Content-based Publish/Subscribe with P4

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von
Christian Wernecke, geb. am 31. Juli 1982 in Berlin

Rostock, 13. Juni 2025

https://doi.org/10.18453/rosdok_id00005146

Gutachter:

Prof. Dr.-Ing. habil. Gero Mühl
Universität Rostock, Institut für Informatik

Prof. Dr. Andreas J. Kassler
Technische Hochschule Deggendorf, Fakultät Angewandte Informatik

Prof. Dr. rer. nat. Clemens H. Cap
Universität Rostock, Institut für Informatik

Tag der Einreichung: 6. Januar 2025

Tag der Verteidigung: 8. Mai 2025

Abstract

Publish/subscribe is a flexible communication pattern for loosely coupled distributed applications. In the content-based variant, each published notification is matched against active subscriptions to determine a set of interested subscribers to which the notification is to be delivered. However, since the set of receivers can be any combination of subscribers, distributing notifications from publishers to subscribers with matching subscriptions is complex. Traditional approaches rely on *broker-based networks* to deliver each message to a unique set of recipients. However, brokers make forwarding decisions at the application-layer and typically perform application-layer multicast by sending multiple unicast packets, which increases network load and delays delivery.

With the advent of *Software-Defined Networking (SDN)*, new distribution strategies are possible that combine the efficiency of *network-assisted multicast* with the flexibility of *application-layer multicast*. In this thesis, we explore how SDN and the P4 programming language can be used to implement fast and flexible notification distribution. We present fundamental strategies that embed the entire delivery tree of a notification directly into the packet header, which conserves bandwidth and reduces latency compared to traditional methods.

Furthermore, we present advanced P4-based forwarding schemes that combine static routing rules in network switches with dynamically encoded additional forwarding information in the packet header. This hybrid approach allows precise control over the trade-off between compact headers and flexibility in notification distribution. Evaluations in an emulated datacenter network demonstrate how these strategies can significantly reduce header size and network traffic.

To further improve delivery efficiency, we introduce innovative distribution strategies that use *virtual trees*. These virtual trees map frequently and simultaneously used paths in the switch infrastructure as partial delivery trees. A specially designed encoding algorithm dynamically constructs notification-specific delivery trees by merging multiple virtual trees and encoding additional routing information in the notification header to extend, connect, or prune virtual trees. Evaluation results confirm that the integration of combined virtual trees with routing information in the header significantly reduces header size and network traffic.

Finally, this work introduces installation strategies that dynamically deploy virtual trees in the switch infrastructure based on various levels of application-dependent knowledge and runtime statistics. These strategies exploit topological information, publisher-subscriber relationships, and notification frequencies to construct efficient distribution trees that require minimal additional routing information in notification headers. Evaluation results highlight the efficiency of these strategies under different churn rate scenarios in real-world network topologies.

By leveraging SDN and P4, this work enables optimized, scalable, and high-performance notification distribution in content-based publish/subscribe systems, significantly reducing delivery latency and bandwidth consumption.

Zusammenfassung

Publish/Subscribe ist ein flexibles Kommunikationsmuster für lose gekoppelte verteilte Anwendungen. In der inhaltsbasierten Variante wird jede veröffentlichte Benachrichtigung mit aktiven Abonnements abgeglichen, um die interessierten Teilnehmer zu ermitteln, an die die Benachrichtigung zugestellt werden soll. Da die Empfänger jedoch jede beliebige Kombination von Abonnenten sein können, ist die Verteilung von Benachrichtigungen von Publishern an Subscriber mit passenden Abonnements komplex. Traditionelle Ansätze basieren auf brokerbasierten Netzwerken, die für jede Nachricht eine individuelle Empfängergruppe adressieren. Broker treffen Weiterleitungsentscheidungen auf Anwendungsebene und führen typischerweise anwendungsseitiges Multicast durch, indem sie mehrere Unicast-Pakete senden. Dies erhöht die Netzwerklast und verzögert die Zustellung.

Mit dem Aufkommen von *Software-Defined Networking (SDN)* eröffnen sich neue Möglichkeiten für Verteilungsstrategien, die die Effizienz von netzwerkbasiertem Multicast mit der Flexibilität des anwendungsseitigen Multicasts kombinieren. In dieser Arbeit untersuchen wir, wie SDN und die *Programmiersprache P4* eingesetzt werden können, um eine schnelle und flexible Benachrichtigungsverteilung zu implementieren. Wir präsentieren grundlegende Strategien, die den gesamten Verteilungsbaum einer Benachrichtigung direkt in den Paketheader einbetten, was im Vergleich zu herkömmlichen Methoden die Bandbreite schont und die Latenz reduziert.

Darüber hinaus entwickeln wir in dieser Arbeit fortschrittliche P4-basierte Weiterleitungsschemata, die statische Routing-Regeln in Netzwerk-Switches mit dynamisch kodierten zusätzlichen Weiterleitungsinformationen im Paket-Header kombinieren. Dieser hybride Ansatz erlaubt eine präzise Kontrolle des Kompromisses zwischen kompakten Headern und Flexibilität bei der Benachrichtigungsverteilung. Anhand von Evaluationen in einem emulierten Rechenzentrumsnetzwerk wird gezeigt, wie diese Strategien die Headergröße und den Netzwerkverkehr signifikant reduzieren können.

Um die Effizienz der Verteilung weiter zu steigern, führen wir innovative Verteilungsschemata ein, die *virtuelle Bäume* nutzen. Diese virtuellen Bäume bilden häufig und gleichzeitig genutzte Pfade in der Switch-Infrastruktur als partielle

Verteilungsbäume ab. Ein speziell entwickelter Kodierungsalgorithmus konstruiert dynamisch benachrichtigungsspezifische Zustellungsbäume, indem er mehrere virtuelle Bäume zusammenführt und zusätzliche Routing-Informationen im Benachrichtigungsheader kodiert, um virtuelle Bäume zu erweitern, zu verbinden oder zu beschneiden. Die Ergebnisse der Evaluierung bestätigen, dass durch die Integration von virtuellen Bäume mit Routing-Informationen im Header die Headergröße und der Netzwerkverkehr deutlich reduziert werden können.

Abschließend untersucht diese Arbeit Installationsstrategien, die virtuelle Bäume in der Switch-Infrastruktur dynamisch bereitstellen, basierend auf verschiedenen Niveaus von anwendungsabhängigem Wissen und Laufzeitstatistiken. Diese Strategien nutzen topologische Informationen, spezifische Beziehungen zwischen Publishern und Subscribern sowie Benachrichtigungsfrequenzen, um effiziente Zustellungsbäume zu konstruieren, die nur minimale zusätzliche Routing-Informationen in den Benachrichtigungs-Headern benötigen. Die Evaluationsergebnisse unterstreichen die Effizienz dieser Strategien bei sich ändernden Abonnements in realen Netzwerk-Topologien.

Durch den Einsatz von SDN und P4 ermöglicht diese Arbeit eine optimierte, skalierbare und leistungsfähige Benachrichtigungsverteilung in inhaltsbasierten Publish/Subscribe-Systemen, die die Zustelllatenz und Bandbreitennutzung deutlich reduziert.

Preface

Acknowledgements

I sincerely thank Gero Mühl and Helge Parzyjegla for their excellent supervision during my PhD. Their expertise and support have been of great value to me. In particular, I would like to thank Gero Mühl for the guidance he gave me during my research. His expertise and ability to see the bigger picture contributed greatly to the success of this dissertation. I would also like to thank Helge Parzyjegla for his attention to detail and many thoughtful comments. His critical remarks sharpened my focus on key aspects and helped to improve the quality of my work. The joint supervision of both mentors, characterized by constructive, open and friendly discussions, created an encouraging and and pleasant working environment for me. Last but not least, I would like to thank my partner Manuela Gutzeit for her moral support.

Contents

List of Figures

Chapter 1

Introduction

In today's interconnected world, the efficient and timely exchange of information is essential across a wide range of industries and applications. As the need for scalable and flexible communication mechanisms becomes increasingly apparent, traditional point-to-point communication models that rely on request/reply messages are becoming inadequate for dynamic, multi-party environments. This has led to the development of *distributed architectures* [75] and computing systems with advanced filtering techniques.

The evolution of data processing systems, driven by the rise of big data platforms [88] and the integration of big data processing techniques in cloud computing environments [63], has significantly raised expectations for these systems. There is an urgent need for scalability and, in particular, low-latency *stream processing* capabilities to manage massive amounts of data. This demand has accelerated the development of new data processing frameworks [2, 53, 77] that enable pattern matching, instant analysis and decision making on streams.

Stream processing engines find applications in various domains, notably in financial trading [49], where they facilitate the collection and dissemination of real-time market data to traders and stakeholders. This enables them to monitor market trends and make informed trading decisions based on up-to-date, or even real-time, information.

The proliferation of the *Internet of Things (IoT)* [68] and the rise of *machine-to-machine communications*, particularly in contexts like smart factories [121], have further increased the demands on data processing systems. To address the complex requirements of these systems, *event-driven architectures* [84] have been developed as a viable solution. A plethora of event-driven stream processing systems [34] specialize in the real-time processing and analysis of continuous *event streams*. These systems employ *complex event processing* techniques [19]. By decoupling information providers and receivers, these systems enable users to define real-time event processing logic and transformations for incoming data

streams. Their aim is to enable informed decision making through the composition and aggregation of data to identify meaningful patterns and relationships.

For example, in a manufacturing environment, IoT sensors attached to machines can publish real-time performance data as an event stream, triggering event-based responses within the network. This enables dynamic machine interactions and facilitates real-time monitoring and control of the manufacturing process.

A robust and flexible communication infrastructure is essential for distributing events generated by devices or components to interested parties. As a highly flexible model for event-driven architectures, the *publish/subscribe paradigm* has been introduced as a promising solution for efficient and scalable information dissemination in complex and dynamic environments.

The publish/subscribe pattern facilitates the exchange of messages between *publishers*, which provide information, and *subscribers*, which consume the published information. Instead of establishing direct connections, publishers and subscribers exchange messages through an intermediary *notification service*. This middleware handles the transmission of messages between the two parties without the need for either to know the other's identity. Publishers send *notifications* to the notification service, while subscribers express their interest in particular message types or content by sending *subscriptions*. Publishers can also advertise the content of their publications by sending *advertisements* to the notification service. The notification service ensures that messages are delivered to subscribers with matching subscriptions. Subscriptions can be based on *channels*, *topics*, *types* or the *content* of messages.

The earliest pub/sub systems are *channel-based*, where publishers publish their messages to specific channels and subscribers subscribe to one or more of these *hard-wired channels* to receive all the messages of interest, possibly filtering out irrelevant data if the channel does not fully match their interests. More flexibility is offered by *topic-based* systems. Publishers associate notifications with topics, which can be ordered in a hierarchical structure. Subscribers can use this *hierarchical structure* to subscribe to a specific topic or multiple subtopics. The granularity of message selection is further increased by *type-based* systems, where the declaration of application-defined *types* provides the core mechanism for categorizing information. Publishers distribute messages that are instances of particular types, while subscribers receive publications of interest by declaring an interest in a particular type or subtype.

The *content-based* variant, however, provides the finest message selection. Here, subscribers specify complex filter conditions that the message content must meet in order to be delivered to them. This granular control allows subscribers to receive only the information directly relevant to their preferences, eliminating the burden of filtering irrelevant data.

The ability to select messages based on their content offers several advantages, including precise targeting, reduced message overhead, and improved scalability. By ensuring that messages are delivered only to subscribers that have explicitly

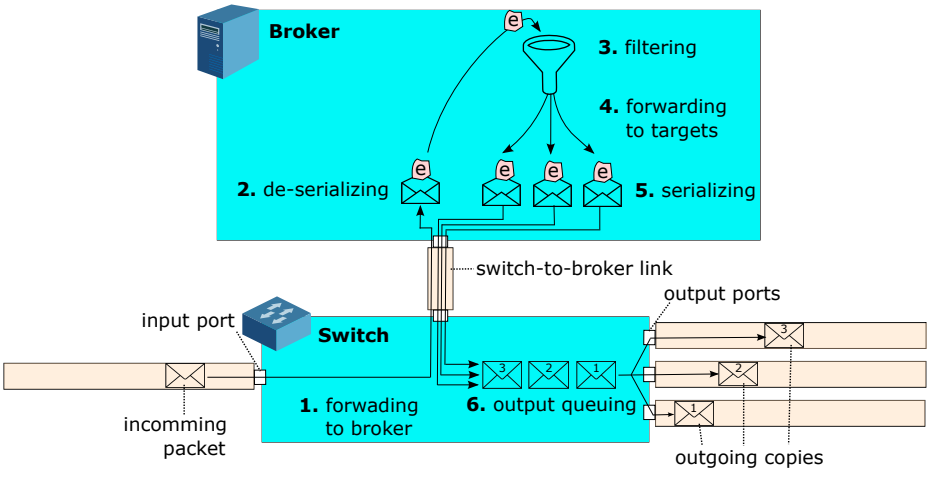


Figure 1.1: Delays introduced by a Broker.

expressed an interest in their specific content, content-based pub/sub systems minimize the dissemination of irrelevant information. This enhances user experience and system performance compared to the channel-, topic- or type-based variant. Content-based pub/sub is particularly valuable in dynamic environments where subscribers' interests may evolve rapidly, or where the availability and relevance of information varies over time.

1.1 Motivation

Implementing publish/subscribe (pub/sub) systems is relatively straightforward for channel-based or topic-based variants. However, the content-based variant poses a complex challenge because each message may be directed to a different subset of subscribers, resulting in 2^n possible subsets, where n is the number of subscribers.

Traditional content-based pub/sub systems rely on a network of *brokers* to ensure flexible message routing and delivery. These brokers register the subscriptions of the subscribers and match them against incoming notification messages. They then filter the messages according to the subscribers' criteria and determine the interested recipients. However, this broker-based approach suffers from performance drawbacks, because they perform the matching in software at the *application layer*, requiring the TCP/IP stack to be traversed when a notification is received or forwarded. Hence, software-based brokers introduce delays and increase network load.

A typical broker setup consists of a switch-broker pair, as shown in Figure 1.1. The switch connects the broker to the pub/sub network, while the broker hosts

the filtering and forwarding software. When an incoming notification message is received by the switch, it does not know initially which ports to forward the message through. The switch forwards the message to the broker via the broker-to-switch link. The broker deserializes the notification message and performs filtering operations on the message content in software. Due to dynamic process scheduling, the execution time of the software is difficult to predict. The software determines which target subscribers should receive the message. Then, since application-based multicasting is typically realised via IP unicasting, the broker copies the original message for each recipient. Consequently, each copy traverses the network stack again in the opposite direction, requiring serialization for each copy. Finally, each packet copy is sequentially transmitted back to the switch via the switch-to-broker network link, to be forwarded by the switch to the next intermediate broker or final destination. This broker-based distribution process continues until every interested subscriber has received a copy of the message from its adjacent broker. As the network scales and the number of brokers increases, this broker bottleneck becomes significant and affects overall message delivery performance.

This research addresses the performance gap by proposing an innovative approach that exploits the power of programmable *network devices* (switches). In particular, we rely on *Software-Defined Networking (SDN)* [72], a promising technology to overcome the limitations of traditional broker-based pub/sub systems. SDN separates the control plane from the data plane. This separation allows the network infrastructure to be managed from a central point, with switches programmed by rules that define how they process network packets.

OpenFlow [80] was the first dominant standard in SDN, enabling switches to process packets by defining match-action rules stored in tables. A rule consists of a filter and one or more corresponding actions that are applied if an incoming packet matches the filter. However, OpenFlow's API is bound to conventional network protocols, restricting customization of OpenFlow tables for pub/sub systems. The *P4 programming language* [32] raises the level of abstraction compared to OpenFlow. Instead of filling out tables, it offers an imperative programming language to program the network's data plane defining how switches have to process incoming packets. We use P4 in favor of OpenFlow because P4 provides the flexibility to define protocol-independent header fields that can be processed by custom forwarding logic on P4 switches.

1.2 Thesis Proposal

We propose the use of P4 to integrate content-based pub/sub functionality directly into the network infrastructure. By using P4, we can implement custom pub/sub protocols that break away from traditional network protocols. Our approach replaces conventional brokers with P4-programmable switches, shifting the processing logic from the *application layer* to the *network layer*, as illustrated in Figure 1.2.

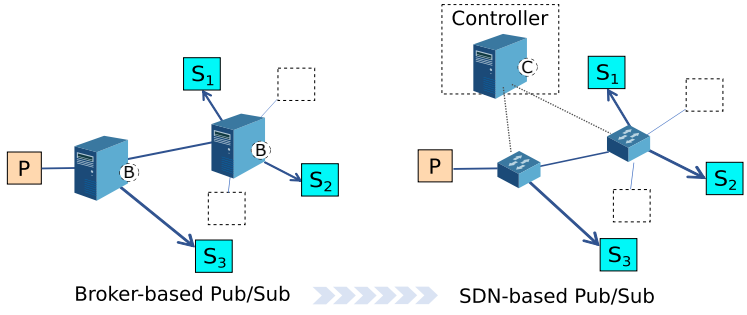


Figure 1.2: Programmable Dataplane for Publish/Subscribe.

This programmatic engineering approach overcomes the limitations of a traditional broker-based network model and aims to quickly route notifications to the interested subscribers. The basic idea is that publishers assign tags to notifications, and switches use these tags to make routing decisions. Each switch processes the tags and forwards the notification to relevant downstream switches, finally reaching interested subscribers. In the simplest case, a tag points to a preinstalled routing tree that connects the publisher to all interested subscribers. In more sophisticated cases, the publisher adds a sequence of tags to the notification. The switches extract their designated tag(s) to evaluate their specific routing information and then perform routing and tag manipulation operations.

We explore three approaches that rely on these tags to distribute notifications: (i) embedding distribution tree information in notification packets, (ii) storing precomputed distribution trees on switches, and (iii) combining forwarding rules in the infrastructure with routing information in the header to balance static and dynamic routing information. The details of this solution are described next.

(i) Compact header-based routing. In content-based pub/sub systems, successive messages from the same publisher may address entirely different sets of recipients. Therefore, an *individual* delivery tree must be created for each published notification message. This ensures that only interested subscribers are notified and avoids unnecessary message delivery to uninterested receivers with no matching subscriptions. To meet this dynamic delivery requirement, our first goal is to develop flexible encoding techniques that can efficiently represent a *complete distribution tree* within the *header* of each notification packet. By embedding all routing information in the header, publishers can encode individual distribution information for each notification. This approach is well-suited to pub/sub systems with frequent changes in published events and/or subscriptions. To avoid possible packet fragmentation, we present techniques that emphasize *compact encoding*, minimizing the size of the packet header. In this way, we preserve payload capacity for message content, thereby reducing network load and increasing overall efficiency.

(ii) Adaptive tree-based routing. Embedding a complete distribution tree in the message header is limited by the maximum packet size defined by the *Maximum Transmission Unit (MTU)*. The MTU sets an upper limit on the size of IP datagrams that can be accepted or reassembled by hosts, typically 1500 bytes [97], as the payload for a standard frame ranges from 46 to 1500 bytes, depending on the user data being transmitted. To cope with this limit while supporting large receiver sets and maintaining scalability, we introduce strategies that offload routing information as *stored forwarding trees* for stable receiver groups into the switch infrastructure, providing a bandwidth-saving alternative to embedding complete trees in message headers. We investigate techniques for extending stored trees to construct notification-specific delivery trees, maximizing their reusability and minimizing redundancy. The aim is to balance switch memory usage with header load usage, leading to optimal resource allocation and minimal bandwidth consumption. We investigate routing strategies where a combination of header-based and infrastructure-based routing can provide optimal performance. We dynamically choose the most efficient trade-off between dynamic and static routing information by considering *network topology*, *publisher-to-subscriber relationship*, and *subscriber notification frequency*.

(iii) Mixed tree- and header-based routing. Stored trees require memory to be stored as flow rules in the switches. However, this memory is limited, so we cannot store delivery trees for every possible subscriber permutation. Therefore, our goal is to reduce the amount of memory required while providing maximum flexibility in addressing arbitrary subscriber sets. To address this challenge, we explore strategies that introduce the concept of *virtual trees*. Virtual trees represent *partial delivery trees* that serve as reusable building blocks for constructing complete delivery trees on demand. We propose techniques for dynamically combining and adapting these partial delivery trees to form complete delivery trees that reach any permutation of subscribers. This will allow flexibility and scalability to evolving network conditions and changing subscriber interests. We also design efficient *tree installation schemes* that use different levels of knowledge about the pub/sub network (e.g., network topology, physical client locations, subscriber interests and notification frequencies) to optimize the placement of virtual trees. This knowledge-based approach leverages available information to preferentially place virtual trees along worthwhile network paths, which are most likely to form the core of any delivery trees.

This research exploits the programmability of P4 to replace application-layer brokers with P4-enabled switches. The proposed solutions focus on delivery efficiency by minimizing message header overhead and network traffic through effective encoding and routing strategies. Our approach adapts to dynamic network and subscriber distributions by either encoding full distribution trees in the message header or dynamically combining header-encoded information with preinstalled routing trees. The resulting distribution trees ensure fast and reliable message delivery under evolving network conditions and changing subscriber interests.

1.3 Challenges

Replacing brokers with switches in content-based pub/sub systems presents unique challenges that require careful consideration. While switches offer advantages such as programmability and potential improvements in routing efficiency, there are limitations in mapping the functionality of brokers to the data plane. The following provides an overview of these challenges.

1. *Constrained programming with P4 language*: Network devices are limited in their programmability. P4 supports only a subset of the functionality available in general-purpose programming languages. With P4, complex filtering and forwarding operations must be implemented using tables with flow rules. A table can refer to other tables, but it cannot refer back to previous tables that have already been processed. The P4 specification [32] guaranties that the flows of the tables are acyclic to ensure a high throughput of incoming packets. Unfortunately, this prohibits programming constructs that rely on loops or recursion, limiting the potential complexity of filtering and forwarding operations.

2. *Restricted packet inspection*: While SDN switches are programmable, they do not support deep packet inspection. This prevents them from examining the content of messages. Consequently, the switches can only read and manipulate information within the header of a message packet. However, it is difficult to embed pub/sub-specific routing information in the header, as its size should be kept small in favor of the payload.

3. *Additional processing overhead*: Incorporating routing information into the headers of notification packets introduces additional processing overhead at both the producer and consumer ends: Pre- and post-processing operations, such as adding routing information to outgoing packets and manipulating or removing it from incoming packets, add complexity and overhead to the system. For our multicast-based approach, however, this message header processing is necessary in order to provide each switch with the routing information it needs.

4. *Limited header load and switch memory capacity*: Efficiently encoding routing information in the message header or offloading it to the infrastructure is challenging. The expressiveness of content-based filtering contrasts with the finite header load or memory capacity of switches. Therefore, techniques for compact mapping and effective maintenance of distribution information are required to map routing information, especially in scenarios with large numbers of subscribers.

This research addresses these challenges by replacing brokers with a programmable data plane that satisfies the constraints. We deal with the programming constraints by restricting the packet processing logic to packet header inspection and

manipulation, and we address the header load and switch capacity constraints by compactly encoding routing information and effectively balancing the routing information encoded in the message header and the switch infrastructure.

1.4 Contributions

This work significantly improves pub/sub systems by proposing a P4-based approach that eliminates the broker bottleneck, enabling flexible message delivery with minimal latency. The proposed middleware implementations exploit the programmability of P4-enabled switches to manage diverse message flows and adapt to changing subscriber interests and network conditions. By shifting processing logic from the application layer to the network layer, fast processing within P4 switches is achieved, promoting responsiveness and real-time delivery.

We introduce distinct multicast-based routing strategies tailored for both stable and dynamic receiver sets, each with an efficient delivery tree encoding technique. For highly dynamic scenarios, we propose source-based routing strategies that encode the complete delivery tree of a notification message directly into the packet header. For stable receiver sets, we present strategies that store parts of multicast routing trees in the switch infrastructure and extend them with additional routing information in the notification packet header. Our advanced strategies combine multiple partial delivery trees stored in switch memories and augment them with different types of header-encoded routing information, allowing the creation of notification-specific forwarding trees.

Each strategy is evaluated through extensive simulations in emulated pub/sub networks, measuring various metrics in different topologies and pub/sub scenarios. Our findings offer valuable insights into the strengths and weaknesses of each strategy, helping to identify the optimal strategies for real-world scenarios.

The details of these contributions are provided below.

(i) Source-based routing strategies. We propose various strategies for the timely delivery of messages in content-based pub/sub systems, using custom protocols implemented as P4 programs that operate independently of existing network protocols. Our *basic strategies* encode the *complete distribution tree* of a notification packet in the packet header, allowing P4-capable switches to route the packet to interested subscribers based on header information. Publishers encode notification-specific delivery trees as sequences of header labels to address any subscriber set. By controlling the paths through which the message flows, publishers ensure precise targeting of recipients. The encoding techniques focus on compactness, optimizing the use of header space to maximize payload capacity and minimize network load. These strategies are designed for highly dynamic pub/sub environments where each message may address a completely different set of receivers.

(ii) Combining static with dynamic routing information. To address the limitations of embedding complete routing information in message headers, we propose strategies that offload routing data for stable receiver sets into the network infrastructure as *stored trees*. Publishers can reference these preinstalled trees and supplement them with additional routing information in the message header to construct a full delivery tree. We also propose *advanced strategies* based on *virtual trees* – partial, reusable delivery trees that can be combined, extended, or pruned to form complete delivery trees on demand. These strategies are based on an encoding scheme that dynamically builds a delivery tree for each notification by combining preinstalled virtual trees with additional routing information in the header. This flexible approach allows for efficient message delivery by leveraging both preinstalled and dynamically encoded routing information, offering a scalable solution for content-based pub/sub systems.

(iii) Virtual tree installation schemes. For the advanced strategies based on virtual trees, we propose several schemes for installing these trees in the switch infrastructure. These *virtual tree installation schemes* are based on different degrees of knowledge about the pub/sub network. In the simplest cases, we rely on *topological knowledge*; in more sophisticated cases, we rely on *pub/sub-specific knowledge*, such as the location and relationship of publishers/subscribers, as well as their notification frequencies. This approach minimizes the amount of information that needs to be encoded in the notification header and in the switch infrastructure. By referencing *stored rules* in the switches and combining them with *dynamic routing information* contained in the notification packet headers themselves, our strategies achieve significant reductions in header size and improved flexibility in the usability of the static rule base, thereby enhancing the overall performance of content-based pub/sub systems.

(iv) Evaluation of routing strategies and virtual tree installation schemes. We conduct extensive simulations to evaluate the performance of our basic and advanced strategies, considering different message payloads and varying numbers of subscribers per publisher. We compare the performance of our *basic strategies* against each other, against three different broker-based approaches, and against message distribution based on OpenFlow. Key factors such as *header size*, *network load*, and *delivery delay* are examined. For our *advanced strategies* based on virtual trees, we simulate different scenarios with evolving subscriptions and evaluate the trade-offs between dynamic routing information in the header and static forwarding rules in the switches. Finally, we evaluate our tree installation strategies in real-world networks provided by the *Internet Topology Zoo* [70]. We generate different distributions of publishers and subscribers and examine the effectiveness of these strategies under different migration scenarios. By analyzing the results, we gain valuable insights into the strengths and weaknesses of each strategy, allowing us to identify the best strategies for different practical scenarios.

1.5 Impact

Our contributions address the challenges of replacing brokers with switches in content-based pub/sub systems by focusing on compact encoding, fast processing, and flexible message distribution. By precisely targeting receivers and either entirely relying on stateless routing information or combining stateless and stateful routing with optimized installation schemes for stable forwarding rules, our approach improves the scalability and efficiency of notification message delivery in programmable switch environments. These advances have the potential to have a significant impact on a wide range of applications and services that rely on the timely and reliable distribution of events.

Moreover, our work bridges the gap between traditional broker-based systems and modern network infrastructures by leveraging the capabilities of software-defined networking (SDN) and P4 programming. By integrating broker processing logic directly into the data plane, we eliminate the performance bottlenecks associated with application-layer processing and reduce the latency associated with filtering and forwarding operations implemented in software. This shift not only accelerates the message distribution process but also enables the system to scale more effectively as the number of subscribers and the volume of notification messages increase.

Our approach also allows a high degree of adaptability to dynamic network conditions and subscriber patterns. By using P4 to program the behavior of switches, we can dynamically adjust routing strategies based on network conditions, subscriber changes, and the specific content of the event being transmitted. This level of responsiveness is essential in environments where timely information dissemination is important, such as in emergency response systems, online gaming, and live content distribution networks.

We propose compact distribution tree encoding techniques with our basic strategies, which minimize message header size and allow a specific delivery tree for each notification. Our advanced strategies, which combine stored rules with dynamic routing in packet headers, further reduce the amount of information to be encoded in the packet header. By storing partial routing trees and dynamically augmenting them with additional routing information, we balance packet header load and switch memory usage, supporting large and complex pub/sub networks. The various tree installation schemes for our advanced strategies, in which the encoding strategies use tree branches either fully or partially, provide a high quality of service in different pub/sub environments.

In summary, our research pushes the boundaries of content-based pub/sub systems by introducing P4-based notification services with innovative distribution strategies. By exploiting the programmability of P4-enabled switches, we address current limitations in the field, provide new insights, and pave the way for future developments in pub/sub systems in a wide range of application domains.

1.6 Structure of the Thesis

Chapter 1 introduced the concept of content-based publish/subscribe systems implemented using P4. The chapter explained the importance of this research, described the software-defined networking (SDN)-based approach adopted to address the issue, and outlined the objectives, contributions, and potential impact of this work.

Chapter 2 begins with an overview of publish/subscribe systems and explores the fundamentals of SDN. Building on this, it delves into the P4 programming language and its forwarding model, covering the technical aspects of programming the dataplane for publish/subscribe middleware. This chapter provides a comprehensive exploration of these topics to provide a solid foundation for the following chapters.

Chapter 3 introduces *source-based* routing strategies that encode the complete distribution tree within the notification header and employ various encoding schemes to compactly map the routing information. To assess their effectiveness, a comparative evaluation is performed in an emulated network, where these strategies are contrasted with conventional routing methods, including broker-based multicast and OpenFlow multicast.

Chapter 4 focuses on dissemination strategies that store routing information for stable receiver sets within network switches. Additionally, it introduces a *hybrid* strategy that combines source routing with stored trees, allowing less frequently addressed subscribers to be connected to the distribution tree through supplementary routing information in the packet header. A comprehensive evaluation compares the performance of these strategies across different subscriber churn rates.

Chapters 5 and 6 take the concepts presented in Chapter 4 a step further, focusing on an advanced routing mechanism based on *virtual trees*. These virtual trees are small multicast trees that are preinstalled and can be combined with each other, as well as with dynamic routing information in the message header. The two chapters can be read independently of Chapters 3 and 4.

Chapter 5 presents this advanced routing mechanism that allows the virtual trees to be assembled into a customized distribution trees for specific notifications. It describes multiple strategies for installing virtual trees that leverage the topological characteristics of a *datacenter network*, as well as knowledge of publisher/subscriber locations and notification frequencies. The efficiency of these strategies is comparatively evaluated within an emulated network environment.

Chapter 6 shifts the focus to virtual tree installation schemes that are adaptable to *arbitrary network topologies*. It explores different levels of network-specific knowledge for deriving virtual trees and performs extensive evaluations across real-world topologies, including dynamic scenarios with fluctuating churn rates. This analysis aims to assess the effectiveness of various installation schemes in practical and complex settings.

Finally, Chapter 7 concludes the thesis by summarizing the contributions and key findings, and highlighting the advancements made in the field of P4-based notification distribution. In addition, this chapter outlines potential avenues for future research and development, paving the way for further exploration and improvement in this area.

Chapter 2

Background

Contents

2.1 Introduction

This chapter provides the technical foundation for the following chapters, equipping readers with essential knowledge to understand and implement content-based publish/subscribe (pub/sub) systems using P4 programming. The chapter begins with an in-depth exploration of the pub/sub paradigm, followed by an analysis of the architecture and principles of *Software-Defined Networking (SDN)*. It then delves into the capabilities of P4 to implement multicast protocols. The structured exploration of pub/sub, SDN and P4 in this chapter prepares the reader for the P4-based pub/sub protocols developed in the following chapters.

The remaining sections of this chapter are organized as follows: Section 2.2 details the pub/sub model, explaining its core components and message exchange mechanics, with a particular focus on the flexibility and expressiveness of content-based pub/sub systems. Section 2.3 examines various methods for routing event-based messages from publishers (content providers) to interested subscribers (content consumers). This section analyzes the advantages and drawbacks of different routing approaches, focusing on critical factors like scalability and delivery precision. It also introduces our network-based solution to message delivery, combining network-layer speed and application-layer flexibility. Section 2.4 discusses the fundamentals of SDN, providing an overview of its architecture and introducing the *OpenFlow* protocol. It explains how multicast protocols are implemented using OpenFlow in an SDN infrastructure and highlights the limitations of OpenFlow-based notification delivery in content-based pub/sub systems. Section 2.5 addresses these limitations and explores how the P4 language can be used to define arbitrary packet processing behaviors in programmable switches. It examines how P4 enables the creation of custom protocols that overcome the constraints of traditional content-based pub/sub implementations, facilitating flexible, scalable, and low-latency message delivery. Section 2.6 concludes the chapter, summarizing the P4 components that can be leveraged to orchestrate content-based pub/sub systems within a programmable data plane.

2.2 Publish/Subscribe Model

The publish/subscribe paradigm is a powerful communication model for event-driven applications with many message producers and consumers. As shown in Figure 2.1, publish/subscribe involves two roles: *publishers* are on the producer side and *subscribers* are on the consumer side.

Publishers publish messages as *notifications*, and subscribers *subscribe* to the notifications that they are interested in. Publishers optionally *advertise* that they will potentially publish certain notifications. Each notification contains details of an *event* that has occurred in a publisher's domain. An event, in this

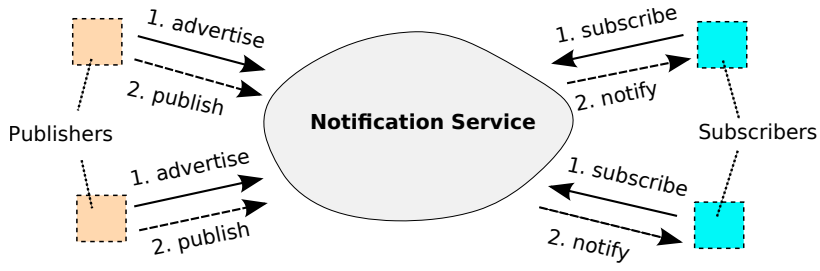


Figure 2.1: Publish/Subscribe notification service.

context, is a detectable change in the state of the system triggered by a particular "*happening of interest*" [50]. Publishers serialize the events that occur in their domain along with their contextual information into notification messages and *publish* them to a *notification service*. It is the responsibility of the notification service to distribute the notifications and to *notify* all subscribers with matching subscriptions. On receipt of a notification, the subscribers deserialize the notification message, review the details of the event and take appropriate action.

2.2.1 Notification Messages

Notification messages can contain any kind of event data. However, they are typically relatively small snippets of information that need to be published at high frequency. For example, IoT sensors at different points in the network publish their measurement results at high frequency, which are subscribed by multiple monitoring processors for processing and further evaluation at other points in the pub/sub network.

Several data models are used to transmit notification messages in pub/sub systems. These data models vary in complexity and flexibility: Simpler systems treat a message as (i) a multi-field record or as an opaque byte array [74] that encapsulates the necessary information for an event, while more sophisticated systems provide (ii) predefined message types such as text or XML messages [35, 124]. The former allow flexibility in content design, while the latter often provide built-in support for processing and interpreting the data. More complex systems use (iii) self-describing message formats or object-oriented concepts [35, 42] that allow programmers to create their own message structure based on a set of basic types or classes. The resulting messages are hierarchically structured or represent a serialized object whose type can be queried at runtime. Overall, the choice of data model depends on the requirements and the desired degree of flexibility in the representation and interpretation of message content.

2.2.2 Notification Service

A notification service is a middleware architecture designed to facilitate the efficient and reliable exchange of messages between senders (publishers) and receivers (subscribers). It acts as an intermediary, decoupling the communication process between the two parties, ensuring that notifications are delivered to the intended recipients without requiring direct interaction or knowledge of each other.

The notification service allows publishers to produce and send notifications without needing to manage the specifics of how or when these messages are delivered to subscribers. Similarly, subscribers can express their interest in specific topics, types, or categories of messages. Typically, subscribers receive only a subset of the messages published. The selection of messages for processing, known as filtering, ensures that each subscriber only receives notifications relevant to its interests.

Notification services are commonly implemented using a loosely coupled architecture. This means publishers and subscribers do not need to know each other's identities, enabling independent operation. Publishers can send notifications without waiting for acknowledgment by subscribers – a concept often referred to as "fire and forget" [40]. Similarly, subscribers can retrieve notifications without regard to the origin of the notifications or the mechanisms used for delivery. This design supports asynchronous communication and improves the scalability and flexibility of the system. In some cases, the notification service may include features such as message caching or persistence to ensure that notifications are not lost when subscribers are temporarily offline.

Eugster et al. [40] identify three key forms of decoupling facilitated by such architectures: (i) spatial decoupling, (ii) synchronization decoupling, and (iii) time decoupling:

(i) *Spatial decoupling* eliminates the need for publishers and subscribers to be aware of each other's identities or locations. Instead, the notification service acts as an intermediary, ensuring that published notifications reach all subscribers with matching active subscriptions. Publishers can produce notifications without managing subscriber details, while subscribers can receive messages from multiple independent sources.

(ii) *Synchronization decoupling* allows publishers to send notifications to the service without being constrained by blocking operations. Publishers can continue other tasks while notifications are asynchronously delivered to subscribers. This control flow decoupling eliminates reliance on point-to-point communication protocols, such as those in the TCP/IP suite associated with the request/reply pattern. Publishers are no longer required to wait for acknowledgments or responses from receivers, enabling greater concurrency and flexibility.

(iii) *Time decoupling* ensures that notifications are preserved when subscribers are temporarily unavailable. The notification service can cache notifications, allowing publishers to distribute events even when subscribers are offline. Subscribers can later retrieve notifications published during their absence. This decoupling enables asynchronous interaction without requiring simultaneous presence. However, it introduces additional storage requirements for the notification service and may need adherence to predefined timing models. However, this thesis does *not* cover time decoupling

By abstracting the complexities of indirect communication, the notification service enhances scalability, supports dynamic client participation, and facilitates interoperability between diverse components in distributed systems. It is widely used in scenarios where reliable, decoupled, and efficient message delivery is required.

2.2.3 Selection Models

There are different ways of specifying what is of interest, which has led to the development of different selection models. In this section, we provide a brief overview of subscription models that are alternatives to the *content*-based model, namely the *channel*-, *topic*-, *type*- and *concept*-based selection models. These models differ in their degree of expressiveness and implementation effort. We then take a closer look at the content-based model, which allows subscribers to define exactly what content is relevant to them.

Channel-based selection. In the channel-based model, publishers send events to one or more designated channels, while subscribers subscribe to one or more of these channels to receive all publications within their chosen category. An example of such a system is CORBA [124], which allows communication between objects across different languages and operating systems. Channels can be efficiently mapped to multicast groups, but suffer from limited selectivity due to the need for pre-established channels. Notifications that do not fit neatly into a single channel may need to be published across multiple channels, imposing additional filtering responsibilities at the consumer end.

Topic-based selection. In the topic-based model, publishers associate notifications with topics, and subscribers express interest in specific topics to receive related notifications. Some systems allow wildcards or multiple topics for more flexibility. Hierarchical topics, structured with dots as separators and resembling a URL-like tree, improve the structure of topic representation [40]. However, the hierarchical structure of topics is only suitable for dividing the notification space into one dimension, which requires predefined topics and limits expressiveness.

Type-based selection. The type-based model refines the topic-based approach for object-oriented programming languages [43, 41, 39]. Here, publishers publish notifications as instances of defined types, while subscribers specify types or subtypes of the objects they are interested in. The type hierarchy supports conformance checking based on type determination, subtype checking or instance-of-type checking. It also allows for relational conditions between types. However, the type-based model lacks support for multi-dimensional notification spaces and still has limitations in terms of expressiveness.

Concept-based selection. Concept-based selection addresses heterogeneity by allowing publishers to describe notifications at a higher level of abstraction, taking into account variations in attributes, units and data encoding [30, 29]. Here, semantic translations based on ontologies facilitate matching. However, concept-based filtering requires complex and resource-intensive message processing. Although XML-based pub/sub systems [24, 25, 107] offer semi-structured data modeling, providing benefits such as interoperability and extensibility, matching algorithms for XML languages can impose significant computational requirements, potentially impacting timely message delivery.

Overall, *channel*-based, *topic*-based and *type*-based selection systems have limitations in terms of granularity, flexibility and coverage. They rely on predefined categories and may not accurately capture the specific interests of subscribers as effectively as *content*-based selection. On the other hand, *concept*-based publish/subscribe, while capable of capturing semantic context, requires advanced processing techniques to analyze and interpret message content, making this complex approach time consuming.

This thesis focuses on content-based pub/sub, which offers high flexibility through fine-grained message filtering, while providing a manageable matching logic that allows subscribers to express their interests based on the actual content of messages. Content-based selection is described in more detail below.

2.2.4 Content-based Selection

In a content-based publish/subscribe (pub/sub) system, subscribers receive only those messages that match their specific preferences. These preferences are communicated to the notification service via filter definitions, which the service uses for precise message selection. Only notifications that satisfy a subscriber's filters are delivered to them. This model provides fine-grained control over message delivery, making it more expressive and flexible compared to simpler models like channel-, topic-, or type-based systems. However, this added flexibility often comes at the cost of increased implementation complexity.

Subscriptions

The filter definitions that subscribers use to express their preferences are referred to as *subscriptions*. Each subscription is essentially a Boolean function, denoted as F, that defines the conditions under which a message, represented as n, is considered relevant to the subscriber. If $F(n)$ evaluates to $true$, the message is considered relevant and delivered to the subscriber. Conversely, if $F(n)$ evaluates to $false$, the message is disregarded.

This Boolean function allows subscribers to specify filtering criteria across multiple dimensions (e.g. $temperature > 21$ *and location* = "*rostock*"). The notification service evaluates each published message against these subscriptions and only forwards those that meet the specified criteria. This selective delivery mechanism reduces network congestion and prevents subscribers from being flooded with irrelevant messages.

Advertisements

In addition to subscriptions, publishers can optionally use *advertisements* to announce the types of notifications they intend to publish [22]. Advertisements act as a proactive declaration, providing a preview of the attributes or content that a publisher plans to disseminate. Like subscriptions, advertisements are represented as filters. An advertisement's filter specifies the attributes or conditions that characterize the notifications a publisher may produce.

The notification service can leverage advertisements to optimize its internal routing and filtering configurations, thereby improving the efficiency of the system [86]. For example, advertisements can help the notification service to configure routing paths in advance.

Note that advertisements are an optional feature. In systems where advertisements are not supported, the default assumption is that publishers can produce arbitrary notifications without prior declarations. While this increases flexibility, it may also limit optimization opportunities for the notification service.

Advantages and Challenges

The filter-based approach of content-based pub/sub systems significantly enhances scalability by enabling dynamic, fine-grained control over message delivery. Subscribers can modify or update their subscriptions at any time, allowing the system to adapt to changing preferences or roles. This flexibility is particularly valuable in distributed systems with diverse and evolving client requirements.

