based publish/subscribe systems. Although top-k computation can be performed locally at the user end, the size of the event stream can constitute a significant bottleneck. Our approach distributes the top-k computation on an overlay network to reduce the number of events flowing through. Instead of forwarding all events, nodes closer to the sources can identify and forward top-k events over the collected stream. Nodes closer to the sinks can then merge the top-k streams into the final result. We use a hybrid chunk-based approach which alternates between full forwarding of events and pre-computed top-k streams at upstream nodes. This hybrid approach allows us to maintain correctness by recreating a possible top-k interleaving of the entire event
stream. Experiments performed over 960 machines using datasets borrowed from Twitter and Facebook with 5k and 30K query subscriptions demonstrate that social workloads exhibit properties that are advantageous for our solution, achieving up to 48% network traffic reduction. We also offer a sensitivity analysis which demonstrates the impact of various parameters on the performance of the solution.

1.2.2 Top-k subscription filtering

In contrast to the previous approach, we investigate top-k subscription filtering, where a publication is delivered only to the $k$ best ranked subscribers (as opposed to selecting the $k$ best publications for each subscriber). Top-k filtering is an effective way of reducing the amount of data sent to subscribers in pub/sub applications such as targeted advertising, or social networks. The naive approach to perform filtering early at the publisher edge broker works only if complete knowledge of the subscriptions is available, which is not compatible with the well-established covering optimization (see Section 2.3.4) in publish/subscribe systems. We propose an efficient rank-cover technique to reconcile top-k subscription filtering with covering. We extend the covering model to support top-k and describe a novel algorithm for forwarding subscriptions to publishers while maintaining correctness. We also establish a framework for supporting different types of ranking semantics, such as fairness and diversity. Finally, we conduct an experiential evaluation and perform sensitivity analysis to demonstrate that our optimized rank-cover algorithm retains both covering and fairness while achieving properties advantageous to our targeted workloads.
1.2.3 Total order

Total ordering is a messaging guarantee increasingly required in applications employing content-based pub/sub systems, such as online games. The main challenge is the uniform ordering of publications from multiple publishers within an overlay broker network to be delivered to multiple subscribers. Our solution integrates total ordering into the pub/sub logic instead of offloading it as an external service. We show that our solution is fully distributed and relies only on local broker knowledge and overlay links. We can identify and isolate specific publications and subscribers where synchronization is required: the overhead is therefore contained to the affected subscribers. Our solution remains safe under the presence of failures, where we show resilient total order to be impossible to maintain. Our experiments demonstrate that our solution scales with the number of subscriptions and has limited overhead for conflict detection. A holistic comparison with group communication systems is offered to evaluate their relative scalability.

1.3 Contributions

In light of the problem statement above, this thesis provides the following contributions:

1. Contravening traditional thinking, the thesis as a whole proposes enhancing expressiveness and raising quality of service in publish/subscribe systems to improve their performance in the context of modern applications.

2. The first concrete contribution introduces the problem of ranked data dissemination for content-based publish/subscribe systems in the context of online social networks. We formalize the problem as a top-k publications filtering problem
and provide a correctness criterion. We develop an efficient distributed top-k publications solution for content-based pub/sub.

3. The second contribution proposes a solution for supporting fair top-k subscriptions filtering. We demonstrate that the problem is non-trivial when subscription covering is employed. We develop an extended covering model to support top-k. The framework we propose can support a variety of top-k semantics. As an example, we show how fairness can be maintained, which employs a novel efficient probabilistic shuffling algorithm.

4. Finally, the last contribution of the thesis shows how total order, a quality of service (QoS) requirement necessary for applications such as online games, can be realized in content-based publish/subscribe. We propose a lightweight solution which leverages the structural properties of the pub/sub topology to eschew the need to perform frequent and costly ordering of publications. We show the safety of our algorithm under failures.

In addition, two more research topics have been investigated and they resulted in co-authored contributions, but are not included in this manuscript. We offer a brief summary below, with references to the associated research papers for further details:

1. Massively multiplayer online games (MMOGs), which are typically supported by large distributed systems, require a scalable, low latency messaging middleware that supports location-based semantics and loosely coupled interaction of multiplayer games components. In such games, each client controls a mobile avatar in a virtual world. The data necessary for a player to interact with its local environment is disseminated through the use of a pub/sub system. This work is a study and evaluation of various pub/sub-based network engine
designs for MMOGs. To the best of our knowledge, this work is the first thorough analysis of the benefits of using an expressive pub/sub language, namely content-based matching, within the context of a real application. The experimental results of the work confirms that the enhanced language offered by the content-based pub/sub improves performance by reducing the subscription load in highly dynamic environments. This work is published in Middleware [20].

2. Modern applications for distributed publish/subscribe systems often require stream aggregation capabilities along with rich data filtering. When compared to other distributed systems, aggregation in pub/sub differentiates itself as a complex problem which involves dynamic dissemination paths that are difficult to predict and optimize for a priori, temporal fluctuations in publication rates, and mixed workloads with both aggregated and non-aggregated subscriptions. This is formalized as an optimization problem for minimizing communication cost to support aggregation in pub/sub, which is shown to be reducible to the problem of minimum vertex cover in bipartite graphs. Dynamic solutions are offered with varied heuristics which explore the tradeoff between computation and communication cost. Details about this work can be found in papers published in DEBS [93] and ICDCS [94].

More details about the current status of these projects can be found in the future work section of Chapter 7.

1.4 Thesis organization

The rest of this document is organized as follows. First, we provide in Chapter 2, a review of established publish/subscribe background knowledge. The techniques,
nomenclature, and taxonomy described constitute general knowledge required for proper understanding of all aspects of this thesis. Second, the related work for each contribution can be found in Chapter 3. Then, each of the core sections focuses on one of the aforementioned contributions: Chapter 4 for ranked data dissemination, Chapter 5 for top-k subscriptions, and Chapter 6 for total order. In each section, we provide additional background material necessary for understanding that particular work. We also provide the targeted applications which justify the need for the work in publish/subscribe systems and our solution to address the problem. Each chapter also provides experimental evaluation. The thesis concludes with Chapter 7, which summarizes our findings and provides an outlook on current and future work.
Chapter 2

Background

We now provide a review of established publish/subscribe techniques found in the literature. Related work which is specific to a particular contribution of the thesis is listed in the next chapter, Chapter 3.

2.1 Introduction

The scale of distributed systems have considerably expanded since the advent of the Internet. Common web applications can involve heterogeneous behavior amongst thousands of peers in geographically separated locations. Furthermore, application servers rely on horizontal partitioning over large clusters of commodity machines.

Our modern usage of the Internet reflects the need for a communication infrastructure which is not only highly scalable, but also flexible. The latter calls for a decoupling of the involved peers: data becomes the central focus, not its location [63]. Thus, consumers and publishers need not to be aware of each others’ presence.

Based on the concept of an information bus [90], the publish/subscribe messaging paradigm is the answer to those requirements. Data consumers (subscribers) have
control on the data content (but not its location) they receive by expressing their interest to the system. Publishers send data to the system irrespective of the identity of the recipients (or even if any exist). This property is referred to as \textit{space decoupling}.

Publications flow from the publishers, through the publish/subscribe middleware, to the subscribers. Thus, messages can be delivered even after the sender has disconnected. Similarly, subscribers can be notified of publications sent while they were offline. This loose relationship is described as \textit{time decoupling}.

Finally, publish/subscribe systems employ asynchronous communication primitives. Data is consumed asynchronously and can be treated as an event-based process, separate from the application code, hence providing \textit{synchronization decoupling}.

The decoupling along those three dimensions is what sets pub/sub systems apart from other communication paradigms [39]. The decoupling allows for looser consistency constraints on the system, increasing its scalability.

\subsection*{2.1.1 General model}

At its core, the publish/subscribe interface usually contains the following operations:

- \textbf{publish}(): Publishes target data to the system. The publication is delivered according to its declared interest.

- \textbf{subscribe}(): Subscribers declare to the pub/sub system which types of publications they would like delivered.

- \textbf{unsubscribe}(): Allows a subscribe to terminate a subscription.

Every operation is submitted from a peer to the publish/subscribe middleware. In addition, subscribers must support notifications sent from the pub/sub system.
Subscriptions are not consumed by publications and remain in the system until unsubscribed.

Some systems also require publishers to call `advertise()` before publishing data. Advertisement is typically used to initialize the system in some fashion. For instance, Scribe requires topics to be created to allocate the necessary data structure on the Distributed Hash Table (DHT) before data can be published on that channel [25]. For SIENA, publisher advertisements are used to compute the delivery tree for publications from that publisher [24].

The operations are illustrated in Figure 2.1. The publish/subscribe middleware continuously receives advertisements (from publishers) and subscriptions (from subscribers). The publish/subscribe system then accepts incoming publications and delivers them to the matching subscribers.

### 2.1.2 Applications

As stated before, the decoupled nature of publish/subscribe is a good fit for modern distributed systems. Publish/subscribe has been used for large-scale distributed applications: NASDAQ stock quotes are being disseminated using SuperMontage [115];
Chapter 2. Background

retailers such as Wal-Mart uses the GDSN (Global Data Synchronization Network) to communicate supply chain information [53]; real-time tennis match scores are also delivered using a pub/sub network developed by IBM [60].

Publish/subscribe systems have also been proposed in the context of service-oriented architectures. \( NIÑOS \) is a decentralized solution for orchestrating a business process execution using an underlying publish/subscribe system [81]. Essentially, processes written in BPEL (business process execution language) are mapped onto publish/subscribe primitives, eschewing the need for a central coordination entity. Furthermore, it has been shown that publish/subscribe systems can support automatic service composition [58].

One prominent domain for publish/subscribe systems is on the Internet. This includes traditional applications such as Rich Site Summary (RSS) readers [103] and online games [11]. Large Internet-based enterprises also use publish/subscribe systems for their flexibility and scalability: Google employs GooPS for integrating various web applications [105], while Yahoo! uses Hedwig for replicating data sets across data centers [104].

The popularity of publish/subscribe systems have led to research in new Internet architectures. Future network models will be content-centric [117], rather than location-based. This calls for an anycast messaging model, where requests can be served by any provider carrying the queried data [72]. Proposed solutions usually involve a routing protocol similar to publish/subscribe either at the network layer [63] or at the naming layer [52]. Here, publisher advertisements are used either to route queries or for server lookup.

Beyond these approaches, publish/subscribe is at the heart of PSIRP, a clean slate approach for a data-oriented network infrastructure [36]. Their architecture rely on an efficient encoding of a publication’s delivery tree into a Bloom filter to achieve
Figure 2.2: Pub/Sub and CEP interaction

routing at line speed [65].

2.2 Interest expression and message filtering

Publish/subscribe systems can be categorized by the expressibility of their subscription language. More powerful language expressions allow for more fine-grained filtering of incoming publications. Functionally, however, any interest that cannot be satisfied precisely can be overcome by equipping subscribers with a CEP (Complex Event Processing) system [82] and/or application logic layer (see Figure 2.2). Note also the utilization of a QoS layer to satisfy non-functional requirements (see Section 2.4).

Given a publication stream (the sequence of all publications entering in the system) and an application data stream for a client (the sequence of all events to be notified to the application layer), the architecture outlined above should produce a subscription for that client which encompasses the application interest (see Figure 2.3). In other words, the events required by the application are either included in a larger set or derivable from the subscription set. The stream of publications matching the subscription is then passed on to the CEP system and/or the application logic itself for further processing to obtain the data stream required by the application.
In such a model, the only concern is where different parts of the filtering takes place, either in the pub/sub system or locally at the subscribers (colored red and blue respectively in the figure). There is thus no loss of functionality from the point of view of the client application when considering a less powerful pub/sub system. Since pub/sub systems are engineered for efficiency, the expressibility of the subscription language must be considered a means to provide that scalability. This is realized in publication filtering techniques which minimize the amount of traffic in the system.

Because the main concern of pub/sub systems is scalability, there exists a tradeoff between network traffic and matching overhead which is exposed by the language expressiveness. Hence, any language feature which compromises the overall scalability of the system is not supported. We will now review existing subscription schemes and the expressions currently supported by pub/sub systems. Details on the protocols employed to implement these languages are provided in Section 2.3.

### 2.2.1 Topic-based

Topic-based pub/sub systems (such as Scribe [25]) limit subscriptions to a set of specific topics or subjects. Each publication is then sent to a specific topic and delivered to its subscribers. This is reminiscent of group communication systems [100], with the difference that group communication systems focus on providing consistency (e.g. total ordering) while pub/sub focus on scalability.

Hierarchical topic addressing is also possible in topic-based pub/sub systems, as
introduced in TIBCO Rendezvous [114], allowing for coarse-grained subscriptions encompassing more specific topics.

2.2.2 Content-based

Content-based pub/sub refers to a very broad class with varying expressive power. At its base, every content-based pub/sub give subscribers the power to “personalize” their subscription by filtering publications through the use of predicates. A commonly cited advantage over topic-based solutions allows content-based systems to define larger topics and have each subscriber submit subscriptions which filter only a subset of that topic, thus reducing the number of topics necessary. For instance, suppose a client $c$ is interested in stock quotes of company $A$ when the price is below $15$. A topic-based system would have to either divide publications for $A$ into two topics (for quotes below and above $15$) or to have $c$ subscribe to all quotes and externally filter only the relevant quotes.

2.2.3 Single publication predicates

Each publication carries a set of attribute-value pairs which are matched against the set of predicates of each subscription. These attribute-value pairs and predicates can be stored as strings, as in the case in SIENA [24], JEDI [31], Gryphon [9] or PADRES [62]. Predicates consist of attributes compared to static values using a basic operator ($\neq$, $<$, $\leq$, $>$, $\geq$). Substring matching, existential operator (PADRES) or even regular expressions (JEDI) are supported. Predicates can be also combined logically (and, or, etc.). Table 2.1 shows examples from the PADRES language. Publication P1 matches S1 since all the attributes of P1 satisfy the predicates of S1, while P2 does not match S1 (because the name does not match).
Chapter 2. Background

| S1  | s [class, eq, 'stock'], [name, eq, 'IBM'], [price, <, 60] | IBM stocks when the price is below 60 |
| P1  | p [class, 'stock'], [name, 'IBM'], [price, 45] | IBM stock with value 45 |
| P2  | p [class, 'stock'], [name, 'GOOG'], [price, 50] | Google stock with value 50 |
| CS1 | cs [class, eq, 'stock'], [name, eq, 'IBM'], [price, >, 50] & [class, eq, 'stock'], [name, eq, 'HP'], [price, >, 40] | IBM stock with price above 50 AND HP stock with price above 40 |
| CS2 | cs [class, eq, 'stock'], [name, eq, 'IBM'], [price, >, 50]] | IBM stock with price above 50 OR HP stock with price above 40 |
| CS3 | cs [class, eq, 'stock'], [name, eq, '$X1'], [price, >, 50] & [class, eq, 'stock'], [name, eq, '$X1'], [price, <, 10] | Stock for any company which had a quote above 50 and another below 10 |

Table 2.1: PADRES language format

Hermes [98] distinguishes itself from other pub/sub system through its use of type-based matching. Instead of allowing arbitrary topics or attributes, publications and subscriptions must specify the object type. This constraint can then be type-checked at runtime.

2.2.4 Composite subscriptions

Publish/subscribe also provides limited support for stream processing (Hermes, Gryphon, SIENA, PADRES, REBECA [46]). Composite subscriptions allow the client to express multiple sets of predicates that must be satisfied by different publications. Once the entire subscription is matched, the set of publications is delivered atomically to the client.

Composite subscriptions differ in behavior across systems. An ordering semantic is defined in SIENA which enforces the composite subscription to be matched in a particular order. In PADRES, the components of a composite subscription can be matched in any order [79] or according to a sequential order where the publications occur within a defined window. They can also be combined logically. CS1 and CS2
are examples of composite subscriptions in PADRES. For CS1, no ordering is enforced on the publications. In the case of CS2, a single subscription that match either of the sub-elements is sufficient to match the entire composite subscription.

2.2.5 Parametrization

As outlined above, predicates compare attributes against static values. PADRES also allow dynamic values through parameterization for composite subscriptions. Values for predicates of one publication can be cross-referenced into predicates of another publication. CS3 shows how parameterization can be used in PADRES.

Dynamic parametrization has also been achieved in [64]. Parametric subscriptions allow subscribers to send update messages which are propagated through the broker overlay, modifying the content of the targeted subscription. The main concerns are performing the routing and matching updates necessary to support the new subscription with minimal disruption to the system.

2.2.6 Aggregation

Higher up on the expressiveness spectrum are aggregation functions operating on arbitrary streams of publications. No known complete pub/sub systems has such stream supporting capability. The closest in achieving this is Caguya [35], a highly expressive pub/sub matching engine. Subscriptions are modeled as non-deterministic finite automata, where state transitions are triggered by the publication stream, outputting matches. However, Caguya is designed for efficiency in the context of centralized matching and has not been implemented as part of a distributed architecture.

Another close match is Agele, a pub/sub system which provides limited support for aggregation [27]. Agele instantiates event gatherer tasks at broker nodes for
the purpose of aggregation. The main contribution of this work is determine the optimal placement of such event gatherers. However, Agele employs aggregation for the sole purpose of eliminating redundant or duplicate publications and cannot support arbitrary stateful operations.

2.3 Infrastructure

Apart from some early centralized approaches such as IBM MQSeries [76] and Oracle Advanced Queuing [92], publish/subscribe systems are distributed for scalability and availability purposes. In this section, we will review the main methods of distribution. In particular, we will focus our attention to broker overlay networks.

2.3.1 P2P

Peer-to-peer pub/sub systems use the clients’ resources as part of the pub/sub infrastructure. More specifically, P2P pub/sub systems are built on top of a distributed hash table (DHT).

Scribe, a topic-based pub/sub system, is built over the Pastry DHT. Topics are created as a key in the DHT. Subscriptions sent by clients for a channel traverse the DHT in hops (see Figure 2.4). The reverse paths of all subscriptions for a given topic form a multicast tree which propagate publications sent to that topic key.

Meghdoot [54] uses a spatial approach to content-based matching. Given a schema of $k$ attributes, a $2^k$ dimensional space over the CAN DHT is formed, with each subscription occupying a point in this space. Publications traverse overlapping regions in this space to search for matching subscriptions. This approach optimizes subscriptions storage (which are only stored in a single peer) to the detriment of publication matching latency (publications must be routed to multiple peers). Since pub/sub
workloads contain more publications than subscriptions, the benefits of this approach are doubtful.

### 2.3.2 Overlay broker network

We now look at the class of pub/sub systems using overlay broker networks. Instead of being fully decentralized like in peer-to-peer approaches, the system employs a set of dedicated servers (*Brokers*) to offload the pub/sub logic from the clients. These brokers are connected to form a topology through which pub/sub messages circulates. Figure 2.5 illustrates this class of systems. More details on routing are provided later in this section.
2.3.3 Topology and overlay construction

In [24], the authors categorize three types of broker network topologies. *Hierarchical* topologies float data upstream to disseminate to other parts of the network. *Acyclic* topologies allow brokers to communicate bidirectionally to their neighbors. However, a single path must connect each peer (clients and brokers) in the network, which simplifies routing. If redundancy links are required (for fault-tolerance), a *general* topology can be employed at the cost of more complex routing algorithms [80].

Scalable overlays can be constructed which minimize the amount of forwarding traffic required to disseminate a publication. This is done by clustering publishers and their subscribers to the same part of the network, maximizing locality [30, 26].

Hermes uses a peer-to-peer approach for overlay constructions [99]. It is built on PAN, a routing substrate similar to Pastry. Brokers communicate with each other by traversing the peer-to-peer network. The advantage of this approach is that the topology is dynamic.

2.3.4 Matching and routing

Matching and routing publications follow two general principles:

- *downstream replication*: A publication should be routed as one copy as far as possible and replicated as close as possible to the subscribers.

- *upstream evaluation*: Filters should be applied as early possible to route publications in the shortest path to their subscribers.

These principles are realized by forming a delivery tree for a publication. Publications are forwarded from the source (publisher) to the next hops by matching the publication against the sum of the subscriptions down each branch. Each broker along
the path then successively compute the next hops away from the incoming edge to disseminate the publication. This is applicable to acyclic topologies only. For general topologies, greater care must be put to ensure loop-free delivery.

In MEDYM [22], this delivery tree is formed by computing a Short-Path Minimum Spanning Tree (SPMST). In Gryphon, a parallel search tree (PST) is maintained where subscriptions form the leaves of the tree and brokers are the internal nodes. Each broker computes the next step in the PST to prune out branches that do not match the publication and propagate the publication downstream to matching nodes.

The technique used by SIENA and PADRES is advertisement-based forwarding (see Figure 2.5). Advertisements are flooded to every broker from the publisher. Whenever a subscription is received, it is propagated back from the reverse (shortest) path of the advertisements to the matching publishers. This effectively forms a delivery tree for publications, as each broker maintains the next hop required to reach a subscriber.

Furthermore, covering can be used in SIENA and PADRES to limit the number of subscriptions to maintain. For a given broker, a subscription $S_1$ covers $S_2$ if the subscription space for $S_1$ contains the subscription space of $S_2$ (see Figure 2.6). If $S_1$ has the same next hop as $S_2$, we know that every publication that matches $S_1$ will

Figure 2.6: Examples of covering relationship
Figure 2.7: Type-based rendezvous forwarding

have to be forwarded down to that hop. Thus, the broker does not need to compare against $S_2$, since there never will be a case where a publication matching $S_2$ will not be forwarded down that hop by not matching $S_1$.

Advertisement-based forwarding works on the assumption that advertisements constitute a small portion of the workload and thus do not generate too much overhead due to their high flooding cost. To avoid this problem, Hermes uses a rendezvous based approach (see Figure 2.7).

Whenever a new type is introduced to the system, it is assigned a rendezvous node (a broker). Instead of flooding advertisements, the advertisements are directed towards the rendezvous node corresponding to its type, with every broker along the path caching the advertisement. Similarly, subscriptions for that type are also routed to the rendezvous node and cached at intermediary brokers. Advertisements are also forwarded towards publishers whenever a cached advertisement is encountered. Publications can then be delivered to the subscribers through the reverse path from any broker with a cached subscriber. Any subscriber not reached in this manner will be delivered through the rendezvous node, which forms a meeting point for advertisements and subscriptions. The disadvantage of this method is that the delivery path might not be optimal, depending on the location of the rendezvous node.
Finally, an alternate approach described in LIPSIN computes the entire delivery tree at the source (source-based routing). Every link in the network is assigned an ID and the source adds every link ID to be traversed by the publication to a Bloom filter [16]. The filter is included with the publication. Each broker then forwards the publication on each outgoing link included in the filter. The advantage of this solution is that the pub/sub logic is removed from the brokers which are stateless in that regards. This allows for very efficient, line-speed forwarding between brokers. The disadvantage is the high overhead of maintaining and computing the entire delivery tree at the source.

2.4 Quality of service

This section reviews some of the quality of service guarantees provided by pub/sub systems.

2.4.1 Persistence

Pub/sub systems do not generally offer persistent storage. In [77], data repositories are maintained at brokers to respond to historical queries. It thus allows a subscriber to receive publications that were published prior to its subscription.

Although only a notification service rather than a full-fledged publish/subscribe system, Thialfi can cache events for offline users [3]. These events are then sent to the subscriber when it is brought online, in accordance with the synchronization decoupling principle.
2.4.2 Reliability

Overlay broker networks can embed fault-tolerance mechanisms. Redundant links found in general topologies can alleviate link failures. For broker failures, [67] allow brokers to bypass failures by maintaining state information about their neighbors. This approach is extended to support Byzantine failures in [69].

On a related note, [57] describes transactional mobility for peers, allowing them to relocate to a different broker, for reliability or performance reasons. [29] is another work which focuses on efficient placement of system resources to achieve maximum efficiency and power consumption.

2.4.3 Total Order

Due to its decoupled nature, pub/sub systems do not usually guarantee delivery ordering across multiple subscribers. In topic-based systems, proposed solutions involve sequencing messages for enforce an ordering. This is achieved by either consolidating publisher streams [12], accessing a sequencer on some global memory [95], or by assigning dedicated sequencers for overlapping topics to resolve ambiguity in the delivery order [83].

In content-based systems, ordering can be achieving by decoupling the total order service from the dissemination service, as in [84].

2.5 Summary

This chapter highlighted the defining feature of publish/subscribe as being its decoupled nature, designed for high scalability. Pub/sub systems can vary in expressiveness of the subscription language, but the expressiveness is always used as a tool for achiev-
ing greater performance. The architecture is typically decentralized and often uses an overlay broker network, which allows for reliability and efficient routing algorithms and topologies. Because of their need for high performance, pub/sub systems are limited in terms of QoS guarantees. The research challenge of this thesis is therefore to design efficient pub/sub systems with greater expressiveness and messaging guarantees.
Chapter 3

Related work

We now provide a detailed analysis of the state-of-the-art related work. We structure this chapter by listing the relevant literature for each of our contributions.