However, the expressive power of content-based filters introduces challenges in implementing an efficient notification service. Since each published message

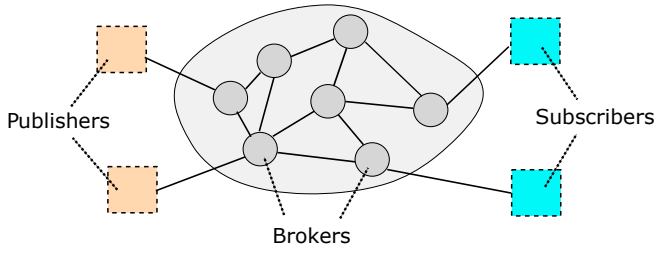


Figure 2.2: Interconnected broker network.

must be matched against potentially thousands of unique subscription filters, achieving low-latency message delivery and minimizing computational overhead becomes complex. Additionally, the highly specific targeting enabled by this model may result in situations where each notification is addressed to a distinct set of recipients, further complicating routing mechanisms.

In summary, while content-based pub/sub systems offer precision and adaptability in message delivery, they require sophisticated mechanisms to balance scalability and performance.

2.3 Realizing content-based Notification Delivery

Delivering notifications to the appropriate set of nodes with matching subscriptions is a fundamental challenge in pub/sub systems. An effective notification service must ensure that each subscriber receives all relevant notifications without missing any (*false negatives*) or being flooded with irrelevant ones (*false positives*). To address this challenge, various methods for notification delivery are employed, each with its own strengths and weaknesses. The most commonly used conventional methods are: (i) *application-layer multicast*, (ii) *broadcast*, (iii) *unicast*, and (iv) *network-based multicast*.

These methods offer different approaches to solving the distribution problem in pub/sub systems. In the following sections, we provide an overview of each method, discussing their mechanisms, advantages, and disadvantages.

2.3.1 Application-layer Multicast

A common approach to implementing a notification service in content-based pub/sub systems relies on a network of cooperating *brokers* [23] (see Figure2.2). These brokers exchange advertisements, subscriptions, and notifications to ensure that a published notification is delivered to all subscribers with a matching subscription.

Typically, brokers operate at the application layer, using an *overlay network* to forward messages. Each broker maintains knowledge only of its immediate neighbors in the overlay and routes messages based on local routing tables derived from stored subscriptions. Notifications traverse the broker network, hopping from one broker to the next until they reach all relevant subscribers.

Broker-based notification services offer flexibility in routing notifications, supporting arbitrary and complex filter expressions. To achieve this, brokers match notifications against the filters of stored subscriptions and route the matched notifications through the network to the final recipients. However, as brokers are implemented in software and perform *application-layer multicast* they introduce significant challenges that affect delivery latency and bandwidth consumption:

(a) *Increased delivery latency:* Each intermediary broker significantly increases latency due to (i) traversing the entire IP stack to serialize and deserialize the content, (ii) scheduling and executing the broker's processing software to match the notification against stored subscriptions, and (iii) forwarding incoming and outgoing packets between the switch and the broker host when the broker software runs on a separate host.

(b) *Increased bandwidth consumption:* Broker-based communication often relies on IP unicast for forwarding notifications. This means that when a broker forwards a notification to multiple neighboring brokers or subscribers, it typically sends separate copies of the notification over the same physical network link. This redundancy can significantly increase bandwidth usage.

These potential drawbacks – high delivery latency and inefficient bandwidth usage – limit the scalability and performance of application-layer multicast. Consequently, there is a need for alternative distribution methods that minimize these inefficiencies while maintaining the flexibility to serve arbitrary subscriber sets.

2.3.2 Broadcast

Broadcasting is one of the simplest distribution methods, where notifications are sent to all nodes in the network, regardless of whether they are interested in the message content. This method can be implemented using flooding, a stateless technique where each node forwards received messages to all of its neighbors except the sending node [79].

In a pub/sub system based on broadcasting, the publisher transmits a notification to all nodes in the network. Each network device forwards the message to its outgoing connections until it reaches every node in the network (see Figure 2.3(a)). Upon receiving the message, each subscriber determines whether it matches its subscription criteria. If the message is relevant, the subscriber processes it; otherwise, it discards the notification.

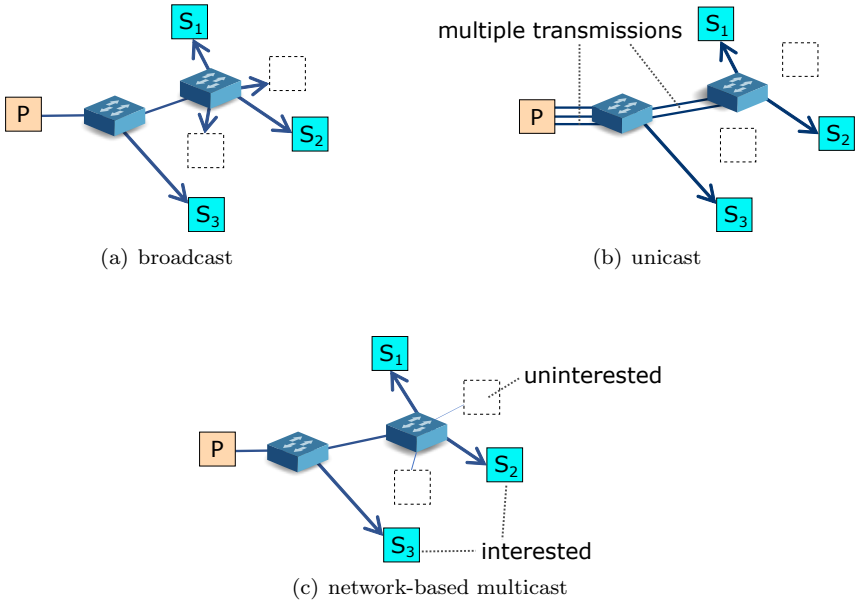


Figure 2.3: Notification distribution in network.

Broadcasting provides a simple and effective benchmark for evaluating other distribution methods, particularly in scenarios with a small number of brokers and a dense distribution of matching subscriptions.

However, broadcasting becomes inefficient in scenarios with a large number of nodes but a sparse distribution of matching subscriptions across these nodes. In such cases, broadcasting results in a significant waste of network bandwidth and processing resources, as a majority of nodes must discard irrelevant notifications (false positives).

2.3.3 Unicast

Unicast involves the direct delivery of a notification from the publisher to each interested subscriber as a separate, point-to-point transmission. Each subscriber is identified by a unique destination address, and network devices send a dedicated copy of the notification to each subscriber (see Figure 2.3(b)).

Unicast delivery is efficient when there are few subscribers or when subscribers have diverse and specific content interests. By targeting only relevant subscribers, unicast avoids unnecessary dissemination of messages to uninterested nodes, reducing bandwidth consumption in such scenarios.

However, unicast becomes inefficient in cases where a notification must be delivered to a large number of subscribers. This inefficiency arises because multiple

copies of the same notification may need to traverse the same network links, leading to redundant bandwidth usage. Additionally, unicast undermines the decoupling principle of pub/sub systems since the publisher must explicitly address each subscriber.

2.3.4 Network-assisted Multicast

Network-assisted multicast offers a middle ground by facilitating one-to-many message delivery directly within the network infrastructure. Instead of sending multiple copies of a notification, the network automatically generates copies at branching points to efficiently forward the message to group members (see Figure 2.3(c)). Unlike broadcast, multicast targets only a specific subset of nodes, defined by a multicast group address.

To enable network-assisted multicast, preinstalled distribution trees must first be preinstalled for each multicast group on the network devices. These trees allow group members to be aggregated under a single group address. However, if a notification needs to be sent to multiple groups, the publisher still needs to send multiple messages, one for each group.

Network-assisted multicast reduces bandwidth consumption compared to both broadcast and unicast in scenarios with multiple subscribers for the same notification. However, it requires additional setup, such as predefining multicast groups and managing distribution trees. Moreover, its support depends on network infrastructure, which is not always be available or compatible with the pub/sub system's requirements.

2.3.5 Combining Network-layer Multicast with Flexibility

Each of these methods – *application-layer multicast*, *broadcast*, *unicast*, and *network-assisted multicast* – has its own trade-offs in terms of efficiency, scalability and complexity. Application-layer multicast, relying on brokers, is flexible but suffers from increased latency and network load due to the indirection and software-based filtering inherent in its design. Broadcast, while simple, becomes inefficient in large networks with sparse subscriptions, creating unnecessary network load. Unicast provides precise, targeted delivery but struggles with scalability when delivering messages to a large number of recipients, as it duplicates traffic on shared links. Network-assisted multicast optimizes bandwidth consumption by leveraging native support in network infrastructure, but its reliance on preconfigured multicast groups and distribution trees adds setup complexity and reduces adaptability to dynamic subscriber needs.

To address these limitations, we propose a solution that merges the bandwidth efficiency of network-based multicast with the flexibility of application-layer multicast. Specifically, we employ *network-based multicast* to efficiently distribute

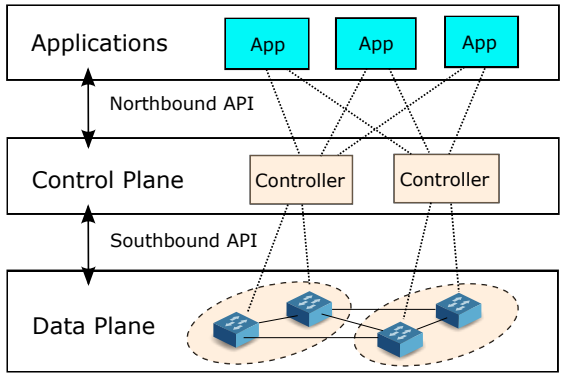


Figure 2.4: SDN architecture.

messages to multiple subscribers, while integrating *P4-based forwarding* mechanisms to dynamically accommodate changing subscriber interests.

The challenge is to shift the multicast message processing logic from the application layer to the network layer. This transition significantly reduces delivery latency while preserving the flexibility required to handle dynamic and complex subscription patterns. The *P4 programming language* plays a key role in enabling this approach by allowing programmable packet processing directly within the network infrastructure.

The following sections delve into the network architectures and technologies that are fundamental to our solution, with a focus on leveraging P4 to create a flexible, network-layer pub/sub system. By bridging the gap between network-layer multicast and application-layer multicast, our model enables delay-minimized, scalable, and flexible message delivery. This solution addresses the limitations of existing methods and lays the foundation for a new P4-based generation of high-performance pub/sub systems.

2.4 Software Defined Networking with OpenFlow

Software-Defined Networking (SDN) is a dynamic and programmable approach to network architecture that has changed the landscape of network control. In essence, SDN centralizes control of the network in software, enabling automation and programmability over diverse network devices.

The SDN architecture separates the network into two distinct planes, as shown in Figure 2.4. On one side of the architecture, the *control plane*, managed by a network controller, determines how packets are to be forwarded and orchestrates routing decisions for network switches. It does this by populating the forwarding tables on these switches with instructions on how to process incoming packets.

On the other side of the architecture is the *data plane*, also known as the forwarding plane, which is responsible for the actual transport of packets from their source to their destination(s). This is where packets are processed based on routing decisions set by the control plane, and then forwarded between different interfaces on network devices, all in accordance with the provided forwarding rules.

As depicted in the figure, the architecture of SDN consists of five primary components: (i) the application layer, (ii) the northbound interface, (iii) the control layer, (iv) the southbound interface, and (v) the data plane [69]. The uppermost layer is hosting network applications that use exposed northbound APIs to define rules and policies for network behavior. The control plane, managed by one or more controllers, receives instructions from network applications, translates them and passes them on to the data plane via southbound APIs. This architecture provides fine-grained control over forwarding nodes that provide a range of network services including routing, monitoring, load balancing and firewalls.

The controller delegates application requirements to network resources at the data plane and configures their routing behavior. *OpenFlow* is the primary southbound API that allows the controller to directly access and control the forwarding nodes. Standardized by the *Open Networking Foundation (ONF)*, OpenFlow addresses the dynamic nature of modern applications while reducing management complexity.

2.4.1 Pipeline Processing

An OpenFlow-based switch (OF switch) is an assembly consisting of one or more flow tables and control channels for communicating with the controller [80]. The flow tables are organized in a sequence to manage the packet forwarding process. Such a table contains a collection of flow entries, each comprising matching fields (the filters) and a set of flow instructions (the actions) to be executed upon a successful match. In particular, the actions can modify a packet header and can either direct the packet to an output port or forward it to another flow table for further processing.

The controller interacts with the data plane switches via the control channels and is authorized to add, update or delete flow entries as required. In addition to filters and actions, a flow entry contains statistical counters that are updated with each packet match, as well as timeouts that determine the lifespan of the entry. Packets in OF switches are processed through a sequence of these flow tables, which together form a pipeline.

When a packet arrives, the switch starts by extracting the match fields. These fields include information such as the incoming port number, source and destination MAC and IP addresses, or transport layer port numbers. The OF switch compares these fields methodically against the entries in the first table and, if a match is found, takes the corresponding action. Based on the match result, the

packet can then be forwarded to another flow table. Here, each flow entry can be assigned a priority. Consequently, if multiple flow entries match the fields, the entry with the highest priority is selected.

Depending on the action of the matching flow entry, the packet could be discarded, forwarded to a subsequent table, or sent to the controller via a packet-in message. Packet-in messages (`OFP_PACKET_IN`) are sent from the switch to the controller for further processing [46]. However, if every unknown packet triggers a packet-in message to the controller, the controller may become overloaded as it has to compute forwarding rules and insert them into the flow tables of the relevant OF switches. Such congestion can contribute to delays in the deployment of flow rules, affecting the overall performance and scalability of the network.

2.4.2 OpenFlow Multicast

The centralized view of an external controller facilitates the implementation of simpler algorithms that can replace complex distributed protocols of legacy network devices, including the development of multicast methods required for pub/sub systems. The following describes the traditional IP-based approach and then explains how it can be improved with SDN and OpenFlow.

Traditionally, IP multicast is used for multicasting. In this protocol, multicast groups are identified by specific IP addresses (e.g. in the range 224.0.0.0 to 239.255.255.255). However, the management of multicast groups requires special protocols such as the *Internet Group Management Protocol (IGMP)* [20]. These protocols are complex and can lead to scalability problems and inefficiencies in large-scale pub/sub systems. In particular, dynamic maintenance of multicast group memberships is challenging, as routers need to track group memberships and update multicast routing tables accordingly.

SDN controllers provide a solution to reduce the burden of managing multicast groups in pub/sub systems, being responsible for installing/modifying flows in switches and accessing the switches via a dedicated control network. These controllers store multicast delivery trees in switches and manage group membership. This is particularly possible with the introduction of the OpenFlow 1.3 protocol, which provides OpenFlow groups [28]. This approach allows packets to be directed to group tables, which specify additional processing and extend the range of packet operations beyond simple forwarding. By controlling the network centrally, the controller can dynamically calculate multicast routes and configure them on switches or routers. This reduces message complexity when managing multicast group memberships, as the controller can compute multicast forwarding rules based on subscriber interests and update the routing configurations of the switches accordingly using OpenFlow.

2.4.3 Constraints of OpenFlow-based Notification Delivery

The interaction between OpenFlow and IP Multicast has inherent limitations. These are related to OpenFlow's relatively low level of programmability, its reliance on standard network protocols, and the protocols' predefined and fixed header fields.

1. *Rigidity in hardware behavior:* OpenFlow operates under the assumption that forwarding devices rely on legacy high-performance chips, characterized by fixed and well-defined behavior specified in a switch ASIC's datasheet. These chips exclusively support a predetermined set of protocols directly implemented in silicon. This lack of adaptability in hardware behavior hampers the introduction of new protocols or functions for managing network traffic.

2. *Static network protocols:* Operations within OpenFlow are limited to the header fields of predefined network protocols. Although OpenFlow has evolved to include 50 different header types, such as IPv6, MPLS, and VXLAN, none of these were designed with pub/sub-specific routing in mind. The limited address space for IP multicast further restricts the scalability of pub/sub systems by limiting the possible recipient groups.

3. *Group membership management:* Dynamic content-based pub/sub systems, where recipient sets vary for each notification, face challenges in creating and maintaining effective multicast groups. Although traditional protocols like IPv4/IPv6 support multicast address creation, maintaining these groups entails significant management overhead. Frequent membership changes are either not supported or inefficient for use in pub/sub. The centralized management of multicast routes through OFM does allow for a degree of customization of the multicast forwarding logic to be implemented. However, updates to flow rules require resource-intensive coordination with the controller, causing delays in notification delivery.

4. *Limited switch memory:* Creating multicast groups for all potential subscriber permutations becomes impractical as the possible receiver combinations increase exponentially with network size. Excessive multicast routing entries can exhaust switch memory and degrade performance, posing a significant challenge to successful multicast implementation in content-based pub/sub systems.

5. *Restricted programmability:* Despite providing a degree of control over network behavior, OpenFlow focuses primarily on populating and configuring routing tables with a fixed set of operations. This limited granularity does not meet the specialized routing and nuanced content-based message distribution requirements of pub/sub systems, thereby limiting their programmability.

Overcoming these limitations necessitates innovative solutions and the development of more flexible and expressive protocols and hardware that can support the dynamic nature of content-based pub/sub systems within an SDN framework.

2.4.4 Related Work

Despite the constraints of OpenFlow mentioned above, researchers have explored various approaches [3, 57] for realizing *OpenFlow Multicast (OFM)*. The authors rely on port forwarding rules at the IP level and store multicast delivery trees in the switches.

To reduce the number of multicast groups, Cao et. al [21] and Opyrchal et. al [87] propose solutions that cluster participants with similar interests. Unfortunately, these approaches require more than a single transmission and can also lead to unintended notifications. If a subscriber receives a notification that does not match their subscription, the message is considered as a false positive, i.e. an incorrect delivery that needs to be filtered out.

In a probabilistic approach [95], publishers integrate a Bloom filter into each notification. This filter contains information about the subscriptions to which the notification relates. The OpenFlow rules installed on each switch then analyze the Bloom filter to make forwarding decisions. Due to the probabilistic nature of Bloom filters, strategies are used to limit the number of incorrect additional notifications sent and to reduce the size of the required forwarding rule base. However, due to the probabilistic nature of Bloom filters [14], false positives cannot be completely avoided.

Alternatively, Tariq et. al [109] recursively decompose the event space and map the subspaces to binary strings represented by IPv6 multicast addresses. However, due to the complexity of building and managing multicast trees and the limited address range for multicast addresses, the use of IP multicast for content-based pub/sub is limited with this and the above approaches.

2.5 Data Plane Programming with P4 Language

Programming Protocol-Independent Packet Processor (P4) [17] raises the level of programmability compared to OpenFlow. Instead of relying on fixed, non-programmable switch chips and filling out protocol-dependent tables, P4 provides a programming language to program the data plane of the network independently of traditional network protocols. Protocol-independent header fields and programmable components in P4-enabled switches allow to precisely define the behavior of network devices and how packets are processed in the P4 switches. It can describe the forwarding behavior for any type of switching device, whether it is a software switch or a high-performance chip.

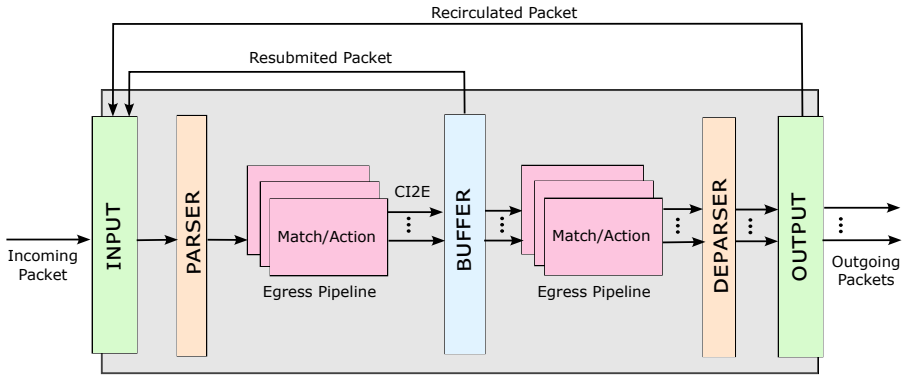


Figure 2.5: P4's abstract forwarding model.

P4 leverages reconfigurable switch chips that can process packets at speeds comparable to fixed-function switches. Built on the innovative *Protocol Independent Switch Architecture (PISA)* [18], these chips enable custom networking code execution without relying on embedded binary code. Instead of hard-coded functionality, switches interpret programs written in P4, a domain-specific language, and rely on reconfigurable match-action tables for decision-making.

P4 allows these match-action tables to be fully defined and configured. Each table consists of entries that map user-defined match fields – such as header fields or metadata – to corresponding actions. These actions modify header or metadata fields, trigger packet forwarding operations, or execute other user-defined behaviors. By implementing customizable match-action pipelines, programmers can determine the exact flow of packets through these tables, facilitating the design of individual protocols and forwarding procedures. This approach also enables decentralized control of networking functions without requiring an explicit central controller. Overall, protocols expressed as P4 programs offer greater flexibility than traditional switches with fixed-function forwarding engines.

2.5.1 Abstract Forwarding Model

In the first standardized version, $P4_{14}$, packet processing is aligned with an *abstract forwarding model* [32], as shown in Figure 2.5. This model is the basis for the explanation of packet processing in the following.

When a packet arrives at the switch, the packet header fields are deserialized by a *parser*. The parser is programmed to handle arbitrary protocol headers. It instantiates header objects to store the parsed protocol fields and additional metadata information (e.g. input port). The parser then passes the header objects to the *ingress pipeline* for inspection and/or modification. This is done using match-action tables against which the header objects are checked. If a rule

matches, the corresponding action is performed. In addition, metadata can be manipulated or created.

The packet can optionally be replicated within the ingress pipeline to send packet copies to multiple output ports. At the end of the ingress pipeline, the packet and the potential copies are passed to a *buffer subsystem*, which assigns the packets to internal queues corresponding to the output ports of the switch. In the case of unicast, the buffer assigns the packet to a specific output port; in the case of broad- or multicast the buffer assigns multiple copies of the packet to distinct internal queues corresponding to the respective output ports.

The buffer subsystem is connected to an *egress pipeline* for further processing, which operates similarly to the ingress pipeline but focuses on tasks specific to outbound packet transmission. It may involve additional inspection and modification of packet headers and or metadata based on match-action tables. However, the egress port of a packet instance cannot be changed at this stage. When the packet is replicated for forwarding to multiple egress ports, each copy passes through the egress pipeline individually. This ensures that the appropriate possible action is taken for each copy, based on the configured forwarding rules.

After completing processing in the egress pipeline, the packets are prepared for transmission by the *deparser*. The deparser serializes the header objects, incorporating any modifications made during processing, and re-inserts them into the packet. Finally, the resulting packets are sent out through the designated egress ports for transmission to their intended destinations.

2.5.2 P4 Program

A P4 program is a high-level program that implements a forwarding model. It specifies how packets are matched against predefined rules, how headers are parsed, modified, added or deleted, and how packets are forwarded through the network. The following is a step-by-step description of the components and instructions that must be written within a P4 program to define how network packets are processed.

Header fields. Programmers initially define header types, which are essentially *ordered lists of fields*. Each field is assigned a specific bit width and a name. This allows the switch to extract incoming bit fields as meaningful and typed data that a programmer can refer to. This gives the programmer granular control and the ability to manipulate header contents within the p4 program.

Metadata fields. In addition to header fields, programmers can define and reference metadata fields. These fields are divided into *user-defined* and *intrinsic* metadata. User-defined metadata can be used by the programmer to store various packet-associated data, such as processing flags or calculated values, while

intrinsic metadata represents control registers or signals defined by the device architecture, such as the input port through which a packet was received. The format of these fields is target specific. Each packet carries its own instances of intrinsic and user-defined metadata through the P4 processing pipeline.

Parser. Based on the specified header/metadata fields, programmers define a parser. The packet parser operates as a *state machine*, progressing from a start state to a stop state, sequentially extracting header fields. Functional blocks within the parser, representing states, form a parse graph delineating relationships between headers.

Match-action tables. After writing the parser, programmers define match-action tables with their *match-keys* and *actions*. A table defines restrictions on the computed metadata or parsed header fields (e.g. exact match of a value) and what action (e.g. removing a header from the packet) to take when a match is found.

Match-keys. These keys determine which header fields or metadata are employed to match packets within the table. Three types of matching are supported: *exact match*, *longest prefix match*, and *ternary match*. Exact match checks for an exact match between the input and stored value, longest prefix match finds the longest matching prefix in a set, commonly used in routing tables, and ternary match provides flexibility by matching against a range of possibilities using a value and a mask, useful for wildcard-based matching like IP addresses with subnet masks.

Rules. Match-keys and their actions are linked by rules. The rules of a match-action table define which action is to be performed for which header or metadata content. Each rule defines two parts: a match pattern corresponding to the match keys of the table and an action to be applied to the packet when the pattern matches to the header/metadata fields of the current packet instance. The rules can be populated and modified at switch runtime via an API.

Actions. When a rule's filter matches, the associated action is triggered. Actions, similar to functions in conventional programming languages, have signatures and are represented as blocks of code. They consist of primitive operations that modify packet header/metadata fields or invoke other actions.

Control flow. Finally, programmers define the order in which match-action tables are executed by implementing the control flow, analogous to the "main" function in traditional programming languages. Using conditionals (`if` statements) and by evaluating metadata or header fields, the control flow determines

how packets traverse through the match-action tables. This control flow is typically segmented into distinct pipelines based on the underlying architectural model. For example, in the abstract forwarding model (see Fig. 2.5), the control flow is divided into two primary stages: the *ingress pipeline* and the *egress pipeline*. This division allows for modular processing of packets as they enter and exit the switch.

2.5.3 P4 Limitations and Workarounds

The P4 programming language has certain limitations compared to general-purpose programming languages. As a domain-specific language designed for programming network devices, P4 is less flexible than languages such as C. It lacks built-in data structures and algorithms, relying instead on match-action tables to define forwarding logic at the data plane level within an architectural model. Furthermore, P4 does not support iterative constructs like "for" or "while" loops, which means that a packet cannot be matched multiple times in the same match-action table [33]. This design choice ensures that the computational complexity of a P4 program remains linear relative to the total size of all headers, but it also limits the implementation of complex logic.

To overcome these limitations, *external objects* and new *native primitives* can be used. External objects provide predefined APIs for vendor-specific functions, while native primitives enable specific actions, such as packet recirculation or duplication. These mechanisms allow more complex logic to be implemented.

External Objects and Native Primitives

External objects implement services that can be invoked by P4 programs via predefined APIs. They provide the ability to interact with vendor-specific elements within the architecture without altering the core syntax or semantics of P4 [104]. These elements cannot be modified by P4 and have predefined functionality. External objects can hold a state that can be read and written by the data or control plane [33]. However, the P4-based strategies proposed in this thesis do not store state across different packets, but only store packet-specific state information that is discarded when the packet leaves the switch.

In addition to external objects, P4 also supports architecture-specific primitives. This thesis focuses on the primitives supported by the *abstract forwarding model*, namely `clone_ingress_pkt_to_egress`, `resubmit`, and `recirculate`. These primitives are used to create clones of packets or to send packets repeatedly through the control flow.

Packet Resubmission and Recirculation

Although P4 does not support native loops, the abstract forwarding model allows for packet recirculation to the ingress or egress pipeline, as depicted in

Figure 2.5. The associated primitives are particularly useful in situations where the packet processing cannot be completed in a single pass, but needs to be processed multiple times within the pipeline. These include complex forwarding information extraction or iterative header stack iteration (as exploited in the P4 programs presented in the next chapter).

Recirculation is achieved by activating an intrinsic metadata field, i.e., a flag that marks the packet for recirculation [32]. In the ingress pipeline, the packet can be flagged for recirculation by calling the `resubmit` primitive, while in the egress pipeline, the packet can be flagged for recirculation by calling the `recirculate` primitive. In both cases, the packet traverses the entire pipeline and is returned to the parser at the end of the ingress or egress pipeline. When setting the resubmit/recirculate flags, it can be specified which metadata fields are to be preserved during recirculation, with all others being reset to their default values. On the next control flow pass, a particular intrinsic metadata field of the abstract forwarding model (`instance_type`) indicates in the ingress pipeline whether the current packet instance has been recirculated. Note that the P4 programs of this thesis only use the `recicrulate` primitive, not the `resubmit` primitive.

Packet Cloning

The abstract forwarding model also supports packet cloning using primitive called `clone_ingress_pkt_to_egress` (CI2E). This primitive can be invoked at any point in the ingress pipeline to activates the intrinsic CI2E metadata flag, indicating that the packet is to be cloned. The copy is created by the buffer system after the ingress pipeline has finished. Importantly, any header changes made within the ingress pipeline are not propagated to the clone.

After CI2E is called in the ingress pipeline, two packets enter the egress pipeline. The first packet is the original packet processed by the ingress control flow. The second packet is the unmodified copy from the ingress pipeline. When the CI2E flag is set, the clone can be configured to keep or clear user-defined metadata field values. When a packet clone enters the egress pipeline, an additional intrinsic metadata flag identifies it as such, enabling the original and cloned packets to be processed differently.

2.5.4 P4 Specifications

There are two specifications of the P4 language: $P4_{14}$ [32] as the first standardised version of the language, introduced in 2014, and the current specification $P4_{16}$ [33], introduced in 2016. Both versions are open and public, fostering innovation and collaboration within the networking community. $P4_{14}$ consists of more than 70 keywords. $P4_{16}$ is an evolutionary enhancement of $P4_{14}$ from which a large number of language features have been eliminated from and moved into libraries including counters, checksum units, meters, etc. With fewer than

40 keywords, $P4_{16}$ offers a smaller core language while also providing a library of basic constructs necessary for writing most P4 programs.

$P4_{16}$ is an evolution from $P4_{14}$ by introducing the concept of specific P4 architectures. Unlike $P4_{14}$, which relies on a generic abstract forwarding model, $P4_{16}$ supports a more flexible approach by supporting multiple architectures for different types of network devices, beyond just conventional switches. These architectures serve as programming models that define the capabilities and logical structure of the P4 processing pipeline for a given target device. A $P4_{16}$ program is written for a specific architecture, which acts as a blueprint for how the program interacts with the target device. This architecture specifies the programmable components, their interconnections, and target-specific features and constraints, enabling developers to exercise fine-grained control over the system. To achieve this, the architecture binds machine-level instructions to intrinsic metadata values within the processing pipeline. For instance, when initializing a packet or byte counter with a specific intrinsic labeling value, the architecture ensures that this value is mapped and stored in a control register of the target device. By leveraging such target-specific features, developers can optimize their P4 programs for improved performance and efficient resource usage.

However, the various architecture models are not interchangeable. Since the interpretation and storage of intrinsic metadata depend on the underlying architecture, it becomes difficult to deploy the same P4 program across devices that implement different architectures. To address this, vendors must provide both a P4 compiler and an associated architecture definition for their target devices. Despite this limitation, P4 programs designed for a particular architecture can be deployed on all devices that conform to the same architecture.

2.5.5 Behavioral Model Version 2 and V1Model

Through the support of abstract architectures, P4 programs can be compiled for various types of execution machines, such as general-purpose CPUs, FPGAs, System(s)-on-Chip, network processors, and ASICs. Software-based P4 targets are packet forwarding programs that run on a standard CPU. With the first public release of P4, *p4c-behavioral* [91] was introduced. It is a combined P4 compiler and P4 software target that translates a given $P4_{14}$, program into an executable C program. To address the limitations of p4c-behavioral, *behavioral model version 2 (BMv2)* [10], the second version of the P4 software switch, was introduced. Unlike p4c-behavioral, the source code of BMv2 is static and independent of P4 programs. Its open-source nature allows contributions from the community and continuous improvement. It is under active development. External functions and other extensions are frequently added by extending the C++ source code of BMv2. Although BMv2 is not intended as a production-ready software switch but rather for testing purposes, throughput rates of up to 1 Gbit/s have been reported for a P4 program with IPv4-LPM routing [96].

The behavioral model does not represent a single target, but contains code for a collection of targets, namely `simple_switch`, `simple_switch_grpc`, and `psa_switch` [10]. The `simple_switch` target was developed first and supports the *Simple Switch Thrift API* [105], specifically designed for BMv2. Thrift enables the configuration of flow rules for match-action tables in software switches. The `simple_switch_grpc` target was developed on the basis of `simple_switch` and supports the *P4Runtime API* [51], a runtime control framework for P4 devices that enables remote control of switches. It is independent of P4 programs and facilitates the (re)configuration of switches. The `psa_switch` target is based on the *Portable Switch Architecture (PSA)* [89], which builds on the success of the *Protocol Independent Switch Architecture (PISA)* embedded in the $P4_{14}$ and adds new functionality. The *P4 Architecture Working Group* [90] created and continues to develop this architecture with the aim to create portable P4 programs across various devices, facilitating collaboration among different parties to compose functionality, and establishing a common platform that vendors can collectively support to simplify data-plane programming.

However, among these different targets, the `simple_switch` target is the default choice for most users. The Thrift-based control API enables communication between the switch and external controllers. The target supports both $P4_{14}$ and $P4_{16}$, as it uses the *v1model architecture* [93] of $P4_{16}$. This architecture is consistent with the abstract forwarding model described in the $P4_{14}$ specification, and is the de facto standard for most users.

The *v1model* was designed to be largely identical to the $P4_{14}$ switch architecture, allowing for easy automatic translation of $P4_{14}$ programs into $P4_{16}$ programs that rely on the *v1model*. There are some minor differences between the *abstract switch model* of $P4_{14}$ and the *v1model* of $P4_{16}$, mainly in the names of some metadata fields.

Within this thesis, all P4 implementations of our pub/sub notification distribution strategies are written for `simple_switch` with the *abstract forwarding model* in $P4_{14}$ or with the *v1model* in $P4_{16}$. We translate the $P4_{14}$ or $P4_{16}$ programs with the *p4c compiler* [92] for the `simple_switch` target and run the compiled result with the `simple_switch` binary. This build process is described in detail below.

2.5.6 P4 Compilation

A P4 compiler translates P4 programs into target-specific configuration binaries, enabling execution on the designated target hardware. A detailed survey of P4 compilers can be found in [108]. The compilation process is illustrated in Fig. 2.6. It solves the table configuration problem by translating logical P4 constructs into hardware-specific configurations while considering hardware constraints and the programmed control flow.

However, as highlighted by Jose et al. [67], the abstract forwarding model fails to account for realistic hardware limitations, such as concurrency constraints,

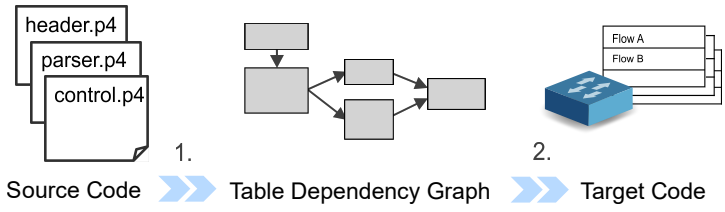


Figure 2.6: Deployment of a P4 program.

finite table space, and a limited number of processing stages. Consequently, the compiler requires two additional inputs: (i) a *table dependency graph (TDG)* derived from the P4 program, which provides insight into concurrency possibilities by illustrating the dependencies between match-action tables, and, (ii) *hardware constraints*, which define the physical limitations of the target switch, including memory capacity, processing stages, and concurrency levels.

The TDG is a *directed acyclic graph (DAG)* in which vertices represent logical tables, and edges represent dependencies between these tables (first step in Fig. 2.6). It captures the relationships between match-action tables in the P4 program, dictating which tables must execute sequentially and which can operate concurrently. These dependencies significantly influence the arrangement of tables within the pipeline, guiding the compiler in optimizing the table layout for maximum concurrency and minimal resource consumption. The optimized layout is then mapped onto the target switch (second step in Fig. 2.6), considering its specific physical constraints. Advanced optimization techniques, such as solvers for *Integer Linear Programming (ILP)*, are often employed to achieve efficient table mappings. These solvers represent switch constraints as linear constraints on integer variables and define an target function to be optimized [67].

Most P4 compilers consist of a common frontend and a target-specific backend. The frontend performs parsing, syntactic and target-independent semantic analysis, and produces an *intermediate representation (IR)* that is then consumed by the target-specific backend, which performs target-specific transformations. For the p4 programs of this thesis, we rely on the *p4c compiler* [92], which is written in C++ and uses a C++ object-based IR. This compiler has a generic frontend that supports both $P4_{14}$ and $P4_{16}$ code, along with reference backends for various P4 targets.

As mentioned above, we use *p4c* to translate a P4 program for the `simple_switch` software switch. Upon successful translation, the compiler generates two files: a file with the extension "`.p4i`", which is the result of the execution of the preprocessor in the P4 program, and a file with the extension "`.json`", which represents the IR in JSON file format expected by the BMv2 `simple_switch` target. This JSON file serves as input for the BMv2 software switch and is passed as a parameter when the switch is initiated. The BMv2 software switch then parses the JSON file, initializes the tables and configures itself, considering

possible additional parameters. The additional parameters are used to configure the ports of the target device, for example. This deployment process enables the software switch to operate on the basis of the defined table relationships and dependencies, ensuring consistent execution of the P4 program logic.

2.6 Conclusion

SDN and P4 offer an innovative solution to meet the demanding requirements of content pub/sub systems for maximum flexibility and minimum latency. By integrating a programmable P4 data plane, there is a transformative opportunity to replace software-centric broker functionality with P4 programs that use match-action tables for efficient message distribution. This paradigm shift allows distribution decisions to be made at the network layer, minimizing latency.

P4's match-action tables allow us to define conditions for message routing based on header content, enabling complex distribution strategies within the data plane. Metadata fields in P4 facilitate the processing of routing information and the caching of processed header details for packets. In addition, P4 architectural models, such as $P4_{14}$'s *abstract forwarding model* or $P4_{16}$'s *v1model*, support multi-stage packet processing and separation of responsibilities between the ingress and egress pipelines, enabling dynamic multicasting in the ingress and individual post-processing of packet copies in the egress pipeline.

Our approach leverages P4's ability to define packet header formats that encapsulate distribution information for content-based pub/sub systems, including packet header fields for encoding distribution trees. Custom header formats and parsers allow complete control over the distribution of message packets, optimizing routing operations. By designing pub/sub-specific routing protocols and control flows that process header and metadata fields, we eliminate reliance on traditional IP suite protocols, allowing fast and flexible multicast forwarding.

However, implementing broker-like behavior within a P4 program presents significant challenges. P4 programs are based on tables with match-action rules, which require translation into acyclic directed table graphs. Mapping each component of a P4 program to tables, and representing the program flow as a sequence of table lookups, requires careful modelling of the pub/sub routing logic using match-action pipelines, all within the constraints of the P4 specification.

In the following chapter, we explore how SDN and P4 can address the limitations of traditional broker-based systems. By leveraging SDN principles and P4 capabilities, we enhance the network infrastructure with the programmability needed to optimize pub/sub operations, resulting in reduced latency and improved notification delivery performance.

Chapter 3

Source-based Routing

Contents

3.1 Introduction

In a switch-based publish/subscribe (pub/sub) network, the efficient mapping of distribution information for notifications is essential to ensure that notifications reach their intended recipients while minimizing unnecessary network load. A practical method for the distribution of notifications is to store routing information in the form of distribution trees within the network infrastructure and to reference these trees during delivery via tags embedded in the notification message header. Here, the message tag specifies the tree to be used for delivery. However, this approach requires switches to maintain information for each tree, which can be time-consuming and impractical, especially in dynamic content-based pub/sub systems where subscriptions change frequently. An alternative approach is to encode the entire distribution tree of a notification in the message header. This eliminates the need to store state on network devices and avoids updating the rule base when subscriptions change. This chapter delves into the latter approach.

Within this chapter, we present SDN-based implementations for notification distribution, where all routing information for a notification is stored in the notification's message header for processing within switches rather than brokers. We show how the P4 language can be used to enable source routing. The core concept is that a publisher tags a notification and sends it to a switch, which processes the packet and forwards it to selected neighboring switches until all switches with interested subscribers have received the notification. In particular, the complete distribution tree is encoded within the notification message header using specially defined header fields that directly address all interested subscribers. A publisher encodes the entire distribution tree as a sequence of header fields within a published message and transmits the enriched message to the switch network. The network devices involved in message distribution derive their routing information for message delivery solely from these header fields. To facilitate this, we use the P4 language to extract and apply routing information and remove any processed information from the header. This flexible approach allows for the individual encoding of distribution trees for each published message. We present several strategies for distributing notifications along these distribution trees using source routing.

The rest of the chapter is structured as follows: In Section 3.2, we explain the use of variable-length header fields to implement pub/sub-specific multicast, with a focus on extracting and interpreting routing information from packet headers. These header fields are organized sequentially as *header stacks*. In Section 3.3, we discuss our source-based routing strategies that use this mechanism to encode distribution trees within the header stack. Each strategy employs different labeling types and processes these labels using individually designed match-action tables. In Section 3.5, we evaluate a representative P4 strategy against alternative approaches using criteria such as header size, network load, and worst-case delay. Finally, in Section 3.6, we discuss related research, and in Section 3.7 we summarize our conclusions for the chapter.

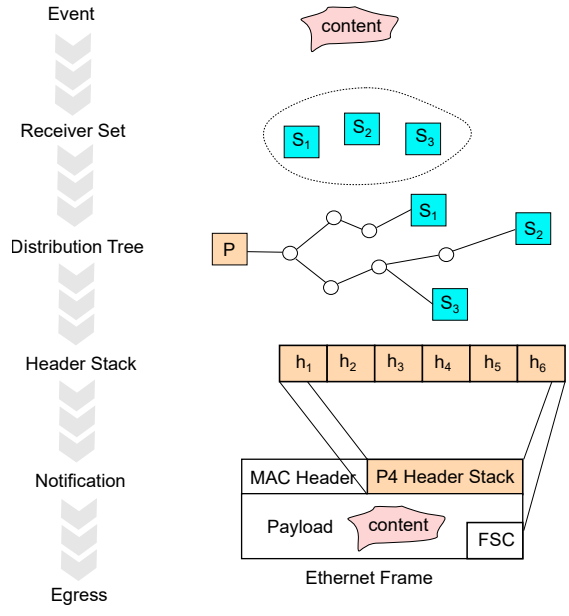


Figure 3.1: Notification encoding.

3.2 Notification Distribution with Header Stacks

Our approach resembles multicast source routing, where we encode the distribution tree of a message in a P4 header stack of variable length. P4 header stacks are capable of encapsulating a variable number of header elements of the same type. They are often used to manage multiple consecutive labels, such as *Multiprotocol Label Switching (MPLS)* [98] or VLAN [81] labels. In the P4 programming language, header stacks are represented as arrays, similar to those in C, with individual entries accessible by specifying an index [33].

In our approach, the complete distribution tree of a message is embedded in its packet header. The distribution tree is represented as a header stack and attached to the message packet by the publisher, which needs to be aware of all its subscribers and their active subscriptions for this purpose. Each entry in the header stack corresponds to a switch or switch port involved in the distribution. This design allows each receiving switch to forward the notification to the correct ports without storing forwarding rules or updating rule bases as subscriptions change. Encoding the distribution tree directly into the packet header gives us the flexibility to address any receiver set, provided that the total size of the header and payload does not exceed the maximum Ethernet frame size. We pay special attention to the compact encoding of the distribution trees, as this directly reduces the message header size. In large systems with many publishers and subscribers, a compact representation of the distribution tree reduces header overhead, saving bandwidth and enhancing the overall network performance.

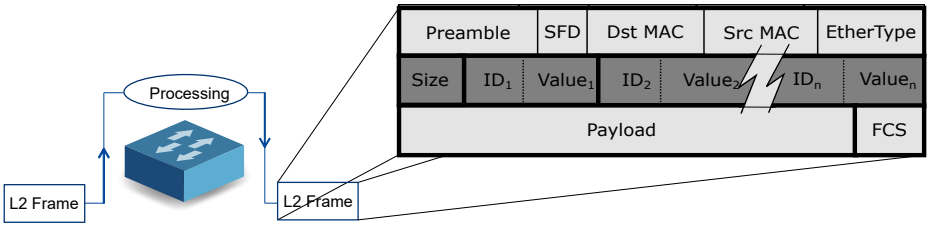


Figure 3.2: Header stack embedded in an Ethernet (L2) frame.

Figure 3.1 illustrates the process by which a publisher encodes and embeds the header stack, involving several intermediate steps triggered by an event occurrence:

1. *Receiver set identification:* The publisher identifies all recipients interested in the event based on its knowledge of active subscriptions.
2. *Distribution tree computation:* A distribution tree that covers all interested subscribers is computed.
3. *Header stack encoding:* The distribution tree is encoded into a header stack. The encoding method depends on the selected strategy for mapping the tree to forwarding behavior.
4. *Notification packet construction:* The header stack is embedded into the packet header, and the event attributes are serialized into the payload.
5. *Egress of packet for transmission:* The fully constructed packet is injected into the switch network.

As the packet traverses the network, each switch processes the header stack to determine the appropriate output ports for forwarding. The stack elements, which represent the structure of the distribution tree, may be encoded using different tag types. Depending on the encoding scheme, switches may remove already-processed stack elements to reduce packet size and conserve bandwidth.

3.2.1 Header Stack Embedding

In our strategies, the header stack is positioned between the header fields of the Ethernet frame and its payload, as shown in Figure 3.2. However, note that not all the fields within the Ethernet frame are necessary for our purposes. In particular, our system does not use the MAC header with its source and destination MAC fields. Nonetheless, these MAC fields are still essential for producers and consumers when sending or receiving packets via conventional sockets [101]. Furthermore, the Ethernet frame contains two fundamental network communication elements: (i) the *Preamble* field, which enables network devices to synchronize

their clocks, and (ii) the *Frame Check Sequence (FCS)* field, containing a cyclic redundancy check value used to identify errors in a received MAC frame [58]. Actually, the entire MAC header can be deleted once a notification packet has been injected into the P4-based network via an ingress switch. The packet is then routed within the core network via the header stack entries. Finally, the MAC header can be restored at an egress switch, just before the packet is sent to the final subscriber. However, our implementation avoids removing the MAC header.

Our routing strategies use different encoding schemes for the distribution tree and rely on a specialized mechanism to extract stack entries. Each switch in the pub/sub network uses this mechanism to extract the relevant stack entries from incoming notifications for processing and forwarding. We describe the extraction mechanism used by our source-based routing strategies below.

3.2.2 Extraction of Routing Information

When a notification with our header stack structure arrives at a switch, the switch's control flow extracts its designated stack entries to determine the routing information that needs to be applied at the switch. To achieve this, the publisher encodes a sequence of labels containing routing information for the entire delivery tree. These labels are globally visible, allowing each switch to access and interpret all entries within the header stack.

As shown in Figure 3.2, the first field, *Size*, specifies the number of subsequent stack elements. Alternatively, the P4 parser allows the end of the stack to be marked by a *terminator element*. Following the *Size* field, the header contains the stack elements, each representing routing instructions for a specific switch.

Each stack element is structured as a tuple consisting of two fields: a switch identifier (`id`) and a `value`. The `id` uniquely identifies the switch, while the `value` encodes the distribution information that specifies forwarding actions, such as which port(s) the notification should be sent to. For every switch involved in distributing the notification, there is a corresponding stack element that details the forwarding instructions for that switch. Collectively, these ordered entries form a header stack that encodes a distribution tree connecting the publisher to multiple subscribers.

To retrieve its routing information, each switch iterates through the stack elements sequentially, inspecting each entry. When the `id` field of an entry matches the switch's identifier, the associated action is executed – for example, forwarding the notification to a specified port.

In P4, these stack elements are implemented using standard `header` objects. Listing 3.1 demonstrates an example header type definition in $P4_{14}$ notation. The header type encapsulates the switch identifier (`id`) and the forwarding information (`value`) to be applied by the switch. Both fields are represented as

```
1  header_type entry_t {
2    fields {
3      id: 8;
4      value: 8;
5  }}
6
7  header entry_t stack[STACK_SIZE];
```

Listing 3.1: Stack entry type definition in $P4_{14}$ Syntax.

octet fields in this example, but the bit widths are configurable and depend on the size of the network, particularly the number of switches. This flexibility ensures adaptability to networks of different sizes.

3.2.3 Packet Recirculation

Stack iteration involves additional overhead due to particular constraints imposed by P4's guarantee of fast processing. Indeed, to allow fast forwarding for a large number of targets, P4 programs are assumed to have fixed computational complexities for table lookup operations and interactions with external objects [33]. The overall complexity of a P4 program is linear to the total size of all headers, independent of the program's packet processing state. Consequently, P4 does not support direct loops or recursion.

Despite this constraint, it is possible to iterate a header stack by repeatedly sending the packet through the switch's control flow. Figure 3.3 shows such a repeated circulation of an incoming packet by an exemplary header stack consisting of three elements. Each stack element requires one iteration of the control flow, indicated by the solid arrows. At each iteration, the top element is taken from the stack and checked. This process is repeated until the element ID matches the switch ID. If a match is found, the corresponding forwarding instruction is decoded and executed. In the example provided, this forwarding instruction generates a clone of the original packet destined for a specific output port. Each clone then leaves the switch via its designated output port, as indicated by the dashed arrows in the figure.

Once an element is processed, it is removed from the stack, or alternatively, the P4 program increments a metadata field counter to keep track of the number of processed elements. This flexible approach accommodates header stacks of varying sizes.

Instead of recirculating the packet, another approach is to employ a set of m replicated switch tables. In this setup, the match-action rules of these tables are assigned to different stack elements, allowing them to simultaneously examine the top m stack entries. However, the number of allowed clone operations is restricted to one per control flow pass, because the processing pipeline of the

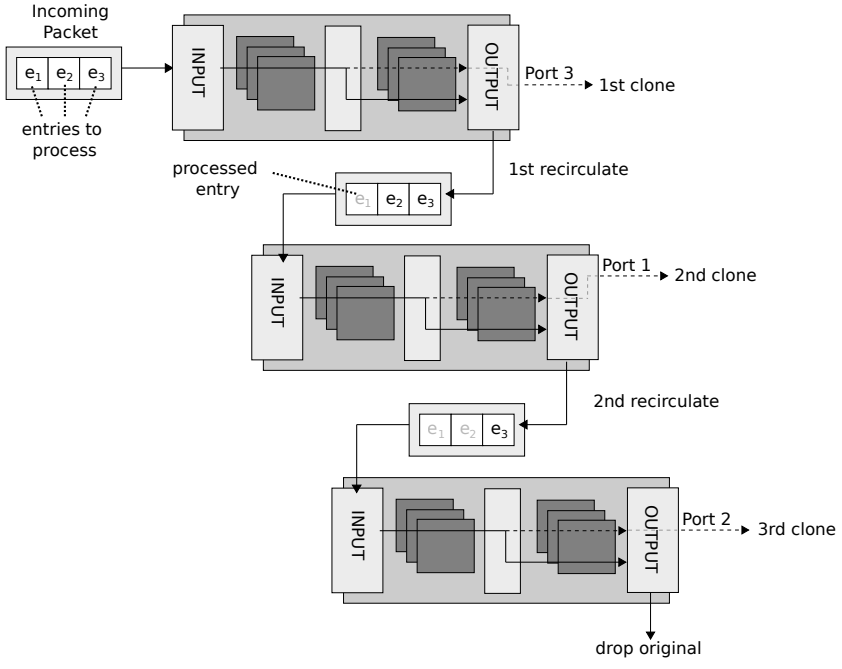


Figure 3.3: Header stack embedded in an Ethernet (L2) frame.

tables is interrupted whenever a match is found. The remaining stack elements must then be processed in subsequent control flow passes.

3.2.4 Stack Element Ordering

P4 enables manipulation of any part of the header stack, allowing information that is no longer needed to be skipped or removed. We exploit this functionality by removing processed entries or entries that reflect irrelevant subtrees of the distribution tree. This reduces the packet size progressively as it traverses the network. To enable subtree pruning and ensure no element is removed prematurely if still required by downstream switches, we order the distribution tree's stack elements according to *preorder traversal*.

For example, consider a scenario in which a message containing distribution information for two subtrees arrives at a switch, with each subtree reachable via a different port. In this case, the header stack first lists a sequence of entries for the first subtree, followed by a sequence of entries for the second subtree. The switch processes the header stack entries sequentially from top to bottom, removing each processed entry from the stack as it progresses. As the first subtree's distribution information is processed first, the generated clone for that subtree contains all the header stack elements that follow the element that caused

the clone's creation. Therefore, it contains all the elements for both subtrees. However, the second clone, generated by the first element of the second subtree, does not contain any entries from the first subtree, since these have already been removed by the time processing reaches the first entry of the second subtree. By systematically popping header entries as they are processed, the packet header shrinks in size as the packet progresses through the network.

3.3 Source-based Encoding Schemes

The header processing capabilities of P4 allow the implementation of different routing strategies. Several source-based strategies are presented below. These strategies encode the entire routing tree in the header of the notification packet. Common to these strategies is that they encode stack elements consisting of a switch identifier (`id`) and an associated routing information (`value`). However, they differ in the way they represent the routing information, i.e. the value.

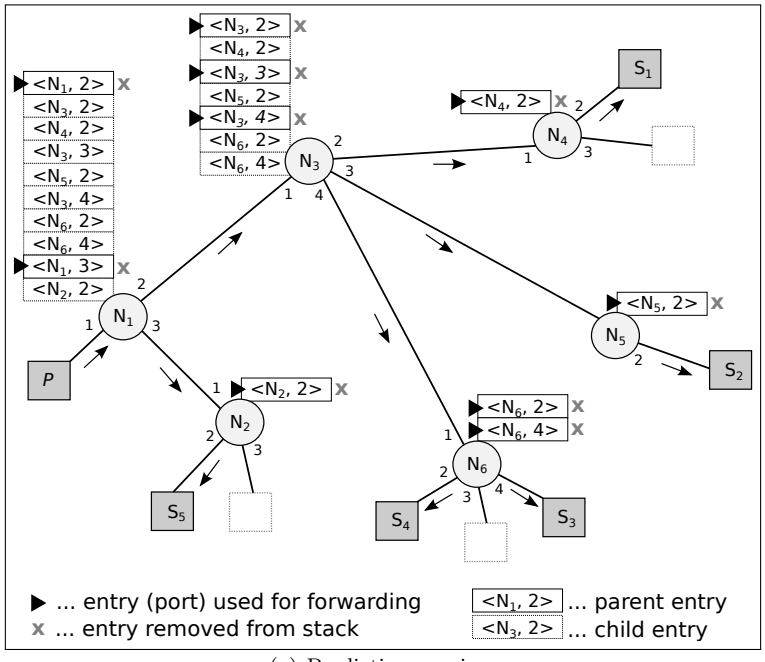
Using the header stack extraction mechanism described above, a switch extracts its own routing information from the header stack, which defines the output ports for that the switch will initiate forwarding. Once the routing information has been processed, the corresponding element is removed from the header stack as it is no longer needed by subsequent switches.

3.3.1 Switch/Port Pairs

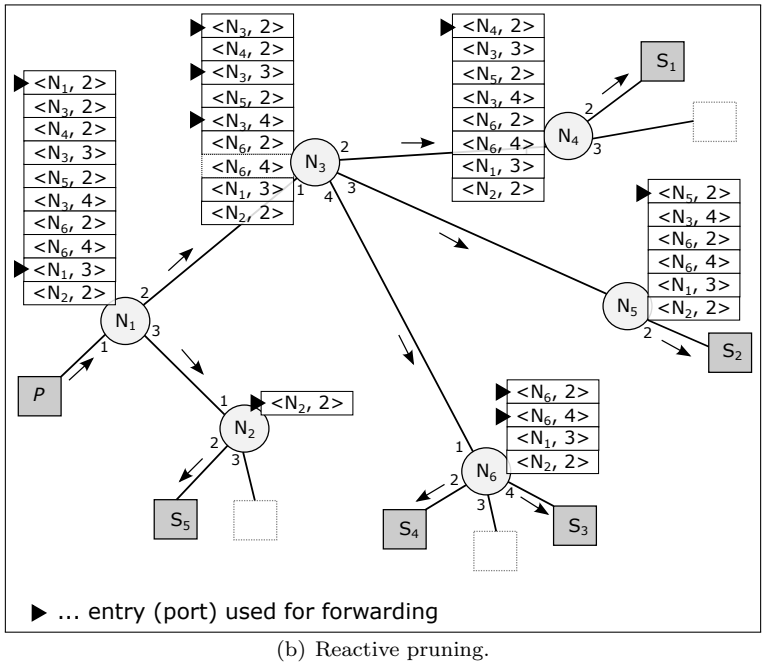
The first strategy encodes the distribution tree of a notification as a list of *switch/port* pairs, where each pair consists of a switch `id` and a corresponding port number. The port number is stored in the `value` field (cf. Fig. 3.2) and directly references the switch port to which the notification packet should be forwarded. This approach also allows for forwarding a packet to multiple destinations by specifying multiple entries in the stack for the same switch, each corresponding to a different destination port.

Figure 3.4(a) illustrates the distribution of a notification using this strategy in an example network, and shows the stack elements processed at each switch. At the initial switch N_1, the header stack contains the complete distribution information for the entire distribution tree. Since N_1 needs to forward the message to its neighbors N_2 and N_3, there are two entries for N_1 in the stack, indicated by black triangles.

The strategy implements a header pruning method that requires the stack entries to be arranged according to a preorder traversal. This enables irrelevant subtree information to be removed. For example, the second entry destined for switch N_2 comes after the complete distribution information for the first subtree (listed at N_3). Consequently, when the header stack processing reaches the $< N_1, 3 >$ entry, a clone containing distribution information for only the subtree at N_2 is created, while the information for the other subtree is removed.



(a) Predictive pruning.



(b) Reactive pruning.

Figure 3.4: Distribution using a list of switch/port pairs.

Header Stack Pruning

Stack element pruning can be done in different ways, depending on how the header stack is processed and checked for forwarding information at each switch. When dealing with switch/port pairs, we can distinguish between two methods of pruning: *reactive pruning* and *predictive pruning*.

The two pruning methods are contrasted in Figures 3.4(a) and 3.4(b), both illustrating the delivery of a notification through the same distribution tree, differing only in their pruning approach. The comparison shows that *predictive pruning*, as opposed to *reactive pruning*, results in smaller headers being forwarded to downstream switches.

Reactive pruning involves the subsequent removal of processed elements – and therefore obsolete information – from the stack by downstream switches. This method automatically removes the preceding elements up to the entry containing the identifier of the current switch, as the preceding entries refer to switches further upstream and are unnecessary for the current subtree. However, *reactive pruning* has the disadvantage of potentially passing irrelevant information to downstream switches.

Predictive pruning addresses this drawback by removing unnecessary information as early as possible. In particular, we assume that the distribution information for each subtree is organized in such a way that it always starts with a switch/port pair for the current switch. Therefore, before a duplicate leaves the switch, the switch thoroughly examines its header stack to check if it contains another matching stack element. Once the switch has identified such an element and any subsequent stack elements, it can confidently conclude that these elements belong to a different subtree. Consequently, the switch can safely remove them from the header stack.

The example shown in Figure 3.4(a) illustrates the application of *predictive pruning*. In particular, the message header stack at switch N_3 no longer contains any elements destined for switch N_2. In this method, each switch only receives labels that are relevant to its own output ports and downstream switches. In contrast, the *reactive pruning* method shown in Figure 3.4(b) does not differentiate between subtrees, but simply removes the top element(s) from the stack. Consequently, switch N_3 also receives a stack element $(< N_2, 2 >)$ destined for switch N_2. Nevertheless, switch N_2 only receives routing information for its own subtree (like in the *predictive pruning* method), because the initial top stack elements that refer to another subtree have already been removed by switch N_1.

It can be concluded that *predictive pruning* is preferable to *reactive pruning* as it reduces header load and consequently bandwidth consumption. However, predictive pruning is more complex to implement as it requires subtree detection.

$\mathbf{P4}_{14}$ Implementation

```
header_type meta_t {
  fields {
    num_seen_labels : 8;
    delete_flag : 1;
    clone_port : 8;
}}

metadata meta_t meta;
```

Listing 3.2: $P4_{14}$ metadata for header stack processing.

In the following, we provide an illustrative implementation of the *switch/port* strategy as a $P4_{14}$ program. This program employs *predictive pruning* to efficiently remove processed information from the header stack of a notification as early as possible.

To facilitate post-processing of the stack based on control flow circulations, the program defines following additional metadata fields:

1. A counter (`num_seen_labels`) to track the number of processed items.
2. A flag (`delete_flag`) to mark stack entries for deletion.
3. A field (`clone_port`) to store the egress port of each replicated packet.

Listing 3.2 presents the definition of these metadata fields in $P4_{14}$ syntax. These fields are initialized for a cloned packet and keep their state during possible multiple passes through the control flow. Listing 3.3 relies on these fields and represents the implementation of the *ingress* and *egress* control blocks of the $P4_{14}$ program. The following text focuses on the processing of both the original and cloned packet instances, with the processing of the original packet instance explained first.

When processing the *original packet instance*, the top stack element (label at position 0) is automatically selected and stored in a metadata field (line 12). It is then checked for the existence of a matching flow rule for this entry (line 15). If a match is found (line 16), a clone is created for the specified output port (line 20), otherwise the entry is ignored. In the egress pipeline, the processed stack entry is removed (line 45). However, if there are remaining stack elements (line 52), the packet is recirculated (line 53). This applies to both the original packet instance and the cloned instances.

To process *cloned packet instances*, a newly created clone enters the egress pipeline, initially preserving the clone's output port information. This step is necessary to retain the `egress_port` information when the packet is reinjected into the switch during recirculation, as each recirculation resets the intrinsic metadata representing the control registers/signals [33]. Additionally, for

```
1  /* Ingress Control Block */
2  control ingress {
3    // restore output port of a recirulated clone by metadata
4    if (meta.clone_port != 0)
5      apply(restore_egresss_port);
6
7    // check if there are unseen labels on the stack
8    if (front.num_valid > meta.num_seen_labels // extraction possible?
9        and [...] ) { // further checks
10
11     // skip processed entries and extract next entry from header
12     apply(read_entry);
13
14     // check if current label (switch/port pair) refers this switch
15     apply(check_entry) {
16       hit {
17         // distinguish between original and clone
18         if (meta.clone_port == 0)
19           // matching entry has been found in original's stack
20           apply(clone_pkt); // create clone for port given by entry
21         else
22           // another subtree has been found in clone's stack
23           apply(set_delete_flag); // delete all entries from now on
24       }
25     }
26   } else if (meta.clone_port == 0) {
27     // discard original, if there aro no more entries on stack
28     apply(drop);
29   }
30 }
31
32 /* Egress Control Block */
33 control egress {
34   // ignore packets that are marked for drop
35   if (standard_metadata.egress_port != DROP) {
36
37     // preserve ouput port info over multiple control flow passes
38     if (standard_metadata.instance_type == INGRESS_CLONE)
39       apply(store_egress_port);
40
41     // decide if current entry needs to be deleted
42     if (meta.delete_flag == 1  // marked for deletion?
43         or [...] ) { // further checks
44       // remove current entry and update number of valid entries
45       apply(rmv_stack_entry);
46       apply(dec_num_valid);
47     } else {
48       // increase counter for processed entries
49       apply(inc_seen_labels);
50     }
51     // recirculate if there are still unseen entries on the stack
52     if (front.num_valid > meta.num_seen_labels)
53       apply(recirculate_pkt);
54   }
55 }
```

Listing 3.3: $P4_{14}$ control blocks of switch/port strategy implementation.

```
/* Table for header stack label extraction */
table read_entry {
  reads {
    meta.num_seen_labels : exact;
  }

  actions {
    read_index_0;
    read_index_1;
    read_index_3;
  } size : 3;
}

/* List of actions, each related to a specific header index */
action read_index_0() { modify_field(entry, stack[0]); }
action read_index_1() { modify_field(entry, stack[1]); }
action read_index_2() { modify_field(entry, stack[2]); }
```

Listing 3.4: $P4_{14}$ example for stack header indexing.

a newly created clone, the `delete_flag` is initialized to `false`, and the top stack element is removed from the clone (line 45) that caused the clone's creation.

For subsequent stack elements, a decision has to be made whether to keep or remove them from the clone's stack. The P4 program, therefore, scans for an entry with the same switch identifier that serves as the root of another subtree. As long as no other subtree root is found, the entries belonging to the current subtree are kept and skipped. However, as soon as another subtree root pointing to a different switch port is detected, this and all subsequent entries are deleted.

Therefore, the P4 program iterates over the clone's stack, skipping the top elements that should be kept. This skipping process continues until either the first element matching the switch ID is encountered or the end of the stack is reached. During the skipping process, the seen entries are counted and the clone is repeatedly resubmitted through the control flow. In particular, counter field `num_seen_labels` is used within the ingress control block, to keep track of the skipped elements (line 12) and to examine the next unseen label on the stack (lines 15). Once a matching element is found on the clone's stack (line 16), the `delete_flag` is set (line 23). From that point on, each subsequent stack element is iteratively removed (line 45), and `num_valid` header field is updated accordingly (line 46), before the packet is finally sent. This sequential processing of the stack elements results in a *predictive* pruned header for each clone.

Specifics of Header Stack Processing

Read/write access to a stack entry works like in C-like arrays, with the difference that *"some architectures may impose the constraint that the index expression evaluates to a value known at compile time"* [33]. In practice, this constraint

```
table_add read_entry read_index_0 0 =>
table_add read_entry read_index_1 1 =>
table_add read_entry read_index_2 2 =>
```

Listing 3.5: Exemplary configuration of table *read_entry*.

means that for all possible values in the counter field `num_seen_labels` a corresponding action must be written, such that the number of actions equals the maximum stack length.

In Listing 3.4, we present a possible implementation of `read_entry` as a match-action table in $P4_{14}$ syntax. This table consists of a filter (defined within the `read` block) and several possible actions (defined within the `actions` block). Depending on the value of `num_seen_labels`, one of the stored actions is executed. Each action corresponds to a particular stack position. In essence, each stack entry requires a separate action to access it using a constant number and retrieve its contents for subsequent processing. In this simplified example, we copy the value of the current stack position into a metadata field (`entry`) and support stacks of up to three entries. Expanding to larger stacks would necessitate additional actions.

The `read_entry` table is preconfigured with the commands listed in Listing 3.5. The syntax follows that of the *Command Line Interpreter (CLI)* of BMv2 [10]. Each of the three actions in the table is associated with a filter value. These filter values correspond to single integer values placed between the action name (e.g. `read_index_0`) and the arrow symbol. Each filter value corresponds to a specific position in the stack, in this case the first, second or third position.

In scenarios where a filter consists of more than one filter attribute, these attributes are written sequentially, separated by spaces. Input parameters for the action follow the arrow symbol and are also separated by spaces. Note, however, that the actions in this example do not require any input parameters.

3.3.2 Switch/Bitmask Pairs

The second distribution strategy uses stack elements consisting of *switch/bitmask* pairs to determine packet forwarding. However, unlike the *switch/port* strategy, only one pair is encoded for each participating switch. Each of these pairs specifies a switch and an associated bitmask, where each bit corresponds to an output port of that switch and indicates whether the packet should be routed through that particular port. If the i-th bit of the bitmask is set to 1, it indicates that the notification packet should be forwarded to port i. To evaluate the bitmask, ternary filter tables are used with ampersand masking on the bits.

```
#define DROP 511
#define TRUE 1
#define FALSE 0

[...]

table check_bitmask {
  reads {
    entry.bit_mask : ternary;
  }
  actions {forward_pkt; _drop;}
  size : 1000;
}

action forward_pkt(clone_port, bit_pos) {
  // set a flag telling that the switch's entry is found
  modify_field(meta.found_flag, TRUE);

  // clone packet to port that the bit stands for
  clone_ingress_pkt_to_egress(clone_port);

  // reset processed bit to zero
  add_to_field(entry.bit_mask, -bit_pos);
}

control ingress {

  // check if current entry refers this switch
  apply(check_dev) {
    hit {

      // clone packet to an output port represented by a 1-bit
      apply(check_bitmask);
    }
  }
}

control egress {

  // drop original packet after all 1-bits are processed
  if (standard_metadata.egress_port != DROP) {

    if (standard_metadata.instance_type != INGRESS_CLONE
        and (entry.bit_mask != 0 or meta.found_flag == FALSE))
      // recirculate original packet as long as 1-bits are found
      apply(recirculate_pkt);
    else
      // remove processed header from clones
      apply(pop_header);
  }
}
```

Listing 3.6: $P4_{14}$ implementation of switch/bitmask strategy.

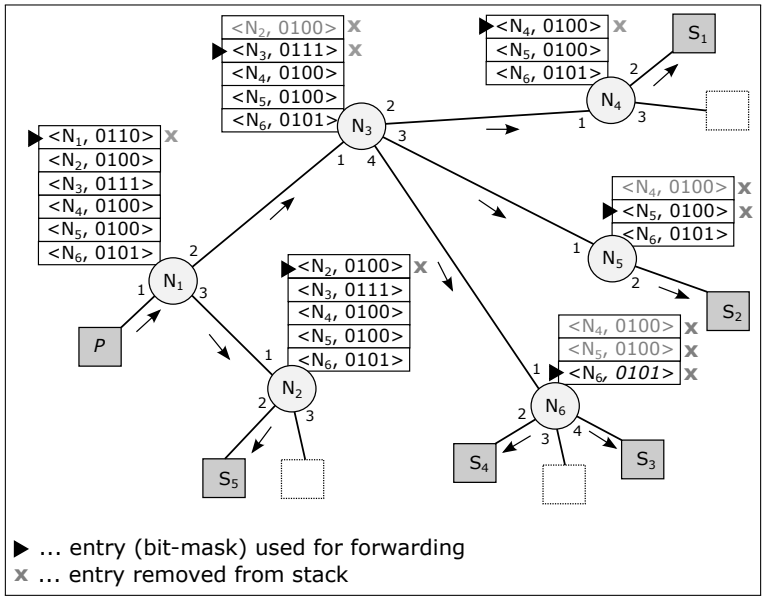


Figure 3.5: Distribution using a list of switch/bitmask pairs.

P4$_{14}$ Implementation

Listing 3.6 presents snippets of an exemplary implementation. In the ingress control block, the process starts by checking the `check_dev` table (line 29) to determine if the next element on the stack is destined for the switch. If it is, the table flow proceeds to the `check_bitmask` table (line 33). The filter and actions for this table are detailed in lines 7 to 13. The filter interprets the element as a bitmask and scans for 1-bits in this bitmask. If no 1-bit is found, it assumes that all ports have already been served for forwarding, and the original packet is discarded using the `_drop` action.

However, if a 1-bit is found, indicating that the corresponding port is to be served, the `forward_pkt` action takes over (lines 15 - 24). In this action, the original packet is duplicated for the port that the bit stands for (line 20). The action is invoked with two parameters: The first parameter specifies the port; the second parameter resets the detected 1-bit in the stack of the original packet to zero (line 23), to prevent another clone being created for that bit during subsequent bitmask processing.

After duplication, the header stacks for both the original and cloned packets are updated. This update is done via the egress pipeline. For each clone, the top entry in the clone's stack, which encodes distribution information for the current switch, is removed (line 49). However, the original packet is resubmitted through the ingress pipeline (line 46) to scan for more 1-bits in the next iteration. This process is repeated until clones have been created for all intended output ports.

If the original packet's bitmask contains no more 1-bits, the packet is discarded.

During subsequent message delivery, the remaining stack elements within the headers of the cloned packets are processed by subsequent switches. However, these clones may contain irrelevant information from other subtrees because this strategy does not support predictive pruning through post-processing of header stacks. Consequently, each switch involved must locate its designated switch/bitmask pair by popping the top stack elements until it identifies one that matches its own ID.

Example

Figure 3.5 shows an example distribution tree encoded as a stack of switch/bitmask pairs. In the example, publisher P distributes a notification to subscribers S_1, S_2, S_3, S_4 and S_5 by sending the notification to switch N_1. The header stack of the notification contains one entry for each switch involved in the delivery, where the bitmask of each entry is 4 bits. The length of 4 bits means that each switch can have a maximum of 4 ports and can therefore forward a maximum of 3 copies of a received packet. Switch N_1 receives the packet through port 1 and forwards it through ports 2 and 3 because the second and third bits in the bitmask are set. The first bit in the bitmask that corresponds to the input port of N_1 is set to 0 in order to avoid a loopback situation.

During header stack processing, the preceding stack elements are removed, while subsequent labels for different distribution subtrees remain in the header stack. For example, when considering switch N_5, distribution information for N_1, N_2 and N_3 is already removed by upstream switches. However, all switches in the example receive distribution information for N_4, N_5 and N_6, which are at the bottom of the stack and thus remain longest in the notification message header.

Unlike the *switch/port* strategy, the *switch/bitmask* strategy requires only one element per switch in the stack, regardless of the number of branches on a switch. This saves header space compared to the individual entry encoding required for each port in the *switch/port* strategy. However, the bitmask approach limits header stack pruning because subtree identification, as in the *switch/port* strategy, is not possible. As a disadvantage, irrelevant information may be passed to subsequent switches.

3.3.3 Switch/Multicast-Group Pairs

In both the *switch/port* and *switch/bitmask* strategies, the number of circulations on a switch is equal to the number of switch ports to be used for forwarding the respective notification packet. In particular, when analyzing bitmasks in the *switch/bitmask* strategy, $m - 1$ circulations are required when m bits are set to 1. To mitigate the circulation overhead, another distribution strategy is possible that uses *local multicast groups* to interpret multiple bits at once. This approach

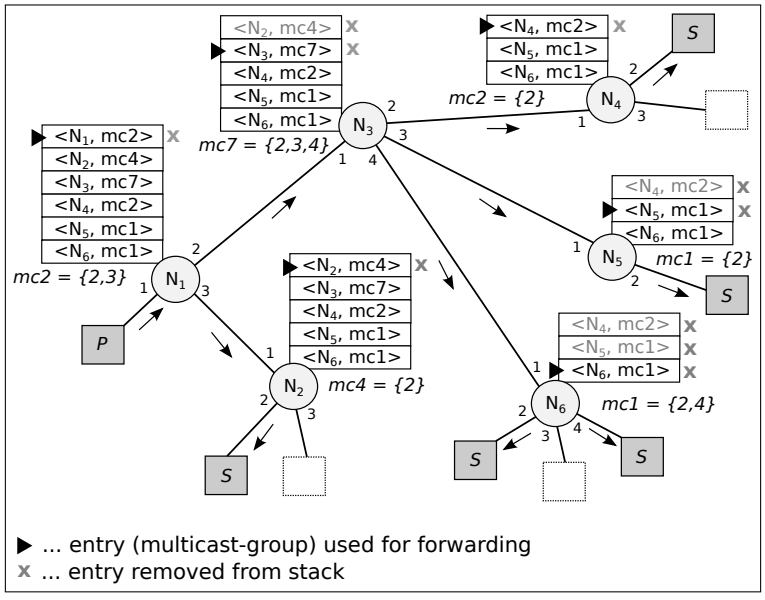


Figure 3.6: Dissemination through listing multicast groups

allows the creation of all required clones in a single action by interpreting a bitmask as the identifier of a preinstalled multicast group.

A *multicast group (mc-group)* refers to a single switch and combines a port permutation of that switch under a single label. These predefined groups can be created in advance for any possible port permutation of a switch using the runtime *Command Line Interpreter (CLI)* of BMv2 [10]. A P4 program can apply such a multicast group within the ingress pipeline by setting a specific metadata field (`standard_metadata.mcast_grp`). In response, the buffer subsystem (traffic manager) creates the necessary packet copies. Finally, the header stack of each replicate is updated in the egress pipeline to remove the currently processed element. In the egress pipeline, each cloned packet keeps identical header information. The replicates differ only in their output ports.

Since each switch maintains its own multicast groups, the header of a message must contain a multicast label for each switch involved. Each stack entry corresponds to an ordered pair of a *switch identifier* and a *multicast group identifier*. When a notification with such a stack reaches a switch, the switch extracts its entry and applies the multicast group contained therein. Figure 3.6 illustrates a switch network based on this header stack structure. For instance, switch N_3 has multicast group *mc*7, which includes ports 2, 3 and 4. Consequently, switch N_3 initiates forwarding on these three ports when the label is applied.

As with the *switch/bitmask* strategy, the *switch/mc-group* strategy does not support post-processing of packet clones to remove irrelevant subtrees from the

```
1  table multicast_forward {
2    reads { entry.device : exact; }
3    actions { do_multicast; _nop; }
4    size : 1;
5  }
6
7  action do_multicast() {
8    modify_field(intrinsic_metadata.mcast_grp, entry.mcast_grp);
9  }
10
11 control ingress {
12   // check if there is at least one more entry on the stack
13   if (front.num_valid > 0)
14     // check device ID and apply multicast group if ID is found
15     apply(multicast_forward);
16   else
17     apply(drop); // drop original if all entries are processed
18 }
19
20 control egress {
21   if (standard_metadata.egress_port != DROP) {
22     apply(pop_header); // remove processed header
23
24     // recirculate original if no match occured yet
25     if (standard_metadata.instance_type != REPLICATION
26         and intrinsic_metadata.mcast_grp == 0)
27       apply(recirculate_pkt);
28   }
29 }
```

Listing 3.7: $P4_{14}$ implementation of switch/mc-group strategy.

header stack. Instead, once the multicast group is applied, the switch removes its entry from the stack of each clone generated in the egress pipeline. Downstream switches are then responsible for locating and extracting their routing information from the remaining stack.

Note that in a particular implementation of this strategy, the multicast groups for each switch can encompass the same port permutations, with each multicast group ID representing a fixed port combination. This simplifies the process of determining the appropriate multicast group for each switch. Although it can streamline the implementation, a fixed mapping is not mandatory.

$P4_{14}$ Implementation

Figure 3.7 shows snippets of our implementation in $P4_{14}$, including the ingress and egress control blocks and the match-action table that calls the multicast groups. In the ingress control block, the first step is to check that there is at least one element on the header stack (line 13). If the stack is empty, the packet is dropped (line 17). Otherwise, the `multicast_forward` table is applied

(line 15), which does an `exact` match of the device ID of the top element against the switch ID (line 2).

On match, action `do_multicast` is invoked (lines 7 to 9), extracting and applying a multicast group from the entry. This will cause the packet to be forwarded through multiple ports, as defined by the multicast group.

On miss, the switch continues to scan the stack by recirculating the original packet (line 27) until a matching entry is found. Each local multicast group corresponds to a specific port permutation that is preconfigured via the CLI. When a multicast group is applied, all the necessary clones for the ports within the group are created simultaneously. Once the clones have been created, each clone instance passes through the egress pipeline, where the processed entry of the current switch is finally removed from the header (line 22).

Reducing Local Multicast Groups

The number of storable multicast groups is limited on a switch. But a switch with n ports requires $2^n - 1$ local multicast groups to cover all possible port permutations. This requires a reasonable selection of multicast groups to be installed. As one option, only actually occurring port combinations could be mapped as multicast groups. However, possible receiver combinations, including their location in the network, must be well-known for determining effective multicast groups, and if combinations change, an update of the rule base is necessary. As another option, the ports of a switch could be divided into non-overlapping subsets and for each subset the possible port permutations could be mapped as groups. In this case, n ports are divided into c subsets, with each subset requiring $2^{(n/c)} - 1$ groups. This gives a total of $c \times (2^{(n/c)} - 1)$ multicast groups. For example, instead of $2^{16} - 1 = 65,535$ groups, a switch with 16 ports requires only $2 * (2^8 - 1) = 510$ groups if two subsets of 8 bits are used. On the downside, splitting the ports into disjoint subsets causes increased circulation requirements, since each subset must be processed separately. This results in $c - 1$ additional circulations per involved switch.

An alternative way of reducing the number of multicast groups is to install multicast groups only for port combinations that are frequently used for delivery and would require a significant amount of packet circulation. For all other port permutations, stack labels from the *switch/port* or *switch/bitmask* strategies described above could be used. This results in a *mixed* strategy, which combines different strategies and different label types.

3.3.4 Mixed Entries

The *switch/{port, bitmask, mc-group}* strategies described above can be used in combination for message delivery by mapping elements from multiple strategies into a single header stack. Such merging is facilitated by pre-installing different

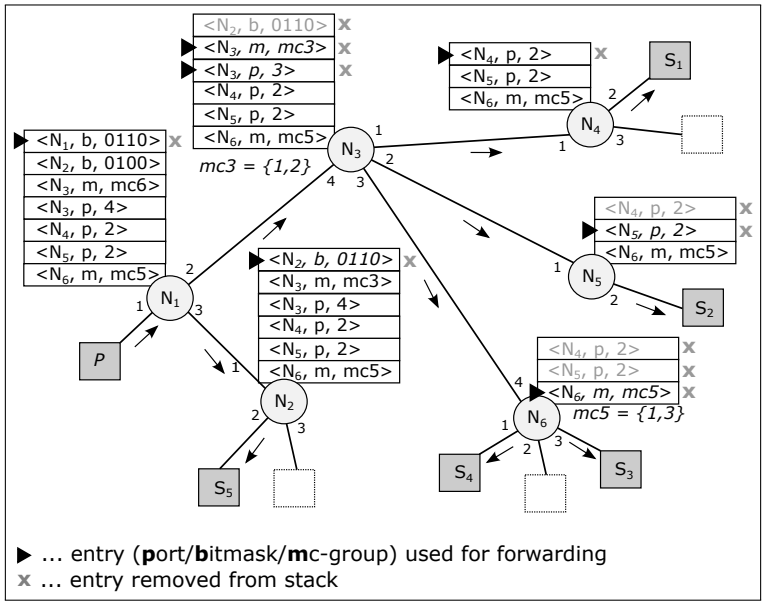


Figure 3.7: Distribution using a list of switch/{port, bitmask, mc-group} pairs.

routing rules in the switches according to the intended strategies. In a simple setup, each switch supports a specific single strategy, while in more complex setups, multiple routing strategies are supported per switch. It is the responsibility of the network designer to select the supported strategies for each switch according to the capabilities of the switch. For example, if switches have different memory capacities, high-capacity ones might use preinstalled multicast groups with the *switch/mc-group* strategy, while low-capacity ones might rely on the *switch/bitmask* or *switch/port* strategy.

However, dealing with mixed header stacks reflecting output ports, bitmasks and multicast groups is complicated. Firstly, an adequate routing strategy has to be selected for each switch, and the corresponding element in the header stack has to be encoded. Secondly, it is important that each element in the stack occupies the same number of bytes to ensure a consistent extraction mechanism, including skipping elements on the switches.

Each switch must be able to interpret the routing information determined for it. This can be achieved either by pre-assigning a fixed strategy to each switch, or by additionally specifying for each stack entry which strategy to use for that entry. In the first approach, a switch is programmed for one strategy only, so that it is not necessary to distinguish between different strategies during label processing. In the second approach, all switches support all forwarding strategies, so that a combination of strategies is possible not only between different switch instances, but also within the same switch instance. To ensure correct execution on the switch, a *strategy-tag* is attached to each stack element, indicating the forwarding

strategy the switch must apply for the element.

Figure 3.7 shows the distribution of a sample notification using a mixed header stack. Publisher P encodes header stack entries as tuples. Each tuple contains a switch ID, a strategy-tag and routing information. The message is sent with the initial header stack to neighboring switch N_1, which extracts the routing information, interprets the strategy-tag and forwards the packet to switches N_2 and N_3 using the bitmask strategy via ports 2 and 3 respectively. The processed elements are then removed from the stack. Subsequent switches process the packet sequentially using their own strategies. For example, N_3 uses two strategies for two consecutive entries. The first entry triggers multicast group forwarding, while the second uses the output port strategy, resulting in packet forwarding through ports 1, 2 and 3. The entries are removed from the stack as they are processed. This continuously reduces the packet header size. Note that a switch has to examine each entry of the header stack in order to apply multiple elements. Therefore, it must circulate the packet for each entry as in the *switch/port* strategy.

3.4 Implementation Remarks

This section discusses potential adaptations to our P4 distribution strategies and examines the differences in header stack operations between P4_{14} and P4_{16}. Firstly, we examine potential adaptations for seamless interaction with non-P4-capable switches and integration with IP suite protocols. Then, we delve into the basic header manipulation operations of the two different P4 specifications and highlight the differences in their implementation.

3.4.1 Customizing P4 Strategies for Implementation

The implementation of P4-based strategies can be customized to the unique characteristics of the pub/sub network. For instance, in a heterogeneous environment with both P4-compatible and non-P4-compatible switches, tunneling mechanisms must be employed to accommodate legacy infrastructure. Similarly, when distributing notifications to a large number of recipients, where the distribution information exceeds the capacity of a standard Ethernet frame header, additional measures such as message splitting or frame size adjustments are necessary. Below, we outline key challenges and propose potential adaptations to address them.

Heterogeneous Networks with Mixed Switch Capabilities

In pub/sub networks that include both P4-capable and legacy (non-P4) switches, tunneling mechanisms can facilitate smooth integration and ensure end-to-end

functionality. To archive this, the P4 header stack and the trailing notification message can be encapsulated in an IP header for forwarding through conventional switches. This encapsulation preserves the P4 header stack and its forwarding logic while allowing the packet to traverse non-P4 switches. Upon re-entering the P4-capable portion of the network, the encapsulated packet can be decapsulated to restore the original P4 header stack.

Before final delivery to the recipients, *header cleanup* is an essential step to remove any residual P4 header stack fields in order to avoid compatibility issues. This may include populating standard protocol headers – such as IP addresses and port numbers – with valid values to ensure that the packet reaches the intended host or application. This cleanup is best performed at the "egress" switch closest to a receiver, ensuring compatibility while maintaining proper delivery semantics.

Splitting Large Distribution Trees

In cases where a notification needs to be delivered to a large number of recipients, the total distribution information may exceed the capacity of a standard Ethernet frame header. To address this, the following solutions can be employed:

1. *Splitting the delivery tree into subtrees*: The distribution tree can be divided into smaller subtrees, each corresponding to a specific network region. For each subtree the publisher generates a distinct messages, resulting in multiple packets per notification. This segmentation shortens the headers of individual packets and ensures that the size of the distribution information remains within the limits of standard Ethernet frames.

2. *Leveraging jumbo frames*: In environments where supported, the use of *jumbo frames* – Ethernet frames larger than the standard *maximum transmission unit (MTU)* – can allow for the inclusion of more extensive distribution information within a single packet. These Ethernet frames support an MTU of up to 9,000 bytes, compared to the standard 1,500 bytes [71]. However, while the P4 specification itself imposes no frame or header size limits, this approach requires all switches along the path to support jumbo frames.

Splitting delivery trees or leveraging jumbo frames both come with drawbacks for large recipient groups in large-scale networks. Chapters 4, 5 and 6 address the issue of too-large headers by offloading routing information into the switch infrastructure and varying between routing information in the header and state in the switch memories.

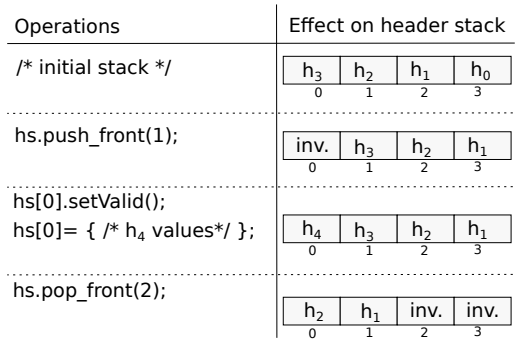


Figure 3.8: Exemplary header stack operations.