3.1 Ranked data dissemination

Generally speaking, the problems related to retrieving the most relevant answers have been studied in different contexts including database (distributed) top-k querying ([42, 61, 7, 23, 87]) and publish/subscribe (pub/sub) matching techniques [124, 4, 41, 21, 120, 48, 107]. Another active research area for content dissemination is information-centric networking (ICN) that is designed for Internet-scale deployment [51]. The primary goal of ICN is to allow users to retrieve data based on its content rather than its physical location; however, in this work, we take a further step and allow users to rank and filter unwanted content, formulated as a top-k problem.

The most widely adopted database top-k processing model [42, 61] differs from our proposed top-k model in an important respect: Our top-k model solves the reverse problem. In the database context, top-k querying means finding the $k$ most relevant
tuples (events) for a given query (subscription). But in our pub/sub abstraction, matching means finding the relevant subscriptions (queries) for a given event (tuple). Furthermore, the database literature is best suited for lower dimensional data [61], while in the pub/sub context, dimensionality on the order of hundreds is commonplace [124, 4, 41, 21, 120, 48, 107, 108], which is orders of magnitude larger than the capabilities of existing database techniques.

Broadly speaking, two classes of matching algorithms have been proposed for pub/sub: counting-based [124, 41, 120] and tree-based [4, 21, 107, 108] approaches. A fresh look at enhancing pub/sub matching algorithms is to leverage top-k processing techniques to improve matching performance. An early top-k model is presented in [85]; however, this model leverages a fixed and predetermined scoring function, i.e., the score for each expression is computed independently of the incoming event. In addition, this approach is an extension of the $R$-Tree, the interval tree, or the segment tree structure; as a result, it is ideal for data with few dimensions [85]. In contrast, a scalable top-k model that supports up to thousands of dimensions while incorporating a generic scoring function is introduced in [120], which relies on a static and single-layered pruning structure. To alleviate these challenges, a new dynamic and multi-layered pruning top-k structure is developed in [108]. However, our proposed top-k model attempts to solve an orthogonal problem, namely, distributed top-k processing; therefore, our model can leverage any of the existing top-k work as building blocks for optimizing local top-k processing at a broker (e.g., [85, 120, 108]).

Another important aspect of pub/sub top-k matching is to explore and identify plausible top-k semantics. Unlike in the database context, formalizing top-k semantics in pub/sub is more involved and not limited to a single interpretation [101, 37, 38]. The most widely used pub/sub top-k semantics is defined with respect to subscribers, i.e., consumer-centric semantics, in which the subscription language is extended (with
a scoring function) in order to rank each incoming publication (over a time- or count-based sliding window); thus, delivering only the top-k matched publications to each subscriber \([101, 37, 38]\).

Alternatively, the top-k semantics can be defined with respect to a publisher, i.e., producer-centric semantics, which extends the publication language for ranking subscribers and delivering publications only to the top-k matched subscribers \([85, 120]\). Produce-centric semantics is suitable for targeted advertisement (e.g., targeting a specific demographic group) and diversified advertisement (e.g., reaching out to most eligible members within each demographic group). Finally, hybrid semantics can be foreseen such that both subscribers and publishers have control on how data is received and disseminated, respectively. In our work, we focus primarily on the well-adopted consumer-centric top-k semantics, although some of our techniques could be leveraged for both semantics.

Top-k processing, being a stateful operation, can also be thought as a special case of aggregation, which has been studied in the context of publish/subscribe, such as in our own work \([94, 93]\). Aggregation processing takes entire windows of publications and produces a result output according to the specified operator. A top-k result set would then contain the \(k\) highest ranking publications in a window. Top-k processing has a particular property in that the result set consists of publications in their original form. These publications can therefore be delivered individually and disseminated to multiple matching subscribers, whereas general aggregation results cannot usually be shared. The solution provided in this chapter leverages properties specific to top-k processing to produce an efficient distributed solution which cannot be generalized to all-purpose aggregation.
3.2 Top-k subscription filtering

Being related to the above work, the relevant literature is also similar, especially with regards to top-k computation itself. The work presented in Chapter 4 is also a window-based top-k matching solution that also enables covering, but unlike the work in Chapter 4, its focus is on consumer-centric filtering.

A recent work argues for the importance of top-k matching based on the relevance of subscriptions (i.e., producer-centric) using a non-monotonic and dynamic scoring function [32]. However, they do not pay attention to the covering routing techniques employed by pub/sub systems and other top-k ranking semantics such as fairness and diversity, which is the focus of our work.

In Chapter 4, we focus primarily on producer-centric semantics that is publisher-driven and enables ranking the consumers or subscribers. More importantly, we further enrich the producer-centric matching semantics to satisfy ranking semantics beyond basic relevance scoring and to include fairness and diversity while addressing the subscription covering challenges that arise.

3.3 Total order

There has been considerable work in developing total order broadcast and multicast algorithms [34]. One technique is to route all messages to one or more dedicated sequencers to order messages [66, 50, 13]. A concern with these algorithms is the scalability of the approach as the sequencer(s) must have the capacity to route and process all messages in the system. By contrast, the solution in this thesis (see Chapter 6) is fully distributed and avoids global sequencing to resolve ordering conflicts.

Another class of algorithms record the history of interactions among a group of
nodes, and nodes use this history to deterministically compute a message order [74, 96]. The solutions in this space assume a well-defined group of nodes that participate in the protocol. Such a model is not appropriate for a distributed content-based pub/sub system. First, in such systems, group membership knowledge is distributed and there can be constant churn in the set of subscribers and subscriptions. Moreover, subscriptions can express fine-grained interest—as opposed to a broad interest in all messages sent to a group—resulting in an exponential number of groups. For anything but small subscription populations, it is not feasible to form groups. The algorithm in this thesis locally determines a set of subscriptions and advertisements that need to participate in resolving the order of a given publication. By performing this detection on-demand for every publication, and using only local knowledge, the problem becomes tractable.

Previous work on ordering in pub/sub systems focuses on topic-based systems only and uses sequencer nodes to enforce an order in ambiguous situations [83]. The drawback of sequencer nodes is that the traffic is rerouted to be delivered from the sequencer, independent of the source of the publication. This limits the advantage of using overlay broker networks since it can create a bottleneck near those sequencers. Furthermore, global knowledge of the subscriptions is required for the sequencers.

FAIDECS offers a fair decentralized solution for event correlation in event-processing systems [121]. The solution relies on a DHT lookup to identify merger processes which are responsible for ordering specific subsets of event types. Publications must flow through this network of mergers before being delivered. Because the solution makes no assumption on the underlying model, all events must be processed through FAIDECS. In contrast, our solution is integrated directly as a lightweight component installed on pub/sub brokers, and can leverage local broker knowledge about the underlying overlay topology to limit the ordering overhead.
The solution in this thesis orders publications by synchronizing brokers along the path of publications. It does not require additional knowledge from the brokers. Furthermore, it is a solution for content-based systems.

In [5], FIFO links are used to preserve an ordering enforced by merger nodes. Our solution differs in that it requires no modification to the topology or dedicated nodes for ordering. We also leverage FIFO links as the foundation of enforcing a certain order in our system. Brokers then detect situations where FIFO links are not sufficient to maintain total order and run a resolution protocol to reorder the stream.
Chapter 4

Ranked data dissemination

This core chapter provides the details of our ranked data dissemination work. Section 4.1 introduces social networks as a motivating scenario for the work. Section 4.2 provides a formalization of the ranked data dissemination problem in the context of content-based publish/subscribe. Section 4.3 presents general concepts we supply for the solution, including extensions. Section 4.4 has the core algorithms of the solution, with proofs of correctness in Section 4.5. Section 4.6 contains performance results for our solution. Finally, we summarize the findings of this chapter in Section 4.7.

4.1 Introduction

The importance of social networks is prevalent with their ever growing user base that has already surpassed a billion users\(^1\). Social networks (SN) at this scale introduce new challenges for both the underlying infrastructure and the perceived user experience.

Examples are the feeds produced and consumed by users and their friends’ up-

\(^{1}\)As of December 2013, Facebook has over a billion active users; http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.
dates of news feeds, videos, application feeds, and location-based events, offers, and coupons.

Other sources of information are rooted in thousands of applications running on SNs that also generate content on behalf of their users (e.g., the social game FarmVille [18]). More recently, the linked data movement is becoming yet another source of publishing information [15, 14]; for instance, social tags (e.g., keyword annotations, “like” tags) on many websites represent a new way of publishing data on SNs. Social tags will (soon) be enabled on billions of web pages, which are visited by millions of users daily. The generated (and delivered) amount of information at its current growth will soon be overwhelming to the consumer [119].

In order to improve the user experience, we aim to enable users to (globally) rank their feeds and be notified only of the top results of interest to them. In SNs, the homepage feed contains only the most relevant stories. More importantly, many of these feeds naturally have notions of proximity and locality associated to them, for example, the value of location-driven marketing and advertising is estimated to be $1.95 trillion by 2022 [18]. In location-based services, people with close proximity and similar interests (e.g., students on campus) constitute an ideal basis for joint top-k filtering. Similarly, traditional coupon distribution is also highly localized: People in the same area may receive similar offers, which constitutes a further opportunity to cluster similar interests based on location.

From the infrastructure perspective, the social network must carry large volumes of data between its users. From the user experience perspective, large volumes of data must be adequately (pre-)processed to be consumable by users. Thus, it is essential for the user to simplify consumption of the volumes of data, which requires the capabilities to filter and rank information and to selectively pick information. This puts the burden on service providers to offer fine-grained control over the delivered
information; yet, it also brings forward new opportunities to optimize the infrastructure to reduce message traffic through the adequate placement of interest queries and information routing operations towards interested users.

The underlying infrastructure that processes these sheer volumes of data, which are prevalent in today’s social network domain such as [40, 89, 43, 1], is inevitably running in a distributed computing environment. Indeed, a rising trend is to establish a decentralized architecture in social networks such as Diaspora, Tent, and DSNP [33]. The pub/sub model, known for its scalability and decoupled nature, then becomes a logical candidate as the dissemination substrate for distributed social networks, and pub/sub designs for social networks have recently surfaced [112, 123, 111]. Therefore, our goal is to provide a solution for distributed ranked data dissemination for pub/sub systems within the context of social networks.

Thus, in this chapter, we make the following contributions:

1. We present a lightweight approach to aggressively filter at the dissemination network’s edges to substantially reduce the overall network traffic (Section 4.2).

2. We develop novel algorithms that achieve early pruning of messages that are guaranteed not to be part of a subscriber’s top-k results while maintaining semantic correctness (Section 4.3 and proofs in Section 4.5).

3. We propose adaptive buffering extensions to avoid re-sending messages that fall in the top-k windows of multiple subscribers (Section 4.4) and to optimize the solution according to the current state of the system.

4. We evaluate and compare the performance of our solutions vis-à-vis a baseline algorithm, offer a sensitivity analysis of the various parameters related to top-k semantics and those specific to our solution, and employ datasets borrowed
Chapter 4. Ranked data dissemination

from Facebook and Twitter, containing 5K and 30K subscription queries, to demonstrate the scalability of our pub/sub architecture by identifying the major properties of social network workloads that impact our solution (Section 4.6).

4.2 Semantics and correctness

We develop a solution for supporting top-k data dissemination within the context of a pub/sub broker overlay network [45]. This model allows publishers to publish data to a broker, which forwards the data to consumers who have subscribed to the data while remaining decoupled. Brokers in the pub/sub system form an overlay and collaborate to disseminate publications from publishers to the appropriate subscribers.

Subscribers define their interest by specifying a conjunction of predicates. Publications contain a set of attribute-value pairs. Those attributes are compared against the predicates of each subscription. If a publication satisfies the predicates of a given subscriber, the publication is considered matched and must be delivered to the subscriber.

We extend the functionalities of pub/sub systems to allow top-k filtering on a publication stream. Subscribers include a scoring function with their subscription which assigns a numerical score to each input publication. Windows of publications are derived from the stream of matching publications according to parameters provided by the subscriber. For each window, the $k$ publications with the highest score are determined and delivered to the subscriber, while the rest are discarded. Windows can be either count- or time-based. A count-based window contains a fixed number of events, while a time-based window contains publications within a certain time range. The focus of this work will be on the former only.

Although the top-k stream can be computed a posteriori (at the consumers), our
approach distributes the top-k computation to the upstream brokers. Top-k filtering is applied on partial streams collected upstream and disseminated towards a target subscriber. These top-k streams are smaller than the original streams; thus, reducing traffic within the system. These are then merged downstream to form the final top-k results.

With each subscription, the subscriber specifies the following top-k parameters:

1. $W$: the window size. $W$ is the number of elements each window contains.

2. $k$: the number of highest ranking elements within a window to be delivered.

3. $\delta$: the window shift size. $\delta$ is the number of elements (publications) to shift over for each successive window. $\delta = 1$ corresponds to sliding windows, $\delta = W$ corresponds to tumbling windows.

We formalize the correctness of our distributed top-k solutions by comparing it to centralized top-k solutions. We show that there is always a stream interleaving that yields identical results for both distributed and centralized solutions. This formulation enables us to compute top-k results in a distributed manner with minimal
synchronization across independent stream sources. We name this correctness criterion \textit{stream reconstructability}:

\textbf{Definition 1} (Stream Reconstructability). \textit{Given a set of finite event streams }$E = e_1...e_n$\textit{ and any window semantics }$WS$\textit{, there exists a combined permutation (interleaving) of all events in }$E$\textit{ that yields our distributed computed top-$k$ solution }$T(E, WS) = t_1...t_n, t_i \in E$\textit{ that requires only to preserve event ordering within each stream thereby not imposing any cross stream event ordering.}

The correctness criterion (Definition 1) states that a correct solution selects the same set of top-$k$ events as the centralized solution, where events are all delivered to the subscriber before top-$k$ is applied. Due to the asynchronous nature of pub/sub, several interleavings of publications exist, depending on the ordering guarantees provided by the system. Because our windows are count-based, different permutations of publications may produce different results. According to our correctness criterion, there exists one interleaving of publications that can be processed by the centralized solution to obtain a sequence of top-$k$ results which is identical to the distributed solution. Because this correctness criterion is agnostic to the scoring function, a distributed solution must in general consider every window from a possible interleaving of all publications to guarantee that all top-$k$ publications will be selected. Note that Definition 1 does not guarantee a specific order for the correct set of top-$k$ events.

For the remainder of this chapter, we will focus on a model where messages in the network can be arbitrarily delayed but never lost or reordered. Per-publisher order is therefore enforced: Multiple publications from a single publisher are delivered to all interested subscribers in FIFO order. This could be supported by the underlying network layer, (eg. using TCP).

To illustrate the complexity of our correctness criteria, we first present an incorrect
naïve distributed approach: Source brokers simply filter local publications and select the top publications out of each window to disseminate downstream. Non-source brokers then simply forward any received publications to the subscriber. Figure 4.1 is a sample execution of the naïve distributed algorithm for sliding \( (\delta = 1) \) count-based windows. Each source broker independently selects and forwards the top-k publications out of their respective windows. For instance, the top broker forwards publications \([a, d]\) for the first window and only \([b]\) for the second window (publication \(d\) is selected again but since it has already been disseminated, it is omitted). The top-k results from each source broker are delivered in a certain interleaving to the subscriber broker.

Suppose the sequence of top-k publications sent is \([a, d][2, 3][b][4, 5]\). To obtain such an interleaving of top-k results, the reconstructed stream of original publications should be \([a, b, c, d, 1, 2, 3, 4, b, c, d, e, 2, 3, 4, 5]\). However, applying our sliding window semantics to this stream would also require results for windows such as \([b, c, d, 1]\) and \([2, 3, 4, b]\). In the naïve distributed algorithm, such windows are not considered since each source broker considers only windows of publications originating from their own publishers. Because it is impossible to construct an interleaving of publications where there does not exist at least one sliding count-based window that includes a publication from both brokers, this approach fails the correctness criterion.

### 4.3 Distributed chunking solution

Our algorithm for efficient top-k pub/sub dissemination consists of three key ideas: (1) maintain correctness using chunking, (2) avoid sending duplicate messages using buffering, and (3) defer (adaptively) covering to farther points in the network to exploit both distributed top-k computation and subscription covering gains.
Note that efficient (centralized) top-k computation by a single broker is complementary to the focus of this chapter and has been thoroughly studied elsewhere [37, 108]. We are concerned with the distribution of the top-k computation, not the actual computation itself.

We also note that a centralized top-k solution does not necessarily indicate that the system is itself centralized. The notion of centralized top-k processing refers to the top-k computation of a single subscriber occurring entirely at a single location. It does not mean that a single node is in charge of all the top-k computations (ie., it is possible for different brokers to process different subscriptions), nor does it mean that only a single broker is in charge of publication dissemination (ie., it is possible for a set of brokers to disseminate publications to a single broker in charge of computing top-k for a particular subscription).

4.3.1 Chunking

As shown in the previous section, the naïve distributed approach is incorrect because it fails to consider windows that cut across publisher boundaries. In other words, when publishers are located at different brokers, the naïve distributed approach will omit certain results because certain combinations of publications coming from multiple publishers are not considered.

The intuition behind our solution is to disseminate additional publications downstream such that the broker connected to the subscriber can process those missing windows. This means that every publication found in those windows must be sent downstream. Our solution is therefore hybrid in nature: Publisher edge brokers switch between full forwarding of publications necessary to maintain correctness, and dissemination of top-k results only. Additionally, the missing windows are ones that
contain publications from multiple sources. We thus need a mechanism to pick an interleaving of publications that reduces the occurrence of such windows.

We introduce the notion of **chunks** and **guards** to maintain correctness. Each publisher edge broker stream is divided into contiguous chunks. Every broker or subscriber can independently choose the chunk size (a parameter referred to as $C$). Chunks can be count- or time-based independent of the properties of the top-k window semantics.

For each chunk $C$, the first and last $W$ events must be included in the chunk and serve as **guards** (left guard and right guard, respectively). For all windows of length $W$ within the chunk, the top-k events are computed and forwarded. Publications that are not part of any top-k or guards are discarded. The parameter $W$ has an impact on performance: Large chunks will reduce forwarded messages and reduce top-k processing at downstream brokers, but may increase the delivery delay since more publications must be received to completely fill the chunk before it is delivered.

The broker may forward events in a chunk as they arrive or as a whole. In either case, the downstream broker must process the events in the chunk as a whole. In particular, it must not interleave other events within those in a chunk.

Upon receipt of a chunk, an edge broker can compute the final top-k results for its subscribers through a process called **dechunking**. Publications located within a chunk (outside of the guards) are already part of processed windows upstream and are simply delivered to the subscribers. The guards, which are complete windows of publications, are processed to compute windows from the inter-chunk top-k zone (i.e., windows that contain publications from two different chunks). Only publications that have been identified as top-k within those guards will ultimately be delivered. The example of Figure 4.2 is dechunked as follows: Events from $LG_1$ marked as top-k events will be delivered. Then, windows $w_1$ to $w_n$ will be delivered as is. Finally, top-
k events will be computed and delivered for all windows with publications between $RG_1$ to $LG_2$. The process is repeated for Chunk 2 and any subsequent chunks.

The correctness of our solution depends on Lemma 1 and 4 detailed in Section 4.5. These two lemmas establish the stream reconstructability property: They demonstrate that our chunking algorithm creates valid interleavings of events, and that the guards provide enough publications to compute all the necessary windows.

One extension we provide allows intermediary brokers to perform “rechunking” by merging multiple chunks into a single larger chunk. To do so, a broker can reconstruct a stream by concatenating incoming chunks in any order, with the constraint that events within a chunk must be kept contiguous in the reconstructed stream. The broker then (logically) recomputes the top-k of the reconstituted stream. This only requires that it computes the top-k among the $W$ events of the two guards at the chunk boundaries; the other events can be forwarded without any further processing. If a broker merges chunks, it can decide on a new chunk size, and it effectively becomes the “publisher” of the chunk.

4.3.2 Buffering

Due to the heterogeneous top-k specifications of the subscriptions (i.e., different window semantics), it is possible for the same publication to be matched at different
times for different subscribers. For instance, it is possible for a publication to be a

guard event for some subscription and later be matched as the top-k of a window for

another subscription.

In such cases, the same publication may be forwarded down a different path

through the overlay network. However, it is possible that the paths to the differ-
ent subscriptions share common brokers. Each broker is equipped with standard

knowledge to understand to which outgoing links a publication can potentially be for-

warded. We require only storage of the standard subscription set and the last hop of

each subscription. This information already exists in the pub/sub broker through the

operation of pub/sub routing (e.g., [45]). If dynamic reconfiguration of the topology

occurs, we assume the brokers eventually receive the local information necessary to

continue functioning.

The brokers can, therefore, determine if a publication needs to be buffered to

serve later requests and propagate the publication down a previously unforwarded

link. This process is called deduplication. Upstream brokers then simply need to tell

the buffered brokers which links they now have to serve. The buffered brokers then

perform rehydration to insert buffered publications back into the stream to be sent

to downstream brokers. If this information is piggybacked on existing messages, we

obtain a net reduction in traffic messages sent.

4.3.3 Covering

The pub/sub concept of subscription covering allows for brokers to stop the prop-

agation of smaller subscriptions whose interest space is covered by a larger one, a

concept formally defined in [78]. Covering raises the scalability of pub/sub systems

by reducing subscription traffic without compromising correctness, since publications
will be matched to every matching subscriptions. Unfortunately, the latter is no longer true when dealing with top-k subscriptions. The top-k publications computed for a covering subscription is based on the content of its windows. However, covered subscriptions will generate different windows due to the count-based semantics since they do not match every publication found in the covering subscription, which will affect the top-k selection, even if the scoring function used is the same for all subscriptions. Therefore, early filtering of low scoring publications will produce incorrect results at downstream brokers where covered subscriptions are processed.

We provide support for covering subscriptions by deferring top-k processing to downstream brokers. Instead, brokers will forward all matching events downstream until the events reach the point of cover, where top-k filtering will then be performed for the covering subscription and those it covers. Within this context, the decision to cover a subscription is regulated by a cost model which evaluates the traffic cost for propagating a covered subscription versus the difference in the cost of disseminating all matching events for the covering subscription instead of top-k publications only.

Given that $d(b_1, b_2)$ is the length of the path from $b_1$ to $b_2$ as measured in number of broker hops between the two brokers, we define two costs as follows:

\[
c_d(s, b_1, b_2) = d(b_1, b_2)
\]

The cost to propagate a subscription $s$ from a broker $b_1$ to another broker $b_2$ is the length of the path from $b_1$ to $b_2$.

\[
c_e(s, b_1, b_2) = |m(s, b_2, b_1) - t(s) - \text{sent}(b_2, b_1)| \times d(b_1, b_2)
\]

The set of additional events to send to $b_1$ from $b_2$ due to subscription $s$ is the set of publications which matches $s$ at $b_2$ to be sent to $b_1$ ($m(s, b_2, b_1)$), minus the
publications which are already selected as top-k for \( s(t(s)) \), as well as the publications which are already sent to satisfy other subscriptions coming from \( b_1 \) (\( sent(b_2, b_1) \)). The cardinality of such set multiplied by the number of hops these publications must travel is the cost to propagate events due to an uncovered subscription \( s \).

Therefore, the cost model evaluates whether:

\[
\omega_1 c_d(s, b_1, b_2) \geq \omega_2 c_e(s, b_1, b_2)
\]

Where \( \omega_1 \) and \( \omega_2 \) are the weight parameters for subscriptions and publications, respectively. If this inequality is true, then the subscription should be covered. Otherwise, it should be propagated.

### 4.4 Chunking and buffering algorithms

The broker pipeline is shown in Figure 4.3. The following high-level procedure is employed: (1) the publisher edge broker \textit{matches} events against subscriptions and performs \textit{chunking}; (2) any broker may optionally perform merging of chunks (\textit{rechunking}). Optionally, any pair of neighbouring brokers may eliminate duplicates. We say that the source broker in this pair \textit{deduplicates} the event stream, and the target broker \textit{rehydrates} the stream.

Note that every broker maintains a queue of incoming events, which are collected and grouped into chunks of windows. Each broker then continuously processes events stored in the queue as it is being filled.

In this section, we present each component of the pipeline. The chunking component has two variants: a synchronous one that queues all events in a chunk and outputs one chunk at a time and an incremental one that outputs the events with
Figure 4.3: Broker pipeline for top-k processing of count-based windows

Matching: For each incoming event $e$, the matching component simply outputs a set of event-subscription pairs for each subscription $s$ that matches $e$. As well, the output event is tagged as a matching event. These event tags are used only in the incremental chunking algorithm.