3.4.2 Differences between P4$_{14}$ and P4$_{16}$

Both versions of P4 provide operations to add and remove stack elements to and from a packet header. These operations manipulate the elements either at the top of the header stack or at arbitrary stack positions. However, there are minor differences in the specification of these operators between P4$_{14}$ and P4$_{16}$.

Manipulation of Elements at Top of the Stack

According to the P4$_{14}$ specification [32], adding and removing elements from the top of the stack is done using `push(int count)` and `pop(int count)`, respectively. In contrast, the P4$_{16}$ specification [33] defines the `push_front(int count)` and `pop_front(int count)` primitives to add and remove stack entries.

The `push` or `push_front` primitive generates a specified `count` of new elements at the top of the stack and shifts existing elements to the right by that count, i.e., towards the end of the stack. It moves an element from index n to index $n+count$ and creates the given number of new elements from index 0 to $count-1$. This primitive is used to add new headers to the front of a stack. Importantly, it keeps the size of the array constant. If elements are moved beyond the static array size, they will be lost.

Figure 3.8 shows an example for P4$_{16}$ with a header stack `hs` consisting of four elements. After invoking `hs.push_front(1)`, each element in the header stack `hs` is shifted one position to the right. However, because the stack size is limited to four elements in the example, the last header (h_0) is pushed beyond the maximum stack position and thus becomes lost. Furthermore, the new element at index 0 is flagged as invalid. For this purpose, each header instance representing a stack entry has an additional validity flag to indicate whether or not it is included in the packet header. In P4$_{16}$, the status of this flag can be assessed using the `isValid()` primitive, and can be set explicitly using `setValid()` or `setInvalid()`. In the example, the h_4 header is assigned to position 0 of the

header stack, but the element's flag remains $false$. In $P4_{16}$, it is necessary to explicitly validate the newly added element by invoking `setValid()` [33]. In contrast, the `push` operation in $P4_{14}$ automatically marks newly initialized elements as valid, eliminating the need to call a primitive.

The `pop` or `pop_front` operation is used to remove the first `count` header elements. It shifts the elements in the header stack to the left by the specified count. This invalidates the last `count` elements, while moving the first `count` elements to fill the empty positions. The primitive is used to delete headers at the top of the stack. In the example, the header stack `hs` initially contains four valid elements after h_4 is added. However, once `hs.pop_front(2)` is called (`count` = 2), `hs` contains two remaining elements (h_2 and h_1), while the first two are invalidated. Note that the `count` parameter must be known at compile time and defined as a constant in $P4_{14}$ and $P4_{16}$ for the push and pop primitives.

Manipulation of Elements at arbitrary Stack Position

P4 offers the flexibility to delete elements not only at the top of the stack, but also at arbitrary position within the stack. This allows the creation of "gaps" at any stack index. In $P4_{14}$, this is achieved using the `remove_header(header instance)` primitive, marking the specified header instance as invalid. In $P4_{16}$, the `setInvalid()` primitive is the equivalent for creating header stack gaps. This functionality is particularly useful for removing processed routing information from the packet header and deleting irrelevant data for subsequent switches.

3.5 Evaluation

We implemented the *switch/{port, bitmask, mc-group}* strategies and conducted a series of experiments to evaluate their performance. The strategies are compared against each other and benchmarked against alternative approaches. In the following, we first describe the experimental setup in detail. Next, we present a comparative analysis of the three P4 strategies. Finally, we benchmark the most promising P4 strategy against competing methods, including OpenFlow multicast, three variants of application-layer multicast, as well as the *unicast* and *broadcast* approaches.

3.5.1 Jelly-fish Topology in Datacenters

The experiments are conducted in an emulated *Jelly-fish* network topology [106], which is characterized by its degree-bounded random graph design. A simplified example topology is shown in Figure 3.9. Unlike traditional datacenter topologies, which rely on network devices with regular port counts, Jellyfish naturally accommodates heterogeneity, allowing for the incremental addition of nodes and network capacity.

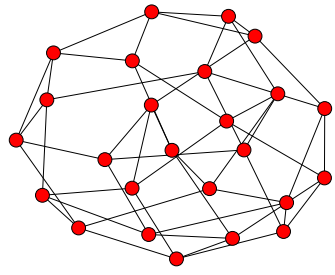


Figure 3.9: Exemplary Jelly-fish topology.

Jelly-fish networks, whether created from scratch or expanded over time, consistently maintain short average transmission path lengths. This property results from the inherent characteristics of r-regular graphs, where the diameter increases logarithmically (to base r) with network size [16].

Compared to fat-tree networks [44], a regular network topology design, Jellyfish offers greater scalability without being limited to specific discrete increments in size. It also supports more participants using the same network equipment [106]. Similar to other datacenter topologies, Jellyfish provides multiple redundant paths between nodes. This improves fault tolerance and, more importantly, gives us multiple ways to propagate notification messages through header-encoded delivery trees. Specifically, within our simulations we can control the node degree and therefore the connectivity, i.e. the amount of redundant links, between nodes.

3.5.2 Setup of Publish/Subscribe Network

In addition to the Jellyfish topology, our simulation setup uses three key tools: *NetworkX* [52], a Python library designed to handle complex network structures; *Mininet* [73] in version 2.1.0, a network emulator designed for SDN prototyping; and the `simple_switch` P4 software switch, a variation of the *Behavioral Model version 2 (BMv2)* [10]. Figure 3.9 illustrates a simplified version of the Jellyfish topology used in our simulations. All experiments rely on a consistent setup with 255 switches, each connected to a single host (peer) capable of acting as a publisher, subscriber, or both. While the distribution strategies vary, the underlying topology and simulation environment remain unchanged.

The simulation of content-based pub/sub scenarios poses significant challenges, as the set of interested subscribers depends on the content of the notification. To address this, we developed a parameterizable simulation model based on uniform distributions for both publishers and subscribers. A key parameter allows us to dynamically vary the percentage of subscribers matching a given

notification (from 10% to 90%) across simulation runs. In each run, a *traffic generator* randomly selects a publisher and determines the appropriate number of matching subscribers, ensuring that each notification reaches a randomly and uniformly distributed subset of recipients.

While uniform distributions may not reflect typical real-world pub/sub traffic patterns, they provide a controlled environment for comparing the relative performance of different distribution strategies. Chapter 6 expands on this analysis by incorporating more realistic distributions (such as Zipf) to capture more complex, real-world behaviors.

To ensure comparability, the traffic generator is initialized with a fixed seed, enabling identical traffic patterns across all evaluated strategies. For each data point, we conducted 100 simulation runs and averaged the results, ensuring statistical robustness and consistency in our findings.

3.5.3 Performance Comparison of P4 Distribution Strategies

We evaluate our source-based strategies in a comparative manner using three metrics: (i) the initial *publisher packet header size*, (ii) the total number of *transmitted bytes*, and (iii) the total number of performed *table lookups*. For the evaluation, we analyzed the log files of the virtual switches. Specifically, we counted the number of match-action rules executed on the switches. We also monitored the virtual network adapters of the switch infrastructure to record the number of bytes and packets transferred.

Initial Header Size

Figure 3.10 shows the initial size of the generated header stacks as the number of subscribers varies for the *switch/*{*port, bitmask, mc-group*} strategies. While the *switch/port* results are shown separately, the *switch/bitmask* and *switch/mc-group* results are shown as one line because both strategies generate headers of the same length. In these experiments, a size of 3 bytes is specified for each entry: 1 byte for the switch ID and 2 bytes for the bitmask or multicast group. This supports up to $2^8 = 256$ switches, each with a maximum of 16 ports. Note that for larger networks or switches with more ports, longer header stack fields must be defined in the P4 program.

The *switch/port* strategy encodes only 2 bytes per entry: 1 byte for the switch ID and 1 byte for the port number. Although the entries are 1 byte shorter with this strategy, it requires a designated entry for each output port used. In contrast, the *switch/*{*bitmask, mc-group*} strategies generate only one entry per switch involved, regardless of the number of output ports used. The results show that for a small number of subscribers, the overhead is nearly the same for all three strategies. In fact, the *switch/*{*bitmask, mc-group*} strategies generate smaller headers than the *switch/port* strategy only when more than 30% of the hosts act as subscribers.

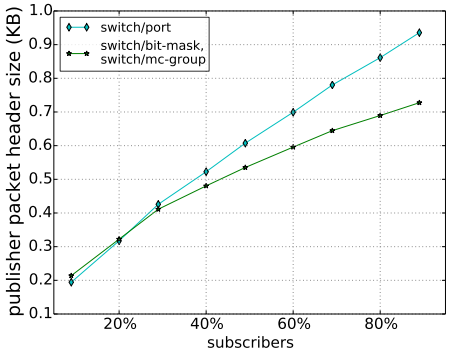


Figure 3.10: Initial header size.

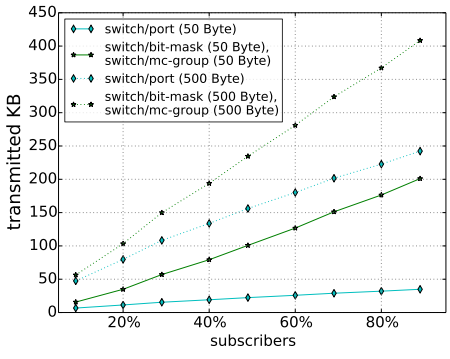


Figure 3.11: Transmitted bytes.

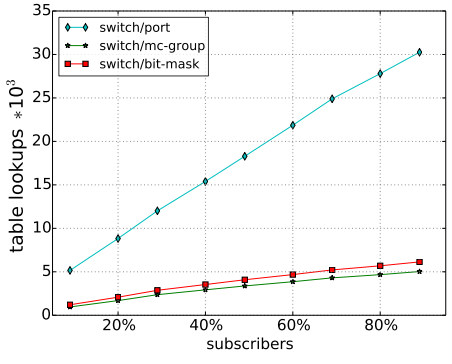


Figure 3.12: Table lookups.

Transmitted Bytes

Figure 3.11 shows the number of *transmitted bytes* as a function of the proportion of subscribers for messages with payloads of 50 bytes and 500 bytes, respectively. Although the initial header size is larger when more than 30% of the hosts are addressed, the *switch/port* strategy generates the lowest network load for all subscriber quantities. This efficiency comes from its use of *predictive pruning*, which removes irrelevant distribution information as early as possible. In contrast, the *switch/*{*bitmask, mc-group*} strategies rely on the less efficient *reactive pruning*.

With reactive pruning, switches do not modify the header stacks of generated packet copies. Instead, downstream switches iteratively remove the top elements of the header stack until they encounter their designated labels. As a result, unnecessary information about unrelated subtrees is passed to downstream switches, leading to longer header stacks and higher network load.

Overall, predictive pruning is more efficient than reactive pruning. While the

differences in network load are marginal for a small subscriber base, these differences become more apparent as the number of subscribers increases.

Table Lookups

Figure 3.12 depicts the number of table lookups for varying fractions of subscribers. The results indicate that the *switch/port* strategy demands significantly more table lookups compared to the other two P4 strategies, with the gap widening as the number of subscribers increases. This is because *switch/port* often processes multiple elements per switch, as each branch of the distribution tree requires mapping as distinct switch/port pairs. Consequently, the number of table lookups grows proportionally with the increasing number of subscribers.

On the other hand, the *switch/bitmask* and *switch/mc-group* strategies require less table lookups, with *switch/mc-group* requiring the fewest. This advantage comes from its use of multicast functionality, which minimizes control flow recirculations and reduces the number of required table lookups. In contrast, the *switch/bitmask* strategy examines the bitmask bit by bit, with each active bit (1-bit) necessitating a separate lookup, leading to slightly higher lookup counts compared to *switch/mc-group*.

In summary, while the *switch/port* strategy offers advantages in terms of predictive pruning, it comes at the cost of higher table lookup demands, especially for larger subscriber groups. Conversely, the *switch/mc-group* strategy requires the fewest table lookups and thus involves the lowest number of necessary control flow circulations among the P4 strategies.

3.5.4 Competing Strategies

Next, we conduct a comprehensive evaluation of a P4-based distribution strategy against alternative message distribution methods. The evaluation focuses on three key metrics: (i) the number of *transmitted packets*, (ii) the volume of *transmitted bytes*, and (iii) the *worst-case delay* in message delivery. Analyzing these metrics provides a comprehensive comparison of the strategies, highlighting their respective strengths and weaknesses.

Based on the previous performance analysis, we have identified the *switch/port* strategy – referred to as *P4-Pub/Sub* – as the most suitable choice for comparison. This strategy is the most promising in terms of message delivery efficiency due to its ability to perform effective header trimming, which reduces packet header size and minimizes network load.

To ensure a fair comparison, we have implemented several alternative message delivery methods, including *OpenFlow multicast (OFM)*, three variations of *application-layer multicast (ALM)*, as well as straightforward *unicast* and *broadcast* approaches. Each strategy has been implemented with SDN-specific adaptations. The following section details their individual implementations.

OpenFlow Multicast

The first strategy for comparison implements *OpenFlow multicast (OFM)* [3, 57], an SDN-based approach that uses flow rules on switches that reference predefined multicast trees. OFM is based on the following principle: a publisher sends a notification to an OFM address, for example consisting of an IP address and a UDP port number [57], which is associated with a multicast tree stored on the switches. OFM trees are realized by corresponding flow rules and are either source-based or shared among multiple senders [12]. For our experiments, we decided to use the source-based variant, which defines a separate tree for each publisher, ensuring accurate multicasting and eliminating false positives. A significant difference is that, rather than using a combination of IP addresses and port numbers, we use Ethernet MAC address fields to distinguish these trees. This adaptation allows a fair comparison with the proposed P4 strategies.

Application Layer Multicast

For comparison with *application-layer multicast (ALM)*, we implemented three broker-based variants. The variants differ in terms of message complexity, i.e. the number of packets transmitted over the network, and are described below, starting with the variant that passes the most packets.

1. *ALM on all nodes:* In this first ALM variant, ALM is implemented on all nodes in the network. Each switch is directly connected to a dedicated broker. In this setup, publishers forward notification messages to their respective brokers. These brokers in turn determine which downstream brokers need to receive the messages and generate duplicates accordingly. The duplicates are sent back to the switch. Thus, incoming messages first traverse the switch-broker link, and subsequent copies of outgoing messages traverse the same link in the opposite direction. This and following ALM variants rely on SDN flows to deliver the messages to downstream brokers.
2. *ALM on fork nodes:* The second ALM variant optimizes the first one by employing the switch-broker combination only at nodes where the distribution tree potentially branches. At other nodes, incoming network packets are tunneled, eliminating the need for broker interaction and accelerating message forwarding.
3. *ALM with mc-groups:* The third ALM variant also focuses on placing brokers exclusively at fork nodes and combines ALM with IP multicast [8]. In this variant, each broker on a fork node uses predefined local multicast groups, each corresponding to a permutation of neighbor nodes. When a notification message arrives at a broker, it determines the correct neighbor brokers for message forwarding and applies the appropriate multicast group. This variant allows outgoing messages to traverse the link between the broker and the switch only once, thereby reducing latency as the brokers are no longer tasked with sequentially forwarding multiple duplicates.

Unicast and Broadcast

To evaluate our P4-based strategies against unicast and broadcast, we installed Ethernet MAC address-based flows in the switches and configured Ethernet frames as notifications sent by the publishers. These frames are routed through the network to individual interested subscribers or to all hosts using the installed flows. Publishers generate an Ethernet frame and configure its destination MAC address before sending it to the switch infrastructure. MAC address-based flows in the switch infrastructure ensure delivery without requiring broker support. In the case of unicast, the sender sends a dedicated notification message to each subscriber, while in the case of broadcast, the notification message is flooded to all hosts via a preinstalled broadcast tree.

Transmitted Packets

Figure 3.13 examines the effect of different percentages of subscribers on the number of packets transmitted. The results show that for all strategies except broadcast, the number of packets transmitted increases with the percentage of subscribers. Broadcast always gives the same result for all deliveries, since the broadcast tree addresses all hosts on every publication. This flooding of the network works well for larger percentages of subscribers. However, broadcast is not suitable for a small number of recipients as it leads to many false positives.

In contrast, unicast gives acceptable results for small numbers of subscribers. However, for large numbers of subscribers, *unicast* suffers from the fact that a packet has to be sent for each subscriber, potentially resulting in multiple copies traversing the same links.

Interestingly, for a receiver quantity of less than 80%, *unicast* is superior to *ALM on all nodes*. This is because this ALM variant requires many additional packet transfers due to switch-to-broker and broker-to-switch communication. *ALM on fork nodes* and *ALM with mc-groups* mitigate this problem by reducing the number of brokers and thus reducing the communication overhead between switch and broker.

ALM with mc-groups is the best performing ALM variant because it only sends a single packet from the broker back to the switch, regardless of the number of packet copies the switch has to send. However, it still sends more packets than *P4-Pub/Sub* and *OFM*. In fact, both *P4-Pub/Sub* and *OFM* send the minimum number of packets, which is also the minimum number possible. This is because both strategies are 'perfect', meaning that a message is only forwarded over a link if it is necessary to reach an interested subscriber. In addition, both strategies converge to unicast when there only a few subscribers, and to broadcast when there are many subscribers.

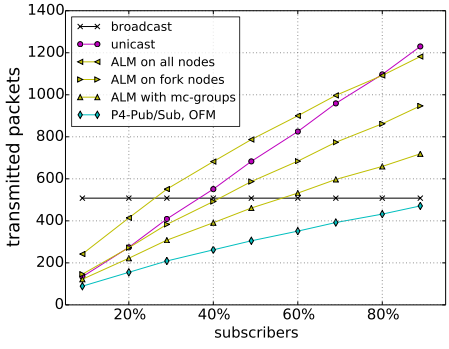


Figure 3.13: Transmitted packets.

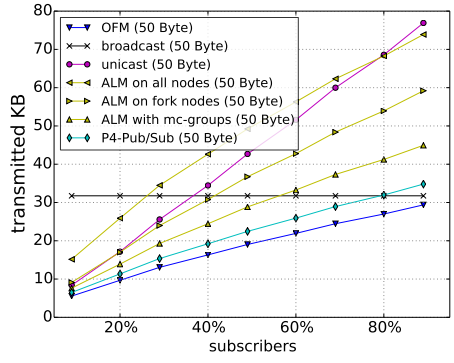


Figure 3.14: Transmitted bytes.

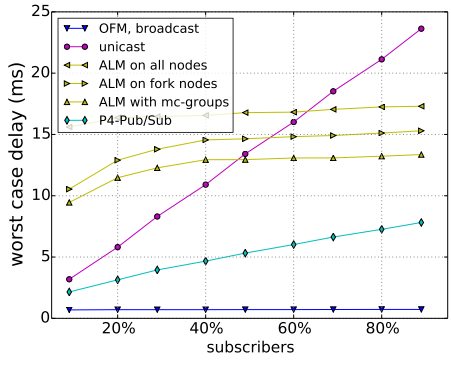


Figure 3.15: Worst-case delay.

Transmitted Bytes

Figure 3.14 depicts the transmitted bytes for notifications with a payload of 50 bytes. The results closely align with those in Figure 3.13, except for *OFM* and *P4-Pub/Sub*. *OFM* transmits the smallest number of bytes. *P4-Pub/Sub* performs marginally worse than *OFM*, with the gap widening as the number of subscribers increases. This is because *OFM* uses the MAC address in Ethernet frames to reference preinstalled multicast trees, which are stored as flow rules in the switches.

However, *OFM*'s efficiency comes with a trade-off: flow rules are only preinstalled for specific subscriber permutations *actively* involved in notification delivery. While this selective rule base minimizes switch memory requirements, it necessitates immediate rule updates whenever subscriptions change. In contrast, *P4-Pub/Sub* encodes all distribution information directly within the notification header, eliminating the dependency on preinstalled trees and providing greater flexibility at the cost of a slightly higher byte count.

Worst-case Delay

The worst-case delay is the maximum possible time that is required to deliver a notification packet to a destination. It is calculated as the difference between the time of sending and the time of latest receipt. When a publisher notifies only one subscriber, the total delay time is calculated from the propagation times required by the publisher and the devices in the delivery path. However, if the publisher is notifying multiple subscribers, the delays of multiple paths must be considered, one for each subscriber. In this case, the *worst-case notification delay* occurs when the last notification to be sent has the longest delivery path.

In order to derive the delays, we use a simple approximation for which we introduce the following times:

(i) network feed-in time per packet at a publisher $0.1\,ms$,

(ii) packet queuing and processing at switches $0.1\,ms$,

(iii) packet processing at brokers $2\,ms$ and

(iv) control flow circulation at a switch $10\,\mu s = 0.01\,ms$.

Note that we do not consider delays caused by network cables. The typical propagation delay for Category 5 cable is around $5\,ns$ per meter and is therefore negligible for a datacenter network with a limited geographical scope.

Figure 3.15 shows the results for the derived worst-case delays for different percentages of subscribers. Broadcast and *OFM* achieve the best results due to the fact that both strategies require only a single table lookup at each involved switch. However, this advantage may diminish under network saturation.

P4-Pub/Sub performs slightly worse in terms of worst-case delay because it requires more lookups depending on the amount of distribution information in the packet header. This is because each switch involved in notification delivery processes all header information to forward and update packet copies.

The three ALM variants have the worst delays caused by software-based filtering and forwarding at the brokers. The more brokers participate in the delivery, the greater the delays. The figure also shows that ALM delays become saturated beyond a certain percentage of subscribers (40%). This means that the number of participating brokers does not increase. *ALM on all nodes* performs worse than the two optimized ALM variants that consolidate fewer brokers for delivery. The lowest delay is actually achieved by *ALM with mc-groups*, as it does not require sequential packet copies to be sent over the switch-to-broker link.

Unicast has shorter delays than all ALM variants for a subscriber percentage of less than 50%. This means that for a small number of subscribers, unicast is a viable alternative. However, the delay of unicast increases steadily with the percentage of subscribers. This is because unicast requires a publisher to

send as many packets over its network link to the pub/sub network as there are interested subscribers. This sequential sending of packet copies causes delays at the publisher side. In the worst case, the copy sent last will be transmitted to the most distant recipient and therefore have the longest delivery path. For example, with 80% subscribers, the publisher has 204 packets to feed into the network and needs $20.4\,ms$ to send them. This time, plus the transmission delay for the last notification, constitutes the worst-case delay.

In our simulation model, broadcast, unicast, and the ALM variants demonstrate robust performance. This is attributed to our assumption of a not fully-loaded system, where we consider a simplified packet queuing and processing delay of $0.1\,ms$. However, it is important to note that for heavily loaded publish/subscribe systems, where numerous notifications from various publishers must be delivered simultaneously, and the paths of distribution trees overlap, we should anticipate longer delays. Our model is simplified by neglecting head-of-line blocking (HoL blocking), which is typically caused by congestion points [56]. HoL blocking occurs when a packet gets stuck in the ingress queue due to switch congestion and subsequently blocks other packets in the same queue. This phenomenon reduces the overall throughput of a packet switching system.

In the case of unicast, congestion scenarios can occur, particularly at the ingress queue, if the switch's forwarding model only allows one packet to be served at a time. Similarly, both *ALM on all nodes* and *ALM on fork nodes* are affected by this performance degradation, as brokers forward outgoing packets to their connected switches via unicast.

On the other hand, *ALM with mc-groups* has a higher internal processing overhead for a switch than unicast switching. In particular, the switch must buffer multiple copies of the packet and forward each copy separately to the appropriate output port via the egress pipeline. This can lead to internal saturation in the switch's buffer subsystem (traffic manager), affecting end-to-end delay.

3.6 Related Work

This section examines various approaches and technologies that have contributed to the discourse on efficient message distribution in distributed systems. By evaluating both traditional protocols and SDN-based methodologies, we aim to highlight the unique characteristics of existing solutions. This will allow us to contextualize and differentiate our approach from established solutions.

3.6.1 Multiprotocol Label Switching

Multiprotocol Label Switching (MPLS) [98] is a widely adopted technique in high-performance networks designed to improve the efficiency of packet forwarding. It achieves this by incorporating link-specific labels into the packet header. Similar

to our proposed P4 strategies, MPLS makes routing decisions based solely on the information contained in these labels, thus avoiding the need to inspect upper layer protocol addresses. These labels act as markers pointing to pre-established *virtual paths*, typically configured by a network controller.

In MPLS, each router along a path performs a label swap: the current label is removed (popped), and a new one – corresponding to the next hop – is inserted (pushed). This dynamic label replacement contrasts with our P4 switch implementation, where header fields are not replaced. Instead, we streamline the header by progressively removing processed or obsolete labels, thereby reducing header length and conserving bandwidth.

Furthermore, the traditional MPLS label format includes a 20-bit value, which is generally sufficient for identifying links in conventional deployments. However, this may be inadequate for our use case. In our *switch/{bitmask, multicast-group}* mapping, we use a more extensive bit length (i.e. 24 bits per label), which is particularly important when dealing with core switches featuring a high number of ports. Increased port capacity requires a correspondingly larger addressable space. To maintain MPLS compatibility while supporting this extended addressing space, one option is to encode our field structure into a stack of MPLS labels. However, configuring such a label stack to reflect our structural requirements introduces significant complexity and may stretch the MPLS paradigm beyond its original design intent.

3.6.2 IPv4/v6 Multicast

Many approaches described in the literature use IP multicast, whether IPv4 or IPv6, to implement pub/sub systems. In IP multicast, reserved multicast addresses play a key role. By sending a message to such an address, a sender can inform all members of the relevant multicast group with a single transmission. This method does not require prior knowledge of the recipients' identities or the number of senders. The dissemination of messages from senders to receivers is achieved through multicast routing protocols, which manage group membership and control the path that multicast messages take through the network. Examples of multicast routing protocols are *Distance Vector Multicast Routing Protocol (DVMRP)* [113], *Multicast Open Shortest Path First (M-OSPF)* [85], *Core Based Trees (CBT)* [7, 6], and *Protocol Independent Multicast (PIM)* [45].

Two types of multicast trees exist: *source-based* trees and *group-shared* trees. Source-based multicast trees can be constructed using a distance vector-style algorithm, which can be implemented separately from the unicast routing algorithm (as in DVMRP). Alternatively, the multicast tree can be built using information from the underlying unicast routing table (as in PIM). Link-state algorithms, exemplified by M-OSPF, are another method to build source-based trees. On the other hand, group-shared trees, as found in CBT, select one router as the root and transmit packets through this root router. However, the delay in the tree is often longer than in a source-based tree.

Both types of multicast protocols use group management protocols that operate between hosts and multicast routers, such as *Internet Group Management Protocol (IGMP)* for IPv4 [20] or *Multicast Listener Discovery (MLD)* for IPv6 [36].

However, while implementing publish/subscribe with IP multicast is straightforward for the channel, subject or topic-based variants, it becomes particularly complex and challenging for the content-based variant. This is because each notification message can be directed to a different subset of subscribers. As a result, a significant number of multicast addresses may be required to deliver a message in a single transmission.

The caveat here is that the IP multicast address space is finite and has different constraints depending on the IP version. For IPv4, the address space (including Class D addresses) is 28 bits [102], while for IPv6 it is 112 bits (prefixed with ff00::/8), with an additional 8 bits reserved for multicast address flags and scope information [83]. However, a portion of this address space remains reserved for various protocols, including routing protocols, low-level topology discovery or maintenance protocols. Consequently, the usable multicast address space is limited for both IPv4 and IPv6.

In response to these address space limitations, several papers have been published [8, 9, 21, 87]. These papers explore different approaches to reduce the number of required multicast groups. For instance, some approaches suggest grouping subscribers with similar interests [21, 87]. However, these approaches have drawbacks, such as delivering messages to uninterested subscribers, or requiring multiple message transmissions rather than a single transmission. In addition, creating, updating and storing multicast trees can be resource-intensive, potentially exhausting switch memory. This makes these approaches less suitable for content-based pub/sub systems with their dynamic nature.

3.6.3 Clustered Group Multicast

In an effort to minimize the number of multicast groups needed, Opyrchal et al. [87] introduced the concept of *Clustered Group Multicast (CGM)*. This approach divides the network into clusters, creating a distinct group for each multicast set. This subdivision effectively reduces the total number of required groups. However, it comes with the potential drawback of duplicate transmissions when a message must reach members of multiple groups.

The authors further refined CGM with a threshold-based variant called *Threshold Clustered Group Multicast (TCGM)*. In TCGM, a cluster is only flooded with a notification message if the number of recipients within that cluster exceeds a certain threshold. Otherwise, the notification is sent only to interested recipients via preinstalled groups. By carefully selecting this threshold, the number of groups required can be further reduced. The downside of this approach, however, is the delivery of notifications to uninterested participants and an increase in the overall network load.

In addition, the authors proposed a method where multicast groups are set up on each broker for any combination of neighboring brokers. This is similar to our local multicast groups (cf. Section 3.3.3), with the difference that the multicast group is not given in the header, but must be computed within the broker before it can be applied. When a message arrives at a broker, it determines the next multicast hop to the destination and selects the appropriate multicast group for forwarding. While this method scales better as the number of brokers increases, it introduces additional processing overhead within the brokers due to broker-specific hop calculations, potentially leading to delays in delivery.

In contrast, our strategies do not rely on IP addresses or calculate multicast hops to intermediate brokers. We encapsulate the entire distribution tree in the packet header, and our strategies are designed so that only interested recipients are targeted.

3.6.4 PLEROMA

Tariq et al. [13] use a recursive method to divide the notification space into distinct subspaces. These subspaces are then associated with strings that serve as representations of specific IP multicast address ranges. A decomposition process that divides an existing subspace into two smaller ones is used to assign IP addresses to these subspaces. These subspaces are distinguished by so-called *dz-expressions (DZEs)*, which are essentially strings of bits used for spatial indexing. Each time a subspace is split, its DZE is transmitted as a prefix to the two newly formed subspaces. Consequently, a *covering dz-expressions* is one that serves as a prefix to the dz-expressions it encompasses.

DZEs allow the decomposition of the event space, provided that the corresponding strings can be mapped to multicast addresses. When a notification is to be distributed, a publisher identifies the subspace that matches the subscriptions and encodes the DZE of that subspace in the IP address field of the notification packet. Specifically, a notification is like a point in this multidimensional space and is distributed by the publisher with a DZE of maximum length corresponding to the smallest subspace that encapsulates the notification.

Subscriptions and advertisements are associated with one or more DZEs that represent the relevant subspaces. To ensure successful delivery of a notification to a subscriber, all switches along the route between the publisher and the subscriber must be equipped with a flow that contains a covering DZE for the correct output port. This is achieved by extending DZEs to IPv6 multicast addresses and incorporating appropriate flow rules into the switches. These rules check these addresses for coverage using standard switch operations commonly used in *Classless Inter-Domain Routing (CIDR)*.

However, in cases where the subscription filters do not match the available subspaces exactly, publishers are forced to either address the next larger subspace or send multiple copies of notification packets, each with different DZEs. Both

of these scenarios lead to increased network load and the potential delivery of false positives, requiring post-processing filtering. In contrast, our source-routed P4 delivery strategies offer precise multicast distribution, but may require more space in a message header.

3.6.5 COXcast

COXcast [64] introduces a novel method of encoding distribution trees into a single identifier called a *Multicast ID (MCID)*. In this approach, an MCID is calculated and embedded in the packet header prior to distribution. As the packet traverses each switch in the network, the MCID is extracted from the header and translated into *Output Port Bitmaps (OPBs)* using node-specific keys. Packet copies are then forwarded based on the computed OPB.

The underlying principle behind this calculation lies in leveraging the properties of the *Chinese Remainder Theorem (CRT)*. By dividing an MCID by different keys, unique remainders are obtained, each corresponding to a distinct OPB. This is achieved through a modulo operation applied to the MCID using its respective node-specific key at each intermediate switch.

A critical prerequisite for this method is the assignment of a unique key to each switch, with all keys in the network being pairwise relative primes. A major advantage of this approach is that it avoids header modifications on packets.

However, a drawback is the potential for the MCID to occupy a wide header field with a significant number of bits. This is because MCID calculations depend on the key values of the switches, and each key must be larger than the port bitmap size of the switch, which typically exceeds 2^n, where n represents the number of switch ports. Consequently, large prime numbers are required, resulting in sizable MCIDs. In addition, due to the irregular intervals between successive prime numbers, the size of the MCID is dependent on the number of switches involved and the range of prime numbers they use.

Regrettably, implementing COXcast on P4 architectures encounters limitations. P4 architectures impose constraints on the maximum size of integers and only support modulo operations that can be calculated at compile time [33]. Computations involving non-constant values are not supported due to the complexity of the modulo operation, which cannot be simplified to simple bit-wise arithmetic. This limitation directly affects the implementation of the modulo operation needed to calculate the output port bitmap. However, *Field Programmable Gate Array (FPGA)* platforms could potentially provide a solution by designing an architecture-specific construct for this operation. The FPGA board's modulo calculator could then be integrated into the P4 program as an external object.

3.7 Summary

This chapter has presented a variety of notification distribution strategies customized for content-based publish/subscribe systems; all implemented using the P4 SDN programming language. The main advantage of our strategies is their ability to make forwarding decisions at the network layer, effectively eliminating the need for application-layer brokers and thus removing a potential bottleneck. Our proposed strategies provide complete control over message distribution trees.

The source-based strategies proposed in this chapter revolve around encoding the distribution tree of a notification message within the message itself, thus avoiding the need to update forwarding rules when subscriptions change. We also avoid mapping routing information to conventional protocol fields, and instead devise custom header fields designed for dynamic sets of receivers. These custom header fields do not require the use of specialized ASICs or hardware extensions. Instead, the proposed implementations are designed to be fully compliant with the P4 specification, using the functions and constructs of P4 to accurately route notification messages. This approach allows our implementations to be compiled into flow tables that can be integrated into the memory of any P4-enabled switch.

In this chapter, we introduced a variety of encoding schemes that have been developed to map the distribution tree of a message. Through an extensive evaluation conducted within an emulated network, we demonstrated the adaptability and performance of these encoding schemes within the complex context of content-based pub/sub systems. Our analysis was twofold: firstly, we compared our different strategies against each other, and secondly, we performed a comprehensive comparison between one of our P4 strategies and conventional approaches, encompassing unicast, broadcast, OpenFlow multicast, and three variants of application-layer multicast. The results of these evaluations demonstrate the efficiency of our P4 strategy. It is almost as efficient as OpenFlow multicast in terms of network bandwidth consumption, while avoiding the need to store extensive forwarding rules. Furthermore, our source-based P4 strategy shows lower worst-case delays compared to both unicast and application-layer multicast.

Nevertheless, the advantages of source-based routing come with some notable challenges. Firstly, publishers must have a complete knowledge of all participants in the network in order to efficiently compute a distribution tree. In addition, encoding the entire distribution tree within the header can result in significant header stack sizes, potentially reducing space for payload content. The following chapter addresses these challenges, focusing on strategies for efficiently offloading parts of the distribution tree into switch memory. This state-based forwarding approach preserves the flexibility to create custom distribution trees, while striking a balance between storing routing information within the infrastructure and within the header stack.

Chapter 4

Forwarding Trees

Contents

4.1 Introduction

In the previous chapter, we introduced source-based P4 strategies that efficiently encode routing details within message headers, enabling the dynamic creation of distribution trees without requiring switch reconfigurations. While these strategies offer notable advantages in terms of adaptability, they have limitations in terms of scalability in pub/sub networks. As the diameter of the network grows and the number of notification recipients increases, the space required within the header stack for accommodating routing information in notification-specific distribution trees also raises. This, unfortunately, leads to a reduction in available payload space.

To address these challenges and improve scalability, this chapter presents alternative strategies that identify stable parts of distribution trees and delegate them to the switch infrastructure as forwarding rules. Meanwhile, publishers can add dynamic parts to these trees on-demand, forming notification-specific distribution trees. These strategies focus on reducing header stack sizes by using the network infrastructure to store state information. The concept revolves around a publisher incorporating tree identifiers into the header stack of a notification packet, along with optional labels that extend branches of the stored trees, while switches apply these two types of labels to route the packet along the notification-specific distribution tree. We evaluate strategies using these labels, combining preinstalled forwarding rules with supplementary header distribution information. We also address open research challenges related to striking the right balance between header size (i.e., what and how much information to encode in the packet header of a notification) and the size of the installed rule base (i.e., how many rules are required to be installed and regularly updated concerning a switch's limited memory capacity and update frequency, respectively).

The remainder of this chapter is structured as follows: Section 4.2 introduces three fundamental forwarding schemes implemented in the P4 language. These schemes rely on dynamic forwarding information within the message header and static forwarding rules installed on the switches. They encompass approaches that (i) store information about neighboring nodes and route packets hop by hop from node to node, (ii) establish subscriber-specific forwarding paths in the switches, or (iii) group stable subscriber sets and establish publisher-rooted multicast trees for each group in the switch infrastructure. Section 4.3 proposes a hybrid routing scheme that combines preinstalled routing rules with additional distribution information stored in header fields. It explores the optimal balance between the information encoded in a notification's packet header and the number of rules that must be installed for stable receiver sets in switch memories. Section 4.4 conducts a comparative evaluation of the presented strategies within an emulated network environment. This evaluation focuses on the trade-off between header size and the number of rules installed. It also examines the impact of changing subscriptions on the static rule base. Section 4.5 examines related work in the field. Finally, Section 4.6 provides a summary of the chapter's key findings and insights.

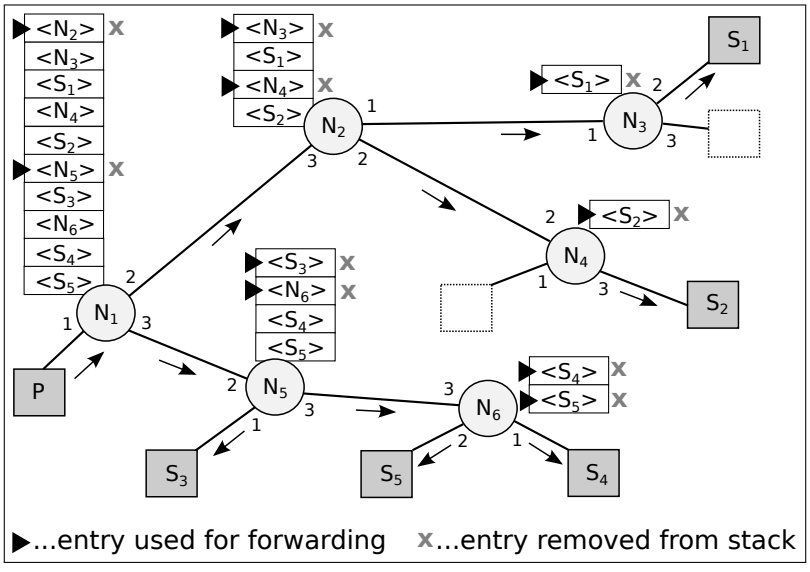


Figure 4.1: Forwarding with switch-hops.

4.2 Stateful Encoding Schemes

In this section, we introduce basic strategies for reducing the size of the header stack, albeit at the cost of storing state information within the switches. Each of these strategies involves varying degrees of information storage within the switches, which can be either the next hop to a direct neighbor, or paths and trees leading to final destinations. By embedding labels in the header of a notification message, the proposed strategies reference and apply the stored information for message delivery.

4.2.1 Hops to Neighbors

The first strategy is called the *hops* strategy. It is based on the source-based *switch/port* strategy discussed in the previous chapter. However, instead of encoding tuples of switch identifiers and ports, this strategy encodes only the ID of the next-hop neighbor to be addressed. The distribution tree for a notification message is derived by listing the IDs of all participating nodes in a pre-order traversal. As switches process the elements in the header stack, they route the notification packet through the listed nodes, ensuring it reaches the intended destinations. If the publisher knows the initial node to send the notification to, the root node can be omitted from the stack.

For this strategy to function as described, each switch must be aware of the IDs of its immediate neighbors, including other switches and subscribers. This requires

the SDN controller to provide flow rules within the network infrastructure for every ID listed in the header stack. Each switch stores a match-action rule for each of its ports, where the match condition is based on the ID of the next node. The corresponding action causes a copy of the packet to be forwarded to that next node.

An example of this strategy is illustrated in Figure 4.1. When network switch N_1 receives a notification from publisher P, it begins processing the header stack sequentially, extracting and analyzing each element. Any irrelevant upstream information is removed from the top of the stack as the switch scans for a known neighbor ID (e.g., N_2 or N_5). Upon identifying a valid ID, the packet is cloned and forwarded to the respective neighbor. Before the cloned packet exits the switch, the header stack is trimmed to remove unnecessary forwarding information, similar to the trimming process in the *switch/port* strategy (see Section 3.3.1).

Because the IDs in the header stack are listed in pre-order, the forwarding information for a neighbor's subtree begins immediately after its ID and extends either to the next neighbor's ID or to the end of the stack. Therefore, only the part of the header relevant to a neighbor's subtree needs to be preserved for forwarding. For instance, when N_1 forwards packets to its neighbors N_2 and N_5, the subtree for N_2 starts directly after N_2's ID and ends at N_5's ID. Similarly, the subtree for N_5 starts right after N_5's ID and continues to the end of the stack. This ensures efficient packet routing while pruning unnecessary data within the header stack.

4.2.2 Paths to Destinations

The second strategy, referred to as the *destinations* strategy, stores dedicated paths for each potential destination within the switch infrastructure. This approach aims to reduce the size of header stacks in packets at the expense of increased memory usage on switches. Unlike strategies that embed the entire distribution tree within the message header, this method encodes only subscriber IDs in the header while relying on preinstalled paths stored within switches to route the message to its intended recipients. These preinstalled paths ensure that all switches in the network are aware of every potential destination, with each switch maintaining a single flow rule per subscriber ID.

When a message arrives at a switch, the P4 program processes the subscriber IDs sequentially from the header stack. For each ID, the switch performs a lookup in its flow rules to determine the corresponding output port. Based on this result, a duplicate of the packet is created and forwarded through the identified port.

Figure 4.2 illustrates this strategy. In the example, switch N_1 receives a packet intended for 5 subscribers (S_1, ..., S_5), represented as five stack entries in the packet header. The switch processes each entry, determining the appropriate output port for each subscriber. For instance, N_1 forwards one copy over port 2

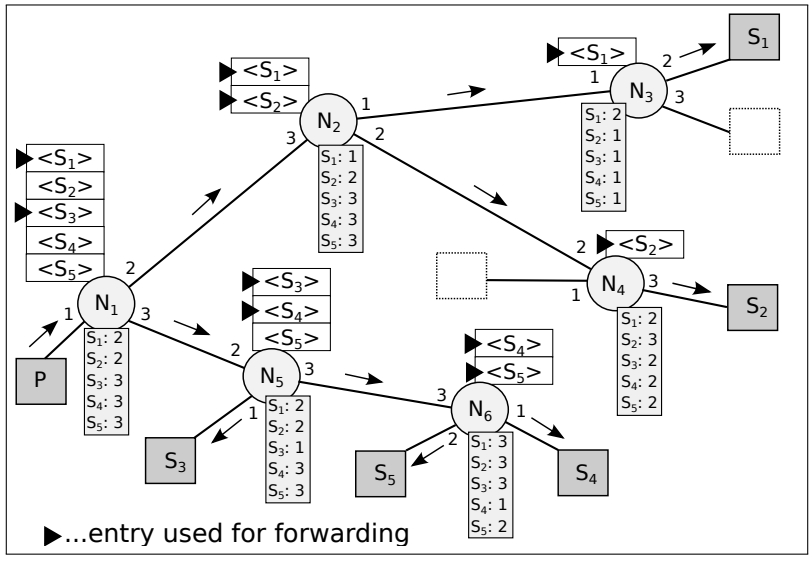


Figure 4.2: Forwarding by listing destinations.

for subscribers S_1 and S_2, and another copy over port 3 for subscribers S_3, S_4, and S_5.

In cases where multiple subscribers share the same output port on a switch, the switch creates only a single packet copy for that port. To ensure this, a *bitmask field* in the metadata is used to track ports that have already been served by a clone. When the switch processes a subscriber ID and identifies its output port, the corresponding bit in the bitmask is checked. If this is the first time the port is being used for a clone, the bit is set, and a clone is created. For subsequent subscriber IDs associated with the same port, no additional clone is generated. For example, the bitmask ensures that N_1 forwards only one packet over port 2 for S_1 and S_2, and one packet over port 3 for S_3, S_4, and S_5.

An essential part of this strategy is pruning irrelevant information from the header stack before forwarding packets to the next hop. This prevents redundant clones from being created at subsequent switches. Before a cloned packet leaves the switch, the header stack is post-processed by comparing the output port associated with each subscriber ID against the output port of the clone. IDs corresponding to different ports are removed from the stack, ensuring that the packet only carries information relevant to the subtree being served by the clone.

The key advantage of this strategy is that it significantly reduces the number of header stack entries required, resulting in smaller message headers than those produced by strategies like *switch/port*. However, this comes at the cost of increased management complexity. Any change in subscriptions requires the rule base to be updated across all switches, since each switch manages a dedicated rule for every subscriber, directing notifications to the intended recipients. This

trade-off highlights the importance of striking a balance between switch memory consumption and how often rule updates are required.

4.2.3 Multiple Publisher-centric Trees

Instead of managing individual paths, the infrastructure can also use entire *publisher-centric* forwarding trees, similar to the concept of IP multicast. This is the *tree* strategy. All subscribers of a publisher are addressed by a single label, and each label is associated with predefined match-action rules installed within the switches. When a packet's label matches one of these rules, the corresponding action generates duplicates for a specific set of output ports on the switch. Using the P4 forwarding model, multiple clones can be created for different ports through *local multicast groups* (see Sect. 3.3.3). This allows all the required clones to be produced with a single action.

While the *tree* strategy downsizes header length by requiring only one label, it faces scalability challenges, especially when dealing with a large number of subscribers. This approach performs best in scenarios involving a moderate number of subscribers, as the rule base must map multicast trees for every possible combination of active subscribers. When the number of receivers grows, particularly in dynamic environments with varying subscriptions, the rule base's size can quickly exceed the memory capacity of network switches, creating bottlenecks.

Additionally, implementing publisher-centric distribution trees necessitates careful planning. These trees must be identified and preinstalled in the network before they can be used, a non-trivial process. For instance, an SDN controller must derive these multicast trees based on comprehensive, real-time knowledge of the pub/sub network. Such knowledge can be acquired through methods like statistical monitoring of communication flows, which allows the controller to analyze patterns and construct meaningful tree structures. However, this adds considerable operational complexity and overhead.

Moreover, the dynamic nature of content-based pub/sub systems poses further challenges. Frequent updates to the rule base are required to accommodate subscription changes, making the *tree* strategy impractical in many cases. This dynamic nature, combined with the potential for state explosion in large-scale systems, limits the applicability of the *tree* strategy.

4.3 Hybrid Distribution Scheme

In content-based pub/sub systems, the composition of the receiver set can vary significantly between different notifications. However, in most scenarios there is some degree of stability in this dynamic behavior. Typically, a subset of subscribers remains constant, expressing interest in almost all notifications from

particular publishers. It is therefore beneficial to build and maintain reusable forwarding trees for these stable subscribers.

To reach the remaining, more dynamic subscribers, additional node identifiers for missing delivery paths or subtrees are embedded within the notification header. This *hybrid* strategy combines the hop-based encoding approach of the *hops* strategy with the stored forwarding trees from the *tree* strategy. By merging these strategies, the hybrid approach offers a flexible trade-off between embedding distribution data directly in the packet header and storing it in the switch infrastructure. It also provides the adaptability to address different subsets of subscribers through separate methods within the same strategy.

This strategy relies on a precomputed, publisher-rooted tree T that includes all potential subscribers for a publisher. From T, a reduced tree T' is derived, containing only the nodes representing stable subscribers. To construct T', only the branches and edges of T that connect to stable subscribers are kept, while the branches leading to dynamic receivers are excluded.

When disseminating notifications, the publisher leverages the stored tree by encoding its identifier in the packet header. If additional subscribers are not covered by the stored tree, the publisher supplements the header with *hop labels* to complete the delivery tree. These hop labels represent the IDs of neighboring switches or subscribers for which forwarding rules are preconfigured in the switches, as outlined in Sect. 4.2.1.

Switches process the header stack sequentially, analyzing each element from top to bottom. When encountering the ID of a known neighbor or tree, the switch creates a clone of the packet and forwards it either to the specified neighbor or along the branches of the corresponding tree. Before the cloned packet is sent, the switch prunes the header stack, removing hop labels that have already been processed, thereby eliminating irrelevant forwarding information for downstream switches. However, tree identifiers are preserved in the header since they may still be relevant for subsequent switches.

Note that instead of hop labels, the *hybrid* strategy can also incorporate other encoding formats, such as *switch/{port, bitmask, multicast-group}* pairs, as described in the previous chapter.

4.3.1 Example

Figure 4.3 provides an illustrative example of our hybrid strategy. In this scenario, subscribers S_2, S_3 and S_5 are reached via the pre-stored distribution tree T_1. The publisher P encodes the ID of this tree in the header stack of a notification. The publisher also includes additional delivery information to attach S_1 and S_4 to T_1 via additional hops. The complete distribution tree thus addresses five subscribers (S_1, …, S_5). For dynamically connected subscribers, the additional information only encodes the missing part of the path leading from a

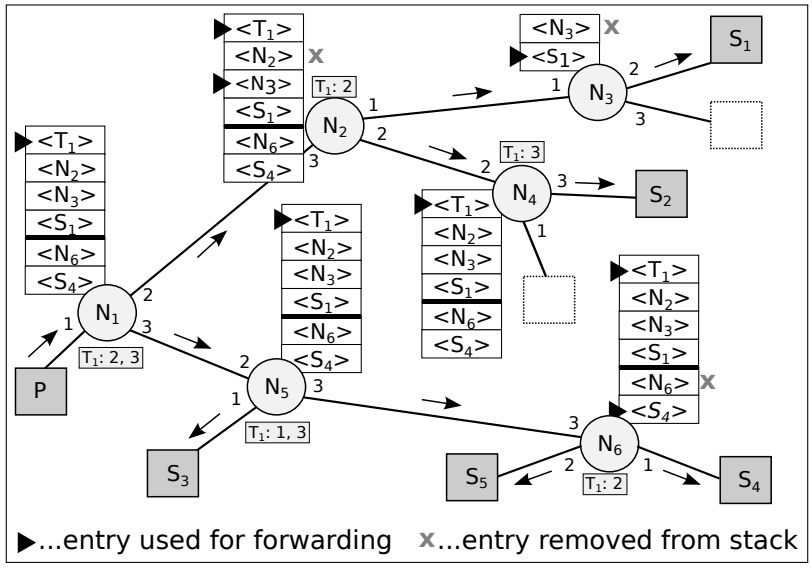


Figure 4.3: Forwarding with stored trees and switch-hops.

node at T_1 to the final destination. As an illustration, the example header stack encodes two routes: first, the path from N_2 to S_1, and second, the path from N_6 to S_4, both achieved by switch hops.

When a switch routes the message according to the rule associated with the T_1 labels, it scans the header stack element by element to check if any other rule matches. Such a match occurs when the switch encounters its own unique ID. This identifier designates the switch as the root of a switch-hop encoded subtree, leading to additional receivers. In the example, both N_2 and N_6 act as such root nodes. Hop labels within the header stack following the root element denote paths to the final receivers. If the header stack encodes several additional subtrees, these are separated by a marker element, as indicated by the bold line in the figure.

However, a notable drawback of this hybrid variant is the limited pruning of the header stack. As the packet traverses the preinstalled forwarding tree, the header stack remains unchanged and may therefore contain irrelevant information for downstream switches. This is because the forwarding switches have no knowledge of which branch is being extended by dynamically encoded forwarding information within the header. Essentially, a switch can only remove an entry from the stack if it matches its own switch ID.

4.3.2 P4 program

The *hybrid* strategy is difficult to implement. It requires the seamless integration of two distinct label types: the *hops* strategy labels, which refer to neighboring

switches, and the *tree* strategy labels, which reference preinstalled multicast trees. Achieving synergy between these two labeling mechanisms presents a significant technical challenge.

Our implementation of the hybrid strategy is based on the use of *bitmasks* for both label types. These bitmasks serve two primary purposes:

(i) *Storing preinstalled multicast trees* in the switches' memory.

(ii) *Identifying additional hops* that extend a delivery tree beyond stored trees.

By employing flow rules that store bitmasks instead of traditional multicast groups, this approach ensures the flexibility required to unify the *hops* and *tree* strategies. Although processing bitmasks introduces a degree of complexity, it facilitates seamless integration of these two approaches into a hybrid strategy.

When a switch encounters a tree ID in the header stack, it retrieves the associated bitmask from its memory, stores it in a metadata field, and creates packet duplicates for every port represented by an active (1-bit) entry in the bitmask. Conversely, when a switch processes a hop label, the bitmask extracted from the label is directly copied into the same metadata field. For each bit in the bitmask (corresponding to an output port), a packet duplicate is created and forwarded through the specified port.

Figure 4.4 outlines the step-by-step packet processing procedure for this implementation:

1. *Packet header parsing:* Upon receiving an incoming packet, the P4 program initiates packet header parsing. At this stage, only the topmost label in the header stack is extracted for processing. Any remaining labels in the stack are ignored temporarily, to be processed in subsequent control flow passes.

2. *Header format validation:* The ingress pipeline validates the format of the extracted label to confirm that it conforms to the expected structure. If validation fails, indicating for example that no header stack is found, the packet is dropped. Conversely, successful validation signals the presence of at least one valid stack label for further steps.

3. *Clone check:* Before processing the label, the program checks whether pending clones still need to be created. This step is essential, as only one clone operation is permitted per control flow pass. A bitmask stored as metadata tracks the remaining ports requiring service. If this bitmask contains active (1-bit) entries, the program prioritizes clone creation before proceeding.

4. *Stack element processing:* If no pending clones remain, the program proceeds to process the top label in the header stack. Two types of labels are distinguished at this stage: *tree* labels (representing preinstalled multicast trees) and *hop* labels (representing additional distribution information).

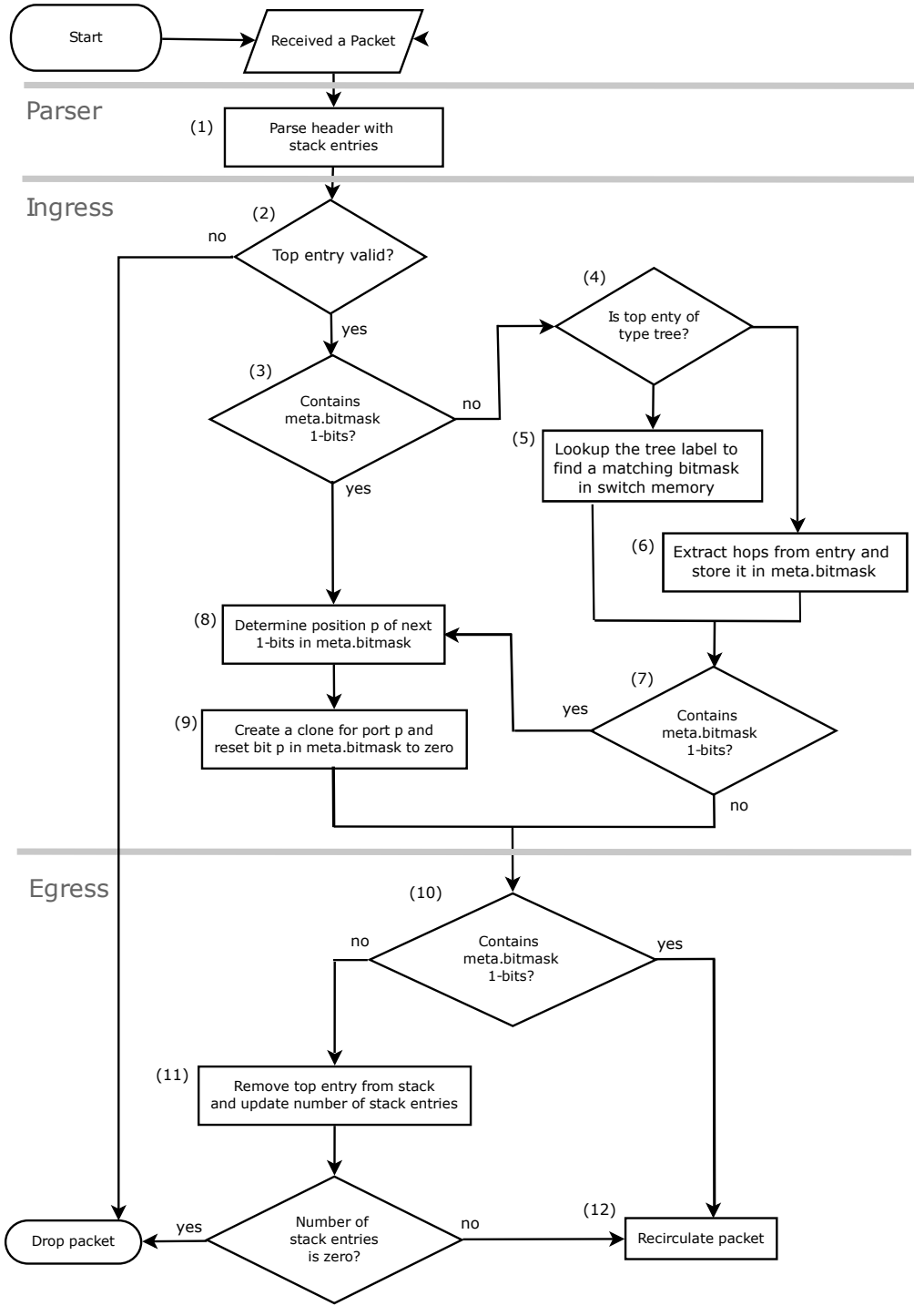


Figure 4.4: Processing of an original packet instance according hybrid distribution scheme as flow chart.

5. *Handling tree labels:* When a label is identified as a *tree* label, the program determines the egress ports associated with the specified tree. To achieve this, the switch checks its memory for a bitmask associated with the tree label. This bitmask specifies the output ports for the required clones.

6. *Handling hop labels:* For a *hop* label, the program extracts the bitmask encoded in the label and copies it into the metadata field. This unifies the handling of forwarding information, treating bitmasks from stored trees and hop labels equivalently.

7. *Bitmask check:* The program checks whether at least one bit in the metadata bitmask is set. If any bit is set, it indicates that a tree or hops label relevant to the current switch has been identified.

8. *Locating next bit:* If the bitmask is non-empty, the program performs a filtering operation to locate the position of the next 1-bit within the bitmask. This position corresponds to the egress port for the next clone.

9. *Clone generation:* A packet clone is created for the identified port, and the corresponding bit in the metadata bitmask is cleared to prevent redundant operations.

10. *Egress pipeline processing:* The packet is forwarded to the egress pipeline for further processing. Depending on the packet's state, this stage may involve additional steps such as packet recirculation or header modifications.

11. *Entry removal:* Once all bits in the metadata bitmask have been processed and the necessary clones are created, the program removes the processed label from the header stack. It then continues to the next label (if any) in subsequent iterations

12. *Packet recirculation:* If the metadata bitmask is empty and additional labels remain in the stack, the packet is recirculated to process the next label.

Alternative Implementation

An alternative implementation can scan *all* stack entries first and then generate the clones for all identified output ports, instead of generating clones for each stack entry separately. The program processes all labels until the end of the stack, activating the corresponding bits in the metadata bitmask using an OR operation for each recognized label. This incremental bit activation method progressively aggregates routing information across all labels for the switch before generating clones.

Once all labels in the stack are processed by this "batch stack scanning", the program sequentially generates packet duplicates for each active (1-bit) entry in

the bitmask. This alternative implementation has the added benefit of preventing duplicate packet generation for the same port, which can occur when hop labels overlap with ports already specified in the stored tree.

4.4 Evaluation

In this section we provide a comparative assessment of our state-based strategies. To perform this evaluation, we implemented the strategies and conducted various scenarios. These scenarios involved manipulating the number of subscribers, the notification payloads, and the rule base deployed within the switches.

4.4.1 Publish/Subscribe Network Setup

We evaluated the proposed notification distribution strategies using the same simulation setup introduced in Chapter 3 (see Sect. 3.5). This includes the use of a Jellyfish topology generated with *NetworkX* [52], the *Mininet* network emulator [73] (version 2.1.0), and the BMv2 [10] P4 software switch compiled for the `simple_switch` target. The base topology consists of 256 switches, each connected to a host capable of acting as a publisher, subscriber, or both.

As in Chapter 3, a *traffic generator* was used to simulate network activity. In each run, a publisher was randomly selected, and the number of matching subscribers was determined using a configurable percentage parameter. The experiments covered three levels of *subscriber density* (10%, 30%, 60%) and three different *payload sizes* (50 B, 250 B, 500 B).

To ensure a fair and reproducible comparison across all evaluated strategies, the traffic generator was initialized with the same random seed used in the previous chapter. This guarantees that each strategy is evaluated under identical traffic conditions. For each data point, 25 runs were performed, and the results were averaged to account for variability.

The evaluation focused on three key metrics:

(i) Initial header size of each notification.

(ii) Total number of forwarding rules installed across all switches.

(iii) Overall network traffic generated during distribution.

4.4.2 Performance Results

Figures 4.5–4.7 present the performance results of the P4 strategies, displayed as grouped bar charts. Each group represents a specific strategy.

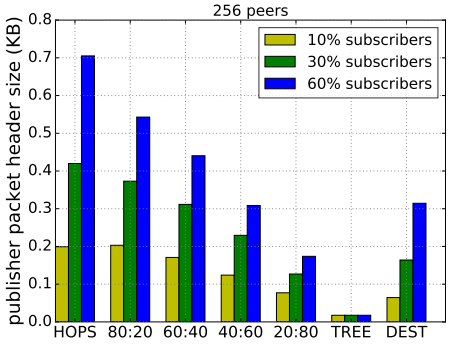


Figure 4.5: Initial header size.

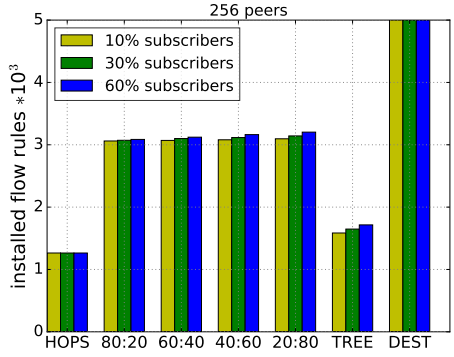


Figure 4.6: Installed flow rules.

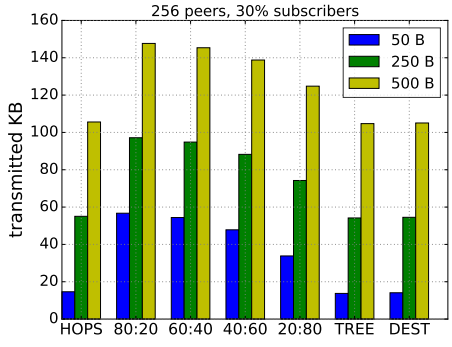


Figure 4.7: Network traffic.

The leftmost bar group (HOPS) represents the results for the *hops* strategy, which encodes the entire delivery tree by hop labels within the header stack. The subsequent bar groups correspond to different variants of the *hybrid* strategy, which combine aspects of the *hops* and *tree* strategies in varying proportions. These proportions are indicated by the strategy labels, which specify the ratio of dynamically encoded information (via hop entries) to statically encoded information (via stored trees). For instance, a label of 80:20 indicates that 80% of subscribers are reached using dynamically added hop entries, while the remaining 20% are reached through references to preinstalled trees. The next bar group (TREE) shows the results for the *tree* strategy, which exclusively relies on stored distribution trees to reach subscribers. This strategy achieves most compact headers by encoding only a tree ID in the header. Finally, the rightmost bar group (DEST) represents the *destinations* strategy, where destination IDs for individual subscribers are explicitly listed in the header stack.

This organization of bar groups facilitates a direct comparison of the strategies' performance metrics, highlighting the trade-offs between dynamic and static encoding approaches, as well as their respective impacts on header size, installed rules, and network load.

Initial Packet Header Size

Figure 4.5 provides an analysis of the initial header sizes transmitted by the publisher to the pub/sub network for a notification. Among the strategies, the *hops* strategy produces the largest headers due to its approach of explicitly encoding subscriber paths within the header stack. The size of the header increases as the number of subscribers grows, making it the least efficient in terms of header size scalability.

On the other hand, the *tree* strategy achieves the smallest header sizes. This comes from its design, which requires encoding only the ID of a preinstalled distribution tree in the header, regardless of the number of subscribers. As a result, its header size remains constant across varying subscriber percentages. However, this strategy involves mapping multicast trees for every possible combination of active subscribers, which is impractical for large-scale networks.

The *hybrid* variants produce header sizes that fall between those of the *hops* and *tree* strategies. The specific header size depends on the proportion of subscribers addressed via hop entries versus those reached through stored trees. As this balance shifts, the hybrid strategy leans toward the characteristics of either the *hops* or *tree* strategy.

Similarly, the *destinations* strategy occupies an intermediate position. It employs compact labels within the header stack to refer to individual stored paths, resulting in a header size that scales with the number of unique paths required. This makes it more efficient than the *hops* strategy but less so than the *tree* strategy, particularly as the number of subscribers increases.

Installed Flow Rules

Figure 4.6 illustrates the number of flow rules required across different percentages of subscribers for each strategy. The results highlight that the *hops* strategy is the most lightweight in terms of flow rules. It requires only a small, constant number of rules per switch, primarily to store the IDs of neighboring nodes. This minimal rule base makes the *hops* strategy appealing for scenarios where memory resources in the network infrastructure are limited.

Conversely, the *tree* strategy demands a larger rule base. It requires a constant number of rules to handle bitmasks, as well as additional rules to store forwarding trees represented in bitmask form. The size of the rule base depends on the number of subscribers included in these stored trees: larger trees, which cover more subscribers, necessitate more rules. Moreover, since the forwarding trees are inflexible and static, the associated rule base must be updated whenever subscriptions change.

The *hybrid* strategy combines labels from the *hops* and *tree* strategies. As a result, its rule requirements are a composite of the two strategies.

The *destinations* strategy is the most memory-intensive. It stores an extensive rule base, requiring n^2 rules in a network of n participants (e.g., $256^2 = 65,536$ rules for 256 hosts). This ensures that any participant can be reached via preinstalled paths originating from any other host, eliminating the need for updates when subscriptions change. However, this strategy sacrifices scalability, as the sheer number of rules can easily exceed the memory capacity of network switches in larger deployments.

Network Load

Figure 4.7 examines the network load in terms of the total volume of bytes transmitted for notification messages delivered to 30% of the hosts, using three different payload sizes. Across all strategies, the results confirm that the network load scales with the payload size, as expected. For notifications with larger payloads (e.g., 500 bytes), the network is stressed the most, regardless of the strategy employed.

The most significant difference between the strategies is how they handle header pruning. The *hops* strategy has the most effective header pruning, which results in a reduced network load. By trimming redundant forwarding information at each switch, the *hops* strategy minimizes the size of transmitted packets as they propagate through the network.

In contrast, the *tree* and *destinations* strategies may carry unnecessary forwarding information through downstream switches. Nevertheless, the results for the *hops* strategy closely align with those of the *tree* and *destinations* strategies, although the latter rely on stored paths or trees, whereas the *hops* strategy encodes most of the distribution information in the header itself.

Interestingly, the *hybrid* strategy generates the highest network load in its 80:20 variant, where 80% of the recipients are addressed using hop entries and the remaining 20% rely on a stored tree. The added overhead from encoding hop entries for a large portion of subscribers amplifies the network load in this case. In contrast, other variants of the *hybrid* strategy perform slightly better, as they cover a larger proportion of subscribers through stored trees, reducing the need for additional hop entries in the header.

The *tree* strategy provides the lowest network load when all subscribers can be reached using preinstalled trees, entirely eliminating the need for hop entries. However, this strategy assumes that subscriber sets remain static for each publisher – a condition that is rarely met in content-based pub/sub systems. As a result, while the *tree* strategy offers an optimal network load under static conditions, it becomes impractical for more dynamic pub/sub environments where subscriber sets frequently change.

Summary

The evaluation underscores the trade-offs between the strategies:

- *Hops*: Minimal memory usage and effective header pruning but larger headers and higher computational overhead for switches.
- *Tree*: Efficient network load and small headers but requires static subscriber sets and a large rule base.
- *Hybrid*: Flexibility to balance between hops and tree characteristics, but performance depends on the ratio of hop entries to stored tree usage.
- *Destinations*: Memory-intensive due to its n^2 rule requirement but eliminates the need for dynamic rule updates.

The choice of strategy depends on the specific requirements and constraints of the pub/sub system, such as subscriber dynamics, memory limitations, and network load tolerances.

4.4.3 Impact of Subscription Dynamics

The *hybrid* strategy combines the advantages of stateless and stateful multicast forwarding by leveraging stored trees while allowing dynamic extensions through additional distribution information encoded in packet headers. However, an important aspect of its implementation is the careful categorization of subscribers into *stable* and *dynamic* receiver sets. The dynamic nature of subscriber groups introduces significant complexity and cost to the creation and maintenance of distribution trees, particularly when subscribers change frequently. A notable overhead arises from updating flow rules within the switches. Each update consumes computational resources at the SDN controller and triggers network traffic between the controller and the switches to establish new flow rules in switch memory tables, incurring communication costs and delays.

To better understand the costs associated with these updates, we conducted experiments to measure the overhead of flow rule modifications caused by subscription changes. Specifically, we evaluated a 30:70 hybrid variant (where 30% of subscribers are dynamically encoded in the header, and 70% are integrated into a preinstalled routing tree) using three different subscriber replacement strategies:

1. *HEAD*: This method prioritizes replacing dynamic subscribers that can only be reached via switch hops.
2. *RULE*: This method starts by replacing static subscribers connected to the distribution tree, requiring tree updates for each change.

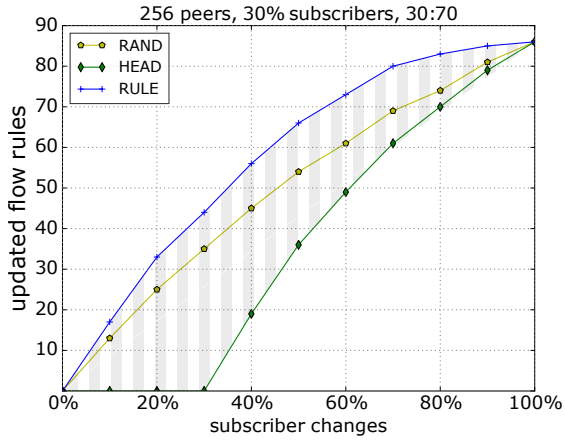


Figure 4.8: Modified flow rules on subscriber changes.

3. *RAND*: This method randomly selects subscribers for replacement, resulting in a mix of the HEAD and RULE methods.

Figure 4.8 illustrates the number of flow rule updates required across varying percentages of subscriber changes for all three strategies. The *HEAD* method represents the best-case scenario, as it initially avoids tree updates while replacing dynamic subscribers. However, once all dynamic subscribers are replaced (at a subscriber percentage of 30%), further changes necessitate updates to the tree's flow rules, causing a noticeable increase in overhead. In contrast, the *RULE* method represents the worst-case scenario, requiring the tree to be updated with every subscriber change, right from the start. The *RAND* method produces results that fall between these two extremes, reflecting an equal probability of replacing either a dynamic or a static subscriber connected to the tree.

The differences between the best- and worst-case scenarios are most significant when subscriber churn is low. As the churn rate increases, the gap narrows, and eventually, all three approaches converge to the same number of flow rule updates. This observation highlights the importance of carefully categorizing subscribers into stable and dynamic participants, especially in systems with low-to-moderate churn, to minimize the impact of flow rule updates on network performance.

4.5 Related Work

This section provides an overview of the existing literature on notification distribution strategies in content-based publish/subscribe systems. In particular, it focuses on strategies that use preconfigured multicast paths or trees. The

primary objective of these efforts is to address the challenges associated with efficient notification distribution in dynamic network environments by minimizing the state to be managed in network devices.

4.5.1 Xcast

Boivie et al. propose XCAST (eXplicit multiCAST) [15] as a solution designed to address multicast communication by embedding an explicit list of destination addresses in the packet header. Analogous to our *destinations* strategy, the sender embeds a list of destination addresses in the packet header. As the packet traverses each router within the network, the header is parsed, the destinations are partitioned based on the next hop, and a copy of the packet is sent to each respective next hop. In particular, the list of destinations is dynamically pruned for each copy, ensuring that each copy contains only the destinations intended for the next hop.

However, XCAST (as well as the *destinations* strategy) has certain scalability limitations. The approach of explicitly listing destinations imposes constraints on the size of multicast packet headers, limiting the system's ability to accommodate a large number of receivers. This limitation affects the group size that can be encoded in the packet header and the payload size of multicast packets, requiring a trade-off between multicast group size and available payload capacity.

XCAST also provides a fallback mechanism for unicast communication. In scenarios where there is only one destination for a particular next hop, the XCAST packet can be transmitted as a standard unicast packet, known as X2U. To facilitate this IP-based tunneling, the XCAST header containing the destination list is preceded by an IP header containing a pseudo-multicast address. While this provides compatibility with non-SDN capable switches, it introduces an additional header and therefore reduces the maximum achievable payload size.

4.5.2 Bit Index Explicit Replication (BIER)

Wijnands et al. [122] present a multicast approach known as BIER (Bit Index Explicit Replication), which encodes a sequence of bits in the header of a packet. Each bit in this sequence represents a particular destination. When a bit is activated, the packet is forwarded and, if necessary, replicated to reach the designated destination. Analogous to our *destinations* strategy, switches within the network maintain forwarding rules for each potential target, each represented by a bit in the bitmask. These rules allow switches to forward copies of incoming packets to downstream nodes.

In alignment with our methodologies, switches compute a new header for each copy, removing previously served destinations from the packet header. This pruning process is essential to avoid duplicates and loops.

To tackle scalability challenges and increase the number of receiver groups, the authors propose the use of a *Set Identifier (SI)* field. This SI field distinguishes between different sets of bit strings, increasing the addressable destinations. However, each packet can carry only one SI. Consequently, if receivers belong to different sets, the sender must send multiple copies of the packet to address them all.

To create clones and reprocess packets, BIER, as well as our approach, relies on packet recirculation. Merling et al. [82] provides a P4 implementation of BIER for *Tofino* switches and discuss the effect of packet loss due to many recirculations. The authors address the challenge by proposing a solution that configures physical ports as loopback ports (*recirculation ports*), which increases recirculation capacity. However, once physical ports are configured as recirculation ports, they are no longer available for regular forwarding purposes.

Eckert et al. [38] propose a new semantic for bit positions in the packet label, called BIER-TE, to indicate adjacencies of the network topology, i.e. links to neighboring routers. Each bit identifies one or more adjacencies and is either activated or deactivated to be used for forwarding or not. When a switch receives a packet, it identifies the activated bit positions in the BIER-TE bit string and creates packet copies for the activated adjacencies. In this way, the complete distribution tree of a message can be encoded, similar to our stateless hop-based encoding. Before transmitting the copies created for the activated adjacencies, the switch clears (as in the original approach) the bit positions in the packet label belonging to its own adjacencies to avoid loops and duplicates.

Although both BIER-TE and BIER can coexist in the same network, a packet is routed either end-to-end using BIER or hop-wise using BIER-TE. Our *hybrid* approach seamlessly combines stored trees and header-encoded supplementary routing information to *extend* stored trees for individual messages as required.

4.5.3 Bloom Filters

Bloom filters offer a probabilistic approach with versatile applications in distributed systems for message dissemination [110]. This data structure provides a resource-efficient method for inserting and testing membership of elements within a set. The core components of a Bloom filter are an initial bit array of m bits, all initially set to 0, and a set of $k << m$ hash functions that uniformly map elements to specific positions in the bit array, ranging from 0 to m–1. When an element is inserted, each hash function computes positions in the array that are then set to 1. To test an element's presence in the set, the same hash functions are employed, and the relevant array positions are examined. If at least one of the checked bits is 0, the element is undoubtedly not in the set. Conversely, if all of the checked bits are set, it is considered to be part of the set, except in very rare cases where other elements have coincidentally set all of the checked bits, leading to a false positive.

In the realm of publish/subscribe systems, XSiena [62] uses Bloom filters for content-based matching at intermediate brokers within a notification's distribution tree. Publishers encode all subscription predicates matching a notification into a Bloom filter attached to the notification. This strategy prevents the costly reevaluation of predicates at intermediate brokers, thereby speeding up the routing of the notification. Nonetheless, forwarding decisions are still made at the application layer, resulting in significant delays.

LIPSIN

Lipsin [66] pioneered the use of Bloom filters in the network layer. Within the proposed routing protocol, each data packet contains a Bloom filter in its header. This filter identifies the network links, represented as hashed link IDs, through which the packet should be routed. A sender computes the Bloom filter, which is inserted into the packet header before distribution.

While Lipsin initially required custom switching hardware (i.e., NetFPGA based deployments), Macapuna et al. [78] provide an OpenFlow-based software-defined network implementation that uses packet-header embedded Bloom filters.

The use of these in-packet Bloom filters [99] for source routing enables not only the encoding of a network path to a single receiver, but also an entire distribution tree for multiple subscribers. Routers along the delivery path evaluate the Bloom filter, determining the correct output links for packet forwarding through bitwise ANDing between the filter and each output link ID. However, due to the probabilistic nature of this approach, there is a risk that the packet may be unintentionally routed through a wrong link, causing additional network traffic and false positives. This also introduces the possibility of forwarding cycles, necessitating countermeasures.

Publish/Subscribe with OpenFlow

Helge et al. [95] further extend the use of OpenFlow-enabled Ethernet switches for notification distribution in a content-based pub/sub system. Publishers integrate a Bloom filter into each transmitted notification. This Bloom filter, in conjunction with rules installed in the switches, controls how messages are routed from the publisher to the subscribers. An SDN-controller, which intercepts subscriptions and advertisements, deploys these rules.

However, as with all Bloom-based approaches mentioned above, the probabilistic nature of Bloom filters can lead to false positives, i.e. unintended messages being delivered. To address this issue, subscribers check the content of received notifications against their own subscriptions in a post-processing step. Notifications are only delivered to the final application when a match is confirmed; otherwise, they are silently discarded. To further limit false positives, forwarding information is distributed among multiple Bloom filters, each associated

with an individual notification copy. In addition, the approach incorporates optimizations to reduce the volume of generated forwarding rules to fit within the limited size of the switch memory. These optimizations exploit similarities in the content-based filter expressions of subscriptions and advertisements, as well as commonalities in the bit patterns of generated forwarding rules.

4.6 Summary

This chapter introduced advanced notification distribution strategies that leverage preinstalled forwarding rules within the switch infrastructure, augmented by dynamic delivery information encoded in packet headers. Additionally, this chapter proposed a *hybrid* strategy that balances flexibility and message size optimization by combining the strengths of these methods. The performance evaluation of these strategies demonstrate their ability to reduce message size while maintaining adaptability for dynamic recipient groups. However, the evaluation also highlights that the costs of adapting the rule base increase as subscription patterns evolve.

Publisher-rooted delivery trees stored in the infrastructure significantly reduce the amount of delivery information required in message headers, but limit the flexibility to create custom delivery trees for individual messages. To address this, we developed the hybrid strategy as a compromise. This approach trims branches leading to infrequently contacted subscribers and allows branch extensions when necessary to construct a complete notification-specific delivery tree. Using this approach, frequently contacted subscribers are addressed with a single tag encoded in the header, while additional routing information in the form of hop labels is appended only for less frequently addressed recipients. As demonstrated in the evaluation, offloading heavily used parts of the distribution tree substantially reduces the amount of state that needs to be stored in message headers. Such pre-computation and storage of distribution trees for stable subscriber groups also reduces network load.

Despite its benefits, the hybrid strategy has a notable limitation: it relies on a single stored tree per publisher to route notifications. This restricts its applicability to scenarios requiring more diverse or flexible routing solutions. The next chapter introduces an enhancement to overcome this constraint – universally applicable stored trees that are both sender- and receiver-agnostic. These generalized trees enable notifications to be propagated through multiple preinstalled trees, improving scalability and removing the dependency on a single publisher-centric tree. This development broadens the utility of stored flow rules while addressing the shortcomings of the strategies discussed here.

Chapter 5

Stitching Forwarding Trees

Contents

5.1 Introduction

The preceding chapter underscored the challenges of using (stored flow rules to map) static distribution trees in dynamic pub/sub systems, particularly due to their limited flexibility and scalability. In such systems, where publishers and subscribers dynamically interact, the rigid nature of stored trees restricts their adaptability to changing notification contents and/or subscriptions. This hinders efficient notification delivery to a large and dynamic client base.

To address these limitations, this chapter introduces notification delivery strategies based on *virtual trees (VTs)*. VTs function as partial notification delivery trees that can be combined to form complete notification-specific delivery trees. Publishers gain the ability to combine multiple VTs and attach additional routing information within message packet headers to tailor VTs to specific notification delivery requirements. Unlike traditional approaches that rely on static delivery trees, VTs offer adaptability in combining stored routing rules with dynamically encoded header information, even as publisher and subscriber dynamics evolve over time.

To facilitate this level of flexibility, we introduce different types of labels that reference VTs and augment them with additional routing information tailored to individual delivery needs. A key contribution of this chapter is the introduction of an algorithm designed to seamlessly combine preinstalled VTs with additional routing information encoded in the message header, using the different label types. We also outline several strategies for installing VTs in datacenter networks, using insights from network topology, application-specific requirements, and statistical analysis to optimize the distribution of notifications. These strategies are essential for calculating efficient VTs in terms of sender-independent usage to be applicable to multiple publishers, and in terms of their combinability with other VTs to form complete trees.

The subsequent sections are structured as follows: Section 5.2 describes the functionality of VTs and introduces the associated label types, which are instrumental in combining, extending, or truncating stored trees to form a full delivery tree. We implement a P4 program that uses these label types to perform routing based on preinstalled VTs and additional encoded header information. We also present a greedy algorithm for encoding and optimizing distribution trees using preinstalled virtual trees. Section 5.3 delves into a datacenter network topology analysis and outlines diverse strategies for installing promising VTs based on varying levels of knowledge about the pub/sub network. This section also introduces a skeleton class that streamlines the implementation of various VT installation schemes. Section 5.4 evaluates the novel tree installation strategies in an emulated network based on a specific datacenter topology, focusing on bandwidth requirements for delivery and generated message header size. Finally, Section 5.5 explores related work, contextualizing the proposed strategies within existing research, whereas Section 5.6 concludes the chapter by summarizing key findings and contributions.

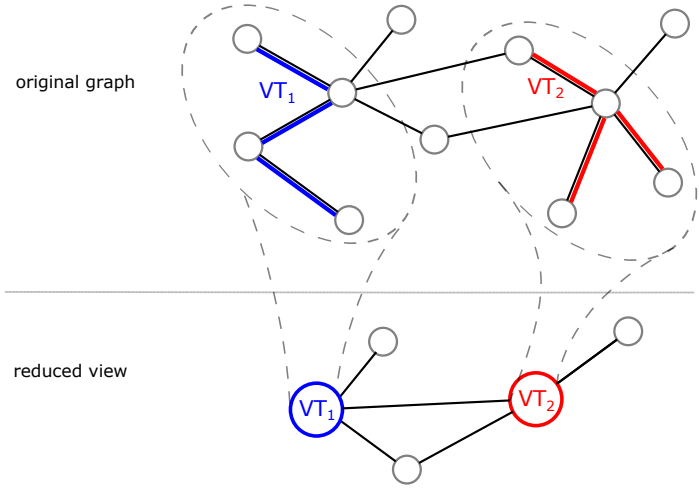


Figure 5.1: Conceptional visualization of virtual trees.

5.2 Virtual Trees

Virtual trees provide an innovative method of message distribution using reusable shared substructures. Unlike *core-based trees (CBTs)* [7, 6], which require group members to join and connect through a central core node, VTs operate without such a rendezvous point. Senders or receivers do not need to explicitly join a tree. Instead, for each message, the publisher dynamically decides which combinations of VTs to use for delivery.

5.2.1 Characteristics

Virtual trees are versatile distribution structures with a high degree of flexibility due to following inherent properties:

- *Bidirectional routing state:* Similar to CBTs, VTs implement a bidirectional routing state. Each switch on a VT that receives a packet forwards it through all interfaces associated with the VT, except the one from which the packet was received. This allows that each node in a VT can act as both a source and a sink.
- *Modular structure:* VTs are not designed to represent full distribution trees. Instead, they serve as modular components embodying partial delivery trees that can be combined to construct complete delivery structures. This modular approach reduces the amount of flow rules that need to be stored for different scenarios. Publishers have the flexibility to employ VTs independently for targeted delivery, or to combine them into larger delivery tree structures for broader distribution.

- *Strategic deployment and composition:* Virtual trees need to be strategically placed along frequently traversed routes, particularly common paths shared by multiple recipients. Each VT is identified by a unique label that encapsulates all its edges, allowing publishers to distinguish between multiple preinstalled VTs. By encoding these labels into the header of a notification packet, publishers can effectively apply the corresponding partial trees. Multiple VTs can be combined by including a sequence of VT labels within the packet header.

5.2.2 Example Scenario

Figure 5.1 depicts a network containing two virtual trees: VT_1 (red) with four nodes and VT_2 (blue) with three nodes, while two additional nodes remain outside these virtual trees. In this example, the virtual trees are disjoint, leaving a gap between them. To efficiently deliver notification messages to subscribers across both trees, it is advantageous to merge the two virtual trees into a single, larger delivery tree. This requires bridging the gap between the independent virtual trees to establish a unified distribution structure.

The gap between VT_1 and VT_2 can be bridged in two distinct ways: (i) adding a single edge directly between VT_1 and VT_2, as shown by the top link in the figure, or (ii) adding two edges that pass through an intermediate node, forming an indirect connection between VT_1 and VT_2 (illustrated by the bottom link in the figure). Importantly, both bridging methods cannot be applied simultaneously to construct a delivery tree, as this would create a cycle in the resulting structure, thereby violating the fundamental tree property.

5.2.3 Challenges of Preinstalled Virtual Trees

Not all branches within a VT may be relevant for every message, potentially leading to wasted bandwidth and false positives. Addressing this issue requires a mechanism for pruning VTs to prevent unnecessary data transmission to uninterested subscribers. However, pruning may not be worthwhile if the VTs only cover a small part of the required delivery tree. In this case, it would be more effective to encode the routes by identifying each edge individually, hop by hop, in a source-routed manner. Conversely, a VT might lack sufficient branches, hindering message delivery to certain recipients. Similar to bridging separate VTs, mechanisms are needed to extend a VT with additional edges to ensure comprehensive coverage.