Synchronous chunking: The synchronous chunking component, used by a publisher edge broker, outputs a chunk for a sequence of incoming events (Algorithm 1). A chunk $c$ has two parts: $c.sub$ refers to the subscription associated with the chunk and $c.evts$ is the sequence of events in the chunk. This component internally maintains a data structure $s.evts$ to store a sequence of events associated with subscription $s$. The $getFirstWindow(evts)$ and $getLastWindow(evts)$ functions return the set of events in the first and last windows in the sequence of events. $getAllWindows(evts)$ returns the set of windows, each containing $W$ events, in the sequence of events. These functions encapsulate the window semantics. Note that the algorithm computes the top-k events for each window within a chunk, even those with events already included in a guard. If a publication is one of the top-k events in a window, it will be tagged as top-k to ensure it will be delivered to the subscriber.

Synchronous rechunking: The synchronous rechunking component merges a set of
Algorithm 1: Synchronous chunking

```plaintext
1 on receiving event e, for all subscriptions s that matches e do
2     s.evts.append(e);
3 on finishing a chunk for subscription s do
4     leftguard ← getFirstWindow(s.evts);
5     rightguard ← getLastWindow(s.evts);
6     topk ← ∅;
7     W ← getAllWindows(s.evts);
8     foreach window w in W do
9         topk ← topk ∪ getTopK(w, k);
10     c.sub ← s;
11     c.evts ← leftguard ∪ topk ∪ rightguard;
12     sort(c.evts);
13     forward(c, s.lasthop);
```

Chunks (Algorithm 2). This component internally maintains a data structure `s.chunks` to store a set of chunks associated with subscription `s`. After concatenating a set of chunks, the algorithm computes the top-k events across the chunk boundaries, namely the windows that span the guard events.

The algorithm does not specify which chunks should be rechunked, the order of these chunks, or when rechunking should occur. These are configuration parameters that can be used to prioritize certain streams, and tradeoff latency for efficiency.

**Incremental chunking:** The incremental chunking component performs chunking but does not wait until the chunk is complete before forwarding events (Algorithm 3). The component indicates which chunk `c` each event belongs to, and also tags outgoing events so that downstream brokers can distinguish between the guards and top-k events. As well, it sends messages to indicate when each chunk begins and ends.

The algorithm forwards the first `W` events as the left guard, and when the chunk ends, the last `W` events are forwarded as the right guard. As each event arrives, it computes a new window (according to the window semantics) and forwards the top-k
Algorithm 2: Synchronous rechunking

```plaintext
1 on receiving chunk c for subscription s do
2     s.chunks.add(c);
3 on performing rechunking of chunks C for subscription s do
4     s.chunks ← s.chunks \ C;
5     all evts ← getEvents(C);
6     left guard ← getFirstWindow(s.evts);
7     right guard ← getLastWindow(s.evts);
8     topk ← ∅;
9     for i ← 1 to |C| − 1 do
10        guard evts ← getRightGuard(C_i) ∪ getLeftGuard(C_{i+1});
11        W ← getAllWindows(guard evts);
12        foreach window w ∈ W do
13            topk ← topk ∪ getTopK(w, k);
14        new chunk.sub ← s;
15        new chunk.evts ← left guard ∪ topk ∪ right guard;
16        forward(new chunk, s.last hop);
```

events that have not already been forwarded.

Internally, the component stores in the s.evts data structure the most recent two disjoint windows of incoming events that match subscription s. We need two windows to ensure there is one full window of events in addition to one window of right guard events in case the chunk ends soon. As well, s.sent evts stores the events the components have already forwarded. s.chunk id is the unique ID of the current chunk for subscription s.

Incremental rechunking: Since we are not allowed to interleave events across chunks, rechunking needs to wait for complete chunks. Therefore, the incremental rechunking algorithm is essentially the same as the synchronous variant but the outgoing events are individually tagged as guard or top-k events.

Dechunking: This component, used by a broker for subscribers directly connected to it, computes the final top-k stream by concatenating chunks, computing and forwarding top-k results over guards and forwarding pre-computed top-k results.
Algorithm 3: Incremental chunking

1 on receive event e matching subscription s do
2     s.evts.add(e);
3     if |s.sentevts| = 0 then
4         // Start a new chunk.
5         s.chunkid ← generateUniqueId() ;
6         send((startchunk, s.chunkid, s), s.lasthop) ;
7     if |s.sentevts| < W then
8         // e is a left guard.
9         s.sentevts.add(e) ;
10        send((eguarded, s.chunkid, s.lasthop) ;
11 else if getNumDisjointWindows(s.evts) < 2 then
12         // Wait for more events.
13 else
14     // Forward top-k events.
15     w ← getFirstWindow(s.evts) ;
16     topk ← getTopK(w, k) ;
17     foreach event e' : e' ∈ topk ∧ e' ∉ s.sentevts do
18         s.sentevts.add(e') ;
19        send((e'topped, s.chunkid, s.lasthop) ;
20     // Expire events.
21     w' ← getLastWindow(s.evts) ;
22     foreach event e' : e' ∈ w ∧ e' ∉ w' do
23         s.evts.remove(e') ;
24        s.sentevts.remove(e') ;
25 on finish chunk for subscription s do
26     w ← getLastWindow(s.evts) ;
27     foreach event e : e ∈ w do
28         send((eguard, s.chunkid, s.lasthop) ;
29        send((endchunk, s.chunkid, s.lasthop) ;
30     s.chunks ← s.chunks \ C ;
31     s.evts ← ∅ ;
32     s.sentevts ← ∅ ;
**Deduplication:** The deduplication and rehydration components operate as a pair to avoid sending duplicate messages between brokers. The deduplication component operates at the tail end of an upstream broker’s pipeline and essentially “compresses” the outgoing chunked event stream. The rehydration component works at the head of a downstream broker’s pipeline and reconstructs the original chunks. To clarify, rehydration restores deduplicated chunks and may be performed at any broker while dechunking refers to the reconstruction of the final top-k results at the subscriber edge.

The deduplication component maintains a history of the events forwarded to each neighbour and never forwards duplicate messages over the same overlay link (Algorithm 4). To ensure the downstream broker can reconstruct the chunk stream, the deduplication component forwards the IDs of the guard events; this is relatively lightweight information but must be done for each chunk. The top-k events that fall between the guards, however, can be inferred by the rehydration component at the downstream broker. The details of this algorithm are described below in the discussion about the rehydration component (see Algorithm 5).

Internally, the deduplication component maintains information about the incoming chunk stream for each subscription $s$. In particular, $s.chunkid$ is the current chunk ID, $s.guardids$ are the event IDs of the guard events, and $s.numTopK$ is the number of top-k events in the chunk.

**Rehydration:** The rehydration component reconstructs the chunk streams (Algorithm 5). For each incoming event, it first finds the matching subscriptions and records the event in a buffer associated with each subscription. This buffer is the sequence of potential events within each subscription’s chunk. The guards within each chunk are explicitly specified by the upstream broker’s deduplication component. To determine the top-k events within each chunk, it waits for the count $n$ of top-k events
Algorithm 4: Deduplication

```plaintext
1 on receive (startchunk, c, s) do
  2 s.chunkStartTime ← NOW;
  3 s.chunkid ← c.id;
  4 s.guardids ← ∅;
  5 s.numTopK ← 0;
6 on receive event e.guard for chunk for subscription s do
  7 sendOnce(e.match, s.lasthop);
  8 s.guardids.add(e.id);
9 on receive event e.topk for chunk for subscription s do
  10 sendOnce(e.match, s.lasthop);
  11 if s.numTopK = 0 then
      // Left guard is done.
  12     send((Lguard, s.guardids, s.chunkid, s), s.lasthop);
  13     s.guardids ← ∅;
  14     s.numTopK ← s.numTopK + 1;
15 on call sendOnce(e, nexthop) do
  16     send((Rguard, s.guardids, s.numTopK, s.chunkid, s), s.lasthop);
17 if e ∉ nexthop.evt then
  18     send(e.match, nexthop);
// Expire old events.
19 oldestChunkStartTime ← min_s(s.chunkStartTime);
20 foreach neighbour n do
  21     n.evt.removeOlderThan(oldestChunkStartTime);
```
within the chunk (sent within the right guard message \(R_{guard}\) from the deduplication phase), and selects the \(n\) highest ranked events in the buffer within the guards.

---

**Algorithm 5: Rehydration**

1. on receive event \(\langle e_{\text{match}} \rangle\) do
2.     foreach subscription \(s\) that matches \(e_{\text{match}}\) do
3.         \(s.\text{evts}.\text{add}(e)\) ;
4. on receive \(\langle L_{guard}, \{e_{\text{Id}}\}, c, s \rangle\) do
5.     send(\(\langle \text{startchunk}, c, s, s.\text{lasthop} \rangle\)) ;
6.     foreach \(id \in \{e_{\text{Id}}\}\) do
7.         \(e \leftarrow s.\text{evts}.\text{get}(id)\) ;
8.         send(\(\langle e_{\text{guard}}, c, s.\text{lasthop} \rangle\)) ;
9.         \(s.\text{evts}.\text{removeOlderThan}(e)\) ;
10. on receive \(\langle R_{guard}, \{e_{\text{Id}}\}, n, c \rangle\) do
11.     guardevents \(\leftarrow \emptyset\) ;
12.     foreach \(id \in \{e_{\text{Id}}\}\) do
13.         \(\text{guardevents} \leftarrow \text{guardevents} \cup s.\text{evts}.\text{get}(id)\) ;
14.         \(\text{potentialtopkevts} \leftarrow s.\text{evts}.\text{getOlderThan}(\text{guardevents})\) ;
15.         \(\text{topkevts} \leftarrow \text{potentialtopkevts}.\text{getTopRanked}(n)\) ; // Compute and send top-\(k\) events.
16.     ;
17.     foreach \(e \in \text{topkevts}\) do
18.         send(\(\langle e_{\text{topk}}, c, s.\text{lasthop} \rangle\)) ;
19.     foreach \(e \in \text{guardevents}\) do
20.         send(\(\langle e_{\text{guard}}, c, s.\text{lasthop} \rangle\)) ; // Send guard events.
21.     ;
22.     \(s.\text{evts}.\text{removeOlderThan}(e)\) ;

---

### 4.5 Chunking correctness

In this section, we prove the correctness of the chunking technique. In particular, we show that the chunking, rechunking, and dechunking components do not violate the top-\(k\) semantics defined in Section 4.2.

Recall that the semantics are defined on a computation of top-\(k\) windows over the stream of events received at a subscriber. Since the network only guarantees
per-publisher ordering of messages, and messages may be arbitrarily delayed, there are many possible interleavings of events at the subscriber.

With chunking, top-k computations are performed on a partial stream of events before they arrive at the subscriber and in fact, before it is even known what the complete event stream at the subscriber will look like. The proof below will argue that performing top-k computations on these partial streams is equivalent to a top-k computation over a stream that represents a possible interleaving of events at the subscriber.

The proof exploits the requirement that chunks must have \( n\delta \) events, where \( \delta \) is the window shift size, and \( n > 0 \). Each chunk then has \( n \) windows. Windows whose events all appear within the chunk are referred to as intra-chunk windows, and windows whose right boundaries extend beyond the chunk are called inter-chunk windows. Note that a chunk can have at most \( \lceil W/\delta - 1 \rceil \) inter-chunk windows, which must all appear as a contiguous set at the end of the chunk. In all the notation below, items are numbered from one: the first event in a chunk or window is \( e_1 \), and the first window in a chunk is \( W_1 \).

**Lemma 1.** The interleaving of chunks will reconstruct a possible interleaving of events.

*Proof.* From our assumptions, the network preserves per-publisher message order, but may arbitrarily delay messages, so it remains to show that the interleaving of chunks preserves per-publisher ordering.

Consider the concatenation of two chunks \( C \) and \( C' \), which generates a reconstructed stream of events that contains two events \( e \) and \( e' \) from the same publisher, such that \( e \) was published before \( e' \).

Now, if \( e \) and \( e' \) belong to the same chunk, then \( e \) must appear before \( e' \), since
event ordering is preserved within a chunk. On the other hand, suppose without loss of generality that \( e \) belongs to chunk \( C \), and \( e' \) to \( C' \). Since \( e \) was published before \( e' \), chunk \( C \) must have been generated before \( C' \). For a broker to concatenate \( C' \) before \( C \), either it received \( C' \) and \( C \) from different brokers, or it received \( C' \) before \( C \) from the same broker. The former case is impossible since \( e \) and \( e' \) originated at a single publisher, and by the acyclicity of the network, there must be a single path from this publisher to the current broker. The latter case is also impossible: Since \( e \) was published before \( e' \), chunk \( C \) must have been generated before \( C' \). Given that \( C \) and \( C' \) traversed the same path to the current broker, and brokers along the path process chunks for a given subscription from the same upstream broker in FIFO order, a broker must receive \( C \) before \( C' \).

Therefore, \( e \) must appear before \( e' \) in the reconstructed stream, preserving per-publisher ordering.

Lemma 2. The windows considered over the events within a chunk correspond to windows in the reconstructed stream of events.

Proof. Let chunk \( C \) contain a stream of \( n\delta \) events \( e_1, \ldots, e_{n\delta} \) and chunk \( C' \) contain a stream of \( m\delta \) events \( e'_1, \ldots, e'_{m\delta} \), where \( \delta \) is the window shift size, and \( n \) and \( m \) are positive integers. By concatenating chunks \( C \) and \( C' \), we get the reconstructed stream of events \( e_1, \ldots, e_{n\delta}, e_{n\delta+1}, \ldots, e_{n\delta+m\delta+1} \), where for convenience event \( e_{n\delta+i} \) is an alias for \( e'_i \) for \( i > 0 \).

There are \( n \) windows on chunk \( C \), where the first event in window \( W_i \) is event \( e_{(i-1)\delta+1} \), for \( 0 < i \leq n \). Likewise, there are \( m \) windows on chunk \( C' \), where the first event in window \( W'_j \) is event \( e'_{(j-1)\delta+1} \), for \( 0 < j \leq m \). The reconstructed stream, having \( (n + m)\delta \) events, has \( n + m \) windows, where the first event in window \( W''_k \) is event \( e_{(k-1)\delta+1} \), for \( 0 < i \leq n + m \).
We now show that each window $W_k''$ on the reconstructed stream corresponds to a window in chunk $C$ or $C'$, that is, $W_i$ or $W'_j$, respectively.

In chunk $C$, window $W_i$ starts at event $e_{(i-1)\delta+1}$, and hence corresponds to window $W_i''$ which also starts at the same event according to the definitions of $W_i$ and $W_k''$ above. By a similar argument, for chunk $C'$, window $W'_j$ starts at event $e'_{(j-1)\delta+1} = e_{(n+j-1)\delta+1}$, and hence corresponds to window $W''_{n+j}$.

Since, each window in each chunk corresponds to a window in the reconstructed stream, and the number of windows in the reconstructed stream is the sum of the number of windows in the two chunks, each window in the reconstructed stream corresponds to a window in one of the two chunks.

The above reasoning can be applied to any pair of chunks, and transitively the argument can be extended to an arbitrary number of chunks.

**Lemma 3.** The top-$k$ events computed over intra-chunk windows are the top-$k$ events in the reconstructed stream. Similarly, events that are not in the top-$k$ in intra-chunk windows are not among the top-$k$ events in the reconstructed stream.

**Proof.** From Lemma 2, we know that intra-chunk windows in a chunk correspond to windows in the reconstructed stream. Moreover, since intra-chunk windows always contain $W$ events, the top-$k$ events found over these windows exactly match those in the reconstructed stream.

**Lemma 4.** All events contained in inter-chunk windows are guard events.

**Proof.** In a chunk with $p$ events, for window $W_i$ to be an inter-chunk window, the first event in the window must be event $e_j$ where $p - W + 1 < j \leq p$. Otherwise, if $j < p - W + 1$, the last event in the window will be $e_{j+W-1}$ which appears in the chunk, and hence $W_i$ will not be an inter-chunk window.
We require that the first and last $W - 1$ events in a chunk be guard events, so all events in $W_i$ will be guard events.

Theorem 5. The distributed chunking algorithms (Algorithm 1 and 3) compute top-$k$ results that are possible were top-$k$ computed locally over the stream of events received at the subscriber.

Proof. From Lemmas 1 and 2, we know that the windows computed over the chunks are aligned with windows over a possible stream of events that a subscriber may observe. From Lemma 3, we know that the top-$k$ results computed over intra-chunk windows are correct. Finally, by Lemma 4, we know that the remaining (inter-chunk) windows consist of guard events that are propagated to the subscriber, where the inter-chunk window top-$k$ results can be computed.

4.6 Evaluation

We experimentally evaluate our various count-based algorithm implementations. Our experiments are divided into two parts. The first part employs a synthetic workload and contains a performance analysis of the following algorithms: (1) baseline where all computations are performed at the subscriber’s edge broker (EDGE), (2) incremental chunking (CHUNK), (3) incremental chunking with the buffering deduplication/rehydration option (B-CHUNK). The baseline comparison establishes B-CHUNK as the superior solution and is the subject of a further sensitivity analysis. The second part evaluates the B-CHUNK algorithm within the context of the online social network use case and is performed using datasets extracted from real applications.
4.6.1 Performance and sensitivity analysis

**Setup:** The algorithms are implemented in Java for the PADRES pub/sub prototype\(^2\). Experiments are conducted on the SciNet testbed in the General Purpose Cluster (GPC) using 24 machines\(^3\).

The workload used for the baseline comparison and the sensitivity analysis is synthetic: Publishers send data every 4 seconds. To maximize the top-k overhead at that scale (worst case scenario), the subscription predicates are identical and match every publication sent. However, the scoring function associated with each subscription can differ, as described below. The workload for the online social networks use case is described separately in Section 4.6.4.

The overlay topology consists of several core brokers connected in a chain (see Figure 4.4). In addition, these core brokers are each connected to 5 additional edge brokers. Publishers are located on edge brokers called source brokers, while subscribers are uniformly distributed on the remaining brokers. This setup models a network of data centers connected through gateways and focuses on measuring the impact of our algorithms on delivery paths with multiple broker hops.

A score is assigned to each publication for each subscriber, generated using a certain distribution function. The top-k consists of the k publications with the highest

\(^2\)http://padres.msrg.toronto.edu/
\(^3\)http://www.scinet.utoronto.ca/
scores within the current window. Since we are focused on the dissemination aspect, we minimize the overhead of the scoring function. Thus, the overhead of the scoring function is not a bottleneck in our experiments, but could be a factor if a real function was used. We also employ a deterministic scoring function (DET) which assigns strictly decreasing scores over time. This deterministic scoring function is used when we want different subscribers to assign the same score to the same publications.

**Parameters:** The various window related parameters, such as $W$, $k$ and $\delta$ vary during the experiments. In particular, we are interested in a sliding window ($\delta = 1$) and a tumbling window ($\delta = W$). We also vary the number of subscribers, publishers, and the top-k distribution. In a uniform distribution, each publication is equally likely to be part of the top-k of a subscription, whereas in a Zipfian distribution, the top-k of each subscription will overlap more. Finally, we also vary the publication delay. In a mixed setup, publications are injected as subscribers join the system. In a 2-phase setup (2P), subscriptions are first submitted before publications start to flow.

**Metrics:** Outgoing traffic — The main advantage of early top-k matching is to allow unselected publications to be discarded early, which reduces the amount of traffic in the system. We are therefore interested in the traffic reduction of our distributed solutions.

End-to-end latency — Although the matching overhead associated with top-k computation is negligible in our evaluation, publications can be deferred until they are selected as part of the top-k. This delay is dependent upon the window parameters used. We measure the end-to-end latency at the subscribers that are the furthest away from the publishers. In our topology, publishers are 6 broker hops away from the measured subscribers. The latency between any pair of brokers is constant.
4.6.2 Performance comparison

We measure the performance of the various solutions presented over an increasing number of subscribers. Outgoing traffic has been measured at different points in the network, namely at the source (publisher) broker, core brokers and edge (subscriber) brokers. We employ a single publisher and each subscriber is interested in the top-k of every publication published using a deterministic scoring function. We use \( k = 1 \), \( W = \delta = 20 \), and \( C = 100 \) (chunk size). These top-k parameters are applied to every subscription.

Figure 4.5(a) shows the relative traffic at the publisher broker of each solution, normalized relative to the EDGE solution. This solution always forwards every publication received since no top-k processing occurs at the source. The chunking solution without buffering performs top-k filtering at the source; however, the top-k publications are sent separately for each subscription. Thus, a publication could be sent multiple times over the same link. This duplication means the traffic increases linearly with the number of subscribers. In our experiments, the traffic of CHUNK starts at an order of magnitude higher to that of EDGE. In fact, the traffic is saturated at around 500 subscribers. With buffering enabled, the chunking solution (B-CHUNK) scales much better than the centralized solution. B-CHUNK shows a constant 57%
traffic reduction with varying subscriber loads. In our setup, a large portion of the remaining traffic is used to forward guard publications.

At the subscriber brokers, the traffic results are straightforward: Each solution presents a 95% reduction. Since the selectivity of the top-k filter is 1 out of 20, only 5% of the publications are ultimately sent to the clients. No further processing is required by the clients themselves: The processed stream sent by the edge broker to a client corresponds exactly to the final result. Therefore, all three solutions are able to produce that stream at the edge broker.

Compared to EDGE, core brokers will receive more inbound traffic for CHUNK, since the source broker generates more outgoing traffic. Each publication received is then forwarded down a single link to the corresponding subscriber. Thus, the outgoing traffic is exactly equal to the inbound traffic. Since core brokers have six outgoing edges, each publication is forwarded to only 16.67% of the links. However, the duplication of publications at the source broker causes core brokers to forward a publication down a specific link multiple times to satisfy different subscribers. Therefore, CHUNK generates more traffic at the core brokers than EDGE, even if individual inbound publications are forwarded down only a single link. For B-CHUNK, every publication received is forwarded down every link, since every subscriber uses the same scoring function. The reduction in traffic at core brokers comes only from the fact that there is less inbound traffic from the source broker.

We compare the end-to-end latency between the various solutions (see Figure 4.5(b)). Because top-k is a stream processing operation, publications can sit in a queue waiting for the window to fill up. In a count-based solution, the delay experienced due to queuing can vary widely between individual publications. In our experiments, we control the variation by using a tumbling window with deterministic scoring. A single publication is sent for every 20 publications received. The sent publication is the
one that entered the window first, since publications have monotonically decreasing scores.

For CHUNK, the latency is dependent on the number of publications and performs much worse than other solutions. This is because that traffic is duplicated for each subscriber. For B-CHUNK and EDGE, the latency is dominated by the queuing time (as seen by the difference when no top-k is used). The queuing time is mostly dependent upon three factors: the scoring function used, the window size, and the publication rate. All three of these factors affect both solutions equally, hence, there is no advantage in choosing one solution over another with respect to latency. Thus, the overall latency is comparable between B-CHUNK and EDGE. The difference is that B-CHUNK queues publications mostly at the publisher brokers, while EDGE queues at the subscribers’ edges.

As explained before, the scoring function used in the experiment incurs minimal overhead. Therefore, it does not factor in our latency measurement. However, the matching overhead is distributed differently depending on the solution used. In B-CHUNK, the source brokers must process a single publisher stream for multiple subscribers, while in EDGE, the opposite is true. Thus, a workload that contains more publishers than subscribers will distribute the matching overhead more evenly.
using B-CHUNK.

**Summary:** The CHUNK solution is simply not scalable due to the publication duplication issue and always performs worse than the EDGE solution in both traffic and latency, even for small number of subscribers. B-CHUNK demonstrates lower traffic than EDGE due to the source filtering of top-k publications. Latency for both solutions is comparable since it is dominated by the queuing time, which is the same.

### 4.6.3 Sensitivity analysis

The parameters used in Section 4.6.2 can be considered optimal in terms of traffic reduction for the B-CHUNK algorithm. We now evaluate the impact of each parameter and explain its effect within the context of our solution.

**Shift:** The shift parameter has a significant impact on the traffic at the core brokers. With a tumbling window, the traffic flows at a constant rate to the subscriber: For every $W$ publications, $k$ publications are sent to the subscriber. For a sliding window, the rate can vary depending on the actual score of the publications. In the best case, a publication can dominate up to $W - 1$ publications. For instance, for $k = 1$, a publication $p$ that has a higher score than the next $W - 1$ publications following it triggers only one message (used to forward $p$). It is only when that
publication falls off the window (at the $W^{th}$ publication following $p$) that a new top-k publication is selected. In the worst case, a publication can be sent out for every incoming publication if the top-k is always at the head of the window. At every shift, one of the current top-k publications fall off the window and is replaced by another which must be forwarded. In this case, once the window is initially filled, maximal traffic overhead is achieved: Every incoming publication generates an outgoing publication.

We evaluate the impact of the shift in the worst case scenario, where the scoring function used generates progressively decreasing scores. Figure 4.7(a) shows the impact of the shift parameter on traffic. With a window size of 20, the first data point corresponds to a sliding window shift of 1 (ratio of 0.05). Except for the first window of each chunk, the communication overhead is maximized and there is no traffic reduction in those subsequent windows. At the other extreme, the tumbling window (ratio of 1.0) provides the best traffic reduction at 43% of the baseline traffic at the source broker.

Outgoing traffic is inversely proportional to latency (see Figure 4.7(b)). A smaller shift will create a more frequent publication rate, which minimizes the queuing time. A longer shift empties the window more quickly, which delays the next top-k computation until the buffer is filled. We observe that this particular tradeoff is not specific to our solution, but rather comes from the window semantics themselves. A workload with decreasing publication scores is sensitive to the traffic issue in sliding windows, but will result in shorter publication delays.

**Chunk size:** The chunk parameter has a major impact on both traffic and latency. Figure 4.8(a) shows that increasing the chunk size reduces traffic, since the size of the guards depends only on the window size. We are therefore increasing the intra-chunk top-k zone where only top-k publications are forwarded.
However, we can observe a tradeoff between traffic and latency only if the number of publishers change. If only a single publisher exists, increasing the chunk size will not affect latency. This is because all publications will be part of sequential chunks that will be processed immediately, since there are no concurrent chunks. However, as soon as we have 2 or more publishers, the increase in latency is linear to that of the chunk size. Publications from different chunks must be queued at the subscriber’s edge broker until the current chunk is finished. A larger chunk size would therefore introduce a longer queuing time on publications of other chunks.

Interestingly, the number of publishers, beyond the initial two needed to introduce concurrently filled chunks, does not affect latency. This is because our algorithm selects filled up chunks before processing incomplete chunks. Multiple chunks can therefore be filled up concurrently and will be processed quickly as soon as the current chunk is done. The overhead impact on each queued chunk corresponds to the time needed to receive the current chunk. This indicates that the latency is bounded by the slowest chunk to finish in the system. Our adaptive rechunking technique can however detect such slow chunks and automatically close them to allow completed chunks to be processed quickly and reduce the queue size.

**Publication delay:** We evaluate B-CHUNK with varying publication delay.
Publication delay refers to the time elapsed before publications are released in the system. In a mixed setup, publishers start their publications while subscribers join the system. This means that earlier subscribers will receive more publications than those who join later. In a 2-phase setup, we first disseminate all the subscriptions before sending publications. Subscribers thus receive the same stream of publications when using the 2-phase setup in conjunction with the deterministic scoring function.

Publication delay has a major impact on the communication overhead of our B-CHUNK solution. This is due to the fact that publication forwarding is a binary decision: A broker must forward a publication down a certain link if at least one subscriber requires this publication at that path. For the source brokers, this means a publication can be dropped only if all subscribers do not require that publication. This occurs when the publication is not selected as part of any top-k window of any subscribers and the publication is not part of any guards of any chunks. In a mixed setup, the subscriptions will start their first chunk at different times, which correspond to the first publication each subscription matches. This means that chunks belonging to different subscriptions can offset one another such that guards for one subscription will overlap into the top-k zone of another subscription. When this happens, publications in that overlap must be forwarded by the source broker to satisfy the guard requirements, even if they are not part of any top-k window computed by other subscriptions. This is illustrated in Figure 4.6. The source broker can only drop publications that are located only in the effective top-k zone, where no subscription is in a guard. In the worst case, a publication stream can be completely covered by guard zones, in which case every publication must be forwarded regardless of the actual top-k computations being performed. In the best case, guard zones are completely disjoint with each others’ top-k zones, which maximizes the effectiveness of the top-k computations.
In our mixed setup, subscriptions are submitted every second (publication rate is once per 4 seconds). At the source broker, the traffic overhead is already degraded after 50 subscribers and is saturated at 100 subscribers and more (see Figure 4.9(a)). At the core brokers (Figure 4.9(b)), there is stable traffic reduction of 20%. Since core brokers have multiple outgoing edges, it is less likely for subscriptions down a certain link to offset each other. In our solution, there are 50 subscribers per outgoing edge, which is not enough to saturate the core brokers, even in a mixed setup.

Nevertheless, our solution remains sensitive to offsets in subscription chunks due to the presence of guard zones which can nullify top-k filtering. It is therefore particularly useful to employ rechunking techniques to synchronize chunks such that their guards overlap each other, maximizing the effective top-k zones.

**Scoring function:** We evaluate the impact of the scoring function used. In our random selection tests, subscribers select different top-k streams according to the score they individually assign to each publication. Again due to the binary nature of publication matching, the traffic overhead of our solution is sensitive to the scoring function used. Scoring functions that produce disjoint sets of top-k publications result in a higher traffic since publications matching at least one subscriber must be forwarded. In contrast, similar scoring functions can be leveraged to reduce traffic by
forwarding publications which will match multiple top-k windows.

We simulate subscriptions with similar interests using a Zipfian scoring function. A random global score is assigned to each publication and each subscription derives its personal score for the subscription by modifying the global score with a random value chosen from a Zipfian distribution. Thus, subscriptions will have similar scores for each publication. In contrast, the uniformly random scoring function assigns an arbitrary value for each publication per subscription.

Figure 4.10(a) shows the relative communication overhead of the various scoring functions at the source broker. The deterministic scoring function is the most efficient since all the subscribers’ top-k publications are the same. Zipfian scoring provides some clustering of top-k publications, which provides a 17% traffic reduction (with respect to EDGE) at 100 subscriptions. However, the outgoing traffic saturates beyond 400 subscribers. The uniform scoring function does not provide any benefit at the source broker, even with a small number of subscriptions.

At the core brokers, we observe that the traffic overhead stays constant with respect to the number of subscribers in all cases (see Figure 4.10(b)). This is due to the fact that each outgoing link for a core broker has at most 50 subscribers. Therefore, a core broker can safely not forward a publication down a specific link if
none of those 50 subscribers require it. This is advantageous for the Zipfian scoring function, which provides a sizable reduction in traffic when the downstream link has a small number of subscribers (e.g., 50).

In general, subscriptions with similar scoring functions should be clustered together. While the source brokers will not benefit from such clustering, core brokers will as they skip forwarding down unmatched links.

**Number of publishers:** We increase the number of publishers and test various deployment strategies. In a *cluster* strategy, we increase the number of publishers at a single source broker, whereas in a *uniform* strategy, the publishers are uniformly distributed around the topology. We keep the rate of publications constant (15 pubs/min), regardless of the number of publishers.

We first observe that the various algorithms are not sensitive to the number of publishers with regards to traffic. Using the same setup as the performance evaluation, but varying the number of publishers, the same pattern is followed: B-CHUNK uses 43% of the original traffic at the source broker, while CHUNK performs increasingly worse as the number of subscribers increase. This pattern holds true no matter how the publishers are distributed.

This result is justified by the distributed and decoupled nature of our B-CHUNK algorithm. When multiple publishers are connected to the same source broker, all publishers’ event streams are treated together as a single merged event stream for that source broker. Because the scoring function is deterministic and is based on the order of publications, the top-k computation produces the same results. When multiple publishers are connected to different brokers, each broker locally performs top-k computations independently. Thus, they will behave and perform identically.

For end-to-end delay, we find that the placement strategy used, rather than the actual number of publishers, is a major factor. Figure 4.11 shows that in a clustered
strategy, the latency remains constant over an increasing number of publishers. This is due to the decoupled property mentioned previously: The output events from the different publishers are merged into a single input stream for the source broker, which is no different from a single publisher publishing at an aggregated rate. When publishers are located on different source brokers (uniform strategy), the latency grows exponentially in the number of publishers. Because the publication rate is kept constant, a greater number of publishers also implies that each publisher will publish at a reduced rate. Since each publisher is publishing at a different broker, each window fills up at a slower rate, thus increasing the queuing time for publications. Furthermore, the edge broker at a subscriber’s end will process one chunk at a time, originating from a single source broker. The edge broker will therefore select one chunk to process and buffer the other chunks. Since each source broker is filling its chunk concurrently, as soon as the selected chunk is completed, the edge broker will quickly process the completely buffered chunks before selecting a new chunk among the batch of newly formed chunks. This means the queuing time of publications in buffered chunks also includes the time required to finish the currently selected chunk. This phenomenon motivates the use of publisher clustering techniques.

Summary: The shift parameter, the chunk size, the publication delay and the scoring function are three major performance factors for B-CHUNK. Selecting a small
shift value can lead to high traffic overhead in certain workloads. However, one must be aware of the traffic-latency tradeoff surrounding the shift parameter. A larger chunk size will increase the performance of the system at the cost of increased latency in concurrent setups. Offset chunk guards can dominate top-k filtering and saturate the traffic. Without chunk synchronization, a mixed workload will not scale as the number of subscriptions increases and the likelihood of guards offset to occur. Synchronizing chunks is necessary in order to correct the offsets. The scoring function can vary the amount of overlap between the top-k sets of the subscriptions. As the number of subscribers increase, the number of publications to be forwarded is likely to increase. Eventually, every publication must be forwarded by the publishers. However, clustering subscriptions with similar scoring functions in the same parts of the overlay will yield benefits for core brokers with multiple links.

With regards to latency, the chunking solutions are sensitive to the number of source brokers and not the number of publishers. As the number of source brokers increases, the number of concurrent chunks also increases, with subscribers only able to process one chunk at a time. This provides an incentive to cluster publishers to reduce the number of source brokers, as well as reducing the chunk size to allow the subscribers to switch between chunks faster. Clustering has been studied in the context of social networks [102] and would therefore synergize well with our solution.

\subsection{Online social networks use case}

We evaluated our solution in the context of online social networks. We construct workloads which are based on public datasets from Facebook [122] and Twitter [73].

**Subscriptions:** Subscriptions are built based on the social relationships between the different users in the network. In Facebook, the \textit{friendship} relation is modeled as
a pair of subscriptions which subscribes one user to another and vice-versa (e.g., a user subscribes to the status updates of a friend). In Twitter, the follow relation is modeled as a subscription between a follower to a followee. All these subscriptions are topic-based: Subscribers are interested in all publications posted by a user. From the original datasets, we extracted a sample set that contained subscriptions for 1000 users, using the technique employed in [112]. The samples retain the original properties of the original datasets, specifically with regards to the relative popularity of the topics. Figure 4.12 shows that the popularity of both datasets exhibit a long tail, with the Twitter tail having more weight than that of Facebook. The Facebook and Twitter samples contain 5K and 30K subscriptions, respectively. For each user, we combine all subscriptions together to form one subscription spanning the various topics this user is interested in. This allows our pub/sub system to perform top-k over the entire stream of matching publications for that given user, rather than compute the top-k publications over each topic separately. The intended purpose of our solution is to provide subscribers with a top-k selection of the most relevant publications among all the topics this user is subscribed to. Thus, our sample actually contains only 1K subscriptions, one for each user.

**Publications:** Publications are generated synthetically at a fixed rate of one per second. This fixed rate models the global activity of the social network, not of
each individual user. Furthermore, our count-based semantics are not sensitive to the rate of publications. The publications represent tweets (Twitter) and status updates (Facebook) posted by users (topics) which must be disseminated to all subscribers of the user (followers and friends). The publications are assigned a topic selected separately for each publication. According to [59], there is a correlation between user popularity and activity. We take this into account by skewing topic selection towards popular users: The probability of a topic being selected for a publication is linearly dependent on the topic popularity.

**Scoring function:** Publications are scored according to the popularity of its source (namely, its topic). A publication coming from a popular topic will gather a higher score than an unpopular topic and thus be more likely to be part of the top-k scores for a window. We argue that since our sample subscriptions come from a cluster of linked users, this popularity-based scoring function is akin to collaborative filtering, which is employed in social networks.

**Topology:** We use a topology similar to the previous experiments, with edge brokers connected to a chain of core brokers, which is modeled after data center networks. Each edge broker is connected to a single publisher. We argue that in a real social network cluster, state updates are handled by an external service which is logically represented as a single source of data for our pub/sub system. Each edge broker is connected to a single subscriber, who is in charge of submitting subscriptions for several users. The subscriptions are allocated randomly to each subscriber. Each pub/sub subscriber logically represents a collection of users that reside on the same machine. Furthermore, even though the subscriptions are assigned randomly, the subscriptions in the sample data are closely related and follow the aforementioned long tail distribution: Subscriptions residing on the same physical machine have similar interests, which adheres to the clustering techniques prescribed by [102].
ing the interest of multiple users to a single subscriber is an effective technique for scaling up a pub/sub-based social network dissemination middleware to the required specifications.

**Top-k parameters:** We use \( k = 2, W = 20, \delta = 20 \) (tumbling window), and \( C = 100 \) (chunk size), with a 2-phase setup. The algorithm employed is B-CHUNK (chunking with deduplication/rehydration). We justify our use of tumbling windows to control the selectivity of the subscription and produce a steady output of notifications for the user to receive which is more intuitive to process.

**Performance evaluation:** Figure 4.13(a) shows the normalized traffic of the B-CHUNK solution over the centralized baseline for varying number of subscriptions at the publisher edge broker. For Facebook, the reduction gain starts at 27.6% for 100 subscribers and deteriorates to 6.7% for 1000 subscribers, while in Twitter the reduction starts at 16.9% and drops to 0.5%. Two main properties of the workload justify these results.

First, the subscriptions have varying degrees of overlap in the publication space. This means that the windows of different subscriptions progress at different rates since they are not filled by every published event (as in our sensitivity analysis). This causes offsets in the chunks and reduces the size of the effective top-k zone, as described earlier. Due to this phenomenon, the results are worse than during the performance evaluation using synthetic data. Synchronizing guards is therefore a priority since it has practical benefits in real workloads. This can be achieved by delaying the start period of incoming subscriptions so that their first window start collecting publications in line with existing subscriptions.

Second, the skew towards popular subscriptions is beneficial to our solution. The publication workload contains a higher frequency of popular publications, which are matching a large number of subscriptions. Therefore, it is likely that for subscribers to
share publications in popular topics across all windows. Due to our scoring functions, these publications are then able to “overshadow” any lower ranking publications and be selected as part of top-k. Thus, any subscriptions that overlap in those popular topics will likely obtain the same top-k results. This allows the publisher broker to safely filter out a large volume of publications very quickly and only keep the higher ranking ones that appear in the top-k list of a large number of subscriptions. It also follows that having a long tail, as exhibited by social network workloads, is a benefit as it allows the solution to “cut” the tail at the publisher broker and keep only a smaller core of publications. This is demonstrated in the results, where the Facebook experiments perform better than Twitter due to their lighter tail. Due to this positive effect, our results are better than the measurements obtained during the publication delay analysis, even though we are still subject to the chunk offset issue.

The positive gain of the popularity skew is more apparent in the core broker performance (see Figure 4.13(b)). Because the core brokers have multiple outgoing hops, the chunk offset occurs only between subscriptions from the same hop. The problem is therefore lessened, and we see the traffic reduction jumps to 48% and 38% at 1000 subscribers for Facebook and Twitter. This means our popularity-based social network experiment outperforms the Zipfian scoring function used in the scoring sensitivity analysis and is closer to the uniform scoring performance.

**Scalability tests:** We also evaluate the impact of the number of brokers on traffic reduction. We extend the topology to 96 and 960 brokers while maintaining the degree of each node constant. For instance, a 960 broker topology consists of 160 core brokers, each connected to 5 edge brokers. For the publisher edge broker, we find no impact from the number of brokers in the system. The traffic reduction is the same: 0.5% for Twitter and 6.7% for Facebook at 1000 subscribers. This is because the algorithm is lightweight: The publisher edge broker computes the top-k of each
subscriber locally and forwards the resulting publications downstream, irrespective of the rest of the topology.

For the core brokers, we find the size of the topology to have no impact on the performance either, other than the fact that the same number of subscriptions is now spread out on a larger number of edge brokers. Thus, the first core broker directly attached to the publisher edge broker is in charge of disseminating for all 1000 subscriptions, and will therefore exhibit traffic reduction in the range of 48% and 38% for Facebook and Twitter, as per Figure 4.13(b). Core brokers towards the end of the chain have a smaller number of subscriptions to disseminate to, and reduce traffic at a higher rate.

**Summary:** The popularity-based scoring function and publication workload, coupled with the heavy tail popularity distribution of topics in social networks, creates large overlaps in the top-k results of the subscribers which increases the traffic reduction capabilities of our solution, especially apparent at the core brokers. On the other hand, the uneven overlaps between the subscriptions create offsets in the chunks which minimize the effective top-k zone, which indicates the solution can benefit from chunk synchronization optimizations.

The size of the topology has no direct negative impact on the performance of the
solution. The benefit gained by increasing the number of brokers is to distribute the subscription load onto a larger number of edge brokers, which is the main factor for our solution. We envision our pub/sub model to scale adequately through partitioning of the social networks according to interests [102].

4.7 Conclusions

We have developed a distributed ranked data dissemination algorithm. The solution performs aggressive top-k filtering closer to the sources within an overlay network. The resulting top-k streams are propagated and recombined by downstream nodes. Early filtering discards low ranking data, thus saving traffic over a centralized solution at the client end. The solution uses a chunking algorithm which switches between full forwarding of a sequence of events and selective forwarding of top-k events. This chunking algorithm ensures correctness by providing enough data to reconstruct a possible interleaving of the original event stream.

We have implemented our solution within the context of pub/sub overlay networks. Brokers closer to publishers are responsible for the top-k computation while brokers closer to subscribers recombine the chunks. Our evaluation shows that we obtain significant traffic reduction in the system compared to the centralized solution, while maintaining comparable end-to-end latency. A sensitivity analysis reveals the importance of rechunking and clustering to maximize the performance of our solution. We also show how workloads can be modeled from the social network use case to exhibit favorable properties for our solution in terms of scalability and efficiency.
Chapter 5

Top-k subscription filtering

The content of this chapter concerns top-k subscription filtering. Section 5.1 presents various motivating use cases for this work. Section 5.2 provides in details the semantics of top-k subscription filtering in the context of content-based publish/subscribe. Section 5.3 describes our proposed algorithms, including the fair extension. Section 5.4 evaluates our solutions with respect to covering effectiveness and fairness. A conclusion is offered in Section 5.5.

5.1 Introduction

The exponential increase in the data volume and velocity is an undisputed trend. Today, data is being generated at an astonishing rate of 2.5 billion gigabytes daily. The explosion is partly due to the Internet of Things (IoT) that increasingly connects data sources (including objects and devices) to form a complex network, one which is expected to exceed one trillion nodes by 2015 [2]. Data is now being viewed as a new natural resource in the enterprise world with an unprecedented degree of heterogeneity and countless possible ways of aggregating and consuming it [2].
Given the sheer size of the data, new challenges arise for distributed dissemination and consumption of data. Therefore, the ability to deliver only pertinent data to selected and interested consumers is an important requirement that exists in many domains ranging from social platforms to enterprise infrastructure.

In the social space, people are engaging on more platforms (e.g., Twitter and Facebook) and directly connecting to a larger audience that results in receiving a greater volume of data. Thus, the underlying system must provide a way to deliver only highly selective data to interested users. A piece of information may have a different face value for different users. The notion of relevance is thus essential: for instance, one person may be interested in following political figures while another may value arts and theater-related content.

From the perspective of the data consumer, there is a limit on the amount of information that can be consumed before overwhelming the user. Thus, the ability to filter data is a must by allowing users to subscribe only to the data of interest. From the perspective of the data producer, there also exist limited resources in terms of communication bandwidth for data distribution. Additionally, the provider may wish to limit the size of the audience to whom data is delivered. For example, promotional offers or coupons are limited and need to be delivered only to the most important individuals, that is, those who are most likely to be interested in the offer or redeem the coupon. Selecting the matching individuals may be based on a relevance ranking along with other objectives such as diversity: the ability to reach out to individuals in a wide variety of demographics. Alternatively, the filtering may be motivated by fairness: every individual must have an equal chance of being informed about new data, irrespective of the size and properties of his or her demographic.

In the enterprise domain, the data dissemination challenges are even more pronounced: consider the ever-growing size of data centers and cloud infrastructures
with Google operating at the million servers scale [75]. Soon, one quarter of all the world’s applications will be running on the data centers of cloud providers [2]. At this level, cloud providers offering hardware and software as services must be able to manage and distribute user-software patches (e.g., application patches, user configurations, or elastic scaling requirements) and system-specific software and security patches (e.g., OS- or VM-related patches). The situation further worsens when taking into account the complexity and size of cloud network topologies that consist of thousands of heterogeneous nodes with a wide range of required SLAs.

We are now again faced with the same challenge of efficient information dissemination (e.g., software patches) to interested consumers (e.g., servers) with well-defined ranking semantics such as relevance, diversity, and fairness. The basic notion of relevance is unavoidable since not every server has to be notified about every software patch, e.g., a server running Windows is not interested in Linux patches. However, it is insufficient for operating a large-scale data center. For instance, a system administrator may want to assess the total risk of releasing a data center-wide security patch; therefore, the platform must support delivering the patch to only a representative sample of all nodes—a requirement satisfied by diversity delivery semantics.\(^1\) Alternatively, important time-consuming system upgrades need to be rolled out to all nodes using a staged deployment: fairness must be respected when selecting the initial set of nodes. Otherwise, situations may arise such that nodes dedicated to certain clients are favored over others.

What is common in both scenarios discussed above is the existence of a complex network of interconnected consumers (i.e., subscribers) and producers of data (i.e., publishers). Furthermore, there is a rising trend to establish a decentralized architec-

\(^1\)Such a semantic could be useful also for incrementally distributing software patches in a diverse manner.
Chapter 5. Top-k subscription filtering

...ture to cope with the data volume and velocity. The pub/sub model, known for its scalability and decoupled nature, is a logical substrate candidate for distributed data dissemination [124, 4, 41, 21, 120, 48, 107, 32].

The problem addressed in this chapter is to efficiently deliver each publication to the $k$ highest ranked matching subscriptions over a pub/sub broker network. Subscription rankings can be based on a variety of criteria including a scoring function (relevance), fairness, diversity, or a combination of these criteria. In addition, this chapter considers the case where it is the publisher that regulates how widely a publication is disseminated by selecting the value of $k$.

Note that the problem of selecting top-k publications, previously addressed in Chapter 4, is complementary to the work in this chapter. Both problems differ significantly since top-k subscriptions is a stateless operation, in the sense that the operation is performed on a per-publication basis. On the other hand, top-k publications is a stateful operation which requires the stream of publications to be stored in windows of events which are evaluated only when said windows are full.

We remark that there is a requirement for the underlying pub/sub system to employ optimized routing protocols and efficient filtering mechanisms for disseminating information flowing from data producers to data consumers. A well-known routing optimization prevalent in pub/sub systems is covering [88]. This protocol reduces the propagation and management of subscriptions that are forwarded to brokers (i.e., nodes) in the pub/sub network. Covering optimization algorithms are compatible only with simple subscription matching and do not extend trivially to more expressive semantics such as top-k filtering using relevance, fairness, or diversity.

In general, without the covering protocol, every edge broker to which a publisher is connected to must maintain global knowledge of all subscriptions in the system, thus making top-k subscription filtering trivial since the rank of all subscriptions can...
be computed locally. But once subscription covering is assumed (popularly supported in pub/sub systems), the edge broker has only a partial knowledge of the subscription space, and top-k filtering becomes a challenging and important problem.

The existing research in pub/sub focuses heavily on efficient routing (e.g., covering [88]) and efficient matching algorithms [124, 4, 41, 120, 48, 107] in isolation. Recent work argues for the importance of top-k matching based on the relevance of subscriptions [32], but without paying attention to the pub/sub covering aspects and without considering the extended ranking semantics such as fairness and diversity that are the main focus of our work.

In this chapter, we first formalize general top-k subscription filtering semantics to realize various ranking objectives, such as relevance, diversity, and fairness. Second, we propose an efficient rank-cover algorithm to reduce the forwarded subscription traffic while satisfying the aforementioned filtering semantics. The key intuition behind our rank-cover algorithm is to construct a subscription covering structure at each broker that consists of a subset of a partially ordered set (POSET) containing only its top ranking downstream subscriptions. Therefore, our rank-cover algorithm substantially reduces the number of subscriptions forwarded upstream while guaranteeing the correctness of top-k filtering semantics at upstream brokers. We make the following contributions in this work:

- We formalize general top-k subscription filtering semantics to express a wide range of ranking objectives including relevance, diversity, and fairness (Section 5.2).

- We develop a novel rank-cover algorithm to enable efficient distributed top-k subscription filtering while supporting covering (Section 5.3.1).

- We introduce an ancestor counting optimization to further reduce the size of
the covering set produced by our rank-cover algorithm and the amount of subscription traffic (Section 5.3.3). We also study the optimal covering pruning (Section 5.3.4).

- We introduce fairness as a top-k ranking objective and develop a weighted sort shuffle algorithm for efficient subscription selection (Section 5.3.5).