In summary, preinstalled VTs present three challenges:

1. *Unnecessary branches*: VTs may contain branches leading to uninterested receivers, demanding the removal of unnecessary branches to avoid false positives.

2. *Insufficient coverage*: Some recipients might not be reachable with existing VT structures, requiring the addition of edges to reach recipients not covered by preinstalled VTs.
3. *Possible gaps*: Combining VTs might be hindered by gaps between them or create unintended paths violating the tree structure. necessitating effective solutions to combine separate VTs while preserving the tree structure during delivery tree construction.

In summary, virtual trees offer a flexible and efficient approach to message distribution. Their modular design and strategic deployment can significantly optimize message delivery, but challenges such as unnecessary branches, insufficient coverage, and possible gaps must be addressed to fully leverage their potential.

5.2.4 Extending and Pruning Virtual Trees

Although VTs form the basis of efficient message delivery, they may not always align perfectly with specific notification requirements due to the above-outlined challenges. To address these issues, we introduce two additional label types that provide flexible control over message distribution: *hop* labels and *stop* labels.

Hop label. A hop label specifies a singular network link for explicit forwarding. In contrast to virtual trees, hop labels enable the creation of unique message routes. They are valuable when no virtual trees are available within a subnetwork or when the required level of control over the routing paths goes beyond what preinstalled trees can cost-effectively provide. Specifically, hop labels facilitate the delivery of notifications to infrequently served subscribers not covered by stored virtual trees, ensuring delivery to all intended recipients. Moreover, sequences of hops allow for the connection of distant VTs, supporting the combination of VTs and communication across different network segments.

Stop label. A stop label serves as a counterpart to a hop label and identifies a connection within a VT where forwarding is explicitly halted. Stop labels enable the removal of irrelevant subtrees within virtual trees, eliminating unnecessary branches and reducing the extent of the VT. This functionality is useful when only certain parts of a VT are required for notification delivery. By disabling unnecessary VT subtrees using stop labels, unnecessary traffic can be avoided in areas of the network that have no interested subscribers. In addition, stop labels can be used to eliminate potential cycles resulting from the overlap of two VTs by cutting off redundant paths, thus preserving the tree property.

The synergy of *tree*, *stop* and *hop* labels enables the construction of compact, notification-specific delivery trees and targeted control of message delivery. Stop labels prune unnecessary branches, whereas hop labels enable fine-grained tree expansion. Together, these labels provide a powerful mechanism for the customized delivery of notifications to specific recipients.

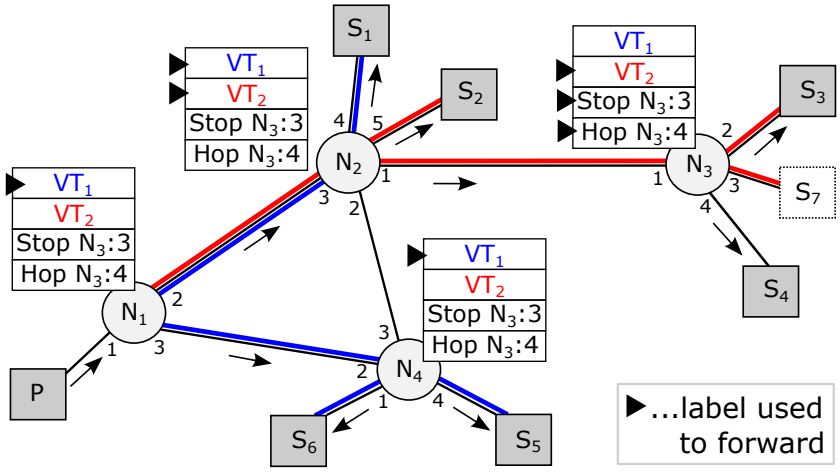


Figure 5.2: Notification distribution using two virtual trees.

5.2.5 Illustration of Notification Dissemination

This section provides two examples of how the switch infrastructure processes notification delivery trees to deliver notifications to interested recipients.

Overlapping Trees

Figure 5.2 shows an example of notification delivery via two virtual trees to subscribers S_1, ..., S_6. Publisher P encodes VT_1 and VT_2 as tree labels into the notification message header, along with a stop and hop label as additional distribution information. Each intermediary switch (N_1, ..., N_4) receives and successively processes the labels of the header stack, starting with the label for VT_1. This first VT includes all edges marked in blue and interconnects nodes N_1, N_2, N_4, S_1, S_5, and S_6. A switch belonging to VT_1 stores forwarding rules for each branch of VT_1. For instance, switch N_1 replicates the packet for port 2 and 3 to forward it towards switch N_2 and N_4, respectively. For each output port of the switch, a corresponding bit in a bitmask field is managed in the switch memory. This bit is activated when a clone packet has been sent via respective ports. The bitmask field is used to detect ports that have already been served to prevent multiple clones from being sent out over the same port. For example, Switch N_1 for the label of VT_2 does not initiate another forwarding via port 2 because the label of VT_1 has already set the corresponding bit in the bitmask field.

Both, incoming packet headers and outgoing clone headers are recirculated on a switch until all labels have been examined. In case of outgoing clones, this post-processing is necessary to detect stop labels referring to the clone. For example, on switch N_3, the label VT_2 generates a clone to initiate forwarding

over port 3. However, subsequent scanning of the clone header detects the stop label that identifies the same port as the clone is created for. Therefore, the clone is discarded and not forwarded towards subscriber S_7. In other words, the stop label truncates an edge of the red tree because there is no interested subscriber beyond it. A hop label, conversely, extends the distribution tree by one edge. In this example, the hop label at the end of the header stack causes the packet to be forwarded from N_3 to S_4.

While hop and stop labels are removed from the stack after application, VT labels are retained. A hop or stop label is dedicated to a switch port and is only used once, whereas an applied VT label may still be relevant to downstream switches because the switch cannot know if the VT extends to downstream switches. Moreover, hop and stop labels may be forwarded to branches where they are never applied, as is the case with switch N_4. To avoid forwarding unnecessary distribution information, additional subtree separators would have to be encoded, thereby increasing the initial header length at the publisher. However, we do not do this in our VT-based distribution strategies in favour of keeping the initial header length as short as possible.

Trees bridged by Hops

Figure 5.3 shows another example with two distant virtual trees which are connected by a bridging hop entry in the header stack to form a full distribution tree. In this example, publisher P sends a notification to subscribers $S_1, \ldots, S_5$ by using virtual trees VT_1 and VT_2. VT_1 comprises the blue edges and covers nodes N_1, N_2, N_5, S_1 and S_2. VT_2 comprises the red edges and covers nodes N_3, N_4, S_3 and S_4. The publisher encodes the labels of the two VTs, along with a stop and two hop labels into the packet header and sends the packet to neighboring switch N_1.

Each switch belonging to VT_1 or VT_2 forwards the packet through its respective output ports. For instance, switch N_1 applies the flow rules for VT_1 and forwards the packet through ports 2 and 3 to switches N_2 and N_5 respectively. Switch N_2 uses the same VT label to deliver the packet to subscribers S_1 and S_2.

For parts of the distribution tree that are not covered by useful virtual trees, the publisher relies on additional hop entries to complete the distribution tree. The header stack of this example contains two hop entries: $N_2 : 1$ and $N_5 : 2$. $N_2 : 1$ is evaluated by switch N_2 and bridges an edge to connect VT_1 to VT_2, and $N_5 : 2$ triggers a forwarding on switch N_5 over port 2 to subscriber S_5. The publisher also encodes an additional stop entry that truncates an edge of VT_2, causing switch N_3 to block forwarding of the packet to subscriber S_6.

After receiving the packet, each switch skips unknown VT labels as well as unprocessed stop and hop labels, until it finds a known label. For instance, switch N_5 will skip four entries before applying the intended one. This functionality is required to obtain potentially relevant distribution information. Similar to

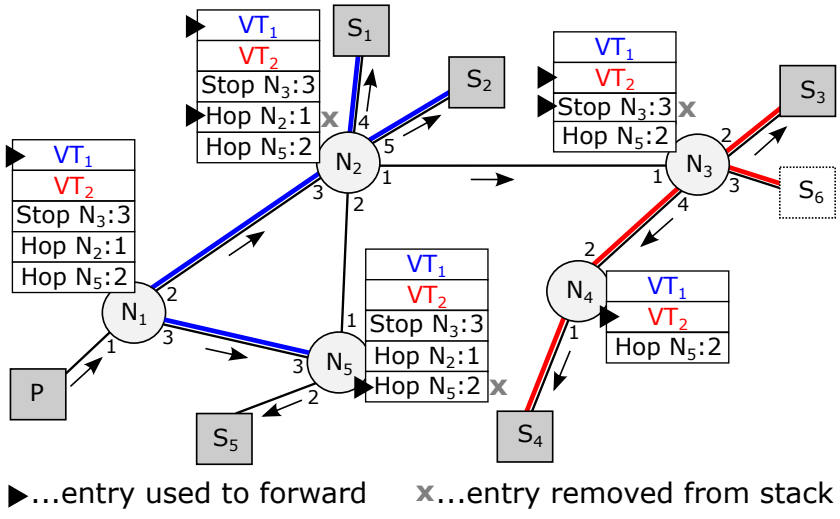


Figure 5.3: Notification distribution by bridging two virtual trees.

the first example, processed VT labels remain in the stack after application because downstream switches may still need them, whereas hop and stop labels are discarded after application because they only refer to a single network edge and are not relevant beyond that point.

5.2.6 P4$_{16}$ Implementation

We developed a P4$_{16}$ program capable of distinguishing between three label types – *tree*, *hop*, and *stop* – and applying them to perform multicasting. The program is designed to manage a header stack that contains these label type, coordinating their corresponding forwarding operations.

The program is structured into four key components: (i) *header field* definitions, (ii) *parser*, (iii) *ingress*, and (iv) *egress control block*. Below, we provide a detailed overview of each component and its role within the program.

Header, Metadata and Parser

In the P4$_{16}$ program, we define a composite header structure consisting of three sub-headers, as shown in Listing 5.1. The first sub-header is the Ethernet header (`Ethernet_h`), which includes the MAC destination and source addresses, along with an EtherType field. Following the Ethernet header is a single 16-bit field, `Size_h`, which specifies the number of elements in the header stack. This field ensures that the program can accurately process the dynamic stack length, accommodating varying numbers of labels. The core of the stack lies in its elements, defined by the `Entry_h` sub-header. These elements encode sequences of

```
1  header Ethernet_h {
2    bit<48> dstAddr;
3    bit<48> srcAddr;
4    bit<16> etherType; // type 9999 identifies our custom header stack
5  }
6
7  header Size_h {
8    bit<16> num_entries; // maintains number of stack entries
9  }
10
11 header Entry_h {
12   bit<16> ident; // id for virtual trees, hop or stop label
13 }
14
15 struct header_fields {
16   Ethernet_h  ethernet;
17   Size_h front; // stack length
18   Entry_h[MAX_SIZE] stack; // stack of labels
19 }
```

Listing 5.1: Header field definitions.

hop, *stop*, or *tree* labels. Each label within the stack is represented as a 16-bit integer. The first two bits of this integer identify the type of label, distinguishing between the three label types. The remaining 14 bits are used to identify specific hops, stops, or trees within the network. This design enables the program to support network topologies with up to $2^{14} = 16,384$ edges and an equivalent number of virtual trees – sufficient for most networks. For networks with larger edge sets, however, the size of the stack entry field would need to be increased to accommodate the additional identifiers.

To maintain state information across multiple control flow passes, the implementation introduces metadata fields, as depicted in Listing 5.2. Among these metadata fields, the most important are: (i) `stack_index`, (ii) `remaining_ports`, and (iii) `served_ports`. The `stack_index` metadata field points to the current position in the header stack, indicating the current label to be interpreted. After each pass through the control flow, `stack_index` advances to the subsequent entry in the stack, ensuring sequential processing of labels. The `remaining_ports` metadata field receives forwarding commands from the header stack labels and keeps track of the output ports through which packets are to be forwarded but have not yet been served, while `served_ports` keeps track of the ports that have already been processed. The P4 program processes the header stack in the ingress and egress pipeline using these metadata fields. The implementation of these two blocks is explained below, starting with the ingress pipeline.

Listing 5.3 lists the states of the parser. The parser is built on the header definition. In state `start` it first reads the Ethernet fields of an incoming packet and checks the `etherType` field. If `etherType` equals value `9999`, the parser proceeds to state `parse_stack`; otherwise the packet is rejected, i.e., dropped by

```
1  struct metadata {
2    bit<16>  stack_index; // pointer to current header stack label
3    bit<16>  remaining_entries; // outstanding entries to parse
4    bit<64>  remaining_ports; // ports to forward the packet for
5    bit<64>  served_ports; // ports that already received a clone
6
7    bit<9>   port; // caches forwarding information of a label
8    bit<9>   ingress_port; // caches input port of incomming packet
9    bit<9>   clone_port; // caches port of a cloned packet
10   bit<1>   remove_flag; // marks entries for deletion
11 }
```

Listing 5.2: Metadata fields used for packet processing.

```
1  parser ParserImpl(packet_in packet, out headers hdr,
2       inout metadata meta, inout standard_metadata_t std_metadata) {
3
4    state start {
5      packet.extract(hdr.ethernet);
6      transition select(hdr.ethernet.etherType) {
7        16w0x9999: parse_stack; // type 9999 identifies header stack
8        default: reject; // wrong EtherType
9      }
10   }
11
12   state parse_stack {
13     // extract number of stack entries and save it as metadata field
14     packet.extract(hdr.front);
15     meta.remaining_entries = hdr.front.num_entries;
16     transition select(meta.remaining_entries) {
17       16w0x0: reject; // empty header stack
18       default: parse_entry; // parse first stack entry
19     }
20   }
21
22   state parse_entry {
23     packet.extract(hdr.entry.next); // extract single entry
24     // update remaining number of entries to parse
25     meta.remaining_entries = meta.remaining_entries - 1;
26     transition select(meta.remaining_entries) {
27       16w0x0: accept; // no more entrie on stack; goto ingress block
28       default: parse_entry; // at least one more entry on stack
29     }
30   }
31 }
```

Listing 5.3: Parser used for extracting header stack labels.

the switch. State `parse_stack` extracts the `num_entries` field from the header stack, indicating the number of entries on the header stack. If no entries are found, the packet is rejected. Otherwise, there is at least one entry on the stack and the parser enters state `parse_entry`, which calls itself recursively to iteratively extract all stack elements. After successfully parsing the entire header stack, the parser enters the final state `accept`, passing the extracted labels to the ingress pipeline, which is explained next.

Ingress Pipeline

In the ingress pipeline, as shown in Listing 5.4, the program first retrieves the next element from the stack (line 4). It then determines whether the received packet is an original or a forwarding copy generated in previous control flow passes. If it is the original packet, lines 8 to 41 are executed.

These lines first store the ingress port in a metadata field (`ingress_port`) if it is a newly received packet (line 13). This step is necessary to prevent possible packet copies for this port during subsequent label processing. Next, the switch retrieves forwarding information either from a cached bitmask, possibly set by previously interpreted labels, or from the next label in the stack (lines 16, 17).

The program first checks if the metadata bitmask field (`remaining_ports`) contains 1-bits, indicating that the corresponding output ports require packet copies (line 16). If this condition is met, and if the bitmask field does not exclusively contain zero-bits, at least one copy must be generated. Such a bit might have been set by label interpretations during previous control flow passes.

If the bitmask is empty (as in the first control flow pass), the program extracts the forwarding information from the next label in the header stack. However, at this stage, the program does not know whether the label represents a hop, a tree, or a stop. Therefore, it tests the different label types through two match-action tables: `check_hop` and `check_tree` (line 17). The `check_hop` table attempts to interpret the stack entry as a hop label. On a hit, the matching rule sets a bit in `remaining_ports`. On a miss, the `check_tree` table tries to interpret the same entry as a tree label. If a matching rule is found, the switch is on a branch of the identified virtual tree, and the action of the rule updates `remaining_ports`. This update sets multiple bits within the bitmask depending on which ports the tree covers on that switch. If neither a hop nor a virtual tree rule applies to the current label, the label is either destined for another switch or switches, or is a stop label. The bitmask remains unchanged in both cases.

Subsequently, the `extract_port` table (line 20) tries to identify the next 1-bit in `remaining_ports`. Upon success, it identifies the next output port. If the possibility that another clone has been created for the port is excluded, a packet copy is created for this port. Verifying this exclusion criterion is necessary because it is possible for two or more trees to share common edges and be used simultaneously for encoding a distribution tree. Without this check, multiple

```
control ingress(inout headers hdr, inout metadata meta,
    inout standard_metadata_t std_metadata) { [...]
 apply {
  act_read_entry(); // extract next entry from header stack

  // distinquish betweeen clone and original
  if (meta.clone_port == 0) { // only clones get an output port
    /*** branch for orginal packet ***/

    // no copies are sent over VT branches identical to ingress port
    if (std_metadata.ingress_port != 0)
      // store input port for multiple control flow passes
      meta.ingress_port = std_metadata.ingress_port;

    // get next hop/VT label forwarding information if cache empty
    if (meta.remaining_ports > 0 // bitmask cache empty?
          || check_hop.apply().hit || check_tree.apply().hit) {

      // extract next output port to serve (from bitmask cache)
      if (extract_port.apply().hit) {

        // compute updated bitmask for served ports
        bit<64> bit_position = (bit<64>)1 << (bit<8>)(meta.port-1);
        meta.remaining_ports = meta.remaining_ports - bit_position;
        bit<64> new_ports = meta.served_ports | bit_position;

        // clone packet for egress, avoiding clones for ingress
        // port or ports that were already served by another copy
        if (meta.served_ports != new_ports
              && meta.ingress_port != meta.port) {
          act_clone_pkt();

          // update bitmask containing all applied ports
          meta.served_ports = new_ports;
        }
      }
    } else {
      // drop original packet if header stack and bitmask is empty
      if (hdr.front.num_entry == 0 && meta.remaining_ports == 0)
        mark_to_drop(std_metadata);
    }
  } else {
    /***  branch for cloned packets ***/

    // drop clone if stop label for clone's egress port is found
    if (check_stop.apply().hit)
      mark_to_drop(std_metadata);
    // otherwise restore ouput port of clone
    else
      std_metadata.egress_spec = meta.clone_port;
  }
}}
```

Listing 5.4: Ingress control block for header stack processing.

packet copies could be sent over overlapping output ports from simultaneously applied trees. To prevent this, another bitmask (`served_ports`) is maintained to track the served ports. It maps served and unserved ports to 1-bits and 0-bits respectively.

A clone for port p can only be created if p's bit in `served_ports` is not already set. To prevent triggering another clone operation for p in subsequent iterations, the bit position of p is calculated (line 23), and the corresponding bit in `remaining_ports` is cleared (line 24). Then, a helper field representing the served ports so far is created (line 25), incorporating p's bit position. It is used to check if p has already been served (line 29). If it is confirmed that p has not already been served by another copy and p is not the ingress port (line 30), the clone operation is executed (line 31), and the list of served ports is updated (line 34). The clone operation has the effect of putting two packets into the egress pipeline – the original and the clone.

However, cloned packets are not sent out immediately, but are recirculated and thus pass through the ingress pipeline again to scan the remaining stack elements for stop labels. The ingress block processes duplicated packets between lines 43 and 51. The `check_stop` table (line 46) contains a stop rule for each port on the switch, verifying whether the current entry is a stop label matching the egress port of the current clone. If it matches, the clone is flagged for drop (line 47) to prevent it from being propagated beyond the edge indicated by the stop label. Otherwise, the clone's egress port is restored from the metadata (line 50). While clones are only discarded if there is a corresponding stop label in the header, the original packet is always dropped (line 40) after all entries in the header stack have been seen.

Upon completing the ingress pipeline, the packet undergoes post-processing in the egress control block to potentially remove the currently processed label from the header stack. This post-processing is performed for every entry on the stack.

Egress Pipeline

In the egress pipeline, depicted in Listing 5.5, the program is tasked with removing processed hop and stop labels from the header stack. This post-processing applies to both the original packet instance and any cloned instances. The egress pipeline exclusively handles packet instances whose header stacks are not empty, i.e., those that have not been marked for drop in the ingress pipeline (line 5).

The removal process uses the `remove_flag` as a metadata field. The flag is activated by either the `check_hop` or `check_stop` table within the ingress pipeline when the current entry is identified as a hop or stop label, respectively. If a check reveals that the flag is set (line 12), the hop or stop label is removed from the stack (line 13), and the header element counter is decremented (line 14).

On the other hand, virtual tree labels and unknown hop and stop labels intended for other switches are kept. These labels are skipped by incrementing

```
1  control egress(inout headers hdr, inout metadata meta,
2      inout standard_metadata_t std_metadata) { [...]
3    apply {
4      // only post-process packet if not marked for drop
5      if (std_metadata.egress_port != DROP) {
6
7        // store egress port of clone in metadata
8        if (std_metadata.instance_type == INGRESS_CLONE)
9          meta.clone_port = std_metadata.egress_port;
10
11       // remove applied hop or stop label and update stack length
12       if (meta.remove_flag == 1) {
13         remove_entry.apply();
14         hdr.front.num_entries = hdr.front.num_entries - 1;
15       }
16       // otherwise increase stack index for pointing to next label
17       else if (meta.remaining_ports == 0)
18         meta.stack_index = meta.stack_index + 1;
19
20       // recirculate as long as there are remaining stack labels
21       if (hdr.front.num_entries > meta.stack_index)
22         act_recirculate();
23     }
24   }
25 }
```

Listing 5.5: Egress control block for header stack processing.

the `stack_index` metadata field, which serves as a pointer to the current label (line 18). Finally, if there are more labels on the stack, the original or cloned instances are recirculated through the ingress and egress pipelines (line 22). This means that the packet "*will begin processing again with the parser, with the contents as they are created by the deparser*" [93]. To hold the output port for a clone during recirculation, we store the clone's output port in an additional metadata field (line 9), which is restored on the next control flow pass within the ingress pipeline (see line 50 in Listing 5.4). This caching step is necessary because the egress port specified by `standard_metadata.egress_spec` is cleared by BMv2 during recirculation.

5.2.7 Encoding Scheme

So far, we have introduced the concept of label types and described a P4 program designed to facilitate the dissemination of notification messages across a network. This approach leverages stored virtual trees (VTs) and a publisher-encoded header stack to efficiently multicast messages. However, a key challenge remains in the construction of a complete distribution tree for each notification message, encoded as a header stack. This task involves assembling an individualized distribution tree by combining relevant VTs, bridging gaps with hop labels, and pruning redundant branches with stop labels to prevent cycles and

unnecessary forwarding.

Not all preinstalled virtual trees are equally useful for constructing a spanning distribution tree. Only a carefully chosen subset of VTs contributes meaningfully to building the notification-specific delivery tree. To address this challenge, we introduce an algorithm that identifies the most relevant subset of VTs and adapts the virtual tree composition to the intended recipient set of the notification. The algorithm not only selects the relevant VTs but also augments them with strategically placed hop and stop labels, ensuring that uncovered network regions are bridged and unnecessary branches are pruned.

Encoding Overview

To construct concise distribution trees for each published notification, publishers employ an optimization process that begins with an initial *hop-minimal* tree. The creation of this initial tree involves tackling the well-known NP-hard problem of constructing a *Minimum Steiner Tree (MST)* [47]. Despite the computational complexity, various Steiner tree heuristics and shortest-path tree algorithms strike a balance between accuracy and efficiency [103], even in large real-world networks.

The algorithm greedily replaces as many links of a distribution tree as possible with preinstalled virtual trees, providing a more compact stack encoding that demands less header space. It starts by encoding the distribution tree as a set of physical edges, where each edge requires a single hop label in the header stack. These physical edges (referred to via hop labels) are subsequently minimized by substituting them with edges from the available VTs in the network.

Therefore, the algorithm then iteratively attempts to incorporate each VT installed in the network and evaluates which individual VT addition contributes the most to reducing the required number of header stack labels. The VT that maximizes the gain is then included in a result set along with any additional routing information required to eliminate unwanted branches of the VT (stop labels) or to bridge gaps not covered by the VTs (hop labels). This VT selection and result set construction continues until no further improvements are possible.

Encoding Details

The detailed functionality of the greedy encoding function is illustrated in Algorithm 5.1. The function takes as input arguments the network graph G, the set of terminal nodes N containing the publisher and the interested subscribers, and the stored virtual trees T_{virt} within the infrastructure.

First, empty sets are initialized for the (i) virtual trees T, (ii) stops E^- and (iii) hops E^+. The algorithm then begins by computing an initial tree using a Steiner heuristic (line 3). Since no virtual tree has been used yet, all its edges

Algorithm 5.1 Greedy encoding of distribution trees.

```
1:  procedure ENCODE( G, N, T_virt )
2:      T ← {}, E_phys ← edges( G ), E_virt ← {}
3:      E+ ← steiner( E_phys, N ), E− ← {}, C ← |E+|
4:      for all t ∈ sort( T_virt ) do
5:          T' ← T ∪ {t}, E'_virt = E_virt ∪ edges( t )
6:          E_dist ← steiner( E_phys ∪ E'_virt, N )
7:          E+_dist ← E_dist ∩ E_phys
8:          E−_dist ← {e ∈ E'_virt \ E_dist | ∃e' ∈ E_dist : adjacent( e, e' )}
9:          C' ← |E+_dist| + |E−_dist| + |T'|
10:         if C' < C then
11:             T ← T', E+ ← E+_dist, E− ← E−_dist
12:             C ← C', E_virt ← E'_virt
13:     return T, E+, E−
```

E^+ must be encoded as source-routed hops, each requiring a header entry. The number of these entries defines the initial cost C, which serves as a benchmark for subsequent efforts to reduce it. These costs represent the aggregated edges of the delivery tree, assuming initial weights of cost 1 for each edge in the network.

Subsequently, the algorithm iteratively optimizes the initial tree by integrating virtual trees and altering the initial edge weights. It systematically loops through the available virtual trees T_{virt} and evaluates each virtual tree t to assess its potential in decreasing the current cost (lines 4 to 12).

Within the loop, current tree t is evaluated by incorporating its links as virtual edges E_{virt} into a temporary graph E'_{virt} (line 5). These virtual edges have zero weight compared to the real edge weights of the physical edges E_{phys}. To validate t, a new distribution tree E_{dist} is computed using the temporary graph, considering the zero-weighted virtual edges E'_{virt} along with the remaining physical edges E_{phys} (line 6). For this recomputed delivery tree, those virtual edges are preferentially chosen that replace source-routed physical edges. The inclusion of t may potentially allow new delivery paths using intermediate nodes that differ from the initial tree E^+.

After recomputing the delivery tree, all persistent physical edges E^+_{dist} (line 7) and adjacent virtual edges E^-_{dist} that branch away from the new distribution tree (line 8) are computed. The physical edges require hop labels, while the adjacent virtual edges require stop labels to prevent messages from leaving the delivery tree and possibly entering forwarding cycles. Forwarding cycles can be an unintended side effect of combining multiple virtual trees.

The new encoding costs C' consider the number of hops $|E^+_{dist}|$, stops $|E^-_{dist}|$, and virtual trees $|T'|$ to include in the header stack (line 9). If this results in a smaller header stack (line 10), the algorithm has achieved a more compact encoding. In this case, t is appended to the result set T, indicating its inclusion

in the ongoing distribution tree construction. Furthermore, dependent variables are updated to enhance the encoding in subsequent iterations (lines 11 and 12).

The loop iterates through all the virtual trees stored in T_{virt}, until no further improvements are achievable. The output of the algorithm (line 13) at the end of the loop is the set of computed virtual trees T, along with the sets of edges E^+ and E^- representing hops and stops, respectively. This provides the necessary labels for a compact representation of the distribution tree.

Importantly, the most significant savings often occur in the early iterations of the loop, allowing for the possibility of an early exit to reduce runtime, especially if little or no improvement is observed after a few iterations. In addition, the encoding heuristic works across different network topologies.

5.3 Tailoring Virtual Trees

The encoding algorithm described above combines virtual trees to construct a complete delivery tree. However, the effectiveness of this combination relies heavily on the preinstallation of VTs that align with the actual message flows between publishers and subscribers. This raises a critical question: How can we design VTs that seamlessly integrate into large-scale distribution trees, thereby optimizing the efficiency of information dissemination in a pub/sub system?

To address this challenge, we propose three distinct strategies, each leveraging progressively deeper insights into the pub/sub network. The strategies evolve from relying solely on the *network's topology*, to incorporating knowledge of *publisher and subscriber distribution*, and finally to using detailed statistical data on *subscriber notification frequencies*.

5.3.1 Datacenter Topology

Modern datacenter networks are built with scalability, performance, and fault tolerance in mind. These structured topologies present ideal conditions for designing flexible, stitchable VTs. Such trees can either interconnect nearby machines within the same rack or span distant network regions to reach multiple subscribers.

As a practical case study, we base our strategy on Facebook's three-tier *Fat-Tree* network fabric, introduced in 2014. This topology, with its regular structure and rich path diversity, offers an excellent foundation for building VTs that can be efficiently combined. Subsequent revisions to this fabric have focused primarily on improving bandwidth and scalability – features we aim to exploit in our VT construction strategies.

Figure 5.4 illustrates the original architecture of this network fabric [4], highlighting the properties we exploit in our design. The fabric is composed of uniformly

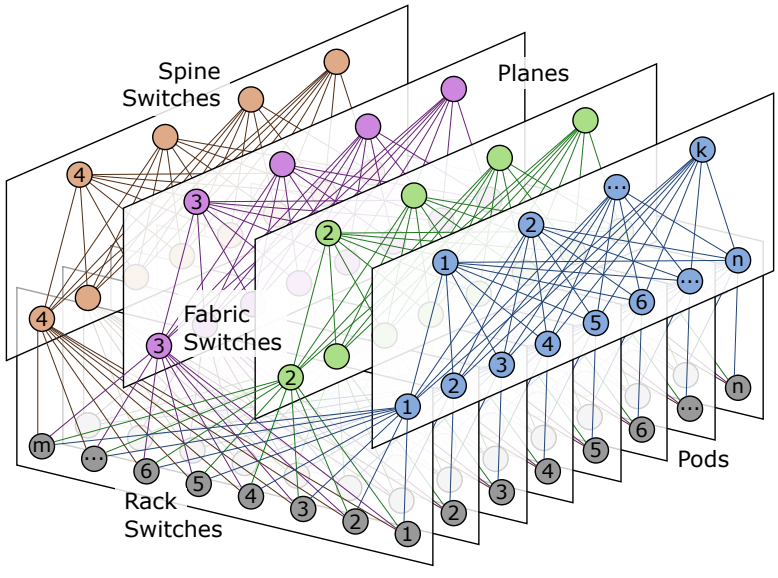


Figure 5.4: Facebook Fabric Network.

structured pods, each containing multiple racks, with every rack equipped with a *top-of-rack (ToR)* switch. Each ToR switch connects a small cluster of machines, which may serve as publishers or subscribers. Importantly, the ToR switches do not connect directly to each other; instead, they communicate via uplinks to *fabric switches*, which form an intermediate communication layer.

Fabric switches connect the racks within a pod using downlinks and provide uplinks to *spine switches*, which are at the topmost layer of the topology. Spine switches ensure connectivity across the entire datacenter, forming the network's backbone. Specifically, the backbone consists of $4 \times k$ spine switches, organized into four planes of k switches each, ensuring robust redundancy.

The hierarchical and redundant design of this fabric offers significant potential the construction of virtual trees. By carefully selecting links between the *ToR*, *fabric* and *spine* switches, it is possible to derive differently shaped virtual trees. Below, we present structured virtual tree designs that exploit the fabric's properties to create stitchable virtual trees for efficient notification delivery.

5.3.2 Broadcast Trees

We begin by leveraging the regular switch layout of the datacenter to construct a set of variously shaped *broadcast trees*, each serving as a virtual tree that interconnects all machines in the network. These trees enable efficient, network-wide addressing by requiring only a single tree label in the message header.

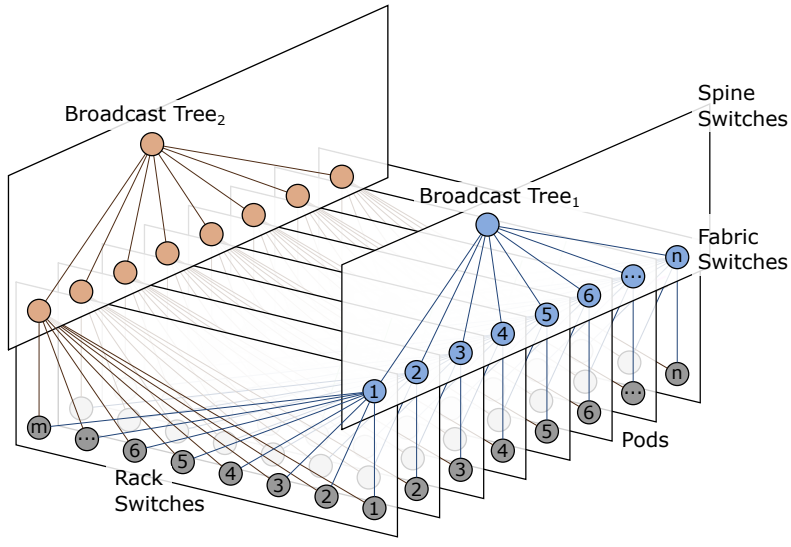


Figure 5.5: Broadcast trees for Facebook fabric network.

Each broadcast tree is designed to balance static network load while ensuring complete coverage of all racks and machines, using distinct combinations of *fabric* and *spine* switches. Figure 5.5 illustrates two examples: *Broadcast Tree$_1$* and *Broadcast Tree$_2$*, each using a different spine and a unique subset of fabric switches, along with all top-of-rack (ToR) switches, to establish full connectivity.

To construct a broadcast tree, we designate a *spine switch* as the root node and recursively traverse all downlinks in the fabric network. Each traversed downlink becomes an edge in the tree. This process gives a tree that includes one spine, a subset of fabric switches, and all ToR switches, ensuring connectivity to every machine in the datacenter. By repeating this process for each spine switch, we generate a unique broadcast tree per spine.

To balance network load, publishers are grouped, and each group is assigned to a specific broadcast tree. A simple mechanism – such as applying a modulo operation to a publisher's numeric ID – can be used to assign publishers evenly. As a result, each group transmits notifications via a distinct combination of spine and fabric switches. Once these trees are implemented in the switch infrastructure, each publisher routes its messages through its designated tree, helping to distribute traffic evenly across the network.

An important characteristic of broadcast trees derived from the same plane is edge overlap. Since all spine switches in a plane use the same set of fabric switches to reach ToR switches, the resulting trees share edges between the fabric switches and the ToR switches. Furthermore, the links connecting ToR switches to their attached machines are identical across all trees because the physical layout offers no alternative paths at this level.

5.3.3 Hierarchical Trees

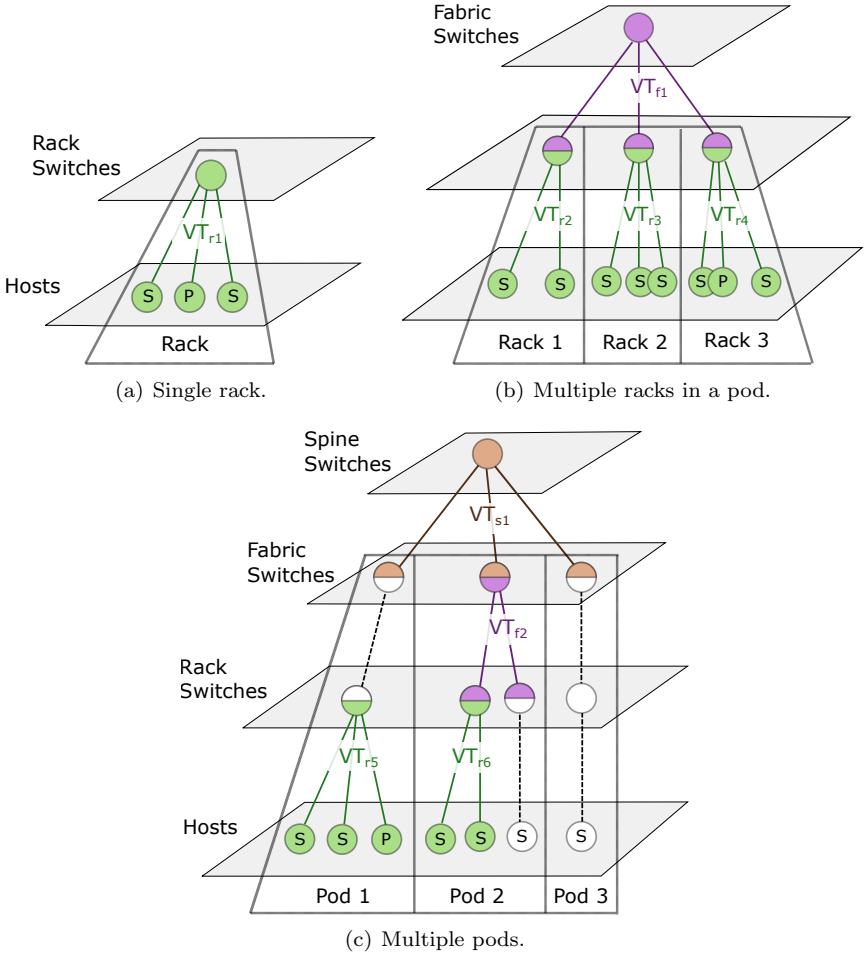
A *topology-based* VT installation strategy exploits the hierarchical structure of the network to craft VTs that do not intersect and can be assembled like modular building blocks. These small VTs are strategically positioned on *spine*, *fabric* and *ToR* switches, enabling their seamless assembly and precise targeting of combinations of downstream switches and machines. By embedding their IDs into the header stack of a notification message, we can target any subset of receivers within the broadcast tree. Using these hierarchical VTs, we distinguish three primary scenarios for assembling a complete delivery tree:

1. *Single rack via ToR switch:* In this straightforward scenario, the notification message remains within a single rack. Figure 5.6(a) illustrates this with two interested subscribers. Originating from one of these machines, the message needs to reach all the other interested machines in the same rack. Here, a single virtual tree, i.e. VT_{r1}, located on a ToR switch, is sufficient to deliver the message to both interested receivers.

2. *Multiple racks via fabric switch:* In a more complex situation, the publisher needs to distribute the message to subscribers located in multiple racks within the same pod. Figure 5.6(b) exemplifies this with a publisher notifying seven subscribers in three different racks. This situation requires the collaboration of a fabric switch and several downstream ToR switches to reach the subscribers. In particular, virtual tree VT_{f1} at the fabric switch is combined with VT_{r2}, VT_{r3}, and VT_{r4} at the respective ToR switches to form the complete delivery tree.

3. *Multiple pods via spine switch:* In the most complex scenario, the notification message must traverse multiple pods in the datacenter and pass through a spine switch to reach remote machines in different pods. This requires the construction of an large distribution tree comprising virtual trees from the spine switch and multiple downstream fabric and ToR switches. Figure 5.6(c) illustrates this with a distribution tree spanning three different pods. Specifically, the publisher combines four virtual trees (VT_{s1}, VT_{f2}, VT_{r5}, and VT_{r6}) with four additional hop labels to form the distribution tree. Since a VT maps at least two branches onto a node, the individual hops (shown as dashed edges in the example) are required when only a single downstream switch or host needs to be connected.

When switch memory constraints prevent the mapping of all possible combinations of neighboring switches or hosts as virtual trees, the focus should be on specific edge combinations. A practical method is to prioritize a reduced VT set based on "power of two" (2^n) edge combinations, where n is incremented starting from $n = 1$ until either the switch memory limit or the number of neighboring nodes is reached. This method reduces switch memory requirements while



(a) Single rack.

(b) Multiple racks in a pod.

(c) Multiple pods.

Figure 5.6: Hierarchical trees for datacenter fabric.

covering a significant portion of the network's connectivity needs. For edge combinations that are not included in the VT set, additional hop or stop entries can be employed to either extend or prune the existing VTs as required.

In scenarios where the publisher and subscriber locations are unpredictable, it is recommended that VTs are distributed evenly across the network nodes. Uniform VT distribution has two advantages: (i) it balances the use of switch memory across the entire network, avoiding localized memory exhaustion, and (ii) it minimizes the likelihood of having to traverse large parts of the network using hop labels.

5.3.4 Advertisements and Subscriptions

Publishers often announce the message content they intend to publish through *advertisements*. These advertisements can be compared with registered *subscriptions* to infer communication patterns between publishers and subscribers. By analyzing these relationships, we can derive publisher-specific virtual trees that connect publishers to relevant subscribers.

As shown in Figure 5.7, three distinct cases arise when comparing advertisements with subscriptions:

(a) *Permanent match:* If a publisher's advertisement overlaps entirely with a subscriber's filter conditions, the publisher consistently generates notifications that are of interest to that subscriber. In this case, every message published by the publisher must be delivered to the subscriber.

(b) *No match:* When a subscriber's filter conditions are completely disjoint from the constraints defined in the publisher's advertisements, there is no overlap between them. Consequently, no published messages will ever match the subscriber's filters, and the publisher's messages do not need to be delivered to that subscriber.

(c) *Potential match:* If a subscriber's filter conditions partially overlap with a publisher's advertisements, there is a possibility that some published messages might match the subscriber's filter. In these cases, each notification must be evaluated individually to determine whether its content satisfies the subscription criteria.

The *tree* strategy is based on these principles. It transforms the set of broadcast trees derived from the network topology by exploiting this relationship knowledge. The key idea is to eliminate both virtual trees and branches of the broadcast tree that lead to hosts without publishers or possibly interested subscribers. The resulting trees connect publishers with the corresponding subscribers where subscriptions either permanently or potentially match.

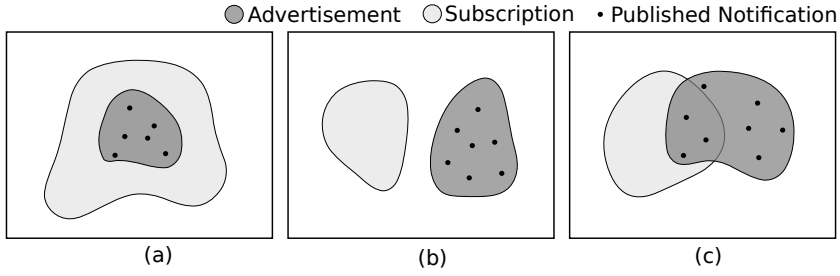


Figure 5.7: Matching of advertisement and subscriptions.

Algorithm 5.2 Pruning of a topological tree.

```
1: procedure PRUNE_TREE( t_broad, N )
2:     t ← t_broad, L ← leaves( t ), L' ← ∅
3:     while L ≠ L' do
4:         L ← leaves( t )
5:         for all n ∈ L do
6:             if n ∉ N then
7:                 remove n from t
8:         L' ← L
9:     if | edges( t ) | > ε then
10:        return t
11:    else
12:        return ∅
```

Publisher-specific Trees

Algorithm 5.2 derives such publisher-centric trees. It takes as input a broadcast tree t_{broad} and a set of terminal nodes N. This N parameter allows the generation of two VT variants: either a *shared* VT for several publishers or a *publisher-specific* VT. For a shared tree, the nodes of several publishers, including the nodes of the associated subscribers are specified as N. For a dedicated tree, only one publisher node and the corresponding subscriber nodes are provided as N. Shared VTs are wider and require more stop labels for notification delivery, as branches to external participants must be explicitly truncated. However, compared to publisher-specific trees, shared VTs save switch memory.

The algorithm works as follows: First, the broadcast tree is copied (t), and all leaf nodes in the tree t are identified and stored in the set L, with a helper variable initialized for a loop condition. Then a loop is executed as long as unseen leaves are found in the tree t. At each iteration a set of leaf nodes is determined and stored in L. For each leaf node, it is then checked whether it is a terminal node hosting a publisher or a subscriber. Terminals remain in the tree, while non-terminal leaves are progressively removed. At the end of the loop, the processed set of leaf nodes is cached in the temporary variable L'. The loop

condition compares the sets L' and L to decide whether the pruning process is complete. If the two sets are different, pruning continues with the next iteration. Otherwise, all non-terminal leaf nodes are removed, and the tree is reduced to its essential edges. Finally, the resulting tree t is returned if it satisfies the edge requirements defined by ε; otherwise, the tree has too few edges and is discarded, returning the *null* value.

The resulting VTs require fewer stop entries than the broadcast trees when applied for delivery. This solution also reduces the number of flow rules required for installation in ToR, fabric and spine switches. In contrast to the *tree* strategy in Section 4.2.3, which requires rule base updates when subscribers unsubscribe or migrate to other network locations, this VT-based approach supports truncation of stored trees by stop labels. These VTs also cover different redundant network links, balancing the network load among different publishers.

Figure 5.8(a) shows two example VTs ($Tree_1$ and $Tree_2$) derived using these algorithms. Both trees are publisher-specific and represent subtrees of different broadcast trees, demonstrating that all branches of the VTs lead to hosts with subscribers. $Tree_1$ spans clients from multiple pods and therefore extends to a spine switch, while $Tree_2$ spans clients from a single pod and therefore only extends to a single fabric switch.

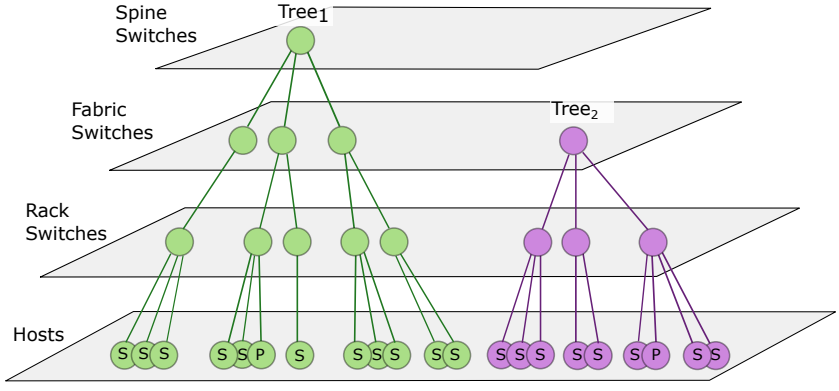
5.3.5 Virtual Paths

While the *tree* strategy incorporates both *permanent* and *potential* matches when evaluating the relationship between advertisements and subscriptions, the *path* strategy refines this approach by distinguishing between these two states. It therefore excludes potentially interested subscribers from publisher-specific trees, focusing only on stable recipients. This optimization allows branches leading to infrequently targeted subscribers to be pruned, freeing up switch memory and reducing the size of publisher-specific virtual trees.

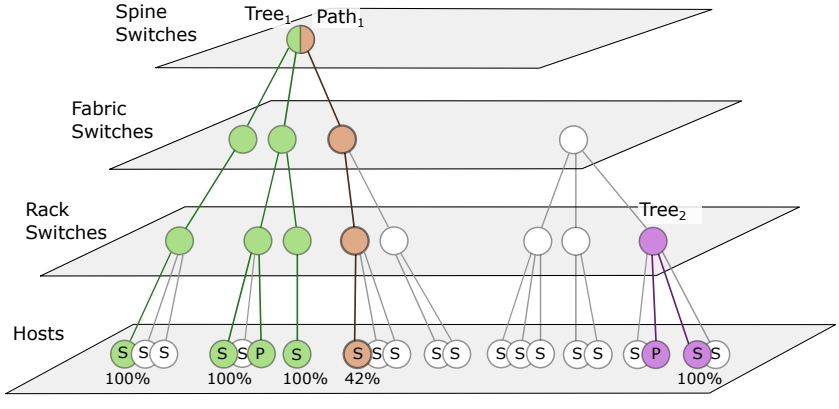
The recovered switch capacity is then repurposed to create additional *virtual paths* – lightweight, bidirectional channels that connect individual machines to a spine switch. Their bidirectional nature simplifies integration with other virtual trees, as only a single additional label is needed to connect an outlying participant to a distribution tree.

Figure 5.8(b) demonstrates this concept using three virtual structures: $Tree_1$, $Tree_2$, and $Path_1$. In this example, $Tree_1$ includes all subscribers with a statistically significant, 100% likelihood of receiving every message from a given publisher. Meanwhile, $Path_1$ provides a downlink to a subscriber with a 42% probability of interest in the publisher's messages. The shared spine node enables $Tree_1$ and $Path_1$ to be combined into a complete distribution tree, encoded using only two VT labels – one for $Tree_1$ and one for $Path_1$.

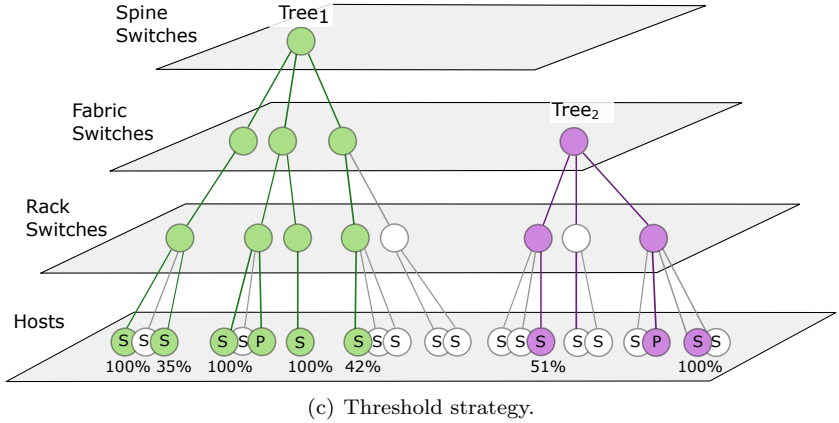
Notably, extending $Tree_1$ without $Path_1$ would require three additional hop labels from the spine switch to the machine, significantly lengthening the packet



(a) Tree strategy.



(b) Path strategy.



(c) Threshold strategy.

Figure 5.8: Tailored virtual trees for datacenter fabric.

header. This illustrates the strategic trade-off between match probability and header efficiency. By isolating infrequently addressed subscribers into separate virtual paths, header overhead is reduced.

In highly dynamic environments with constantly changing subscription patterns, maintaining updated publisher-specific base trees for all possible recipients is inefficient. The *path* strategy avoids building large VTs by excluding *all* dynamic addressed subscribers and merging virtual paths into a delivery tree to attach them as needed. This approach optimizes switch resource usage and reduces packet header size in pub/sub network scenarios with many 'volatile' subscribers.

5.3.6 Notification Statistics

Robust traffic information that accurately reflects actual communication flows is essential for the construction of precise and efficient virtual trees. Although assumptions about the distribution of notifications are essential, relying solely on issued advertisements and subscriptions may not be sufficient to identify high-frequency message flows for the derivation of profitable trees. Instead, collecting *notification statistics* at runtime provides a more accurate understanding of message traffic.

These statistics quantify the notifications sent per publisher, capturing the proportion of notification messages received by each subscriber. This enables the identification of high-frequency message flows per publisher. Ranking subscribers by contact frequency allows us to prioritize message flows when installing forwarding rules on limited switch memory. To implement this nuanced approach, we revise the tree strategy by installing publisher-specific trees and including only those subscribers with a high likelihood of receiving a notification.

For this likelihood, we use a *threshold* based on the fraction of notifications a subscriber has received from the publisher. Subscribers with fractions above the threshold are categorized as frequently contacted and included in the tree, while those with notification frequencies below the threshold are categorized as infrequently contacted and ignored for VT construction. These rarely addressed subscribers can be added to the distribution tree using additional labels encoded in the header.

This *threshold* strategy refines the *tree* strategy and calibrates the expansion of virtual trees by setting a minimum required contact frequency for each subscriber in the tree. The resulting tree sizes range from those of the *tree* strategy to those of the *path* strategy, corresponding to thresholds of 0% and 100% notifications received, respectively.

As an example, Figure 5.8(c) depicts the VTs of two publishers constructed using this threshold-based strategy. With the threshold set to 30%, subscribers with a notification frequency of at least 30% are included in the publisher-specific tree, while those below this threshold are excluded and require explicit inclusion in the distribution tree through additional hop or tree labels.

5.3.7 Implementation Framework

The virtual tree installation schemes described above are designed to be flexible and interchangeable during the runtime of the pub/sub network. This is possible because the VTs are stored as flow rules that can be managed by an SDN controller. The ability to add, delete, or modify virtual trees is particularly valuable in scenarios where initial knowledge of the network is limited. For example, this could be useful when the network topology is known, but the locations of publishers and subscribers are not. In such cases, a simple topology-based tree installation scheme could initially be selected to optimize header space within notification messages to a certain extent. As network knowledge grows, more sophisticated virtual trees can be installed at runtime to further enhance header optimization.

To facilitate this flexibility, we developed a skeleton class that promotes the interchangeability of (i) notification dissemination strategies and (ii) tree installation schemes. This framework uses a class hierarchy to encapsulate key entities, defining their functionality through dedicated class methods. Abstract base classes provide an API to simplify the development of notification distribution strategies, from which specific strategies are derived. Each strategy relies on a specific set of header label types L, such as $L = \{hop\}$ for the *hops* strategy or $L = \{tree, hop, stop\}$ for VT-based strategies, while the VT-based strategies, i.e. the *tree*, *path* and *threshold* strategies, differ in their rule base installation schemes and offer different stored tree composition variants for header encoding.

Class Hierarchy Overview

Figure 5.9 illustrates the entities of our pub/sub model and their relationships. Central to this model is the abstract class `Strategy`, which defines abstract methods to be implemented by concrete strategies. Each concrete strategy specifies its own set of flow rules to be installed on the switches by implementing these abstract methods. The class diagram shows two strategies derived from `Strategy`: `HopStrategy`, which uses only *hop*, labels, and `TreeStrategy`, which uses *hop*, *stop*, and *tree* labels to encode complete distribution trees. Each strategy implements its own encoding function based on the supported labels and its potential rule base.

The `TreeStrategy` class serves as an abstract base class for specific tree installation strategies. Two subclasses are depicted in the diagram: `TopoTreeStrategy`, which provides functions for installing virtual trees on rack, fabric, and spine switches, and `ThresholdTreeStrategy`, which encapsulates the `threshold` attribute for calculating streamlined virtual trees and excluding infrequently contacted subscribers. In `ThresholdTreeStrategy` subclass, the `addBaseTrees` method installs a tree for each publisher, while `addPaths` method adds additional publisher-independent virtual paths as long as there is available switch

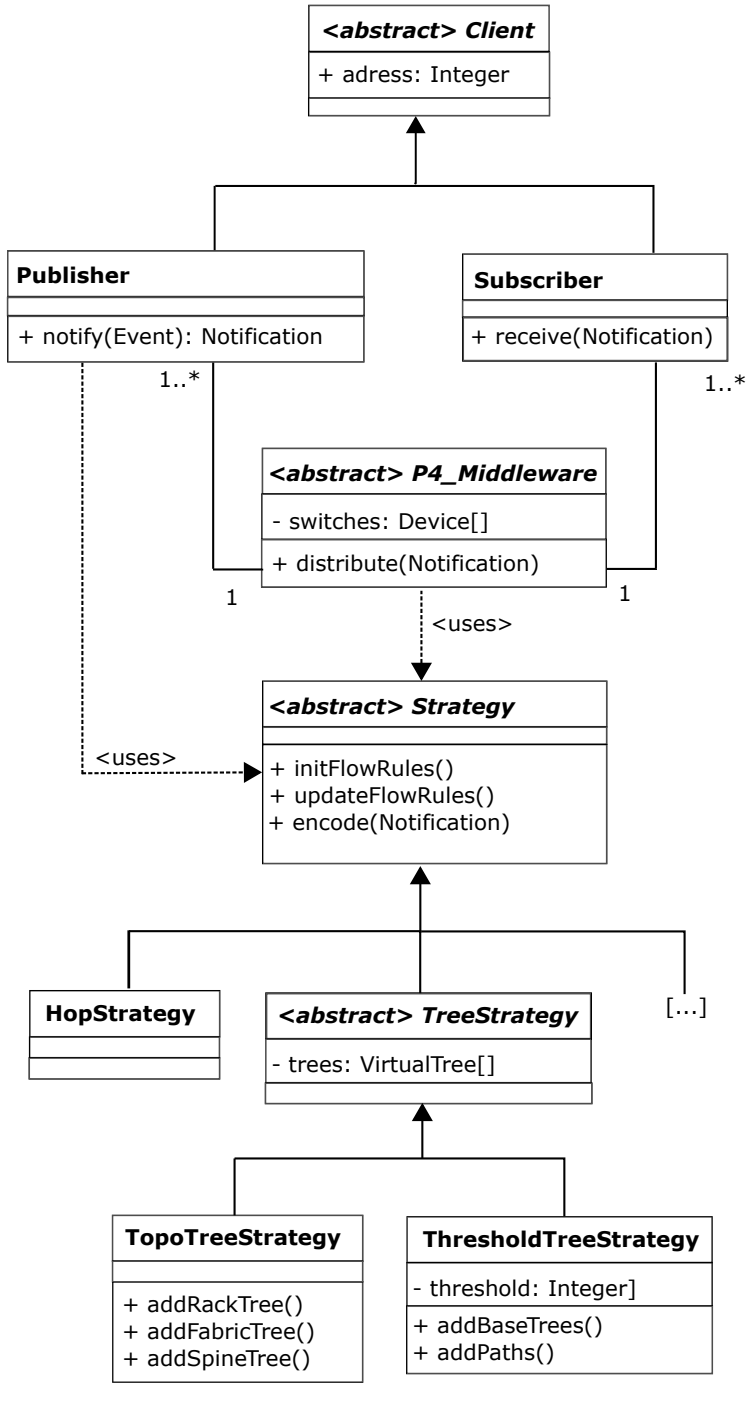


Figure 5.9: P4-based framework for notification dissemination strategies.

memory. The threshold specifies the minimum percentage of published notifications a subscriber must receive to remain in the tree. The *tree* or *path* strategy can be activated by setting the threshold to 0% or 100%, respectively.

The middleware integrates a concrete strategy using the abstract `Strategy` class. A `Strategy` implementation computes and installs the initial flow rules for all publishers and their corresponding subscribers using the `initFlowRules` function. This rule base can be modified at runtime using the `updateFlowRules` function, allowing an already installed tree to adapt to changing communication patterns. Once the VTs are installed, publishers can encode the header stack for a notification message using the `encode` function provided by `Strategy` and send the encoded message to the `Middleware` class by calling the `distribute` function. The middleware encapsulates the switch infrastructure and manages multiple instances of publishers and their corresponding subscribers through the `Publisher` and `Subscriber` classes. Both these classes are subclasses of `Client`, and their instances are uniquely identified by the `address` attribute. While this implementation uses integer identifiers as addresses, alternative implementations can map MAC or IP addresses instead. Publishers use the `notify` method to transmit events through the `P4_Middleware`, while subscribers receive relevant notifications via the event handler method `receive`.

Rule Base Update

Updating the rule base using the `updateFlowRules` function requires careful coordination to prevent errors. To avoid referring to deleted trees, outdated rules should only be deleted once the new rules have been fully installed and the relevant publishers have been informed. Deleting old trees before the new ones are installed could result in packet loss, as switches would no longer recognize messages in transmission that refer to old trees via outdated IDs in the header.

A more serious problem arises from incorrect routing due to restructured VTs, where VT changes are not yet known to the publisher at the time of delivery encoding. In this case, a notification header referring to the old VTs might unintentionally refer to the new, restructured VTs. This could lead to incorrect routing, and in the worst case, form a cycle where the packet circulates indefinitely in the network. To prevent such confusion, IDs of deleted VTs must not be reused and should be made unique, e.g. by incrementing them continuously.

Classes for Label Encoding

Figure 5.10 gives an overview of the classes designed to encode header stacks. The abstract class `Label` contains an identifier attribute and is responsible for labeling notification messages and holding any kind of routing information. Subclasses such as `Hop`, `Stop`, and `Tree` inherit from `Label` and represent different types of routing information. Instances of these label types are generated during

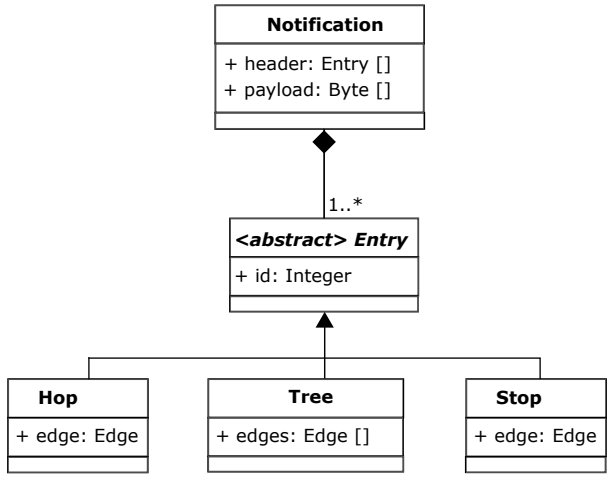


Figure 5.10: Label type entities for strategies based on virtual trees.

the encoding of the distribution tree to form a sequence of labels written into the message header. `Hop` and `Stop` contain an `edge` attribute to identify a particular network edge for the purpose of truncating or extending the distribution tree, while `Tree` contains the `edges` attribute representing an edge set forming a VT.

5.4 Evaluation

This section presents a detailed evaluation of virtual tree deployment strategies in an emulated network environment. The evaluation investigates the performance of strategies ranging from basic topological approaches to advanced techniques that leverage publisher-subscriber relationships and notification frequency statistics to fine-tune virtual trees.

The analysis is structured around three key aspects: (i) the *initial header size* generated by each strategy, (ii) the impact of each strategy on *overall network load*, and (iii) the *impact of VT size on delivery efficiency*. Together, these considerations provide a holistic view of the strengths and trade-offs of each strategy.

5.4.1 Network Setup

The following outlines the simulated network configuration, detailing the variables listed in Table 5.1 used to configure the network conditions.

Topology. The network topology is modeled after the Facebook network fabric [4]. We configure the network with $n = 8$ pods, each containing $m = 4$ racks,

Variable	Range	Description
subscriber_percentage	30% to 70%	Determines the proportion of hosts that act as subscribers for each publisher, influencing the size of the notification delivery trees.
notification_frequency	1% to 100%	Represents the likelihood that a host is interested in a particular notification message content, simulating a content-based filtering model.
max_rules	0 to 50	Defines the maximum number of forwarding rules that each switch can store, simulating practical memory constraints of network switches.

Table 5.1: Key Variables for Simulation Run

and each rack connecting $l = 2$ machines (see Fig. 5.4). As a result, each pod consists of 8 hosts, yielding a total of 64 hosts across the network, interconnected by 80 switches. Every host can function as both a publisher and a subscriber, creating a dynamic, flexible communication environment.

Subscriber density. To evaluate the effectiveness of the VT deployment strategies across various scenarios, we vary the number of subscribers for each publisher. The variable `subscriber_percentage` defines the percentage of hosts that are interested to a publisher's notifications, ranging from 30% to 70%. This variation allows us to assess the strategies under different subscriber densities, where a higher subscriber density leads to increased network load.

Content-based filtering model. The simulation incorporates a content-based filtering model to reflect realistic subscription dynamics. Each host is assigned a random, fixed `notification_frequency` value at the start of the simulation. These values, uniformly distributed between 1% and 100%, represent the likelihood of a host being interested in specific notifications. A higher probability indicates a stronger match between a host's subscription and the content of a notification. On average, a publisher's message reaches half of its subscribers. For example, if a publisher has 30% of total hosts as subscribers, approximately 15% of the total hosts will receive the message on average.

Forwarding rules. The simulations account for practical memory constraints in switches by varying the maximum number of forwarding rules per switch, denoted by the variable `max_rules`, from 0 to 50. A value of 0 indicates that no virtual tree can be installed on the switch, requiring hop-by-hop routing. A value of 50, on the other hand, allows each switch to distinguish between up

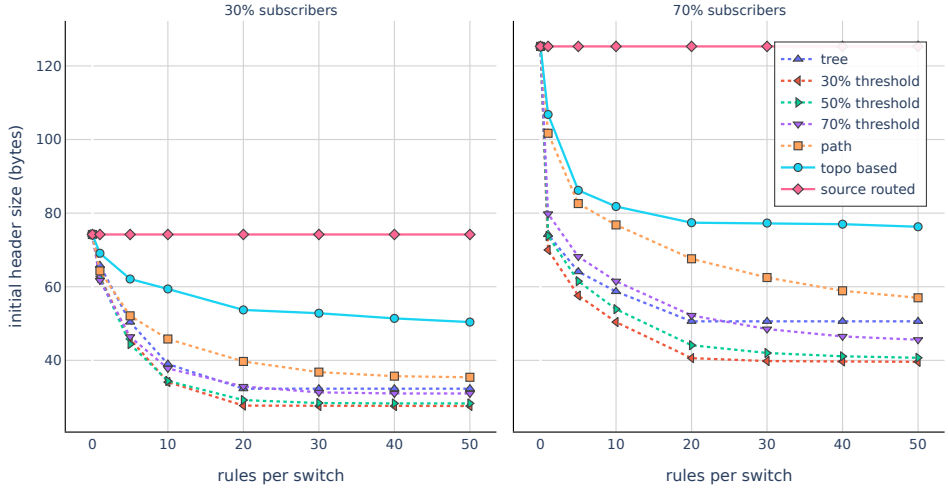


Figure 5.11: Initial header size in Facebook's fabric topology.

to 50 VTs, allowing a variety of VTs to be combined to form a distribution tree. However, the total number of VTs in the network may actually be higher than this, as switches may store different virtual trees consisting of distinct edge combinations. This range simulates limitations in switch memory, testing how well each strategy adapts to varying flow rule constraints.

Key Metrics

The evaluation measures two primary metrics for all strategies: (i) the *initial header size* of the notification message required for routing and (ii) the overall *network load* generated by counting the bytes transmitted per message.

To ensure a robust and representative performance analysis, the results for each data point in the evaluation charts are averaged over 25 independent simulation runs per simulation scenario. In each scenario, the publishers and subscribers are redistributed, the percentage of interested subscribers is changed, and the maximum allowed forwarding rules per switch are adjusted. The evaluation results provide valuable insights into the performance of different VT deployment strategies, which are discussed in the following sections.

5.4.2 Initial Header Size

Figure 5.11 illustrates the initial header size as a function of the number of flow rules available on switches, comparing different dissemination and VT in-

stallation strategies. The chart includes two scenarios with differing subscriber densities: *low density* (30% of hosts as subscribers) and *high density* (70% of hosts as subscribers). This distinction highlights how subscriber density impacts header size. The *source-routed* strategy serves as the *baseline* for both scenarios. Since this strategy operates independently of preinstalled trees, its header size remains constant across all rule limits, appearing as a horizontal line in the plots. By contrast, strategies leveraging virtual trees show significant reductions in header size, particularly as the number of flow rules increases. This demonstrates the benefits of preinstalled routing information.

The analysis and comparison of the plots leads to several significant observations:

1. *Effect of allowed rules on header size:* There is a clear trend across all strategies: increasing the number of allowed rules results in smaller headers. This underscores the role of preinstalled trees in minimizing delivery overhead. From approximately 20 rules per switch, most strategies can map the majority of their distribution trees, regardless of subscriber density. Adding more rules leads to minor improvements for the *path* strategy and the *70% threshold* strategy. This indicates that this rule base covers the most important subscriber permutations by VTs.

2. *Impact of subscriber density:* Header size expands with more subscribers (70%), leading to a higher delivery overhead compared to scenarios with fewer subscribers (30%). For example, with 10 rules per switch, the *topo-based* strategy requires 60 bytes for the initial header when 30% of hosts are subscribers, while the same strategy with the same number of rules per switch requires approximately 78 bytes when 70% of hosts are subscribers.

3. *Efficiency of VTs in dense scenarios:* In scenarios with a large number of recipients, VT strategies show significantly greater relative savings compared to the source-routed strategy. For example, the *topo-based* strategy shows has a larger margin over the source-routed strategy and thus provides more compact headers when messages are distributed to 70% rather than 30% of hosts. This implies that VT strategies can make better use of preinstalled trees when encoding large delivery trees with many recipients.

4. *Advantage of application-specific knowledge:* Strategies that incorporate application-specific knowledge, such as the *path* strategy, generate smaller headers than purely topology-based strategies. For example, for both the 30% and the 70% subscriber scenario, a significant gap between the *path* and the *topo-based* strategy can be seen from 5 rules per switch onwards, which gets wider as the number of rules increases. More sophisticated VTs derived from notification statistics, in particular the threshold variants, are even more effective. In particular, the *30% threshold* variant always produces the smallest headers and reaches its optimum at 20 rules per switch for both subscriber scenarios. More rules do not result in any further reduction in header size.

5. *Effectiveness of large virtual trees:* The *tree* strategy, which relies solely on advertisements and subscriptions, is one of the best performing strategies, although it does not consider notification statistics. Compared to smaller VTs, such as those generated by the *path* strategy, the large VTs generated by the *tree* strategy are better suited to encode distribution trees with a minimum number of labels. Although the *path* strategy gradually converges to the results of the *tree* strategy as the number of allowed switch rules increases, the *tree* strategy consistently gives better results in both subscriber scenarios.

In summary, a small number of large virtual trees (such as those generated by the *tree* strategy) show a better ability to encode distribution trees with a minimum number of labels than a large number of small VTs (such as those generated by the *path* strategy). This indicates that *pruning a large VT with stop labels* is more effective than *extending a small VT with additional hop labels*. This finding highlights the importance of considering the size of virtual trees when reducing packet header length.

5.4.3 Network Load

Figure 5.12 illustrates the total network traffic and highlights the benefits of employing fewer but larger virtual trees. The figure distinguishes between low and high recipient scenarios, with 30% and 70% of subscribed hosts respectively. Each notification message includes a 50-byte payload, simulating the frequent status updates typical of pub/sub systems through small messages.

Due to the relatively small payload size, the notification header contributes significantly to the network load. Consequently, strategies that reduce header size are particularly effective in minimizing the overall network load, as variations in header size have a direct impact on message traffic. This demonstrates how optimized VT configurations can improve network efficiency.

Analyzing and comparing the plots reveals following key observations:

1. *Effect of payload size:* The 50-byte payload size emphasizes the substantial impact of the notification header on network load. Smaller headers result in proportionally less message traffic. This highlights the critical role of header optimization for pub/sub systems with small notification payloads but frequent message delivery to subscribers.

2. *Strategies profiting from larger trees:* Strategies employing fewer but larger distribution trees, exemplified by the *tree* strategy and the 30% *threshold variant*, demonstrate advantages in overall network traffic. The impact of tree size on network load is particularly pronounced in scenarios with higher subscriber densities (70%).

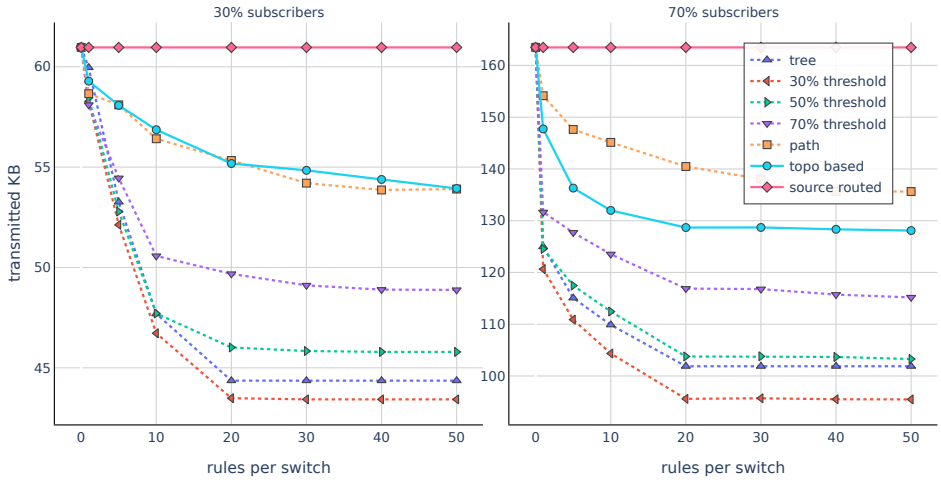
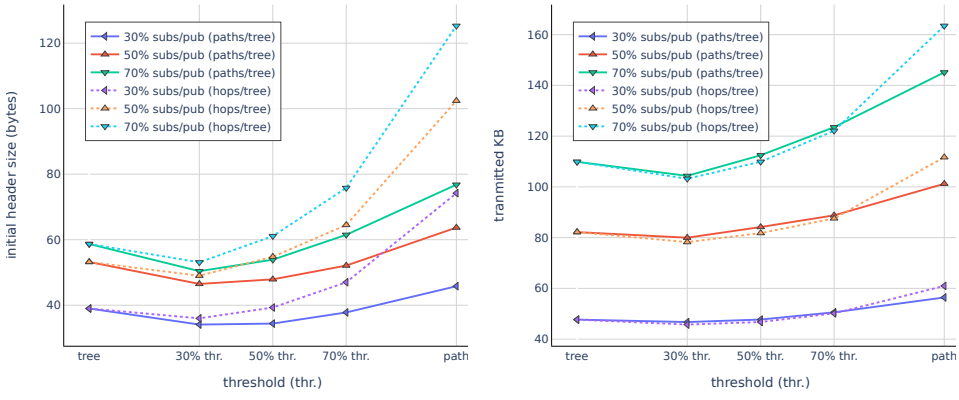


Figure 5.12: Network load in Facebook's fabric topology.

3. *Pruning constraint with tree labels:* Strategies that combine many VT labels, as the *topo-based* strategy or *path* strategy, pose challenges for header stack stripping, leading to increased traffic load. The results always remain clearly above the 50 KB and 120 KB limits for the 30% and 70% subscriber scenarios respectively, while the other strategies are below these limits and therefore perform better. This is because the large number of VT labels of the *topo-based* strategy or *path* strategy remain in the header stack, increasing the network load. In contrast, the hop and stop labels, which are used more heavily by the better-performing strategies, are pruned from the header stack after application.

4. *Advantage of larger virtual trees:* The *tree* strategy and the *30% threshold* variant, using larger virtual trees, allows publishers to adapt to the receiver set with minimal effort. From 20 rules per switch onward, both strategies generate an average network load of less than 45 KB for the delivery of a notification while all other strategies generate more network load. Obviously, these strategies can use stop labels with minimal impact on the initial header size and consistently reduce the network load. In contrast, strategies that rely on small distribution trees, such as the *topo-based* strategy, face challenges in achieving payload reduction.

Efficient network load management is closely related to the label-based refinement of distribution trees. While *stop* and *hop* labels are deleted from the message header after a single use by their dedicated switch, *tree* labels are always kept in the header as they may be relevant to subsequent switches. In addition



(a) Initial header size.

(b) Network load.

Figure 5.13: Tree extension by stored paths vs. hops (10 rules/switch).

to being persistent, small VTs require extensive extensions by additional tree or hop labels, which lengthen the header. Larger distribution trees, especially if pruned intelligently, offer greater benefits in terms of network load.

5.4.4 Optimizing Tree Size with Scarce Switch Resources

To evaluate the effectiveness of installing virtual trees for individual distribution paths under constrained switch memory, we conducted an experiment limiting each switch to just 10 forwarding rules. The experiment examined the impact of different notification thresholds (0% 30%, 50%, 70% and 100%) on various subscriber densities (30%, 50%, and 70% of total hosts). The threshold extremes – 0% and 100% – correspond to the *tree* strategy and *path* strategy, respectively.

We compared two methods for reaching (additional) subscribers excluded from the main distribution tree due to the threshold setting:

(i) *Paths/tree:* Additional subscribers are connected to the base tree by using virtual trees for *individual paths (virtual paths)* leading to them.

(ii) *Hops/tree:* Additional subscribers are attached via *source routing*; all hops from the base tree to the subscribers are encoded in the notification header.

Figures 5.13(a) and 5.13(b) summarize the results for *initial header size* and *network load*. The following observations highlight key trends:

1. *Dynamics of tree size:* The results confirm that the *tree* strategy, which uses large virtual trees for each publisher, performs well with minimal additional labels needed to adapt the distribution tree (e.g., *stop* labels for pruning). By contrast, smaller base trees produced by threshold variants ($30\% \leq$ threshold $\leq 70\%$) generate the smallest headers, as they require minimal augmentation with additional labels. Conversely, the *path strategy*, relying on a small base tree, demands significantly more *hop* labels to encode a complete distribution tree, increasing header size.

2. *Header size reduction:* Compared to the use of hop labels in the source-routed variant, the use of individual paths consistently leads to smaller headers. This reduction becomes significant when the notification threshold exceeds 30%. While the *tree* strategy benefits little from individual paths (since it relies on pruning branches with *stop* labels rather than extending them), threshold-based strategies achieve a balanced trade-off between header size reduction and VT usage.

3. *Effects on network traffic:* Substantial reductions in network traffic are observed only when the notification threshold surpasses 70%. Individual paths only contribute to a significant reduction in network load for small base trees, as it is particularly the case for the *path* strategy. However, larger base trees, which include more stable receivers, reduce the necessity for and the impact of additional paths.

The results show that although VTs provide a significant reduction in header size, the impact of virtual paths on network traffic becomes significant at thresholds above 30%, reflecting relatively small base trees. Larger base trees (below the 30% threshold) inherently accommodate stable receivers, reducing the need for additional paths.

In dynamic environments where publishers deal with fluctuating subscriber sets, combining smaller base trees with virtual paths can effectively minimize both header size and network traffic. These insights highlight the importance of selecting appropriate VT strategies based on the network's specific constraints and operational requirements.

5.5 Related Work

Various approaches for implementing multicast routing protocols have been extensively explored in the literature. Islam et al. [59] offer a comprehensive survey that examines protocols designed to address specific constraints while optimizing various objectives. These protocols generally fall into two categories: (i) constructing shared distribution trees to minimize the state overhead within the switch infrastructure, or (ii) employing redundant trees to achieve a more balanced distribution of network traffic.

In the sections that follow, we analyze these protocols, drawing comparisons with our virtual tree strategies and highlighting the key differences between these established approaches and our proposed solution.

5.5.1 Protocol Independent Multicast

Protocol Independent Multicast (PIM) is an IP multicast routing protocol that establishes shared trees using PIM control messages. It manifests in two primary modes: *sparse mode (PIM-SM)* [45], designed for sparsely distributed networks, and *dense mode (PIM-DM)* [1], designed for densely populated networks.

In *dense mode (PIM-DM)*, routers forward multicast traffic on all interfaces until a downstream router sends a *prune* message to its upstream router, signaling it to stop forwarding. This mode presumes membership for each client on the network and relies on the prune messages to remove unnecessary routers and their connected clients from the multicast tree.

In *sparse mode (PIM-SM)*, multicast traffic is not forwarded on any interface, until a downstream router sends a *join* message to its upstream router, requesting it to start forwarding. This mode allows local group members to actively participate in the multicast distribution tree.

PIM-SM introduces the concept of *rendezvous points (RPs)* for the construction of shared trees. RPs serve as central nodes where senders announce their presence and receivers discover new contributors, enabling dynamic changes in group membership. Senders use *register* messages to announce themselves, while receivers use *join*/*prune* messages to connect or disconnect.

However, shared trees can also cause network congestion on the links around the RPs. To alleviate this, shared trees can be converted to *source-specific distribution trees* [45] by using *join* messages to add group members to a source-specific tree and *prune* messages to disable the original delivery path through the RP once the new source-specific tree is established.

Despite these capabilities, tree membership management can become complex. PIM-SM routers exchange messages to manage the PIM neighborhood and establish shortest path trees. In the context of content-based pub/sub systems, PIM multicast has limitations in terms of individual control and customization.

To address these challenges, the SDN-based approaches discussed below propose clustering multicast sources and delegating their management to an SDN controller for multicast tree construction.

5.5.2 Locality-Aware Multicast Approach

Lin et al. [76] propose the *Locality-Aware Multicast Approach* (Lama), introducing a location-based multicast strategy that favors shared trees over per-source

trees. LAMA clusters multiple multicast groups and constructs shared trees by grouping close senders using a distance-based clustering algorithm. A central node, known as the *rendezvous point (RP)*, is then selected based on its minimal distance to all multicast sources within the cluster.

Unlike in PIM-SM, the RP is not used for registration and discovery, but serves as the root of a shortest path multicast tree, connecting the various multicast group sources within the cluster to common receivers. A packet to be delivered is first routed from the source to the RP as an intermediary, and then distributed from the RP to the receivers. This approach reduces the number of required trees and the volume of flow rules needed for installation.

However, there are limitations to the use of shared trees. Firstly, if the rendezvous node is not on the shortest path of each group member, the inclusion of a shared tree can potentially lengthen the delivery path, leading to longer routes compared to direct sender-receiver communication. Secondly, reconfiguration of a shared tree is essential whenever a new receiver joins the shared tree, requiring updates to the routing tables of the participating routers. In contrast, our virtual trees do not require reconfiguration, but do lengthen the packet header if they fail to cover all interested receivers or include uninterested subscribers, necessitating additional stop or hop labels.

Finally, LAMA's shared trees cover subscribers from multiple publishers, potentially aggregating too many subscribers with diverse interests and possibly leading to false positives. With our virtual trees, on the other hand, publishers keep complete control over delivery paths and tailor virtual trees to ensure that only interested recipients are notified.

5.5.3 Avalanche Routing Algorithm

Iyer et al. [60] introduced the *Avalanche Routing Algorithm* (AVRA), a method designed to improve load balancing in datacenters by leveraging SDN capabilities and network topology characteristics. AVRA constructs hop-minimal distribution trees tailored for different receiver groups, ensuring a more balanced distribution of traffic load across routers.

The core principle of AVRA is the use of distinct routing trees for different multicast groups to take advantage of the inherent path diversity in datacenter networks. Initially, the algorithm creates trees based on shortest-path calculations and dynamically expands them to accommodate additional receivers using a *breadth-first search* approach. When integrating a new peer into the tree, the algorithm exploits the hierarchical structure of the network, randomly selecting up-links to higher-level routers. This randomness increases the utilization of the available path diversity, ensuring better load balancing.

Our approach also exploits the structured architecture of datacenters to derive virtual trees. Similar to the routing trees generated by AVRA, our VTs span

various ToR, fabric, and spine switches, connecting multiple machines in different racks via diverse links. However, unlike random link selection in AvRA, our virtual tree strategies systematically iterate over available switches. At different hierarchical levels of the topology, we construct top-down VTs for permutations of downstream switches or hosts. Moreover, we install VTs for single paths from spine switches to hosts, or redundant broadcast trees that cover all hosts but traverse different fabric and spine switches. In cases, where multiple redundant trees are available to publishers for message delivery, we assign each publisher to a specific tree in a round-robin fashion to ensure a balanced link utilization.

5.5.4 Dynamic Software-Defined Multicast

Rückert et al. [100] present the *Dynamic Software-Defined Multicast*(DynSdm) model, which dynamically establishes multicast groups and constructs multiple distribution trees that can quickly adapt to link failures. In DynSdm, the sender's traffic is divided into sub-streams, with each sub-stream routed through an individual subtree. These subtrees are uniquely identified by a subgroup identifier, consisting of an IP address and port number, to ensure efficient traffic distribution. This process leverages OpenFlow's group table selection feature [46] for subtree mapping.

DynSdm categorizes switches into ingress, internal, and egress switches. When a packet arrives at an ingress switch, an action bucket modifies its destination IP address and UDP port to assign a subgroup identifier, associating the packet with a specific subtree. Internal switches read this subgroup identifier and apply preinstalled rules to forward the packet along the designated subtree. At the final stage, egress switches strip the subgroup identifier and convert the multicast packet back into unicast traffic before delivery to the destination node.

The construction of redundant multicast trees in DynSdm is guided by edge weighting in the network topology, taking into account factors such as current traffic load and quality of service metrics. This adaptive weight calculation encourages the selection of disjoint edges where possible within the given topology. When new clients join a multicast group, DynSdm integrates them into active trees using a *breadth-first search* algorithm. Unlike approaches that recompute the entire tree, this method only requires installing additional rules along the path between the active tree and the new client, minimizing disruption.

In contrast, our virtual tree approach simplifies subscription management by avoiding immediate rule base updates. However, if VTs no longer fully cover all intended recipients, the system compensates by increasing routing information in the packet header.

5.6 Summary

This chapter introduced innovative delivery strategies customized for content-based pub/sub systems, centered around the concept of *virtual trees (VTs)*. VTs serve as reusable, partial delivery structures that can be combined to construct complete distribution trees. By leveraging VTs and embedding additional distribution information within message headers, the proposed approach enables fine-tuning between stateful and stateless techniques, providing both adaptability and efficiency in message delivery.

Virtual trees facilitate bidirectional communication and can be used by any sender on the network, allowing label-driven composition with other VTs. An efficient greedy encoding algorithm merges *tree* labels with additional routing information (*hop* and *stop* labels) to tailor the stored trees to specific notification message delivery requirements. Hop and stop labels allows the encoding algorithm to expand or prune stored trees, resulting in compressed message headers where a minimum number of tree, hop, and stop labels represent a complete delivery tree.

To generate a diverse set of virtual trees, our tree installation strategies leverage either topology-based knowledge, application-specific insights (e.g., pub/sub relationships), or refer runtime statistics (e.g., subscriber notification frequency). By exploiting this knowledge, we prioritize the installation of stored forwarding rules along frequently used paths and balance traffic on redundant network links.

Evaluations show a significant reduction in message traffic when using VTs, with their effectiveness increasing with the level of knowledge used for deployment. While small topology-based trees offer easy composition, they require multiple VT labels. On the other hand, notification statistics enable fine-tuned trees, preserving branches to infrequently contacted subscribers, whereas VTs based solely on publisher-to-subscriber relationships accommodate more branches. The evaluation shows that truncating branches with stop labels from large VTs is more effective than extending smaller VTs with individual delivery paths.

However, designing efficient VTs, especially for dynamic networks with many senders and receivers, is difficult. The dynamic nature of pub/sub systems, including evolving subscriptions and client migration, can impact VT effectiveness. To address diverse pub/sub scenarios, the next chapter examines real-world network topologies and proposes diverse VT installation strategies, considering various client distribution and migration scenarios.

Chapter 6

Computing and Evaluating Virtual Trees

Contents

6.1 Introduction

The previous chapter examined strategies for deploying *virtual trees (VTs)* customized to datacenter network topologies. While manual VT installation works for smaller, stable networks, scalability is a challenge in larger, dynamic networks. Potential migrations of publishers or subscribers add further complications, requiring either adjustments to stored trees or longer notification headers with additional routing information to extend or prune the trees. Therefore, specialized VT strategies that are independent of the underlying topology and the current distribution of publishers and subscribers are valuable for minimizing header size and network load.

This chapter introduces additional VT installation schemes designed for different real-world networks of considerable size from the freely available *Internet Topology Zoo* [70]. It broadens the scope of installation strategies and considers various levels of detail about the pub/sub network. The accuracy of the resulting VTs in their alignment with the actual communication flows depends on the level of knowledge about the network. The less knowledge there is, the less accurate the trees will be. One way to address a complete lack of knowledge is to distribute virtual trees of different sizes uniformly across the network. When the topology is known, chains of connected nodes can be identified and connected by VTs, making them easily combinable. A deeper understanding of the network is needed in order to prioritize VT installation in densely populated areas. Clients in sparsely populated areas, meanwhile, are connected via additional routing information in the notification header. Finally, detailed knowledge enables the positions of subscribers and notification frequencies to be considered, allowing VTs to be derived for each publisher. The aim of all schemes is to use VTs to cover frequently contacted subscribers, while using minimal additional routing information in the message header to connect rarely contacted subscribers to a distribution tree.