- We conduct an extensive sensitivity analysis of our algorithms (on an open-source pub/sub system) with respect to edge broker load reduction (covering effectiveness), end-to-end latency, processing overhead, and fairness semantics (Section 5.4).

5.2 Semantics and naive solution

In this section, we first formalize a model for top-k subscription filtering in pub/sub. Our model consists of a general framework that supports a variety of ranking semantics used for selecting a top-k of subscriptions. In particular, we describe our relevance scoring, diversity, and fairness ranking criteria for top-k filtering. Second, we describe how top-k subscription filtering semantics are propagated through the system by piggybacking top-k data via advertisements. Finally, we demonstrate how covering, in conjunction with the proposed top-k model, is incompatible with a naive solution and is thus a non-trivial challenge to solve.

5.2.1 Top-k model and ranking criteria

Consider a publication $p$ that matches a set of subscriptions $S$. The problem addressed in this chapter is to deliver the publication to the highest ranked $k$-sized subset of $S$. Different ranking criteria can be used to determine the subset. For example:
• Relevance-based (scoring) semantics. There is a scoring function, score\( (p, s) \), that computes a score given a publication \( p \) and subscription \( s \). The subscriptions are ranked by descending score, such that the subscriptions with the top-\( k \) highest scores are selected.

• Fairness semantics. Each rank between \([1, |S|]\) is randomly assigned with equal probability to every subscription in \( S \). Therefore, a fair delivery selects a uniformly random \( k \)-sized subset of \( S \). More precisely, the probability of delivering \( p \) to any subscription \( s_i \in S \) is \( \frac{k}{|S|} \).

• Diversity semantics. Suppose there is a distance metric that quantifies the pairwise similarity of subscriptions: distance\( (s_i, s_j) \). Each subscription \( s \) in \( S \) is ranked based on the value of \( \sum_{j=1}^{N} \text{distance}(s, s_j) \), which is the cumulative sum of pairwise distances between \( s \) and all subscriptions in \( S \). A subset of \( S \) consisting of \( k \) elements whose sums are largest is called diverse. The intention is to avoid delivering the publication to subscriptions with similar interests.

5.2.2 Semantics propagation

The top-\( k \) ranking semantics above are from the perspective of the publisher. In particular, we are not addressing the problem where a subscriber chooses to receive only a subset of matching publications, since this has been covered in Chapter 4.

As a consequence of this design point, the publisher must specify a value \( k \) with each advertisement \( a \) it issues, indicating that publications induced by \( a \) should be sent to the \( k \) highest-ranked matching subscriptions. In addition, a ranking function \( \text{rank}(p, S) \) may be issued along with the subscription or advertisement, or there may be a system-wide function. In this chapter, we assume the most general case of a per-subscription ranking function.
For a publication $p$ and a set of matching subscriptions $S$, the ranking function $\text{rank}(p, S)$ assigns a rank between $[1, |S|]$ to each subscription $s$ in $S$. This ranking function can use any combination of ranking semantics described above. For example, a function could first use relevance to rank subscriptions by score. Subscriptions which are tied in score can then be further demarcated using diversity. This ranking function is used during the top-k filtering process to compute the top-k ranked subscriptions for a publication to be delivered to.

### 5.2.3 Naive solution

This chapter focuses on the forwarding of advertisement, subscription, and publication messages to achieve top-k dissemination. Notably, the algorithm to compute the top-k set for a publication is assumed to be known. More specifically, given a set of subscriptions $S$ and a publication $p$: (i) a matching algorithm can compute the set of subscriptions $S \in S$ that match $p$, and (ii) a top-k algorithm can use a provided $\text{rank}(p, S)$ function to compute a $k$-sized set of top ranked subscriptions $\hat{S}$ among $S$.

If there is no subscription covering, then each broker can compute the top-k ranked subscriptions locally and forward the publications accordingly along with the IDs of the subscriptions to deliver the publication to.

Note that if the broker simply forwarded to each neighboring downstream broker the publication along with a count of subscriptions reachable from that broker, there could be inconsistencies due to subscription churn while the publication is propagating through the system. If this is acceptable, then the publisher can forward the subscription count instead of the IDs, saving message overhead, but incurring additional computation load on the downstream brokers.

In the experiments in Section 5.4, the ID-based algorithm which does not employ
covering is referred to as REGULAR-K.

### 5.2.4 Problem with naive solution

The naive solution works because each broker knows the complete set of subscriptions in the system, allowing for the ranks of matching subscriptions to be computed accurately. A common pub/sub optimization is to forward only a subset of subscriptions to upstream brokers. The subset is chosen to be an appropriate summary of the subscriptions such that the upstream broker can correctly determine which downstream brokers may have matching subscriptions. For example, the subscription covering algorithm avoids forwarding subscriptions whose interest space is subsumed by another subscription [88].

A subscription $s_1$ covers subscription $s_2$ iff all publications that match $s_2$ also match $s_1$. Given a set of subscriptions, and their pairwise covering relationships, the broker constructs a partially ordered set (POSET) that represents the covering relationships among its subscriptions. It then forwards only those subscriptions that are not covered by any other.

Consider the broker topology shown in Figure 5.1, where the publisher is connected to broker $B_1$ and the subscribers are connected to brokers $B_2$ and $B_3$. Also suppose that the subscribers at broker $B_2$ issue the subscriptions $s_1$ to $s_7$ whose covering POSET is depicted in Figure 5.2. When subscription covering is not employed, broker $B_1$ is aware of all the subscriptions and can select the appropriate top-k set as outlined in Section 5.2.3. However, with subscription covering, broker $B_2$ only forwards the roots of the POSET, subscription $s_1$ in this case, to the upstream broker. Broker $B_1$ no longer has enough information to know how many of the top-k subscriptions are reachable from each downstream broker. In Section 5.3, we will develop a solution
to this problem that can achieve correct top-k filtering while retaining most of the benefits of subscription covering.

5.3 Top-k forwarding algorithms

This section presents first a distributed top-k algorithm (RANK-COVER-K) that correctly supports covering, as opposed to the REGULAR-K solution shown in Section 5.2.3. We also present ACO-COVER-K, an optimized version of RANK-COVER-K using ancestor counting. Finally, we propose ACO-FAIR-K, a specific instance of ACO-COVER-K using our proposed fair ranker.

5.3.1 Top-k-aware subscription covering

We now describe a set of algorithms that support subscription covering. The challenge is to decide how many of the $k$ most relevant subscriptions are reachable from each downstream broker.

Without loss of generality, we employ the example shown in Figure 5.1 to ex-
plain the proposed solution. It is important that brokers $B_2$ and $B_3$ only forward a minimized subset of their subscriptions to $B_1$, that is, they should employ covering techniques to reduce the forwarded subscription load. Suppose $k = 2$ for the publisher. Then, when a publication arrives at broker $B_1$, it needs to distinguish among three cases:

- **2-0 case**: The two highest ranked subscriptions are both reachable through broker $B_2$.
- **1-1 case**: One of the highest ranked subscriptions is reachable through broker $B_2$, and the other through $B_3$.
- **0-2 case**: The two highest ranked subscriptions are both reachable through broker $B_3$.

The key insight is that broker $B_1$ only needs to distinguish among the above three cases. In particular, it does not need to know if there are more than $k = 2$ matching subscriptions downstream of a particular broker. Therefore, it suffices for each broker to forward the $k$ best ranked subscriptions for all possible publications that may be induced by advertisement $a$. Note that unlike the REGULAR-K solution, each broker along the dissemination path must perform top-k filtering to ensure that covered subscriptions can be selected.

We will now generalize and formalize this point. Let $S_j$ be the set of subscriptions that match publication $p_j$ at a given broker, and let $s_{j,1},...,s_{j,n}$ be the subscriptions in $S_j$ ordered by decreasing rank. Furthermore, let $\hat{S}_j$ be the top-k set in $S_j$, that is, $\hat{S}_j = s_{j,1},...,s_{j,q}$ where $q = min(n,k)$. Finally, let $\hat{S} = \bigcup_j \hat{S}_j$ be the set of highest ranking subscriptions for all possible publications. Notice that the size of $\hat{S}$ may be larger than $k$. 
Theorem 6. If a broker $B$ receives the subscriptions $\hat{S}$ as computed by each of its downstream brokers, then it can compute the set of most relevant subscriptions for a given publication.

Proof. Suppose by way of contradiction that there exists a subscription $s' \notin \hat{S}$ that is in the top-$k$ set of subscriptions for a given publication. Consider the neighboring downstream broker $B'$ where $s'$ came from. Since $s'$ is in the top-$k$ set of $S$, and since the set of subscriptions $S'$ at broker $B'$ is a subset of $S$, then $s'$ must be in the top-$k$ set in $S'$, that is $S' \in \hat{S}'$. If this is the case, however, then $s'$ will be forwarded to the upstream broker and $s' \in \hat{S}$. This contradicts the original supposition.

We now outline several algorithms to compute $\hat{S}$, the set of subscriptions that each broker must forward to its upstream broker in order to allow top-$k$ filtering with subscription covering.

5.3.2 Rank-cover

Let subscriptions $s_1, \ldots, s_n$ be the subscriptions at a given broker that intersect an advertisement $a$ from an upstream broker.

In this algorithm, the subscription cover is a partially ordered set (POSET) built according to an extended covering definition that is based on the subscription predicates and given ranking functions. We call this the rank-cover definition. In particular, subscription $s_i$ rank-covers $s_j$ iff both these conditions are true: (i) The predicates of $s_i$ cover those of $s_j$. (This is the conventional covering definition.) (ii) The ranking function of $s_i$ covers that of $s_j$. More precisely, for every publication $p$ induced by advertisement $a$, the rank of $s_i$ is greater than the rank of $s_j$.

Intuitively, rank-covering implies that if a publication matches $s_j$, then it will also match $s_i$ with a higher rank.
Each broker then forwards all the subscriptions in the top $k$ levels of the POSET. This will ensure that the upstream broker has enough subscriptions to determine how many of the top-$k$ ranking subscriptions are downstream.

More precisely, this algorithm constructs $\hat{S}$ as $\{s_i | \text{depth}(s_i) < k\}$ where depth$(s_i)$ is the depth of subscription $s_i$ in the rank-cover POSET.

**Theorem 7.** The set $\hat{S} = \{s_i | \text{depth}(s_i) < k\}$ contains the set of top-$k$ subscriptions for any possible publication by advertisement $a$.

**Proof.** Suppose by way of contradiction that there exists a subscription $s' \notin \hat{S}$ that is among the top-$k$ highest ranking subscriptions for some publication. Consider the set of subscriptions $\tilde{S}$ on the path from the root of the POSET to $s'$. By our supposition, there are at least $k$ such subscriptions in $\tilde{S}$. Also by the definition of the POSET construction, for any publication that matches $s'$, each subscription in $\tilde{S}$ will also match $p$ and have a higher score than $s'$. Therefore $s'$ cannot be among the top-$k$ most relevant subscriptions for $p$. \hfill \square

For example, consider the topology in Figure 5.1, and let subscriptions $s_1, ..., s_7$ be the subscriptions at broker $B_2$ that intersect with advertisement $a$ (which has $k = 2$) from broker $B_1$. Suppose also that Figure 5.2 depicts the covering POSET for the subscriptions at $B_2$ that intersect advertisement $a$ according to the rank-cover definition. In this case, since $k = 2$, all nodes with depth less than 2 will be forwarded to $B_1$. So, subs $s_1, s_2, s_3, s_4$ will be sent to $B_1$. Broker $B_1$ now has enough subscriptions from $B_2$ to determine if 0, 1 or 2 of the top ranked subscriptions are from broker $B_2$.

Note that the algorithm must be rerun when there is a change in the set of subscriptions or advertisements, as is true of the traditional covering algorithm.
The rank-covering algorithm is defined in Algorithm 6, and is referred to as RANK-COVER-K in the experiments in Section 5.4.

**Algorithm 6: Rank cover forwarding**

```plaintext
foreach advertisement a do
    /* Construct rank-cover POSET for subs that intersect a and come from a different last hop. */
    S_a ← \{ s | s matches a ∧ s.lasthop ≠ a.lasthop \};
    C_a ← rank-covering POSET among subscriptions S_a;
    /* Construct the forwarding set for a. */
    F ← subscriptions at depth < k in C_a;
    add F in the subs to forward to a.lasthop;
```

### 5.3.3 Ancestor counting optimization (ACO)

The rank-cover covering algorithm (Section 5.3.2) can further be optimized so that certain subscriptions, which have multiple parents, can be removed from the forwarding set. This happens if there are enough overlapping ancestors to cover the subscription and satisfy \( k \) matches.

In this optimization, rather than forwarding all subscriptions in the POSET with depth less than \( k \), we only forward those subscriptions with fewer than \( k \) ancestors in the POSET. This is outlined in Algorithm 7.

More precisely, this algorithm constructs \( \hat{S} \) as \{ \( s_i | \text{ancestors}(s_i) < d \) \} where \( \text{ancestors}(s_i) \) is the number of subscriptions in the rank-cover POSET that are in the path from the root to \( s_i \).

**Theorem 8.** The set \( \hat{S} = \{ s_i | \text{ancestors}(s_i) < k \} \) contains the set of top-\( k \) subscriptions for any possible publication from advertisement \( a \).

**Proof.** The proof is almost identical to that of Lemma 7. \( \Box \)

This algorithm is referred to as ACO-COVER-K in the experiments in Section 5.4.
Algorithm 7: Ancestor counting forwarding

1. foreach advertisement $a$ do
2.   /* Construct rank-cover POSET for subs that intersect $a$ and come from a different last hop. */
3.   $S_a \leftarrow \{s | s \text{ matches } a \land s.lasthop \neq a.lasthop\}$;
4.   $C_a \leftarrow$ rank-covering POSET among subscriptions $S_a$;
5.   /* Construct the forwarding set for $a$. */
6.   $F \leftarrow$ subscriptions at depth $< k$ in $C_a$;
7.   /* Ancestor counting pruning step */
8.   foreach sub $s \in F$ do
9.      if $s.ancestors \geq k$ then
10.         remove $s$ from $F$;
11.      add $F$ in the subs to forward to $a.lasthop$;

5.3.4 Pruned rank-cover

The rank-cover and ancestor counting algorithms above do not compute the minimal set of subscriptions to forward. As a counter example, consider the subscriptions in 5.3(a) which depicts a POSET of subscriptions built according to their predicate covering relationships rather than their rank-cover. Furthermore, suppose that subscription $s_i$ has a higher rank than subscription $s_{i+1}$ for all publications. The rank-cover POSET of these subscriptions is shown in Figure 5.3(b).
When $k = 2$, it is evident that a broker needs to forward only subscriptions \{s_1, s_2, s_3\}. However, both the rank-cover and ancestor counting algorithms would compute an $\hat{S} = \{s_1, s_2, s_3, s_4, s_5\}$.

A smaller $\hat{S}$ can be computed as follows: First, compute the subscriptions as in the ancestor algorithm: $\hat{S} = \{s_i | \text{ancestors}(s_i) < k\}$. Then for each $s_i \in \hat{S}$, if the number of predicate_ancestors($s_i$) > $k$, then discard $s_i$ from $\hat{S}$, otherwise, keep it. Let predicate_ancestors($s_i$) be the number of subscriptions in the predicate-cover POSET that are on the path from the root to $s_i$.

**Theorem 9.** The set $\hat{S} = \{s_i | \text{ancestors}(s_i) < k \cap \text{predicate}_\text{ancestors}(s_i) \leq k\}$ contains the minimal set of top-$k$ subscriptions for any possible publication.

**Proof.** Suppose by way of contradiction that there exists a more optimal set $\hat{\hat{S}}$ that does not include subscription $s' \in \hat{S}$. Consider a publication $p$ that matches $s'$ but none of its descendants in the predicate-covering POSET. By definition of the POSET, all ancestors of $s'$ in the predicate-covering POSET will also match $p$, and no other subscriptions will match $p$. Now, since $s'$ was included in $\hat{S}$, and $p$ only matches the ancestors of $s'$, $s'$ must be among the top-$k$ relevant subscriptions for $p$. This implies that $s'$ must be in $\hat{\hat{S}}$ which contradicts the original supposition. \[\square\]

### 5.3.5 Fairness

This section considers a fair ranker when selecting the $k$ subscriptions to forward a publication. More precisely, given a publication $p$ that matches $n$ subscriptions $s_1, \ldots, s_n$, the probability of delivering $p$ to $s_i$ must be $k/n$. That is, each matching subscription is equally likely to be in the top-$k$ set.

Note that if there is no subscription covering, then the solution is trivial as discussed in Section 5.2.3: the first broker that receives the publication from the pub-
Publisher can compute the set of matching subscriptions, uniformly select among them, and forward the ids of these selected subscriptions to each downstream broker.

One attempt to evaluate fairness on subscriptions known by the broker after applying the covering algorithms is shown in Section 5.3.1. However, this simple-minded approach could produce incorrect results. For example, consider a broker with subscriptions $S$ and corresponding forwarding set $\hat{S}$ as computed by the algorithms in Section 5.3.1. Now add a new subscription $s'$ to get $S' = \{s'\} \cup S'$ and corresponding forwarding set $\hat{S}'$. If $s'$ is a leaf node in the predicate-covering POSET of $S'$, and $k$ is less than the minimum height of the rank-cover POSET of $S'$, then $s'$ will not be in $S'$. Consequently, $\hat{S} = \hat{S}'$, and hence from the upstream broker’s perspective, the case where $s'$ is present is indistinguishable from the case where it is absent, but to achieve the fairness semantics, the addition of $s'$ should affect the routing logic.

The fairness algorithm presented here is an approximate algorithm, which we argue is justifiable since the definition of fairness is itself probabilistic. In this algorithm, subscriptions are forwarded as usual, but each forwarded subscription is assigned a weight equal to the number of descendants in its rank-cover POSET.

During publication forwarding, a weighted shuffle is performed among the matching subscriptions, and the first $k$ subscriptions in the shuffled sequence are selected. The weighted shuffle algorithm, detailed in Algorithm 8 uses a search tree (eg. red-black tree), called $wtree$. This tree keeps track of the cumulative sum of the weights of inserted elements for each entry. A randomly generated value in the range up to the total weight of all the entries can be used to retrieve an element whose key is the least greater to the value selected. This algorithm requires $n$ insertions and $O(n)$ searches for a total running time of $O(n \log n)$. 
Algorithm 8: Weighted shuffle algorithm

```c
/* Construct a cumulative weight search tree wtree: */
1 sum ← 0;
2 tail ← null;
3 foreach subscription s do
4    sum ← sum + s.weight;
5    insert (sum, s) in wtree;
6    tail ← sum;
/* Shuffle the elements with weighted probability */
7 slist ← {};
8 repeat
9    x ← random(1, tail);
10   e ← ceiling(x, wtree);
11   slist ← slist ∪ {e};
12   replace e with a pointer to the range [tail − e.weight, tail] in wtree;
13   tail ← tail − e.weight;
14 until |S| times;
15 return slist;
```

5.3.6 Discussion

The above algorithms can be optimized further if relaxed to accept approximate solutions. For example, if we control the forwarded subscription set recomputation frequency such that the recomputation rate is slower than the subscription churn rate, the number of subscription messages is reduced at the expense of a stale top-k selection.

As stated in Section 5.2.2, combinations of different ranking schemes are possible. For instance, fairness and scoring-based top-k semantics can be achieved by forwarding subscriptions according to the algorithms in Section 5.3.1 as well as assigning a weight to each forwarded subscription as described in Section 5.3.5.

The framework in this chapter can also support diversity, but it remains to devise a reasonable and efficient pairwise distance metric among subscriptions. The definition and justification of such a metric is out of the scope of this manuscript, and is treated...
as an interesting avenue for future work.

5.4 Evaluation

This section contains the results of experiments performed on our various algorithm implementations. We first compare the covering performance of three implementations: (1) our baseline top-k solution with covering (RANK-COVER-K), (2) an optimized version using ancestor counting (ACO-COVER-K), and (3) our baseline covering algorithm without top-k (COVER). These experiments gauge the impact of using top-k over covering and provide grounds for a discussion on the sensitivity of our work with respect to various workload parameters.

We then perform an evaluation of our fair ranker using our optimized top-k solution (ACO-FAIR-K), compared to ACO-COVER-K using a fairness metric. The fairness baseline for this experiment is our top-k solution without covering (REGULAR-K). Each subscription has a fair chance to be selected in REGULAR-K since each broker has complete knowledge of all downstream matching subscriptions when covering is not employed (see Section 5.2.3).

Finally, we measure the processing overhead of our various solutions using end-to-end publication latency to assess the impact of our top-k filtering and fairness selection algorithms on the publication match-and-forward process.

5.4.1 Setup

The implementation is performed in Java using the PADRES pub/sub prototype\(^2\). Experiments are performed on the SciNet testbed using the General Purpose Cluster

\(^2\)http://padres.msrg.toronto.edu/
(GPC)\textsuperscript{3}. Up to 60 distinct machines are used, each equipped with Intel Xeon quad-core processors and 16GB RAM.

The workload used is synthetic, with a single publisher emitting publications at a rate of 30 per minute. A single publisher allows us to better control the distribution of subscribers relative to the publisher which is important for fairness (see Section 5.4.4).

The broker overlay topology consists of 5 to 10 core brokers. Each core broker is then connected to 5 edge brokers. We connect 10 subscribers to each edge broker. We employ between 250 to 5000 subscribers, each with one subscription. For example, an experiment could have 10 core brokers, with 50 edge brokers and 500 subscribers. This setup is a model of a network of data centers connected through gateways (core brokers) and measures the impact of our algorithms on delivery paths with multiple broker hops.

Subscriptions are randomly generated to have between 30\% and 60\% selectivity, uniformly distributed in the publication space. We argue that the high selectivity of the subscriptions is appropriate for our motivating scenario where users are receiving an overwhelming amount of data, hence the need for additional filtering. Furthermore, the narrow range of selectivity creates conservative covering results since neither excessively large subscriptions exist to cover large spaces, nor are small subscriptions present that can easily be covered. The relatively even size of subscriptions creates broad and shallow covering trees which constitute a worst case for our scenario. Better covering results can be achieved using deep covering trees. For the purpose of this experiment, we are interested in the worst-case results for our solutions.

Advertisements are set with a $k$ which is between 0.2\% to 2.5\% of the total number of subscriptions. For instance, in an experiment with 500 subscriptions, a $k$ set to 10 has a selectivity of 2\%. Top-k is useful for suppressing an overwhelming amount of

\textsuperscript{3}http://www.scinet.utoronto.ca/
data, therefore maintaining a lower $k$ to control the selectivity of the publications is desirable.

For all solutions except ACO-FAIR-K, we employ a covering-agnostic random ranking function. Each subscription known by the broker for a publication is assigned a random rank with uniform distribution. Since our experiments focus on fairness, this setup allows us to maximize the size of the pool of subscriptions which must be fairly selected from. Our solution is compatible with any ranking scheme.

The fair ranker used by ACO-FAIR-K extends the random ranker by weighing subscriptions appropriately according to the covered subscriptions (see Section 5.3.5). Furthermore, we asynchronously update the weights for the ranker by propagating the count of covered subscriptions through propagated subscriptions (see Section 5.3.6).

### 5.4.2 Metrics

**Percentage of forwarded subscriptions:** Our main metric for measuring the effectiveness of our covering techniques is the reduction in the number of subscriptions sent upstream (towards publishers). In our evaluation, we measure the ratio between the number of subscriptions at the publisher edge broker and the total number of subscriptions sent by all subscribers. Since the publisher edge broker is the last hop for every subscription in the system (as they all have non-zero selectivity), this number is an overall measure for covering in the system. The lower bound is provided by our baseline covering algorithm COVER which does not support top-k, while the upper bound is 100% for top-k without covering (REGULAR-K). Our proposed solutions fall somewhere in between and allow us to assess the impact of top-k dissemination on covering.

**End-to-end latency:** We measure the publication latency from the time elapsed
between its sending at the publisher to the receipt at a subscriber. This allows us
to measure the aggregated processing overhead of our top-k algorithm being run at
every broker hop encountered. In particular, this allows us to assess the impact of
the weighted fair shuffle algorithm developed in this thesis.

Fairness: We measure fairness empirically by comparing the number of publica-
tions each subscription receives in each implementation to the baseline top-k algo-

rithm REGULAR-K. We sum up the difference in number of publications received
for all the subscriptions during the course of the experiment. This number indicates
how much deviation the implementation has to the expected number of publications
to be received from each subscription. Note that the subscriptions and publications
are generated deterministically for every experiment.

5.4.3 Covering performance

Figures 5.4(a) and 5.4(b) show the covering performance of our various algorithms
relative to the baseline COVER with varying $k$ values and different amount of sub-
scriptions. A lower number indicates that the covering algorithm has been more effec-
tive in reducing the number of subscriptions disseminated. The graphs show that our
solutions, RANK-COVER-K and ACO-COVER-K, are sensitive to increasing values
of $k$. For example, we note that covering is completely nullified (no reduction in
subscriptions, 99.8% subscriptions) for RANK-COVER-K at $k$ set to 2% ($k = 10$) for
500 subscriptions. As described in Section 5.4.1, we selected subscriptions such that
the covering trees are shallow and broad. Therefore, traversing a few levels deep is
enough to encounter a large number of subscriptions. However, we argue that for our
intended applications, low values of $k$ (e.g., less than 1%) are appropriate to justify
top-k filtering.
Chapter 5. Top-k subscription filtering

(a) 250 subscribers

Figure 5.4: Covering effectiveness
We also notice that the baseline propagates only 5.83% of 250 subscriptions, and 3.47% for 500 subscribers. Note that the baseline performance is unaffected by the value of \( k \), since it does not support top-k. COVER therefore demonstrates scalability through the improvement in subscription reduction with increasing subscription loads. RANK-COVER-K does not preserve this trend, as the covering effectiveness decreases for the same relative \( k \) in a workload with more subscriptions. As an example, 86.67% of subscriptions are present for 250 subscriptions which rises to 99.8% for 500 subscriptions when the \( k/\text{subs} \) ratio is set to 2%. On the other hand, ACO-COVER-K is less sensitive to increasing subscriptions: at the same \( k/\text{subs} \) ratio of 2%, the % of subscriptions slightly increases from 75% to 82.04%. Since the optimization leverages the presence of subscriptions with multiple parents to cover subscriptions more effectively, we can infer that the incidence of multiple parents increases as the number of subscriptions increase, which is consistent with the selectivity parameters described in the setup above. Increasing the number of subscriptions does not necessarily increase the depth of the covering tree, but the breadth of existing levels, which is useful for ACO-COVER-K but not for RANK-COVER-K.

We also evaluated our solutions with 5000 subscribers (see Figure 5.4(c)). At that scale, the reduction provided by RANK-COVER-K is less than 1% in the \( k/\text{subs} \) range tested (0.2%-2%). ACO-RANK-COVER provides a 7% reduction (350 subscriptions covered) at a 1% \( k/\text{sub} \) ratio. We attribute these observations due to sensitivity to values of \( k \). Although the \( k/\text{subs} \) ratio is maintained compared to previous experiments, the absolute values of \( k \) tested are in the higher range of 10-100, compared to 1-10. We reason that this sensitivity is due to our experimental setup. The subscription space is kept constant, which results in a large number of overlapping subscriptions residing in upper levels of the covering tree. We, therefore, conclude that covering optimizations for overlapping subscriptions, such as merging techniques
Chapter 5. Top-k subscription filtering

(a) Overall fairness

(b) Fairness by distance

Figure 5.5: Fairness evaluation

proposed in [78], are useful for controlling the number of subscriptions in the upper levels of the tree.

**Summary:** Our covering top-k solutions are sensitive to varying values of $k$. RANK-COVER-K also does not scale to the number of subscriptions: at 500 subscriptions, RANK-COVER-K does not reduce the number of subscriptions for $k$ set to 2% (10). These properties are advantageous to ACO-COVER-K: top-k is employed to enforce a low rate of selectivity, hence does not require a large $k$, and filtering on subscriptions is performed in workloads with a large number of subscriptions, thus requiring scalability in that aspect. Merging techniques are useful in controlling the number of subscriptions found in upper level of the covering tree.

5.4.4 Fairness evaluation

Figure 5.5(a) evaluates the fairness of our optimized solution ACO-COVER-K to ACO-FAIR-K, which is the same optimized solution equipped with our fair ranker. A lower value of fairness is better, as it indicates that the solution sends a num-
ber of publications to each subscription closer to the expected value established by REGULAR-K. We note that our fair ranker outperforms the random ranker at all values of $k$ tested by an average of 224%. ACO-FAIR-K also becomes increasingly fair as the value of $k$ increases. This is due to the increase in notification volume which comes with larger values of $k$. With more subscriptions being selected, the fair ranker’s ability to do a weighted shuffle of those subscriptions is better leveraged. Furthermore, increasing the $k$ value triggers more subscriptions to be sent upstream. Since the metadata used by the fair ranker is propagated via subscriptions, this increased frequency in subscription propagation increases the accuracy of the metadata.

On the other hand, ACO-COVER-K decreases in fairness as more brokers are encountered as the random ranker used further impacts the fairness of the solution. We also note that the fairness of ACO-COVER-K improves by 20.18% between $k = 0.4\%$ to $k = 0.8\%$. As $k$ increases, the number of subscriptions which are uncovered rises by 13.34% (as seen in Figure 5.4(a)). This larger number of subscriptions found at the publisher edge broker benefits the random ranker which now has a more accurate representation of the subscriptions in the system, but this benefit is not enough to offset the inaccuracy of larger selections with higher values of $k$.

In light of these observations, we conclude that ACO-FAIR-K outperforms ACO-COVER-K significantly and does not suffer from issues that arise when the $k$ value is at the extreme ends of the intended range for $k$.

Figure 5.5(b) shows the sensitivity of fairness for subscriptions at varying distance from the publisher. Distance is measured as the number of core broker hops between the publisher and the subscriber. The figure shows the average fairness of subscriptions across the different values of $k$, since the same pattern is observed regardless of the $k$ used. We observe that for both solutions, fairness improves when it reaches 3 broker hops, and then decreases when the number of hops increases beyond. This can
be explained due to the nature of the inaccuracies. The subscriptions closer are more numerous at the publisher edge broker: because they are propagated through less brokers to reach the publisher, there are less opportunities for them to be covered. On the other hand, subscriptions farther away are less likely to reach the publisher, since there are more opportunities for them to be covered by other subscriptions in intermediary brokers. As a result, subscriptions closer are overvalued and receive more publications, while subscriptions further away are undervalued and receive less publications. In both cases, the fairness metric increases as these subscriptions deviate from the expected outcome. ACO-FAIR-K is subject to this trend to a lesser degree, but for a different reason. The fairness ranker relies on the meta-information which is propagated through subscriptions. In the workload used, staleness in the meta-information tends to underestimate the number of subscriptions. Because these inaccuracies accumulate over multiple hops, subscribers which are closer to the publisher tend to be overvalued, while ones farther away are undervalued. Therefore, we observe the discrepancy in fairness for the closest subscriptions because they receive more publications than expected, while the subscriptions further away loses fairness as they receive less publications than expected.

Summary: ACO-FAIR-K, which uses our fair ranker, outperforms ACO-COVER-K, which selects randomly from known subscriptions, by an average of 224% in the target range for $k$ and becomes increasingly better as the value of $k$ increases. Both solutions tend to send publications to closer subscriptions more frequently, while subscriptions further away are less likely to be selected. ACO-FAIR-K is less sensitive in that regard. Additionally, the diameter of pub/sub overlays can be controlled to lessen this effect [91].
Chapter 5. Top-k subscription filtering

5.4.5 Processing overhead

Figures 5.6(a) and 5.6(b) show the end-to-end latency of publication delivery for 250 and 500 subscriptions, respectively. We first note that REGULAR-K is outperformed by ACO-COVER-K for lower values of $k$. This indicates that the processing overhead for top-k is affected by the number of subscriptions stored at a broker. As the value of $k$ grows larger, the covering effectiveness (as explained in Section 5.4.3) decreases which affects the top-k overhead. Using values extracted from the COVER and ACO-COVER-K results, we evaluate the overhead at a broker of top-k processing to be 3% of the total latency, if the same number of subscriptions is stored at the broker. The most significant part of the top-k processing overhead is therefore due to the drop in covering performance, which increases the number of subscriptions residing at each broker, which in turn affects the matching and processing times.

The overhead of the fair sorter has a significant impact on latency in higher values of $k$ tested. The overhead follows a superlinear growth to the number of subscriptions, as observed in Figures 5.7(a) and 5.7(b). This is in line with the $O(n \log n)$ running time of our weighted shuffle implementation which is invoked at each broker when
processing a publication. This result suggests that adaptive solutions which can either limit the number of subscriptions to be shuffled or can avoid the shuffle selection at certain brokers will improve publication latency. It also denotes the importance of our optimization for the weighted shuffle procedure. Nevertheless, the experiments indicate that for values of $k$ less than 1% of the total number of subscriptions, it is possible to maintain fairness and still perform better than a top-k implementation with no covering.

Summary: Top-k covering is shown to improve end-to-end latency by an average of 25% in the range of $k$ tested, with the benefit lessening as the covering effectiveness decreases. The fair ranker has a significant impact on the processing overhead at a broker which is sensitive to the number of subscriptions processed. This denotes the importance of optimizing the weighted shuffle procedure, which is performed at every broker while processing publications.
5.5 Conclusions

In this chapter, we address the problem of subscription filtering with covering in light of our extended top-k semantics that supports general ranking objectives, such as relevance, fairness, and diversity. In particular, we introduce a novel rank-cover algorithm to construct an efficient covering POSET that respects our extended top-k semantics. We further develop various optimizations to effectively prune and reduce the cover size. We conclude our work with an extensive set of sensitivity analyses of our algorithms incorporated in PADRES, an open-source publish/subscribe system, that illustrate the importance of rank-cover with respect to end-to-end latency and fairness.
Chapter 6

Total order

This chapter describes our total order extension to content-based publish/subscribe systems. Section 6.1 uses online games, business process engines, and log monitoring as motivating examples for total order in pub/sub. Section 6.2 presents the terminology used to address the total order problem. Section 6.3 demonstrates the foundation of our solution, which leverages FIFO links to support per-publisher total order. Section 6.4 shows how we can derive a natural total order from per-publisher total order, and how ordering conflicts can be detected and resolved to fully maintain total order. Section 6.5 presents performance results for the different variations of our proposed solution, comparisons to other known pub/sub solutions such as sequencing, and with group communication systems. Section 6.6 offers a summary of this chapter.

6.1 Introduction

Total order has not been an important concern for pub/sub systems so far because of the space decoupling properties between subscribers: since they do not know the identity of others in the system, inconsistencies cannot be detected by comparing
publication streams amongst each other. However, as the popularity of pub/sub increases, so does the variety and complexity of its application specifications. This property is thus useful in applications where peers do have an external communication medium.

Current distributed content-based pub/sub systems do not offer strong guarantees about the order in which publications are delivered to subscribers. For example, due to network delays, it is possible that two subscribers interested in the same set of published events receive them in different order. This chapter develops a total order algorithm to ensure that all recipients of a set of publications will deliver those publications in the same order. This is illustrated in Figure 6.1: the different messages (represented with different colors) must be delivered to the subscribers the same order.

There are many pub/sub applications that could benefit from total order guarantees. For example, in a distributed on-line game ([70, 11]), players receive publications to notify them of changes to the game state, such as the movement of another player, or the points and objects accumulated by a player. In this case, users receiving messages in different order may perceive inconsistent game states. Similarly, in electronic stock tickers, investors are notified of the trades and price updates on a set of stocks [116]. It would be unfair for investors to observe different market behavior due to differences in the order they receive stock updates. Another set of examples
are workflow or business process engines in which the flow of the process is driven by a carefully orchestrated sequence of trigger messages [81]. It is common in these systems to monitor the messages received by elements of the workflow, perhaps to detect an event pattern that reveals a missed business opportunity. Ordering guarantees here are critical, as the event pattern is sensitive to the order in which publications are delivered. Another case for total order arises in situations where a user’s program state needs to be replayed. For example, if the exact sequence of updates received by an air traffic controller’s console could be mirrored by a replica, it can be used by auditors to determine whether the controller’s decisions were justified based on his view of the system. On a related note, developers could monitor the trace of publications received by a software component using an external subscriber to replay a sequence of events and debug faults in the component.

Broadly speaking, there are at least two classes of applications that require some kind of total order guarantee. First are those that need to detect event patterns or composite events [79], such as detecting stock market trends, or observing attack signatures. The others are those where an application’s state is constructed as a function of the sequence of events it receives, that is, those that follow the Command [49] or Event-sourcing [56] design patterns. An example of this includes the online gaming scenario above. Understanding the difference between these two classes in terms of requirements and workload properties is crucial when optimizing our total order solution towards a particular set of applications.

Offering total order guarantees in a distributed content-based pub/sub system is a challenging problem. First, we note that the naïve solution of a centralized sequencer to globally order messages does not scale, violates the distributed nature of the system, and is overkill since it is only necessary to order those messages that are delivered to multiple subscribers.
Existing distributed messaging systems that offer various ordering semantics are not readily applicable to our context. The primary reason is that existing systems typically assume messages are delivered to an explicit and well-formed group of nodes. For example, this is applicable for group communication middleware [118] and channel-based messaging systems [55]. As well, distributed algorithms based on logical clocks assume a relatively stable and small group of participants [74].

In content-based pub/sub, there are no explicit groups; each subscription may overlap the interests of another, and every publication is delivered to a group which contains any subset of subscriptions. Consequently, given \( n \) subscriptions there are potentially \( 2^n \) groups, and it quickly becomes infeasible to manage an exponentially increasing number of groups.

In this chapter, we develop a distributed protocol to ensure total ordering of publications in a content-based pub/sub system. The protocol works in two parts: first during the detection phase, a broker determines whether a publication needs to be ordered or if it can be immediately forwarded. Next, for those publications that require ordering, in the resolution phase the broker collaborates with a small number of other brokers in the overlay to determine a consistent delivery order.

The protocol is encapsulated within the broker network, and the algorithm does not change the system model of popular pub/sub implementations [24, 45, 46]. In particular, the network is assumed to be asynchronous, and the brokers operate independently and have knowledge only of their overlay neighbors. Moreover, the protocol allows ordering to be applied at a fine-grained level: publishers can mark publications that must be ordered, or subscribers can indicate whether the publications matched by a given subscription must be delivered in order.

This chapter makes the following key contributions:
1. Section 6.3 describes how weaker ordering semantics called per-publisher order can be supported using FIFO links between nodes. We also discuss the impossibility of total ordering in crash failure scenarios.

2. Section 6.4 develops a distributed total ordering algorithm in a content-based pub/sub model. The algorithm is fully distributed, and requires no changes to the pub/sub client API. The basis of the algorithm rests on FIFO links which can partially support total order. We also show the safety of our algorithm when failures are introduced to the model.

3. Section 6.5 evaluates the performance of an implementation of the algorithm. The results show that the pairwise total order algorithm scales with the number of subscriptions in the system and the publication size overhead is bounded. We also conduct a holistic performance comparison with group communication systems.

### 6.2 Background

#### 6.2.1 Total order broadcast

We first introduce the notation for our message passing model. \( \mathcal{M} \) is the set of all possible valid messages. \( \Pi \) is the set of all processes in the system. Given some message \( m \), \( \text{sender}(m) \) designates the process in \( \Pi \) from which \( m \) originates, and \( \text{Dest}(m) \) denotes the set of all destination processes for \( m \).

We first review the problem of total order multicast, widely studied in the literature [34], which is closely related to our problem in pub/sub. The definitions of validity, agreement, and integrity are as follows:
• (Validity) If a correct process broadcasts a message $m$, then it eventually delivers $m$.

• (Uniform Agreement) If a process delivers a message $m$, then all correct processes eventually deliver $m$.

• (Uniform Integrity) For any message $m$, every process delivers $m$ at most once, and only if $m$ was broadcast by $sender(m)$.

• (Uniform Total Order) If a process $p$ and $q$ both deliver messages $m$ and $m'$, then $p$ delivers $m$ before $m'$, if and only if $q$ delivers $m$ before $m'$.

We will use some of the above properties as we discuss ordering semantics and develop the algorithms in this chapter.

6.2.2 Publish/subscribe model

We now reformulate the problem in the context of the pub/sub system model. The system consists of clients which can take on a subscriber or publisher role (or both). Messages are multicast to a set of subscribers $Dest(m)$ (called the destination group). Publish/subscribe supports open groups such that for a given $m$, the $sender(m)$ does not need to be in $Dest(m)$. Multiple groups are possible. In other words, given two messages $m$ and $m'$, the condition $Dest(m) = Dest(m')$ is not enforced. Furthermore, destination groups can be overlapping: $Dest(m) \cap Dest(m') \neq \emptyset$ is possible.

We now provide several ordering semantics for pub/sub which are studied in this chapter:

• (Per-Publisher Total Order) For any two messages $m$ and $m'$ and any two processes $p$ and $q$, where $\{p, q\} \in Dest(m) \cap Dest(m')$. If $sender(m) = \ldots$
sender(m′), then p delivers m before m′ if and only if q delivers m before m′.

Essentially, every message sent by a certain publisher will be delivered in same order to all its recipients. This order can be enforced by the publisher alone (e.g., through FIFO ordering). A different property considers the destination groups only [34]:

- **(Local Total Order)** For any two messages m and m′ and any two processes p and q, where \( \{p, q\} \subseteq \text{Dest}(m) = \text{Dest}(m′) \). p delivers m before m′ if and only if q delivers m before m′.

Local total order guarantees uniform order of delivery of publications sent to identical destination groups. Finally, we propose a stronger property which abstracts away the concept of destination group and only consider messages received by common receivers [34]:

- **(Pairwise Total Order)** For any two messages m and m′ and any two processes p and q. If \( \{p, q\} \subseteq \text{Dest}(m) \cap \text{Dest}(m′) \), then p delivers m before m′ if and only if q delivers m before m′.

Section 6.3 shows how per-publisher total ordering can be enforced using FIFO links. We then extend the solution in Section 6.4 to maintain pairwise total order.

Ordering semantics can also be qualified as weak if they do not guarantee reliability. We define the properties of liveness and safety within the context of failures as follows:

- **(Liveness)** If a message m is sent by a process p to its destination group \( \text{Dest}(m) \), then all processes in \( \text{Dest}(m) \) will eventually deliver m.
• **(Safety)** If messages $m$ and $n$ are delivered to both processes $p$ and $q$, then $m$ and $n$ are delivered to both $p$ and $q$ in the appropriate order.

A resilient pub/sub system should maintain both safety and liveness. However, a system with weak total ordering semantics maintain only safety but not liveness. In other words, processes are not required to deliver every message they receive as long as they maintain the right order amongst delivered messages [8]. Note that a trivial solution which does not deliver any message guarantees safety. In practice, weak total order solutions must provide some limited form of liveness (i.e., deliver a non-empty set of publications).

Figure 6.2 illustrates the various types of total order. In this example, we have three publishers (A-C) and six subscribers (1-6). Publications by A are delivered to the group $G_x$, while B and C publish to $G_y$. Some subscribers are part of both groups (Subscribers 1 and 5). The order publishers send their publications and subscribers receive those publications is from top to bottom in the figure. Per-publisher total order is enforced (note that the subscribers’ ordering does not necessarily correspond with the order publications are sent by the publication). Local total order is not respected since the delivery order of $G_y$ publications is not uniform (because of Subscriber 3). Thus, pairwise total order is not respected. Also note that the delivery order between the red triangle and the hexagon at subscribers 1 and 5 violates pairwise total order, but not local total order.

It is interesting to note that in a topic-based environment, local total order requires less information to maintain than pairwise total order. Recall that in this model, each publication is assigned a single topic and is delivered to nodes subscribed to it. The subscription matrix, that is, the relationships among publishers and subscribers, is well defined and allows sequencing to be separated by topic. On the other hand,
Figure 6.2: Comparison of total order semantics
pairwise total order requires synchronization between messages of different topics.

In a content-based environment, using the subscription matrix to verify whether local total order should be enforced requires more effort than verifying for pairwise total order. Messages are ordered only when two messages have identical delivery groups. Since subscriptions to a message are computed dynamically, the system must first compute all matching subscribers per message before making a decision by comparing these subscription sets. For pairwise total order, it suffices to show that the messages are doubly-overlapping \[83\]: they have at least two subscribers in common, regardless of the actual composition of the groups.

6.2.3 Notation

We consider an overlay topology as an undirected graph \( G = (V, E) \), where \( V \) contains all the nodes in the system (brokers and clients), and \( a, b \in E \) if and only if \( a \) and \( b \) are directly connected in the topology. \( \text{pub}(p) \) is the publisher for some publication \( p \), while \( \text{sub}(p) \) is the set of subscribers which will receive \( p \).

A path \( P \) from \( x_0 \) to \( x_n \) is a sequence of vertices \( (x_0, x_1, ..., x_n) \) where \( x_i, x_{i+1} \in E \), for all integers \( i \), such that \( 0 \leq i \leq n \).

\( nh(P, x_i) \) refers to the next hop from \( x_i \) on a path \( P \). If \( P = (x_0, x_1, ..., x_n) \) then
\[ nh(P, x_i) = x_{i+1} \] for \( 0 \leq i < n \), and \( nh(P, x_n) = \bot \) when \( i = n \).

\( \text{sub}(P, x_a, x_b) \) refers to the subpath of \( P = (x_0, x_1, ..., x_n) \) from \( x_a \) to \( x_b \): \( \text{sub}(P, x_a, x_b) = (x_a, x_{a+1}, ..., x_b) \), given that \( 0 \leq a \leq b \leq n \).

The concatenation of paths \( P = (x_0, x_1, ..., x_n) \) and \( Q = (y_0, y_1, ..., y_n) \), which is denoted as \( P \cdot Q \), is \( (x_0, x_1, ..., x_n, y_1, ..., y_n) \) assuming \( x_n = y_0 \).

In an acyclic connected overlay topology, for any nodes \( v_1 \) and \( v_2 \) in \( V \), there exists a unique non-trivial path \( P \) from \( v_1 \) to \( v_2 \) and the reverse path \( P^{-1} \) is the unique path
from $v_2$ to $v_1$. A non-trivial path is a path which does not contain the same node twice.

In this chapter, we will consider only topologies where each client is connected to at most 1 broker, i.e. $\text{degree}(c) = 1$, for all clients $c$.

### 6.3 Analysis of ordering semantics

This section sets the stage for the distributed total order algorithm developed in Section 6.4. First, we describe how a pub/sub overlay with FIFO links can be used to build a per-publisher total order algorithm. The FIFO property is utilized again later in the chapter to support pairwise total order. We also discuss the notion of liveness and safety and show the impossibility of total order in a system with faulty processes.

#### 6.3.1 Per-publisher total order

In a failure-free content-based model with FIFO links, per-publisher total order is easy to achieve if the delivery path from the publisher to a given subscriber is the same for all messages destined to that subscriber (see Theorem 11). The order the messages are sent by a publisher will be maintained as long as the messages are processed and forwarded in the received order by each broker. The client can then safely deliver the messages as they are received in total order.

In cyclic topologies, each publisher must maintain the same delivery path to each subscriber across publications in order to leverage the natural per-publisher ordering described above. Otherwise, FIFO links are not sufficient to maintain total order. Consider Figure 6.3, where the dashed (blue) message is sent before the continuous (red) message; they should therefore be delivered in the same order at both sub-
Figure 6.3: FIFO links counter-example

scribers. However, the continuous message takes a different path from the dashed message to reach $S_1$. Thus, there is no ordering guarantee between the messages sent by $B_1$ and $B_2$ to $B_3$. $S_1$ can therefore receive the messages out-of-order. For the remaining of this chapter, we focus solely on acyclic topologies, where unique paths exist between each pairs of publishers and subscribers, thus eschewing the case above.

**Observation 10.** In an acyclic topology, any two publications from a publisher $p$ to subscriber $s$ must traverse the same path $P$ from $p$ to $s$.

*Proof.* This follows from the acyclic property: there is an unique path from $P$ from $p$ to $s$. 

We use the previous observation to prove the per-publisher total order theorem:

**Theorem 11.** Per-publisher total order is maintained using FIFO links in an acyclic topology.

*Proof.* Consider any two publications $p_1$ and $p_2$ from a publisher $p$ delivered to two subscribers $s_1$ and $s_2$. According to Observation 10, publications are delivered to $s_1$ using an unique path $P_1$ and similarly a path $P_2$ for $s_2$. Without loss of generality, suppose $p$ decides to publish $p_1$ before $p_2$.

We now consider the first hop after $p$ on the path $P_1$. Due to the FIFO property, it will receive $p_1$ before $p_2$. Due to the acyclic nature of the topology, the next hop in
$P_1$ will receive $p_1$ before $p_2$. By induction, $s_1$ will therefore receive $p_1$ before $p_2$. The proof is similar for $P_2$. Therefore $s_2$ also receives $p_1$ before $p_2$.

Per-publisher total order can be supported in the general content-based case, where no assumptions are made on link quality or the topology. A simple solution exists if a subscriber is interested in every publications made by a publisher. This is effectively the case when a publisher can only publish to a single topic in a topic-based system. In this case, per-publisher total ordering is trivial since the publisher itself can add a sequence number for each topic to each message it is publishing. Subscribers always deliver the message of the next expected sequence and defer the delivery of any messages received out of order.

However, if the publication space of a publisher is not fully contained within the subscription space of a subscriber, it is possible that the subscriber does not receive every publication made by a publisher. This is possible in a content-based model where a subscriber can provide additional predicates such that it may not receive all publications associated to a topic. In this case, “gaps” are formed in the delivered sequence numbers received by a subscriber. Thus, the subscriber will wait forever for a sequence number which will never arrive since it belongs to a publication which does not match the subscription.

To resolve this problem, we enable brokers to keep track of sequence numbers forwarded for each publisher. Thus, a broker can deliver to a subscriber the missing sequence numbers along the next expected delivery.

The algorithm requires a new object $LS$, a table which keeps track of the last sent sequence number per interface $i$ (ie., an outgoing node: broker or client) for a publisher $p$ at each broker. A record is accessed using the function $get(LS, i, p)$ and updated using $update(LS, i, id)$, where $id$ is a sequence number. In addition, both
brokers and clients requires a counter $LR_p$ for each publisher $p$ which tracks the last received sequence number by $p$.

Upon receipt of a message, brokers and clients execute the code in Algorithm 9 to enforce total ordering.

<table>
<thead>
<tr>
<th>Algorithm 9: messageDelivery(m,p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $LR_p = m.ack$ then</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4 else</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

This algorithm takes as input $m$, a publication message received, with publisher $p$ as the sender of the publication. Each publication $m$ carries the field $ack$ which is the id of the previous publication message sent to the current. $ack$ is compared to the last delivered id stored ($LR_p$). If they are the same, then $LR_p$ is updated to the value of the current message ($id$). Note that, we assume that if this is the first message sent by a publisher, both $LR_p$ and $ack$ have a value of 0. We then call the $deliver(id)$ method, which delivers in increasing order every message received starting from $id$. If the comparison fails, the publication is queued using $defer(m)$ in anticipation of a future delivery.

Delivery of a publication once $deliver(m.id)$ is invoked operates differently for brokers and clients. Clients will internally process the publication, while brokers need to forward the publication to the next hop. The details of this forwarding algorithm are given in Algorithm 10.

This algorithm takes as input $m$, the publication to be sent, $p$, the source of the publication, and $interfaces$, a list of outgoing interfaces this publication must be sent to. For each outgoing hop, the broker retrieves the last recorded sequence number in $LS$ for $p$ and inserts it in the $ack$ field for $m$ before sending it to that hop ($send_i(m)$).
Algorithm 10: brokerForwarding(m,p,interfaces)

1. \textbf{foreach} \( i \in \text{interfaces} \) \textbf{do}
2. \hspace{1em} \( m.\text{ack} \leftarrow \text{get}(LS, i, p) \);
3. \hspace{1em} \text{update}(LS, i, p.\text{id})
4. \hspace{1em} \text{send}_i(m);

\( LS \) is also updated to the id of \( p \).

6.3.2 Impossibility of total order under failures

We now show that local total order is impossible in a topic-based publish/subscribe system with a general topology. Since consensus has been proven impossible under the presence of failure in asynchronous systems [47], local total order is also impossible under the same model if consensus can be reduced to it.

We first formally define the properties of consensus:

- Validity: Any value decided is a value proposed.
- Agreement: No two correct processes decide differently.
- Termination: Every correct process eventually decides.
- Integrity: Every process decides at most once.

Theorem 12. Consensus can be reduced to local total order using a resilient topic-based publish/subscribe system.

Proof. Suppose we have a set of processes \( P \), each subscribed to a consensus topic. Each non-faulty process \( p_i \) receives an input proposed value \( x_i \) and outputs a decided value \( y_i \). We now construct a consensus algorithm in Algorithm 11 using a resilient local total order topic-based pub/sub system:
Algorithm 11: consensus($x_i$)

1. publish($x_i$, “consensus”);
2. repeat
3.  wait
4.  until receive($x_m$, “consensus”);
5.  decide($x_m$);

In this algorithm, every process is subscribed to the consensus topic “consensus”. Then, every process publishes their proposed value. Since local total order and liveness are preserved, every process will deliver the same message, hence the same proposed value, first. This first message received becomes the decided value. We show how each property is satisfied:

- Validity: A process can only decide on a value proposed by a process, since it only decides based on received values, which are published according to each process’ proposed value.

- Agreement: Since local total order is enforced, every process will receive the same publication first.

- Termination: Because the pub/sub system is resilient, every process will eventually receive the first message published. Furthermore, each process must first publish before waiting for a message. Therefore, each process will at the very least receive their own message, which may be the decided value.

- Integrity: Processes decide based on the first value received; subsequent messages are discarded.

Because the subscription to a particular topic can easily be accommodated using content-based pub/sub, the impossibility proof also applies to content-based pub/sub.
Furthermore, pairwise total order, being a more general model than local total order, is more than sufficient to support this algorithm.

Maintaining both safety and liveness is therefore not possible. As a consequence, we focus our attention on the safety of our algorithm when dealing with failures by maintaining weak total order. However, it is possible to design a system where liveness is prioritized over safety. For instance, a timeout mechanism can be used to control the time allocated to the total order component. If total order is not achieved within this time, the message is forwarded immediately to the recipients, ensuring progress. Publications which arrive out-of-order after the timeout period are immediately delivered, thus violating safety.

6.4 Total order algorithm

We present an algorithm for maintaining pairwise total order in content-based publish/subscribe systems. The algorithm is integrated within the broker logic and does not require any external processes. Scalability is achieved by enforcing and ordering only when necessary: FIFO links are sufficient to preserve total order when certain conditions are met. If not, potential ordering conflicts are detected using local broker knowledge and resolved amongst a small subset of relevant brokers.

6.4.1 System model

We assume an advertisement-based pub/sub protocol that employs reverse path forwarding [24]. More specifically, a publisher first issues an advertisement message that defines the set of publications it will publish. These advertisements are broadcast across the overlay and create a spanning tree rooted at the publisher. Subscriptions from subscribers are routed hop-by-hop along the reverse path of matching adver-
tisement trees until the reach the publisher. Finally, publications are routed along the reverse path of matching subscriptions until they are delivered to interested subscribers. There is no restriction on the publication and subscription language (e.g. content-based); we only assume it is possible to match publications against subscriptions, and find overlaps between the interest spaces specified by advertisements and subscriptions.

We further assume an acyclic overlay topology of brokers with FIFO links between brokers. We believe this is a reasonable assumption since the underlying network can support it (i.e., using TCP) or could also easily be implemented at the pub/sub level by exchanging sequence messages at both ends of a network link. Under this model, per-publisher total ordering is provided as described in Section 6.3.1.

In order to support the above pub/sub model, brokers maintain a list of advertisements and subscriptions received at the broker. Brokers are also equipped with a routing table which maintains next-hop information to reach the source of known advertisements and subscriptions.

Our system guarantees weak pairwise total order under the presence of failure. Nodes can fail by crashing, but all links are reliable & maintain FIFO order.

The above properties of the system model are common in a number of distributed content-based pub/sub system [24, 45, 46].

### 6.4.2 Ordering conflicts

We define a conflict as an out-of-order delivery of publications to two or more subscribers. We wish to detect any possible conflicts and resolve them prior to delivery, making our solution pessimistic in nature. The use of FIFO links provides a “natural” ordering for many situations.
Figure 6.4: Natural total order using FIFO links

Theorem 13. (Natural Total Order) In an acyclic topology, given publications $p_1$ and $p_2$, which are delivered to both subscribers $s$ and $t$, $p_1$ and $p_2$ are delivered using path $S$ to $s$. Similarly $p_1$ and $p_2$ is delivered using path $T$ to $t$, respectively. If $S \cap T \neq \emptyset$, then FIFO links preserve pairwise total ordering between $p_1$ and $p_2$ for $s$ and $t$.

Proof. Suppose there exists a broker $n$ which is common to both paths. This broker receive publications $p_1$ and $p_2$ in some order. We can then consider $n$ as a publisher who is sending $p_1$ and $p_2$ to subscribers $s$ and $t$. According to Theorem 11, per-publisher total order is preserved for publications $p_1$ and $p_2$. Thus, subscribers $s$ and $t$ will receive these two publications in the same order, preserving total order. \qed

According to Theorem 13, it is sufficient for publishers to share a common broker along the path to all overlapping subscribers to guarantee total ordering using FIFO links. Figure 6.4 illustrates this property: Broker B3 is common to all paths from publishers to S1, S2, S3 and will enforce some ordering on the publications which will be retained by FIFO links.

Conflicts can therefore only arise when the overlapping subscribers do not have a common broker (see Figure 6.5). In this example, the path from P1 to S1 (P1-B1-S1) does not share a common node with the path from P2 to S2 (P2-B2-S2). If P1 and P2 publish in parallel, B1 will receive the publication B from P1 before publication R
from P2. Similarly, publication R from P2 are received at B2 before B from P1. Since FIFO order is preserved, S1 will then receive the publications in the order enforced by B1 while S2 uses the ordering by B2, where those orders are not guaranteed to be uniform. To resolve this conflict, three solutions are possible:

1. Mobility and topology reconfiguration: Natural total ordering can be restored by migrating peers to different brokers. Brokers can be reorganized into a different topology to alter the delivery paths. Clustering algorithms are particularly useful here to partition the topology by identifying groups of subscribers with similar interests (e.g., subscribed to the same topics).

2. Adaptive routing: Publications can be routed differently to include a common broker on the delivery path. This is notably relevant in general topologies where multiple delivery paths can be employed.

3. Conflict resolution: A conflict resolution process is engaged when a conflict is detected to enforce an ordering on the publications.

We will focus on the third solution, as it maintains the distributed nature of the system better than the other two (which seek to centralize instead) and is relevant to
the system model (i.e., acyclic topologies with FIFO links). The two other solutions are outside the scope of this thesis.

6.4.3 Conflict detection

We now present a conflict detection algorithm for brokers. The algorithm requires no global knowledge: any additional information maintained by the algorithm can be derived or obtained from the usual process of advertisement-based forwarding [24].

The algorithm is triggered upon receipt of a publication and can only detect potential conflicts based on advertisements received from other publishers. A broker cannot instantly determine whether there exists another publication in transit conflicting with this one. It can, however, determine if there is a possibility of a conflicting publication coming from a certain publisher. If this possibility is detected, the conflict resolution algorithm is triggered (Theorem 18). In other words, the algorithm determines, based on the local knowledge of the broker at the moment the publication is received, whether or not to trigger conflict resolution.

Upon receiving any publication \( p \), the algorithm, executed at every broker along the delivery tree, must specifically detect the following:

1. A set of hops \( H \) where \( |H| > 1 \), and \( h \in H \) if there exists a subscriber that matches \( p \) whose next hop is \( h \),

2. At least one advertisement \( a \) such that \( h_a \), the next hop to reach the source publisher of \( a \), belongs to \( H \),

3. There exists two subscriptions \( s \) and \( t \) which both match publication \( p \) with next hops \( h_s, h_t \in H \) respectively, and the interest space \( a \cap s \cap t \neq \emptyset \), where the next hop \( h_a = h_s \) and \( h_a \neq h_t \).
Point 1 detects whether all subscribers share the same next hop. If they do, then the current broker constitutes a common broker for the delivery of all publications upstream of this broker. Any potential conflicts occurring downstream will be processed by the next hop brokers. Thus, the current broker does not require further processing.

Point 2 detects which publishers are potentially conflicting. If all the known publishers do not have the same next hop as any matching subscriber for the publication being processed, then the current broker is again common to delivery to all subscribers, regardless of the next hops of the subscribers.

Point 3 verifies that some publisher downstream can issue a conflicting publication. In particular, we detect whether it is possible for such a publisher to create a publication $q$ which will be delivered to a subscriber $s_1$ which also matches $p$ without passing through the current broker and have this publication $q$ delivered to another subscriber $s_2$ which also matches $p$ through the current broker. This would indicate that the delivery path from $p$ to $s_2$ and $q$ to $s_1$ would be disjoint, in which case a conflict can occur. If the publisher only matches subscribers at the same next hop from the current broker, then the next hop broker can detect and resolve any conflict.

\begin{algorithm}
\caption{conflictDetection(p, in)}
\begin{algorithmic}[1]
\If{$|\text{next}(p)| = 1$}
\State $\text{forward}(p, h), h \in \text{next}(p)$;
\Else
\State \For{$h \in \text{next}(p)$}
\State \For{$s \in \text{match}(p)$, where $\text{nh}(s) = h$}
\State \For{$a \in \text{pubs}(h, s)$}
\If{$X \neq \emptyset$, where $X = \text{nhs}(a) \cap \text{next}(p) \setminus \{h, \text{in}\}$}
\State \For{$t \in \text{match}(p), nh(t) \in X$ where $\text{isMatching}(a, s \cap t)$}
\State $\text{conflictResolution}(p, h, nh(t), s)$;
\EndFor
\EndIf
\EndFor
\EndFor
\EndFor
\EndIf
\end{algorithmic}
\end{algorithm}
Algorithm 12 uses two hash tables: the pubs(h, s) table is indexed by subscribers, returning the list of advertisements matching s, where a must have next hop h; the nhs(a) table is indexed by advertisements and returns list of next hops for subscribers matching a. These data structures are updated during advertisements, subscriptions, and unsubscriptions.

The function match(p) returns matching subscribers for p on outgoing links and is provided by the broker logic. nh(s) indicates the next hop to reach subscriber s. The function next(p) computes the set of next hops for those matching subscribers. forward(p, h) forwards the publication down the link h. conflictResolution(p, h, g, t) is a special conflict resolution protocol for forwarding a publication down to a hop h, knowing there is a conflict with hop g due to subscription t. nh(s) is the next hop for a subscriber s. isMatching(s_1, s_2) determines whether the interest spaces of two subscriptions s_1 and s_2 are intersecting or not.

The input for the algorithm is the publication p and in, which is the incoming edge.

We now prove the correctness of the algorithm:

**Lemma 14.** Let p and q be publications delivered to both s_1 and s_2 using paths P_1, P_2, Q_1 and Q_2 respectively. If P_2 ∩ Q_1 ≠ ∅, then P_1 ∩ P_2 ∩ Q_1 ∩ Q_2 ≠ ∅.

**Proof.** Let x_c be some broker in P_2 ∩ Q_1. Then Q_1 can be decomposed as sub(Q_1, x_q, x_c) • sub(Q_1, x_c, s_1), where x_q is the publisher of q. Furthermore, P_2 can be decomposed as sub(P_2, x_p, x_c) • sub(P_2, x_c, s_2), x_p being the publisher of p. Thus the path sub(Q_1, x_q, x_c) • sub(P_2, x_p, x_c) exists and must be Q_2 since the topology is acyclic. Likewise, the path sub(P_2, x_p, x_c) • sub(Q_1, x_c, s_1) exists and is P_1. Therefore, x_c ∈ P_1 ∩ P_2 ∩ Q_1 ∩ Q_2. □
Lemma 15. Given publication $p$, which is delivered to $s_1$ and $s_2$ using paths $P_1$ and $P_2$ respectively, there exists at least one node $n$, $n \neq \text{pub}(p)$ where $n \in P_1 \cap P_2$.

Proof. Suppose not, then $P_1 \cap P_2 = \{\text{pub}(p)\}$. However, our model does not allow multiple connections for clients, which means there exists a single edge broker $b$ to which $\text{pub}(p)$ is connected to. Therefore, $b \in P_1 \cap P_2$. \qed

Lemma 16. Given publication $p_1$ and $p_2$, which are both delivered to $s$, using paths $P_1$ and $P_2$ respectively, there exists at least one node $n$, $n \neq s$ where $n \in P_1 \cap P_2$.

Proof. Suppose not, then $P_1 \cap P_2 = \{s\}$. However, our model does not allow multiple connections for clients, which means there exists a single edge broker $b$ to which $s$ is connected to. Therefore, $b \in P_1 \cap P_2$. \qed

Lemma 17. Let $p$ and $q$ be publications delivered to both $s_1$ and $s_2$ using paths $P_1$, $P_2$, $Q_1$ and $Q_2$ respectively. If $P_1 \cap Q_2 = \emptyset$, then $P_2 \cap Q_1 \neq \emptyset$.

Proof. Let $y_q$ be a broker in $Q_1 \cap Q_2$. $y_q$ must exist since $Q_1 \cap Q_2 \neq \emptyset$ according to Lemma 15. Let $y_1$ be a broker in $P_1 \cap Q_1$, which must exist due to Lemma 16. $y_1 \neq y_q$ since $P_1 \cap Q_2 = \emptyset$. There is then a subpath $\text{sub}(Q_2, y_q, s_2)$. There is also a subpath $\text{sub}(Q_1, y_q, y_1)$. Since $P_1$ crosses $y_1$, there is a path from $x_p$ to $s_2$ constructed as follows: $\text{sub}(P_1, x_p, y_1) \cdot \overline{\text{sub}(Q_1, y_q, y_1)} \cdot \text{sub}(Q_2, y_q, s_2)$ (recall that the topology is undirected: reverse paths can be employed). Since the graph is acyclic, this path must be $P_2$. Therefore $P_2$ and $Q_1$ are not disjoint, sharing at least $y_1$ and $y_q$. \qed

Theorem 18. Given a publication $p$, Algorithm 12 detects all potential ordering conflicts with $p$.

Proof. Suppose there exists an advertisement $a$ from which a publication $q$ can be generated to be delivered to subscribers $s_1$, $s_2$ using paths $Q_1$ and $Q_2$ respectively.
$p$ is delivered to subscribers $s_1$ and $s_2$ using paths $P_1$ and $P_2$ respectively. From Lemma 13 and Lemma 14, FIFO links guarantee total ordering if $P_2 \cap Q_1 \neq \emptyset$. Upon dissemination of $p$, each broker part of the delivery tree is guaranteed to receive the publication, since the algorithm only defers forwarding to subscribers at the edge brokers (see Section 6.4.4).

According to Lemma 15, $P_1$ and $P_2$ will share a common broker. Furthermore, all common brokers must form a consecutive subpath of both $P_1$ and $P_2$, otherwise there exists a loop in the topology, which is not permitted.

Let $b$ denote the last broker of that common subpath of $P_1$ and $P_2$. By definition, $nh(P_2, b) \neq nh(P_1, b)$. Under the advertisement-based forwarding scheme, advertisements are flooded while subscriptions are held at every broker along the delivery path. Thus, $b$ holds the subscriptions of $s_1$, $s_2$ and the advertisement $a$.

Advertisement $a$ was propagated to broker $b$ from publisher $x_q$ using some path $A_b$. Then, $nh(P_2, b) = nh(A_b, b)$, where $nh(P_2, b)$ is the next hop from broker $b$ to reach $s_2$. If the previous claim is not true, then $A_b \bullet sub(P_2, b, s_2)$ constitutes a loop-free path which must be $Q_2$. This would mean that $b \in Q_2$, which cannot be true since $b \in P_1$ and $P_1 \cap Q_2 = \emptyset$.

The detection algorithm at $b$ will detect that $a$ has the same next hop as $s_2$. As stated earlier, $nh(P_1, b) \neq nh(P_2, b)$. Therefore, the path $Q_2$ must be disjoint from $P_1$. Suppose it is not, then there exists a broker $c$ which is common to $Q_2$ and $P_1$ which is located either before or after $b$ on $P_1$. If it is before $b$, then the path $A_b \bullet sub(P_2, b, s_2) = Q_2$. In this case, $nh(P_2, b) \neq nh(A_b, b)$ which contradicts the above. If it is after $b$, then the path $sub(P_1, x_p, c) \bullet sub(Q_2, c, s_2) = P_2$. This means $nh(P_1, b) = nh(P_2, b)$ which contradicts the former statement that $b$ was the last common broker on paths $P_1$ and $P_2$.

Since the algorithm iterates over every matching advertisement and subscription,
it must detect all conflicts with \( p \).

\section*{6.4.4 Conflict resolution}

Once a conflict is detected at the time of forwarding a publication, the broker must defer delivery to directly connected subscribers until it is resolved. The broker employs a conflict resolution scheme to determine whether there are conflicting publications (from conflicting advertisements) in transit at the time. If none, the broker can safely deliver the publication. Otherwise, the broker must deliver those conflicting publications in the correct order. This order is enforced using publication IDs. Each publication’s ID is a combination of its publisher’s ID and a sequence counter which is monotonically increasing for that publisher. The publishers’ IDs are unique and provided by the pub/sub system.

The resolution is processed through acknowledgments. The broker will piggyback a request on the conflict publication to brokers who are the next hops to the source of conflicting advertisements. Those brokers process the request and send an acknowledgment back if it can determine that there is no conflict. Otherwise, the request continues to be carried downstream.

Note that the conflict resolution process does not prevent a broker from forwarding the publication to other brokers. If a conflict is detected and the next hop is a broker, the publication is still forwarded, but tagged as a conflict. The next broker processes it as usual, but understands that it must wait for an acknowledgment from the incoming edge before delivering to subscribers who match the specified advertisements from the incoming edge. This is formalized in Theorem 19.

Algorithm 13 takes as input a publication \( p \), hops \( h \) and \( g \) which contain subscriptions which are involved in the conflict, and \( t \) is a subscription involved in the conflict.
whose next hop is $h$.

The algorithm maintains $\text{acks}(p, h)$, a list of pending acknowledgment messages which must be received before delivering $p$ to $h$. Publication $p$ contains additional data: the request $R$ and wait flag $W$. These flags can be turned on using the operations $\text{flagR}(p, h)$ and $\text{flagW}(p, h)$ for the copy of the publication $p$ destined for hop $h$, respectively. The status of these flags can be queried using $\text{isFlagW}(p)$ and $\text{isFlagR}(p)$. $\text{addSub}(p, t)$ stores a subscription which conflicts with the publication. $\text{delay}(h)$ delays all subsequent publications to $h$ until $\text{deliver}(h)$ is called.

Algorithm 13 is invoked at Line 9 of Algorithm 12.

\begin{algorithm}
\textbf{Algorithm 13:} conflictResolution($p, h, g, t$)
\begin{algorithmic}
\State $\text{flagR}(p, f) ;$
\State $\text{flagW}(p, h) ;$
\State $\text{addSub}(p, t) ;$
\State append $f$ to $\text{acks}(p, h) ;$
\If {$h$ is a subscriber}
\State $\text{delay}(h) ;$
\EndIf
\end{algorithmic}
\end{algorithm}

We now integrate the conflict resolution mechanism with the detection. $\text{brokerForwarding}$ (Algorithm 14) is invoked upon receiving a publication $p$ from an incoming edge $in$. $\text{isFlagR}(p)$ and $\text{isFlagW}(p)$ are used to determine if the publication $p$ carry flags $R$ or $W$, respectively. $\text{conflictDetection}(p, in)$ is the conflict detection algorithm presented earlier (Algorithm 12). $\text{safeDetection}(p, in)$ checks if a conflict is possible or resolved and flags the publication appropriately. $\text{returnAck}(p, h)$ returns an acknowledgment for $p$ on hop $h$. $\text{subsC}(p)$ returns all subscriptions added to the publication through $\text{addSub}(p, t)$. Note that the publication does not retain the flags from the incoming publication; they must be set again through $\text{conflictDetection}()$ by this broker. Forwarding only occurs at the end of the algorithm; publications are buffered until then.
Algorithm 14: brokerForwarding(p,in)

if isFlagW(p) then
  flagW(p,h) ;
  append in to acks(p,h) for h = nh(c), \forall c \in subsC(p) ;
conflictDetection(p,in) ;
if isFlagR(p) then
  safeDetection(p,in) ;
  if acks(p,in) = {} then returnAck(p,in) ;
else
  append acks(p) to acks(p,h) for all h \neq in, where h is a subscriber
  commitForwards() ;

Algorithm 15: safeDetection(p,in)

foreach h \in next(p) do
  if acks(p,h) = {} then
    foreach s \in match(p) where nh(s) = h do
      if pubs(h,s) \neq {} then
        append h to acks(p,in) ;
        flagR(p,h) ;
      else
        append h to acks(p,in) ;
        flagR(p,h) ;

Algorithm 16: ackProcess(p,in)

if acks(p,h) = \{in\}, \forall h \in match(p) then
  if h is a subscriber then deliver(h) ;
else
  returnAck(p,h) ;
else
  remove in from acks(p,h) ;
safeDetection($p, in$) (Algorithm 15) ensures that there are no additional conflicts detected and that subscribers matching $p$ are on separate links as other advertisements matching $s$.

Finally, we present the code executed upon receiving an acknowledgment when there are more acks required before delivering a publication to a given hop. If there are none, the buffered messages are delivered in the correct order (if the hop is a subscriber) or an acknowledgment is sent (if it’s a broker). Algorithm 16 shows how an acknowledgment for a publication $p$ from $in$ is processed.

In summary, the algorithm waits until all pending acknowledgments are received in order to deliver a publication along a certain hop. These acks are received after all conflicting publications and therefore indicates that the broker can proceed to deliver these publications in the correct order.

We prove the correctness of the overall solution in maintaining total order:

**Theorem 19.** Given a publication $p$, which has been detected to have a conflict with advertisement $a$ (with possible publication $q$) at broker $b$, the resolution algorithm maintains total order.

**Proof.** At $b$, we can divide subscribers into two groups: subscribers on the same hop as $a$ are on the request side and have a disjoint delivery path as subscribers on other hops (called waiting subscribers). For every request-side subscriber of $p$, there are two cases: either $q$ is already deferred at the edge broker, or not. If $q$ has been deferred at one subscriber of $p$, then it has been deferred at all request-side subscribers of $p$, since per Theorem 18, $q$ has triggered a resolution process through the detection mechanism for these subscribers. If it has not deferred, then either $q$ does not exist yet or has not reached any request-side subscriber of $p$. In this case, the subscribers downstream of the request hop can safely deliver the publication without being delayed, since the
delay is triggered only if $q$ has been deferred first. According to Lemma 17, the path for $p$ to all request-side subscribers have at least one common broker with the path for $q$ to the waiting subscribers of $p$. One of these brokers must see the next hop towards the source of $a$ as being different as all the matching subscribers. Otherwise, $q$ can be delivered to the subscribers without crossing any of these common brokers, which is a contradiction that there was a conflict on $q$. Using the safeDetection algorithm, this broker will propagate an acknowledgment back to $b$, freeing the edge brokers to deliver all waiting subscribers.

If the request-side subscribers were deferred because of $q$, then $p$ will be queued to those subscribers as well. However, since deferrals occur only at edge brokers and only for subscribers, every broker on the delivery path will receive the publication (along with its request flag) and some broker will be able to propagate an acknowledgment back. Note that this acknowledgment will be propagated after $q$, such that $b$ will disseminate $q$ before receiving the acknowledgment (since FIFO links are used). Therefore, all the edge brokers of waiting subscribers of $p$ will deliver both $p$ and $q$ in a sorted order. Similarly, the algorithm is symmetric to $q$: $p$ has been received and deferred before the acknowledgment. If it is not, it means some broker on the delivery path to the request-side subscribers of $q$ sent an acknowledgment back before receiving $p$. The broker which can acknowledge back for $q$ must be on the delivery path of $p$ to the request-side subscribers of $q$, therefore if the acknowledgment is sent before $p$, all the request-side subscribers of $q$ will receive $q$ and deliver it right away before $p$ triggers a conflict resolution process. Therefore, $p$ must be received before the acknowledgment of $q$ and again the waiting subscribers of $q$ (which corresponds to the request-side subscribers of $p$) deliver $p$ and $q$ in sorted order. Thus, all the overlapping subscribers of $p$ and $q$ will deliver the two publications in the same order. \[\square\]
6.4.5 Safety under failures

The algorithm is proven to be safe in a system with crash failures (see Theorem 20 with proof). Links are assumed to be reliable and FIFO (i.e., TCP). Messages can be lost at brokers and clients during failures. However, the set of publications actually delivered by each client has the same order, thus maintaining weak pairwise total order.

Theorem 20. Given publications $p$ and $q$ from $x_p$ and $x_q$, which are delivered to both $s_1$ and $s_2$ using paths $P_1$, $P_2$, $Q_1$ and $Q_2$. Under the presence of failures, if $s_1$ delivers $p$ before $q$, then $s_2$ only delivers $p$ and $q$ only if $p$ is delivered before $q$.

Proof. Consider the delivery of $p$ and $q$ to $s_2$.

If no conflicts are detected at any brokers along their respective paths, the publications are simply propagated in FIFO order to $s_2$. No additional messaging is required, therefore only the publications themselves can be lost due to failures. If a broker fails for either publication, it will not deliver that publication to $s_2$: the algorithm does not employ any form of retransmission or reordering.

If conflicts are detected, then the algorithm triggers a conflict resolution process. Suppose a conflict is detected for $q$ on some broker $b$ in $Q_2$. Delivery of $q$ is deferred until an acknowledgment is returned to $b$. If, due to failures, this acknowledgment never makes it to $b$ (either because the request or ack was lost), then $s_2$ will never receive $q$ since its delivery is deferred indefinitely. If the acknowledgment does come back, then failures can only affect the delivery of $p$ to $s_2$ via $b$: $p$ can only be lost but cannot be reordered with the acknowledgment.

Therefore, for both $p$ and $q$ to be delivered at both $s_1$ and $s_2$, no failures should occur in the system affecting their delivery. If any message loss occurs, $p$ and $q$ may not be delivered to both subscribers, which only violates liveness, not safety (no
out-of-order delivery).

6.4.6 Optimizations

We offer two optimizations to our basic algorithm. One weakness of the original algorithm is that every ack must be received in order to deliver publications. A livelock situation can occur if the rate of new conflicting publications received is greater than the rate of acknowledgments received, since the queue of deferred publications will always be blocked as there are pending acks. Progressive delivery of the publications is possible by scanning from the head of the queue of deferred publications and delivering every publication until a non-acknowledged publication is reached. This subset of acknowledged messages must be sorted according to publication IDs, as usual.

Our second optimization seeks to minimize the detection overhead and sets a bound on the metadata attached to conflicting publications. For a given hop, the detection algorithm stops checking as soon as it determines that it is conflicting with another hop. The algorithm therefore determines every pair of hops <r, w>, where the hop w is waiting for an acknowledgment to a request sent along r. During the safeDetection phase, the downstream brokers from the request side do not exactly know which advertisements and subscriptions are conflicting; instead they assume that advertisements for subscribers matching the publication are also overlapping with subscribers upstream. Likewise, brokers downstream from the waiting side do not know exactly which subscribers are conflicting: they simply assume every matching subscriber they have is conflicting and must be deferred.

There exists a tradeoff between detection overhead and false positive conflict rate. This has an impact on end-to-end delay of non-conflict subscribers and resolution overhead. On the other hand, publication size is fixed since no advertisement or
subscription information is necessary; the request and wait flags are sufficient.

6.5 Evaluation

In this section, we experimentally evaluate our solution and its optimizations against two baselines: one which provides only per-publisher total ordering using FIFO links and another which uses a central sequencer. We also compare our approach in terms of performance to Spread [6], a group communication system.

Setup: The algorithms are implemented in Java as a module for the PADRES pub/sub prototype.\(^1\) Our experiments run in two testbeds: a cluster of 24 machines each with four 1.86 GHz Xeon processors and 4 GB of RAM (referred to as PADRES-CLUSTER), and for larger experiments, on SciNet \(^2\) using 96 GPC (General Purpose Cluster) machines. We use a synthetic workload, where each publisher sends a publication every 4 seconds and each publisher emits a single advertisement.

The network topology consists of several central brokers, connected in a chain. Each core broker is then connected to 5 edge brokers. This topology is modeled after interconnected data centers. Communication across data centers must then pass through the core brokers. For 96 brokers (on SciNet), there are 16 central brokers, each connected to 5 edge brokers. Advertisements and subscriptions are uniformly distributed across the edge brokers. In particular, every advertisement matches every subscription at a particular edge broker. This setup maximizes the number of delivery paths containing only one broker, which increases the number of potential conflicts since delivery paths are less likely to have a common broker.

The publication rate, number of subscriptions, advertisements and interest overlap constitute the parameters which vary across the experiments.

\(^1\)http://padres.msrg.toronto.edu/

\(^2\)http://www.scinethpc.ca/
Metrics: We use the following metrics:

Detection delay: We measure the processing time required at each broker to detect conflicts for a publication. This is an important metric since the overhead is incurred on every subscriber, regardless of their conflict status. We seek to minimize this overhead as much as possible to limit the impact of our algorithm on non-conflicting subscribers.

End-to-end delay: This covers the detection and resolution overhead required before finally delivering the publication. The end-to-end delay with and without conflicts are compared to get an estimate of the resolution overhead. The metric only evaluates delivery of publications directly connected to the same broker as the subscriber: the impact of total order is greatest in those cases as the delivery without total order only involves one broker, whereas conflict resolution now requires waiting for acknowledgments from other brokers in the topology.

Outgoing traffic: We count the outgoing messages at each broker. This number is expected to increase with total ordering due to the presence of control messages.

Ordering degree: For the FIFO approach, we compute the ordering degree to evaluate how out-of-order the delivery stream is [10] for publications common to two subscribers. The publication ordering degree is computed as \( \sum_{D=0}^{S} (S - D)f(D) \), where \( S \) is the length of the stream (number of common publications), \( D \) is the distance the publication was displaced, \( f(D) \) is the frequency (number of publications) which was displaced by distance \( D \). This is then normalized by dividing by the maximum ordering degree possible, \( S^2 \). Total order provides a perfectly ordered stream that has a normalized degree of 1. The requirements for ordering is application-dependent.
Chapter 6. Total order

6.5.1 Performance results

Unoptimized algorithm: The unoptimized algorithm waits for acknowledgments of every pending publication before delivering the whole buffer. Publication rate becomes an important concern here: a livelock can occur if the delivery queue for publications fills up faster than the acknowledgment rate and no publications ever get delivered. In our setup, a rate of 120 conflicting publications/minute is enough to saturate the system, with three publishers and subscribers. Thus, no meaningful figures or data are produced for our regular solution as it simply does not scale with respect to publication rate.

Progressive stream delivery algorithm: This optimization avoids the livelock problem by delivering a subset of the pending queue at each acknowledgment. This is possible by scanning the queue and delivering publications from the head of the
queue until the next unacknowledged publication.

We measure the relative detection delay when varying the number of subscriptions and advertisements, as well as the fraction of overlapping publications on PADRES-CLUSTER. In other words, a 50% conflicting workload signifies that each publication will conflict with 50% of the subscriptions and advertisements. Furthermore, the advertisements and matching subscriptions at each edge broker are identical to each other. This will reduce the number of brokers for each publication’s delivery tree. The publication rate also increases linearly in the number of subscriptions and advertisements.

Figure 6.6(a) shows that at 100%, the detection overhead can take up to 10 seconds at each broker when there are 300 combined subscriptions and advertisements at the broker. Figure 6.6(b) shows the detection overhead at the core broker. The overhead for the 100% workload peaked at 7 seconds while for the other cases it is less than 100 ms. Since core brokers do not serve as edge brokers themselves, this suggests that processing and storing publications at the edge is more expensive. Figure 6.6(c) shows the number of outgoing messages at a broker during a fixed period. The conflict percentage greatly affects the traffic volume through the number of acknowledgements generated, as demonstrated by the peak of 9000 messages for 100%, 6000 for 50% and 2000 for 25%. End-to-end delay also increases with the number of subscriptions and advertisements and is proportional to the fraction of conflicting publications, as shown in Figure 6.6(d).

We also notice that with 100% conflict, some of our brokers will exceed the allotted Java heap space of 512MB when storing publications. This is due to the queue storing publications of unbounded size, since each publication must carry the entire set of advertisements and subscriptions it is conflicting with. Combined with the long end-to-end delay, the queues at the edge brokers are growing and consume more and more
Chapter 6. Total order

(a) Detection overhead  (b) Outgoing traffic  (c) End-to-end delay

Figure 6.7: Fast detection algorithm

memory.

In summary, this optimization is not enough to achieve our desired scalability. The detection overhead is proportional to the number of advertisements and subscriptions since it must check every one of them for conflict at every broker. Furthermore, the publication message size is unbounded and depends on the number of conflicts.

**Fast detection algorithm:** The fast detection algorithm addresses the issues encountered with the previous algorithm. We focus on the performance of the fast detection algorithm by constructing a workload with no false positives (using the PADRES-CLUSTER). In Figure 6.7(a), the overhead of the detection algorithm is proven to be stable over varying loads of subscriptions and advertisements, averaging less than 0.8 ms. Also note that the algorithm incurs more delay on average when there are fewer conflicts (e.g. 25% vs. 50%), since every advertisement and subscription must be checked on a given hop to conclude that there is no conflict.

The traffic curve behaves similarly as in the previous algorithm, since it improves only detection time and the workload does not incur any false positives (see Figure 6.7(b)).

The end-to-end delay also benefits greatly from the scalability of the fast detection algorithm. Figure 6.7(c) shows that the end-to-end delay stays stable over an increasing number of advertisements and subscriptions. This is due to the uniform
distribution of subscriptions and advertisements over the entire topology, which signifies that the number of brokers involved in the delivery tree of a publication does not change when more subscriptions are introduced.

6.5.2 Comparison with FIFO and a centralized sequencer

We compare the performance of our optimized algorithm using fast detection to two baselines. First, we look at the pub/sub system using only FIFO links, which maintains per-publisher ordering. Second, we implemented a central sequencing service. Publishers must request a sequence number before sending the publication. The setup for this experiment uses 800 subscriptions and 240 advertisements which are conflicting at a rate of 100% over a topology of 16 core brokers and 80 edge brokers running on SciNet.

In Figure 6.8, the growth rate of end-to-end delay is linear to the number of subscriptions and advertisements in all cases: our solution (FD-Opt), per-publisher (FIFO) and central sequencer (SEQ). Although our overhead is the highest, it shows that our solution is scalable. However, the latency variance is much greater for our solution: publications have a maximum latency of 10 seconds, while remaining under 200 ms for the baseline solutions.

While the central sequencer performs better in terms of latency, the sequencer throughput becomes a bottleneck. Throughput saturates at 130 publishers: additional publishers are not able to send any publications because their sequencing requests are discarded. In comparison, all 240 publishers are functional in our solution’s experiments.

We evaluate the ordering degree of the FIFO solution by sampling random pairs of subscribers from different brokers and comparing their publication streams with each
other. On average, ordering discrepancies were found once every 14 publications. The displacements were never more than 2. The normalized ordering degree was calculated to be 0.9952.

### 6.5.3 Comparison with Spread

Spread is a popular group communication toolkit written in C, notably used for distributed logging in Apache servers. It supports total order (referred to as agreed ordering) by circulating a token around the daemons [8]. We also compare against Spread without any ordering (called reliable).

As a group communication system, Spread fulfills a different role than pub/sub. Groups are at the same level of expression as topics in pub/sub. On the other hand, our content-based system would be analogous to having per-client filtering of messages within each group.

Brokers in our topology are mapped to daemons in Spread and can only communicate through unicast links (i.e., separate segments), as in PADRES. Furthermore, daemons in Spread are required to have complete knowledge of the domains, as they are arranged in a fully connected network. In contrast, our brokers can be arranged in any acyclic topology and are provided only with local information (e.g., direct

---

Figure 6.8: Baseline comparison: end-to-end delay
neighbors). Note that Spread has a soft limit of 20 segments and therefore can only accommodate our 20 edge brokers (no core brokers).

Experiments are conducted on PADRES-CLUSTER, with a 100% conflict rate. We measure the end-to-end latency as the number of daemons (brokers) increases (see Figure 6.9), with each daemon carrying 50 publishers and 50 subscribers.

With 1 daemon, Spread with total order (Spread-AGREED) performs better than our solution (FD-Opt), with less than 5 ms delay, since the token algorithm incurs no overhead. With 2 daemons, the delay immediately jumps to 200 ms, which is still comparable to the 100 ms provided by our solution. But we can quickly see that Spread does not scale to larger number of daemons, hovering at a latency of 700 seconds for 20 brokers and 1000 subscribers. In comparison, our solution remains in the sub-250 ms range.

To determine the impact of total ordering on Spread, we also measured its performance without any ordering guarantee. Although the performance of Spread-REL deteriorates rapidly after 16 daemons, the latency is still 41 s even at 2000 subscribers, which means the total order overhead increases the latency by a factor of 17. On the other hand, our solution produces an increase by a factor of 10.
6.6 Conclusions

This chapter investigates the issue of total ordering in content-based pub/sub systems. The semantics of total ordering are defined and compared with topic-based pub/sub and other messaging models. The main contribution of this chapter is a solution to the content-based pub/sub model using reliable FIFO channels. The solution does not require any global knowledge and is implemented directly in the brokers, eliminating the need for an external sequencing service. The solution leverages FIFO links as much as possible in order to contain the ordering overhead to publications and subscribers that truly require it. An evaluation of a real implementation of the algorithms shows that it scales well with the number of subscriptions and the publication size overhead is bounded.
Chapter 7

Conclusions

This chapter provides a summary of the findings of this thesis in Section 7.1, as well as an outlook on current and future work in Section 7.2.

7.1 Summary

Traditionally, pub/sub systems were optimized for latency and scalability, while being lightweight on the communication semantics provided. This was sufficient for earlier applications which required only simple, topic-based communication. However, this thesis argues that emerging applications for pub/sub now require systems that are more expressive and possess better QoS support. Although these specifications can be satisfied by applying the end-to-end principle to install complex event processing engines at the event consumers’ side, an integration of such functions within the pub/sub layer itself leads to improved performance. This thesis revolves around this hypothesis and makes concrete contributions in three different aspects.

In Chapter 4, online social networks are presented as an application which feeds overwhelming amounts of data to the users through a publish/subscribe system.
Therefore, ranked data dissemination is called upon to intelligently filter and retain the most relevant data for the users. This requirement is captured in the form of top-k publications filtering for content-based publish/subscribe systems. In order to quickly reduce the publication load carried by the pub/sub layer, we propose to distribute the top-k processing tasks to the brokers themselves. We show that the naive solution of processing early at the publisher edge brokers, while simple and effective, does not maintain correctness. We thus devise a hybrid solution which switches between chunks of top-k and full windows of publications. This chunking solution retains enough data to satisfy the correctness criteria, while benefiting from a reduction in load by distributing the top-k filtering. Our evaluation shows that online social networks exhibit properties which are advantageous to our solution.

In Chapter 5, we investigate the related problem of top-k publications filtering. Instead of ranking publications for a subscription, we are instead scoring subscriptions for a given publication. Each publication is thus disseminated to a subset of matching subscriptions. Applications of interest in this work include targeted marketing, online social networks, and location-based applications. We identify the challenge of supporting top-k publications due to its apparent incompatibility with subscription covering, a routing optimization technique frequently employed in large-scale pub/sub systems. We propose an extended covering model which allows enough subscriptions to be disseminated to support top-k filtering. In addition, our solution offers a framework for supporting other selection semantics, such as fairness and diversity of subscriptions. In particular, we show how fairness can be realized through the use of a novel efficient probabilistic shuffling algorithm. The evaluation shows that our solution is able to provide good covering performance when the top-k selectivity is low, which is the case in our targeted applications.

Finally in Chapter 6, total order is studied as a valuable QoS requirement for
certain pub/sub applications, such as online games. Providing total order through an external service (i.e., a sequencer) leads to unacceptable overhead for our desired workloads. Instead, our proposed solution leverages properties of the broker topology, as well as FIFO links, to achieve total order naturally. Essentially, certain brokers in the topology which are on the common path of multiple dissemination routes can act as sequencers and order publications. We also show corner cases where natural total order is not sufficient and provide an algorithm to detect such ordering conflicts. This algorithm is lightweight in the sense that it can be run locally and independently by each broker while servicing publications. We also propose a resolution algorithm to resolve any detected conflicts. This resolution algorithm is efficiently integrated along the usual dissemination paths employed by the conflicting publications in the topology. Our evaluation shows that in the usual case our solution outperforms alternatives by avoiding costly ordering operations. We also demonstrate the scalability of performing total order in publish/subscribe compared to group communication systems.

7.2 Future work

The objective of this thesis, which is to extend language expressiveness and quality of service support for publish/subscribe systems, is part of a more general goal to adapt event processing systems to suit a variety of demands put forth by modern applications. Besides developing extensions for event processing systems, another aspect of this overarching goal is to integrate them with new technologies, for instance software-defined networking (SDN) and MapReduce. We highlight some of the ongoing and future work in both research directions.
7.2.1 Extending event processing systems

The contributions of this thesis fall in this category. Using our methodology, we identify emerging application scenarios for event processing. We first motivate how those applications require or could be designed around event processing. We then elicit in more details requirements from the application which drive our research in extending event processing systems.

As previously mentioned in Chapter 1 and omitted from this thesis, we have performed a study to design network engines using publish/subscribe for online games [20]. In such games, each client controls a mobile avatar in a virtual world. The data necessary for a player to interact with its local environment is disseminated through the use of a pub/sub system. To the best of our knowledge, this work is the first thorough analysis of the benefits of using an expressive pub/sub language, namely content-based matching, within the context of a real application. After evaluating our designs using existing pub/sub semantics, we are currently developing extensions to cover the perceived weaknesses resulting from our study. In particular, we are developing the concept of “evolving” subscriptions: the ability to define subscriptions whose properties change automatically during its lifetime. This ongoing work is inspired by parametric subscriptions [64], except it can efficiently alter the subscription without requiring additional communication from the subscriber. This work is motivated by observations rising from the aforementioned study we conducted on online games. Due to the dynamic nature of online games, subscriptions and unsubscriptions represent a significant amount of the load experienced during our experiments. We propose to leverage the predictable nature of player movements (i.e. employing dead reckoning [28]) to reduce the amount of subs/unsubs needed to express the interest of a player with its local area.
In the aggregation work (also not part of the manuscript, more details can be found in [94, 93]), the model currently only considers per-NWR binary decisions. In other words, for a given window (NWR), a broker can decide either to aggregate the entire window, or to forward its content. It is not possible to partially aggregate a subset of publications for that window, while aggregating a partial result for the rest. Extending the solution space to allow for such decisions will likely push the complexity to NP-Hard [109], but should provide more opportunities to optimize the performance of our system.

We are also looking to continue progressing existing works contained in this manuscript. For the ranked data dissemination work, there are opportunities to optimize the top-k filtering effectiveness by rendering the solution dynamic. In particular, we want to explore in more details the rechunking aspect of the work which solves the guard offset problem. A cost model conveying the tradeoff between latency and filtering overhead needs to be formulated.

For the top-k subscriptions work, the framework we defined can support a variety of semantics. We are mostly interested in diversity, where the selection of top-k subscribers selected are the most dissimilar. However, enabling diversity with covering requires additional data to be propagated upstream to represent the subsumed subscriptions. We are considering using spatial clustering representations such as centroids [80] or employing space filling curves to measure the diversity distance between subscriptions [109].

For total order, we are looking to relax the assumptions made in our model. In particular, we are looking to support general topologies. This should be possible by tagging the tree identifier (tid [80]) to publications and detecting conflicts based on the tid, rather than the publishers. We are also considering optimizing the conflict resolution stage by offloading it to the subscribers rather than processing it at the
brokers.

### 7.2.2 Integrating new technologies

We show two examples of event processing works which incorporate new technologies:

**PLEROMA [113]:** Software-defined networking (SDN) is becoming an increasingly popular way to achieve agility and reduce the complexity of deploying large-scale dynamic networks. This paper suggests that SDN can be leveraged to achieve higher matching and delivery performances by integrating publish/subscribe matching semantics into the switches, while routing is defined at the controller. The design of the PLEROMA middleware employs a novel space partitioning technique to encode pub/sub data into IP-Multicast addresses which can be matched against by OpenFlow-enabled switches.

**StreamMine3G [86]:** StreamMine3G is an event processing system which is built on MapReduce principles by decoupling each specific operator and aligning them in a pipeline fashion. This allows for horizontal scaling and elasticity, as new nodes can be deployed and assigned to specific operators where congestion occurs. Recognizing that MapReduce scaling techniques can be applied to event processing is the strength of this work.

In these two examples, emerging technologies such as SDN and MapReduce are integrated in various ways. In the first case, the event processing middleware is adapted to operate using the target technology. In the second, the integration is less explicit: instead of employing the chosen technology, the core principles are extracted and applied to event processing systems. From this observation stems our key insight in this type of work: both types of interaction must be considered.
Chapter 7. Conclusions

Our novel two-way approach to integrating new technologies can generate key insights that would not be found under a straightforward treatment of the matter. As an example, one current work includes our own research on SDN, called SDN-Like [126]. Unlike PLEROMA, our work not only considers employing SDN as the underlying networking layer for a pub/sub system, but also demonstrate how pub/sub decoupling can be enhanced using the separation of the data and control plane. We have also studied the integration of our publish/subscribe system with Android and Apple smartphones [125]. In the future, we plan on studying in more depth Future Internet Architectures (FIA) such as CCN [17], as well as BigData systems such as Spark.
Bibliography


SIGMOD International Conference on Management of Data, SIGMOD ’07, pages 1100–1102, New York, NY, USA, 2007. ACM.


[42] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-


[53] GS1. GS1 GDSN is the GS1 Global Data Synchronisation Network. www.gs1.org/gdsn.


[62] Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramanayam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The


[87] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proceedings of the 31st Interna-


USENIX Association.


[120] Steven Euijong Whang, Hector Garcia-Molina, Chad Brower, Jayavel Shanmugasundaram, Sergei Vassilvitskii, Erik Vee, and Ramana Yerneni. Indexing