The subsequent sections are structured as follows: Section 6.2 outlines basic tree installation strategies that work without prior knowledge of the pub/sub network. Section 6.3 introduces strategies that integrate topology knowledge into the installation process and adapt to different topologies. In addition to topological insights, Section 6.4 considers client locations and publisher-to-subscriber relationships, as well as subscriber notification frequencies, to derive accurate virtual trees. These strategies apply the *hub-and-spoke paradigm* by combining large backbone trees with smaller location-based VTs. Sections 6.5 and 6.6 evaluate these VT installation schemes across various real-world topologies. Section 6.5 categorizes and evaluates strategies in representative topologies under static scenarios, whereas Section 6.6 focuses on dynamic scenarios and investigates the effectiveness of strategies by varying the subscriber churn rate. Finally, Section 6.7 provides an overview of related work on multicast trees, whereas Section 6.8 summarizes the key findings of this chapter.

6.2 Basic Trees

This section introduces basic strategies for efficiently distributing notification messages using virtual trees without any prior knowledge of the pub/sub network. These strategies are independent of network topology or client distribution. Their main objective is to uniformly cover the network with VTs, ensuring that clients at any network location can be reached via these stored trees with a minimum of additional routing information. These strategies are recommended when location and publisher-to-subscriber relationship knowledge is unavailable.

6.2.1 Broadcast Tree Strategy

The simplest strategy we employ involves creating a single *broadcast tree* and integrating it into the network switches as a VT. This tree connects all hosts across the network and requires only one flow rule per switch. Once established, this broadcast tree becomes the primary structure for notification delivery for all publishers, with unnecessary parts of the tree intelligently pruned using stops. A publisher uses the common broadcast tree instead of hop labels when a message needs to reach the majority of nodes in the network. Any branches of the broadcast tree that are not needed for delivery are truncated by stop labels.

Label Complexity

Consider a network with n nodes, including r recipient nodes ($r < n$) to be served with a notification message. In the best-case scenario, the broadcast tree strategy requires 0 stop labels and 1 VT label when all nodes except the sender are receivers. Conversely, in the worst case, the publisher and its subscribers are leaf nodes of a tree topology with the longest distance between sender and receivers. This requires a *maximum* of $(l - r) - 1$ stop labels plus 1 tree label to encode the distribution tree, where l is the number of leaf nodes and r is the number of interested subscribers. The number of stop entries can only be reduced if *several* uninterested subscribers to be excluded can be pruned from the VT with a *single* stop entry. This works if these subscribers are behind a branch in a common subtree with no interested subscriber in the same subtree. The number of labels thus ranges between 1 and $l - r$, reflecting the best and worst cases for the *broadcast tree* strategy.

In contrast, the *source-routed* approach relies only on hop labels and does not use a stored tree. In the best case, the publisher and its interested receivers r are arranged in a row, requiring r hop labels, with each hop addressing the next receiver in the row. In the worst case, the sender and its receivers are leaf nodes in a tree topology, requiring a *maximum* of $n - (l - r)$ hop labels to address r receivers, where n is the number of nodes and l is the number of leaf nodes. The number of labels thus ranges between r and $n - (l - r)$, reflecting the best and worst cases for the *source-routed* approach.

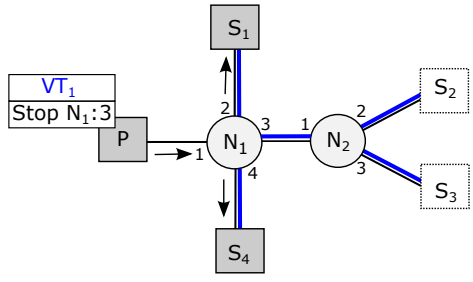
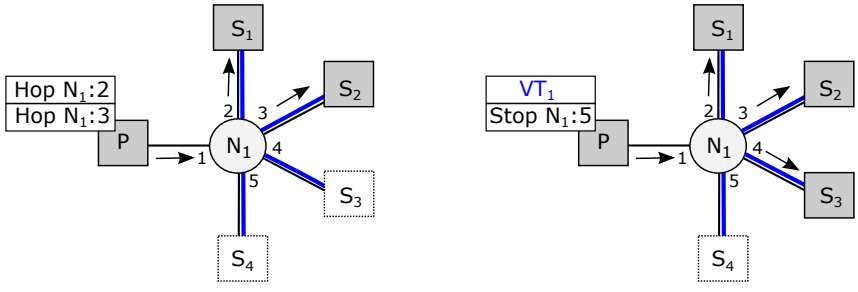


Figure 6.1: Pruning multiple subscribers from a broadcast tree by one stop label.



(a) *Scenario 1:* Addressing 2 of 4 subscribers via source-based hop encoding.

(b) *Scenario 2:* Addressing 3 of 4 subscribers via virtual tree encoding.

Figure 6.2: Addressing different receiver sets within a simple star network.

Exemplary Topologies

Figure 6.1 shows an example, where the broadcast tree spans two switches (N_1, N_2) and connects four subscribers (S_1, S_2, S_3, S_4). Each switch connects two subscribers. Publisher P communicates with S_1 and S_4, both connected to switch N_1, by encoding the broadcast label with a stop label in the message header. This stop label truncates the tree behind N_1 towards N_2, excluding subscribers S_2 and S_3 from the tree. Alternatively, the publisher could achieve the same header length and forwarding effort by encoding two hop labels in the header to reach subscribers S_1 and S_4 from its neighboring switch N_1.

Figure 6.2 illustrates another example, differing between two scenarios in a star topology. Here publisher P is directly connected to four subscribers (S_1, S_2, S_3, S_4) through a central switch. The blue edges represent a stored virtual tree.

a) *Scenario in Figure 6.2(a)*: The publisher communicates with two of the four subscribers (S_1, S_2) using *source-based hop encoding*. This method is more efficient here because it minimizes the number of header entries; using the blue VT would require additional stop entries to exclude the non-participating subscribers (S_3 and S_4).

b) *Scenario in Figure 6.2(b)*: The publisher communicates with three of the four subscribers (S_1, S_2, S_3). Here, *VT-based encoding* is more efficient, requiring only one VT label and one stop label to encode the entire distribution tree. By comparison, source-based hop encoding would necessitate three separate labels, one for each network edge.

In summary, hop-based routing is usually preferable for smaller groups of receivers, whereas the VT-based broadcast tree becomes more efficient when the majority of nodes are receivers. The usage of the broadcast tree depends on the number of subscribers being addressed.

6.2.2 Random Tree Strategy

To increase the number of available VTs arbitrarily, a simple approach would be to install multiple VTs by connecting randomly selected network nodes. This *random* strategy begins by selecting a random set of nodes, referred to as *terminal nodes*. A *Steiner tree* is then computed to efficiently connect these nodes while minimizing the total number of edges. The number and selection of terminal nodes are determined at random. The resulting tree is then integrated into the switch infrastructure by installing the corresponding flow rules. This process is repeated iteratively until the desired number of VTs has been installed or the memory capacity of the switches is exhausted – specifically, when a switch on the new VT is at full capacity and cannot store the required flow rule.

Figure 6.3 demonstrates an example network with three virtual trees generated using Steiner tree computations based on randomly selected terminal nodes. In the figure, black circles represent terminal vertices, and bold lines mark the edges of the Steiner trees. The trees share common vertices but are connected through different paths. If, for example, the first and second trees in Figure 6.3 were combined, the overlapping node subsets would form a cycle, making the merged tree impractical for message delivery.

Relevance of Tree Size

We conducted experiments using VTs of varying sizes. We did this across different network configurations. These experiments revealed the importance of minimizing excessive overlap in randomly generated VTs. Overlapping trees with shared node subsets can compromise composability by creating cycles within the network. Although stop labels can break these cycles by ensuring that each terminal node is uniquely connected to the publisher, they also increase the length of the message header. Our experiments confirmed that the likelihood of cycles forming when merging two VTs grows with the average size of the VTs due to the higher probability of overlapping node subsets.

To mitigate overlap when combining VTs, it is beneficial to control the size of the trees during their construction. This consideration is especially important

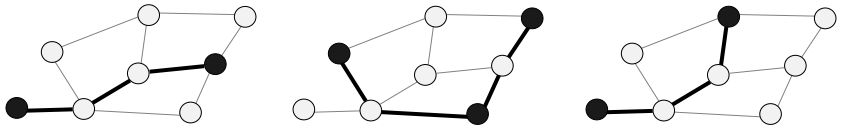


Figure 6.3: Exemplary Steiner trees created by random strategy.

in small-world topologies, where high-degree nodes connect numerous neighbors, and most nodes can be reached within just a few hops. In such densely connected networks, smaller VTs that span fewer edges are preferable to larger ones to reduce redundancy and prevent overlap.

To address this challenge, we propose the *random-variant* strategy, which introduces constraints on tree size during the construction process. By limiting the size of individual VTs, this variant minimizes the likelihood of overlap and enhances the composability of the trees. Details of this refined strategy are presented in the following section.

Determining a Meaningful Tree Size

The *random-variant* strategy refines the *random* strategy by dynamically adjusting the size of the VTs to be installed. This process begins with the construction of a large spanning tree that encompasses all network nodes, similar to a broadcast tree. Subsequently, the tree sizes are gradually reduced for the subsequent VTs to be installed.

This controlled downsizing aligns with the encoding algorithm described in Section 5.2.7, which seeks to maximize the replacement of hop labels with tree and stop labels. Initially, the encoding algorithm selects a relatively large VT to replace the maximum number of hop labels with a single tree label and the necessary stop labels, providing a significant shorter header. In later iterations, progressively smaller VTs are chosen to replace any remaining hop labels with a minimal number of tree and stop labels. By targeting smaller subtrees in later iterations, the strategy reduces the likelihood of interference with previously installed VTs, minimizing the probability of cycle formation and the associated need for additional stop labels. However, this comes at the cost of diminishing optimization gains, as smaller virtual trees inherently replace fewer hop entries.

Figure 6.4 illustrates the *random-variant* strategy applied to an example network with eight nodes. The network is logically represented as a graph $G = (V, E)$, where V denotes the vertices (switches) and E denotes the edges (links) connecting these vertices. Each vertex corresponds to a switch connected to a host, which may function as a publisher, subscriber, or both.

The strategy begins by constructing a spanning tree encompassing all vertices in V. This tree serves as the largest VT in the sequence and functions as the baseline for subsequent partitioning.

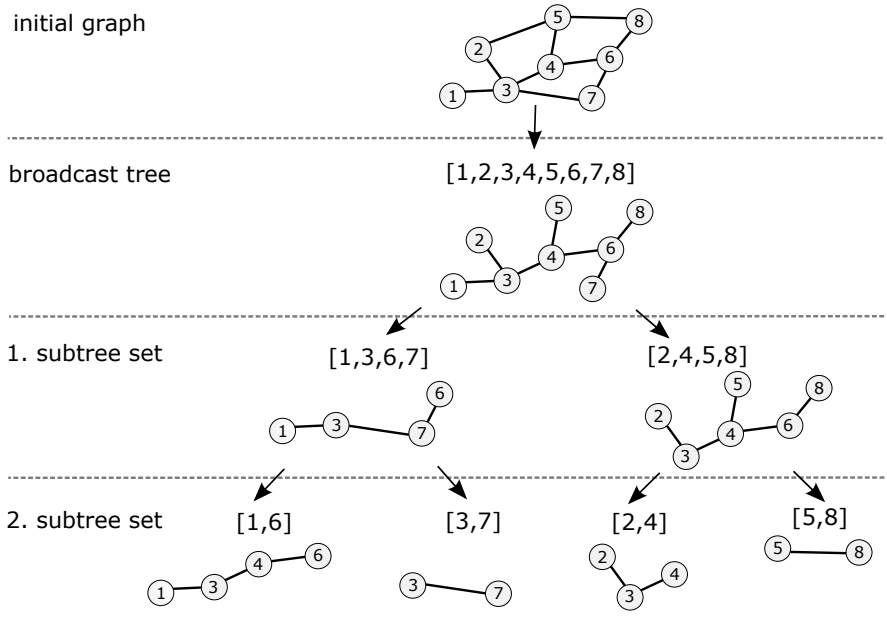


Figure 6.4: Deriving random trees of different sizes.

After constructing the initial broadcast tree, V is divided in a series of steps to create progressively smaller VTs. Each step halves the number of terminal nodes per VT while doubling the number of VTs to be installed. This iterative subdivision produces a set of VTs that cover smaller, distinct subsets of nodes and edges within V, ensuring reduced overlap and increased combinatorial flexibility for the encoding algorithm. The subdivision continues as long as each VT connects at least two terminal nodes and contains at least two edges – the minimum installation requirement.

When constructing the delivery tree, the encoding algorithm actively avoids merging VTs that would result in cycles. For example, merging the two VTs from the first partitioning step would create a cycle with an edge set of $E' = \{(3,7),(7,6),(6,4),(4,3)\}$, where $E' \subset E$. This would require an additional stop entry to break the cycle. Furthermore, the VTs generated by the terminal nodes $[3,7]$ and $[5,8]$ from the second partitioning step each contain only one edge. This is fewer than the minimum required, so these are not installed as VTs.

The *random-variant* strategy allows modifications to the downsizing process and tree installation criteria. Alternative implementations may vary the maximum and minimum number of terminal nodes per VT or adjust the size reduction steps between iterations. This flexibility is beneficial for adapting to specific network characteristics. For example, networks with large diameters and low hub node densities may benefit from larger VTs that cover broader areas. Conversely, networks with small diameters and high hub densities may benefit from

smaller VTs, which are better suited to the denser interconnectivity and reduce unnecessary overlap.

6.3 Topology-aware Trees

The *random* strategy discussed above does not consider the topological properties of the network, resulting in an irregular distribution of VTs across network nodes. This can result in two significant problems:

1. *Overlapping VTs:* Certain network regions may become densely covered by overlapping VTs. When these VTs are combined into a delivery tree, the overlaps can create cycles. Additional stop labels are required to break these cycles, which increases the header size and reduces the efficiency of VT combinations.
2. *Uncovered regions:* Other regions may lack VT coverage entirely, requiring additional hop labels to bridge uncovered areas of the distribution tree. This is particularly problematic in networks with widely separated VTs, as it leads to longer headers and increased processing overhead.

To address these shortcomings, this section introduces advanced *topology-aware* strategies that exploit the network's structural properties to construct VTs. These strategies aim to overcome the limitations of the *random* strategy by promoting seamless VT concatenation. They achieve this by identifying chained nodes or grouping densely connected nodes to derive VTs of different sizes: small local VTs to connect group members, and larger network-wide VTs to connect the groups. By considering the underlying network topology, these strategies minimize tree overlaps and gaps, improving VT composability and reducing the need for hop and stop labels.

6.3.1 Cycles Strategy

The first topology-based approach, the *cycles* strategy, is designed to exploit cyclic patterns within the network topology for virtual tree construction. This strategy creates dedicated VTs for each identified cycle and their combinations, capitalizing on the fact that sequentially arranged clients within a cycle can be covered by a single tree. The resulting VTs can be seamlessly interconnected at hub nodes, requiring only a single stop entry per tree to terminate forwarding after the last receiver in the chain.

The number of VTs installed at a switch is directly proportional to the count of simple cycles in the topology. Specifically, for n simple cycles, up to $2^n - 1$ trees are required to account for all possible cycle combinations. For example, in the topology illustrated in Figure 6.5, there are 3 simple cycles: (a, b, e, c, a),

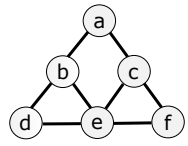


Figure 6.5: Topology with 3 simple cycles and a total of 7 cycles.

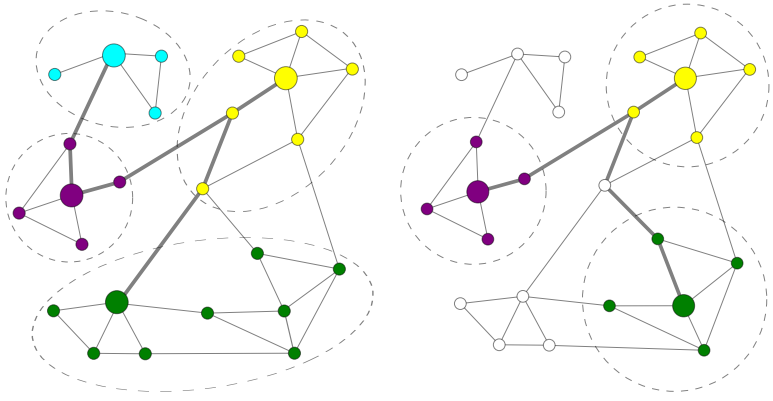
(b, d, e, b), and (c, e, f, c). If all combinations of these cycles are considered, the total number of VTs would be $2^3 - 1 = 7$. However, the *cycles* strategy prioritizes installing VTs for individual simple cycles first and only considers their combinations if sufficient switch memory is available.

This strategy optimizes VT placements by exploiting cyclic structures to promote VT concatenation. However, it has a notable drawback: receivers located outside the identified cycles must be connected to the delivery tree via additional hop labels, increasing complexity. Furthermore, in topologies with non-existent cyclic structures, such as tree topologies, the *cycles* strategy defaults to source-based hop encoding. For these topologies, alternative strategies are more practical, as outlined next.

6.3.2 Partitions Strategy

An alternate topology-based strategy, the *partitions* strategy, leverages the *hub-and-spoke paradigm* [111] to construct hierarchical virtual trees. In this paradigm, the entire network is segmented into partitions, each anchored by a dedicated hub node and covered by its own *partition-internal VTs*. When a publisher sends a notification to interested subscribers that all belong to the same partition, the message is delivered directly to the receivers via the VTs within that partition. For messages destined for recipients in other partitions, on the other hand, the notification passes through a number of *hub nodes*, following indirect paths rather than direct paths from the publisher to the subscribers. Hub nodes are interconnected by backbone VTs and support network-wide communication. The notification is first routed to the hub of the source partition and then passed through a sequence of hubs via a backbone tree, with each hub acting as a gateway to another partition. Finally, the hub of a target partition distributes the notification to the final recipients using the internal VT of the target partition. This hub-and-spoke approach supports message delivery to multiple destinations across multiple partitions.

Multiple VTs are established within each partition. Each VT covers various node permutations within a partition, including the hub node. The resulting VTs are constrained by the partition boundaries. They therefore have a small diameter and avoid overlapping with VTs from other partitions. For wider network communication, the strategy also creates larger backbone VTs connecting



(a) Partitions strategy. (b) Clusters strategy.

Figure 6.6: Clusters vs. Partitions strategy.

permutations of the partition hubs. Because each partition-internal VT contains the partition's hub node, simple composition with a backbone VT to form a comprehensive delivery tree spanning multiple partitions is supported.

The strategy begins by grouping nodes into non-overlapping sets using an *agglomerative clustering method* [31]. This method decomposes the network into a hierarchy of groups based on a given weighting function, prioritizing strongly connected nodes with, for example, many edges between them. After grouping, a hub node is chosen for each group by selecting the best candidate switch. During this selection process, switches within a partition are ranked, and their importance is determined using centrality indicators.

There are a number of different centrality measures, each with a different definition of the importance of a node. Examples include *betweenness centrality*, which emphasizes the number of (shortest) paths passing through a given node, and *degree centrality*, which measures the importance of a node based on the number of direct edges it has to other nodes. A wide range of parameterized network centrality measures are explored in [11]. In particular, we adopt *degree centrality* for hub selection due to its alignment with the hub-and-spoke paradigm and its computational simplicity. Our findings indicate that other centrality indicators either provide comparable results or perform suboptimal, resulting in the selection of less suitable nodes as hubs and consequently increasing communication overhead. The hub-and-spoke-based VTs improve the efficiency of notification delivery in networks characterized by high-degree nodes.

Figure 6.6(a) illustrates a sample network divided into four partitions of varying sizes. Each partition has a hub (depicted as larger nodes) connected by a backbone VT (represented as thicker lines). Within each partition, smaller local VTs connect nodes to the local hub, remaining confined within their respective partition boundaries.

Computational Efficiency

The computational efficiency of the partitions strategy depends on the choice of the clustering method in the initial step. Generally, *k-means* clustering and *hierarchical* cluster analysis are widely recognized as efficient methods for dealing with large networks [123]. However, k-means clustering requires the number of clusters k to be pre-specified, which poses a challenge as determining the optimal value k for a given topology can be computationally intensive. To avoid this challenge, we decided to use hierarchical cluster analysis for our approach.

Several hierarchical clustering techniques exist to identify *communities* in networks. Jarman [61] offers a comparative analysis of these methods. Among the simplest and best known are the *single-linkage* and *complete-linkage* methods, also known as graphical methods. The single-linkage method, or nearest neighbor method, computes the distance between clusters based on the closest pair of nodes from different clusters. Conversely, the complete linkage method uses the maximum distance between any pair of nodes from different clusters to determine the inter-cluster distance. Since these graphical methods calculate the distance between clusters by considering all points within each pair of clusters, their computational complexity can become a limiting factor for large networks. In particular, these methods require $O(n^2)$ time complexity for pairwise node comparison, where n is the number of nodes.

To mitigate the computational complexity and enable hierarchical clustering for large networks, approximation methods can be employed to reduce the complexity of computing pairwise distances between clusters. Specifically, we adopt the *agglomeration algorithm* proposed by Clauset et al. [31]. This algorithm uses a greedy optimization approach to efficiently identify a hierarchical network decomposition into communities, aiming to maximize the global modularity Q as a metric for decomposition quality. The algorithm initially treats each node as an individual community and progressively merges the two communities that provide the largest increase in Q. After $n-1$ merges, the result is presented as a dendrogram computed with a time complexity of $O(md \log n)$, where n is the number of vertices, m is the number of edges, and d is the depth of the dendrogram. This algorithm ensures an effective partitioning of the network into communities, from which we derive VTs for use in our *partitions* strategy.

6.4 Considering Client Distribution

Previous strategies do not consider the distribution of clients, ignoring the spatial arrangement of publishers and subscribers (clients). However, pre-computing virtual trees for an entire network without considering actual client distributions can lead to VTs that do not closely reflect actual communication patterns, requiring additional labels when encoding the distribution tree. This issue can be mitigated by taking actual *client distributions* into account when pre-computing

trees. By considering client positions, more accurate VTs can be strategically built in regions with dense client populations, connecting these high-density areas with additional backbone trees. Even more nuanced trees are possible if *publisher-to-subscriber relationships* and *subscriber notification frequencies* are known. With this knowledge, publisher-rooted trees can be created that focus on frequently contacted subscribers. Below we present strategies that exploit client position information and statistical measurements to derive meaningful VTs based on these principles.

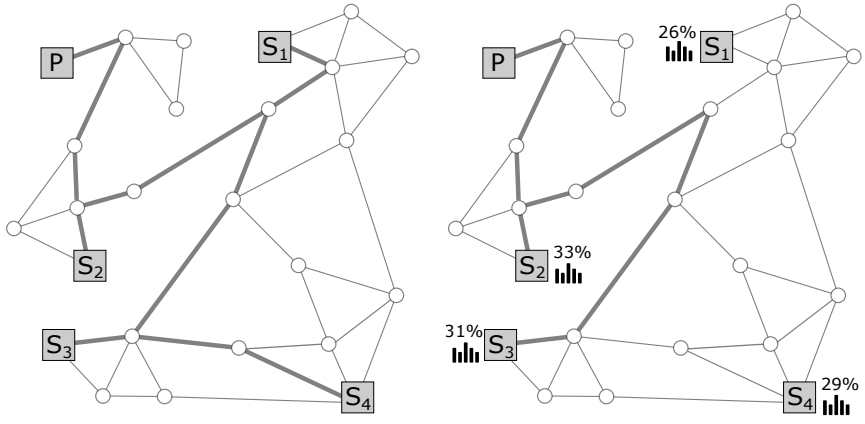
6.4.1 Clusters Strategy

The *clusters* strategy is a variation of the *partitions* strategy. It leverages the geographic distribution of publishers and subscribers, focusing VT installations in network regions with higher client population densities. In areas of the network with significantly higher notification traffic, the *clusters* strategy uses statistical measurements to estimate the location and size of these 'client clusters' and preinstalls VTs exclusively in these high-density regions. The underlying assumption is that links within clusters will experience higher traffic rates, meaning these areas should be prioritized for VT optimization.

Figure 6.6(b) illustrates an example network with clusters represented by different colors. Similar to the *partitions* strategy (see Fig. 6.6(a)), each cluster is covered by a dedicated set of small VTs, while hub nodes within each cluster (depicted as larger vertices) are interconnected by backbone VTs to enable inter-cluster communication. These hub nodes are identified using a centrality indicator, specifically *degree centrality*.

By prioritizing high-traffic regions, publishers can use a more extensive rule base within densely populated clusters, minimizing the additional hops or stops required for message delivery. This localized concentration of VTs reduces resource usage and avoids overlapping VTs that could create cycles during tree combinations. However, unlike the *partitions* strategy, the *clusters* strategy omits VT coverage in sparsely populated areas, leading to higher probabilities of requiring additional hop entries for delivery to outlying clients. This trade-off can increase header sizes for clients outside cluster boundaries.

The effectiveness of the *clusters* strategy relies on accurately identifying client clusters, which may require continuous monitoring and adjustment as network dynamics evolve. If significant shifts in publisher or subscriber distribution occur – such as migrations to previously low-density regions – the client clusters must be redefined to maintain efficiency. Inappropriate recognition of clusters could result in larger message headers and increased network load due to VT adjustments. In contrast, the *partitions* strategy provides broader coverage by dividing the entire network into partitions, encompassing all nodes rather than focusing solely on high-density regions. While this ensures uniform VT availability across the network, it lacks the localized optimization of the *clusters* strategy.



(a) Relation strategy. (b) Frequency-relation strategy.

Figure 6.7: Relation vs. freq-relation strategy.

In summary, the *clusters* strategy generates short message headers when client populations remain concentrated in predictable, high-density regions. However, its performance can degrade in scenarios with highly dynamic subscriber distributions, where population densities are unstable or challenging to predict, resulting in increased header sizes and network load.

6.4.2 Relation and Frequency-Relation Strategy

A more advanced, knowledge-driven strategy builds on the principles outlined in Section 5.3.4, leveraging publisher-to-subscriber relationships to construct highly efficient virtual trees. This is the *relation* strategy. It takes advantage of the intrinsic characteristics of pub/sub systems, where publishers announce the notifications they will produce and subscribers declare their interests using filters. By analyzing and matching these advertisements and subscriptions, overlapping filter expressions are identified, enabling the creation of relationship trees that closely mirror the underlying pub/sub structure. Figure 6.7(a) illustrates an example VT that connects a publisher P to four subscribers (S_1, S_2, S_3, S_4) whose subscriptions overlap with the publisher's advertisement.

Building upon this foundation, the *frequency-relation* strategy refines the tree construction process by integrating notification frequency data. As outlined in Section 5.3.6, this strategy measures the notification frequency for each subscriber. A fixed *threshold* is then applied to filter out rarely contacted subscribers. Only those whose notification frequency meets or exceeds this threshold are included in the final VT, ensuring that the virtual tree focuses on the most frequently addressed endpoints.

These optimized virtual trees are constructed using a Steiner tree algorithm,

where the publisher and high-frequency subscribers serve as terminal nodes. For instance, Figure 6.7(b) depicts a refined version of the tree from Figure 6.7(a), where subscribers S_1 and S_4 have been pruned based on a 30% threshold. Subscribers must receive at least 30% of all published notifications to remain in the tree; those falling below this threshold (e.g., S_1 at 26% and S_4 at 29%) are excluded.

The threshold choice directly affects the size of the resulting virtual trees. Lower thresholds result in larger VTs that cover more subscribers, while higher thresholds generate smaller, more focused VTs that mainly serve subscribers with a high contact frequency. As analyzed in Sect. 5.4, larger virtual trees tend to allow more compact headers for message delivery because a single stop entry can effectively prune an entire branch of a virtual tree. In contrast, adding a missing branch requires multiple hop entries.

While frequency-based virtual trees are efficient in scenarios where recipient sets remain stable over time, their efficiency declines in dynamic scenarios with subscriber churn. Regularly updating the rule base to adapt to these changes can be impractical and impose significant administrative and computational overhead.

To mitigate this limitation, *freq-relation* trees are best deployed in stable regions of the network where subscription patterns remain relatively unchanged. This ensures sustained efficiency without the need for VT updates, minimizing both header length and network load. In dynamic environments, alternative strategies such as the *random* strategy or *partitions* strategy are more effective in dealing with churn scenarios.

Installation Steps

To build *freq-relation* trees, an SDN controller can serve as the coordinator. The process involves the following steps:

1. *Registration and initial tree setup:* Publishers and subscribers register with the SDN controller, enabling it to acquire knowledge of the locations of all notification sources and sinks in the network. The controller constructs endpoint sets for Steiner trees, which serve as initial coarse-grained trees based on relationship information about advertisements and subscriptions. The required forwarding rules are systematically installed on the switches, and publishers are informed about these initially stored trees.

2. *Runtime monitoring and tree adaptation:* During runtime, the controller monitors the subscribers and updates their notification frequency counters, identifying frequently contacted subscribers based on the predefined threshold. The controller uses this monitoring data to create new, reduced endpoint sets, resulting in trimmed versions of the original trees. These frequency-based trees are systematically installed on the network switches through dedicated flow rules.

3. *Switching to frequency-relation strategy:* The controller informs the publishers about the refined trees and deactivates the original trees. Publishers then switch to using the frequency-based trees, allowing the controller to remove the larger original trees.

With these steps, an SDN controller can effectively manage and optimize the deployment of VTs, balancing efficiency with adaptability to network dynamics.

6.5 Evaluation of Static Scenarios

We evaluated the proposed virtual tree strategies through two types of scenarios: (i) *static scenarios* with fixed client locations and subscriptions, and (ii) *dynamic scenarios* with churn, where publishers and subscribers (clients) relocate, altering their network positions, notification content, and subscription filters. This section focuses on the static scenarios, examining and analyzing the strategies across three distinct real-world networks with varying characteristics. Section 6.6 deals with the evaluation of dynamic scenarios with churn.

In this section, we first analyze a set of real-world networks to select representative networks that differ in their characteristics. We then examine the composition of the resulting delivery trees for static scenarios simulated in these representative networks. This involves a detailed analysis of the individual header stacks, consisting of the *tree*, *stop*, and *hop* labels. Subsequently, we evaluate the effectiveness of the tree installation strategies by comparing the number of header entries required for each strategy for notification delivery within the representative networks. The results are shown as graphs showing the best strategy for each type of network.

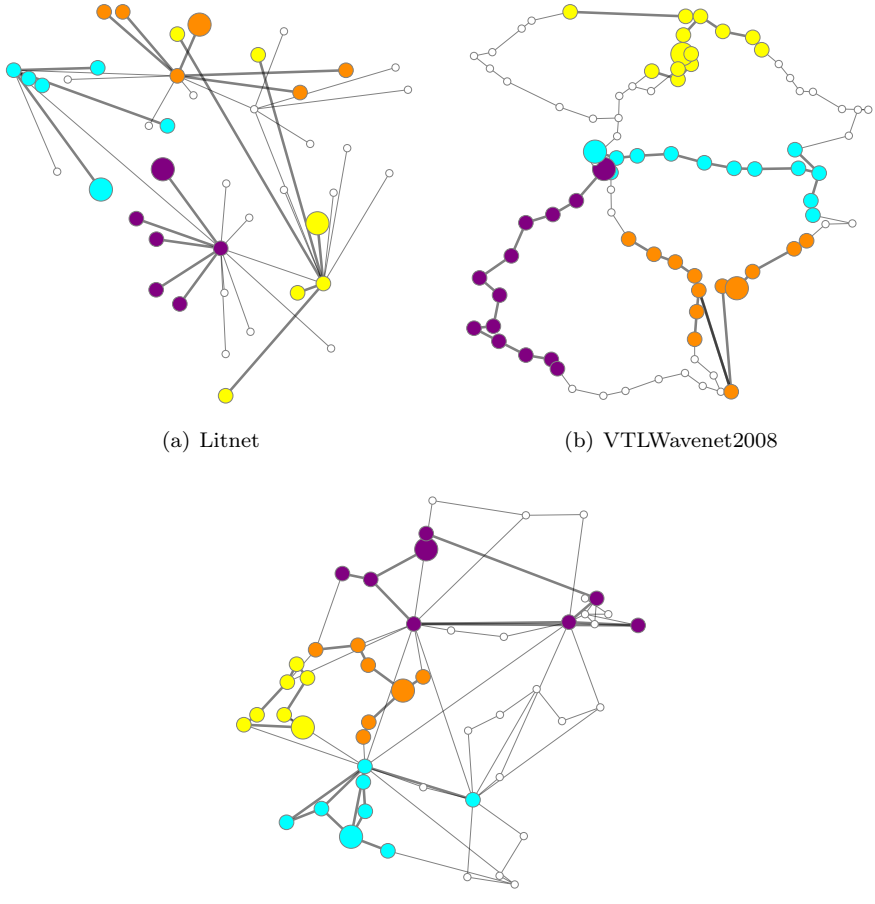
6.5.1 Internet Topology Zoo

To evaluate our proposed strategies, we first examined 25 real-world networks obtained from the *Internet Topology Zoo* [70], an ongoing project that collected more than 250 network topologies from various locations worldwide. We focused on networks of significant size, particularly those with more than 40 nodes. Table 6.1 provides an overview of the topologies, including the following key characteristics: *topology name*, *node count* ($|V|$), *edge count* ($|E|$), *average node degree* ($\langle k \rangle$), *node degree fluctuation* (σ_k^2), and *network diameter* (d).

We classified the networks based on the last two columns, i.e., (i) *node degree fluctuation* and (ii) *network diameter*. The node degree fluctuation describes the variation in the number of links to neighboring nodes, whereas the network diameter denotes the maximum number of hops required to traverse from one *satellite node* to another. Based on these two characteristics, we selected three topologies for detailed analysis: `Litnet`, `VtlWavenet`, and `DFN`. These three candidate topologies are illustrated in Figure 6.8.

Topology Name	$\|V\|$	$\|E\|$	$\langle k \rangle$	σ_k^2	d
Chinanet	42	66	1.5	10.52	4
Litnet*	43	43	0.98	5.04	4
Cernet	41	58	1.32	5.6	5
Ntt	32	65	1.48	7.07	6
Cesnet200706	44	51	1.16	6.27	6
Carnet	44	43	0.98	5.48	6
Dfn*	50	78	1.77	5.31	6
Telcove	71	70	1.59	9.13	7
Forthnet	62	62	1.41	7.72	7
Bellsouth	51	66	1.5	7.55	7
Garr200902	54	68	1.55	5.13	7
Arnes	41	57	1.3	4.53	7
BeyondTheNetwork	53	65	1.48	3.98	7
Uunet	49	84	1.91	7.38	8
Tw	71	115	2.61	5.58	8
Uninett	71	97	2.2	3.12	9
Renater2010	43	56	1.27	3.08	9
Surfnet	50	68	1.55	3.36	11
Iris	51	64	1.45	2.16	11
Palmetto	45	64	1.45	2.57	12
BtLatinAmerica	45	50	1.14	1.87	12
Bellcanada	48	64	1.45	2.59	13
Sanet	43	45	1.02	1.66	13
LambdaNet	42	46	1.05	1.57	13
HiberniaGlobal	55	81	1.84	2.72	16
Ntelos	47	58	1.32	1.92	17
RedBestel	84	93	2.11	0.85	28
VtlWavenet2008*	88	92	2.09	0.11	31

Table 6.1: Tested topoplogies.



(a) Litnet

(b) VTLWavenet2008

(c) DFN

Figure 6.8: Topologies with predefined clusters.

We selected `Litnet` and `VtlWavenet` as they represent the extremes on the quantified feature scale, while `DFN` combines aspects of both. In fact, `Chinanet` has similar characteristics to `Litnet`. However, we have a preference for the latter, as this configuration is less complex to comprehend.

The selected networks are characterized as follows:

(i) `Litnet:` A scale-free network with a high node degree variation and a small network diameter. The high density of high degree nodes (hubs) contributes to short path lengths between any two nodes.

(ii) `VtlWavenet:` Characterized by a large network diameter and composed of extensive ring structures. Within these rings, most nodes have two neighbors. Only a few hub nodes have a significantly higher number of links connecting the rings. The low density of hubs and the large ring size result in longer average path lengths between nodes.

(iii) `DFN:` A hybrid network combining features of both `Litnet` and `VtlWavenet`. It includes central hubs that connect many neighbors and additional alternative paths beside the hubs. The hubs contribute to a small network diameter, while the redundant paths provide multiple connection options.

6.5.2 Simulation Setup

We evaluated our strategies through simulations conducted using *Mininet* [73] and *BMv2* [10], following the evaluation methods established in previous chapters. In our simulated network, each node represents a switch connected to a single host that can act as a publisher and/or subscriber. To simulate authentic pub/sub scenarios, we randomly select 30% of the hosts as publishers and, for each publisher, we randomly select 30% of the hosts as subscribers.

To account for different client population densities, we introduce a distinction between *densely* and *sparsely* populated regions within the simulated network. This is done by distributing publishers and subscribers in a clustered pattern with four distinct clusters. As an example, Figure 6.8(c) shows a clustering result for to the *German Research Network (DFN)*. Therefore, node-specific probabilities are computed during network initialization to ensure preferential selection of certain nodes over others, with high and low probabilities assigned to nodes inside and outside clusters, respectively.

The differentiation between *dense-* and *sparse-populated* areas is achieved through a three-step process during network initialization:

1. *K-means clustering:* Initially, we group all switches in the network using the *k-means* clustering algorithm [5], which divides a set of samples into disjoint groups, with each group represented by the mean of the samples within it. These means, called *centroids*, typically do not correspond to actual sample points.

2. *Center node selection:* Following the clustering step, we identify a *center node* for each group. A center node is a sample point that has the smallest distance from the centroid of its corresponding group.

3. *Population density differentiation:* Finally, for each group, we classify a subset of nodes as densely populated using a *breadth-first search* starting from the center node. Specifically, we expand these subsets in a round-robin manner until a predefined percentage (60%) of nodes are classified as densely populated, while the remaining nodes (40%) are classified as sparsely populated.

This process makes it more likely that publishers and subscribers will reside on hosts close to the center nodes, while hosts further away from the cluster centers are less likely to perform these roles. For clarity, Figure 6.8 illustrates the clustering results for the three candidate networks, dividing each one into four clusters. Densely populated nodes are color-coded, with each color representing a distinct cluster group. The center nodes are visually distinguished by their larger size. This strategic differentiation of population density within our simulated network enables us to realistically model different scenarios and comprehensively evaluate our strategies under different conditions.

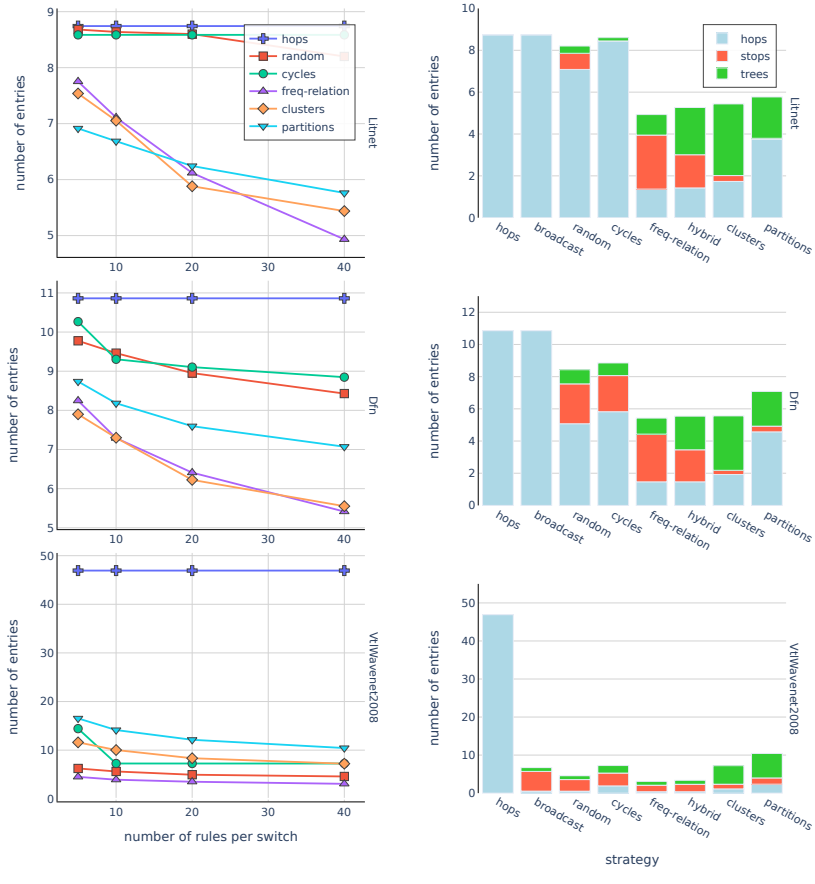
6.5.3 Average Header Load

In the following analysis, we perform a detailed comparative evaluation of the *average header load* for our different strategies. First, we examine the resulting header size for varying numbers of rules per switch. Then we examine the specific composition of the header stack labels, consisting of *tree*, *stop* and *hop* labels, for a fixed number of rules per switch.

Results for varying Rules Count

Figure 6.9(a) presents the simulation results for the three candidate topologies (`Litnet`, `DFN`, and `VtlWavenet`) as line charts. Each chart corresponds to a specific topology and displays the average number of tree, stop, and hop labels encoded in the notification header under varying limits on the number of rules per switch.

The *hops* strategy, represented as the topmost line in each chart, serves as a baseline by exclusively encoding the distribution tree using hop entries. This reference strategy ensures an unbiased comparison with alternative strategies. Since the *hops* strategy does not use stored trees, its simulation results remain unaffected across all maximum allowed rule counts. In contrast, all other strategies employ virtual trees to reduce notification header size, making their performance dependent on the number of rules available in the switch memory.



(a) Results from 5 up to 40 rules/switch. (b) Detailed results for 40 rules/switch.

Figure 6.9: Average number of routing header entries in static scenario.

As expected, the number of entries required decreases as the number of allowed flow rules increases. However, the observed trend varies depending on the specific network topology. For instance, in `VtlWavenet`, most strategies show a slight decline in header size after an initial drop but remain significantly lower than the hops baseline. Most strategies produce good results from 10 rules per switch onwards. This indicates that a small number of rules are sufficient to achieve compact headers, with additional rules offering only marginal gains.

Conversely, for the `DFN` and `Litnet` topologies, performance continues to improve even beyond the 10 rules per switch limit. In these topologies, the installation of additional VTs continues to reduce header size. The *clusters* and *freq-relation* strategies deliver the best results, despite their distinct VT placement methodologies. While the *clusters* strategy focuses on dense client populations, the *freq-relation* strategy prunes large publisher-centric VTs to focus on frequently contacted subscribers. The *freq-relation* strategy achieves best results at 40 rules per switch, where it is able to map (almost) all publisher-rooted trees into switch memories. It also achieves the best performance in `VtlWavenet`, demonstrating its adaptability to different network scenarios.

Interestingly, the performance of the *cycles* strategy is outperformed by the *random* and *freq-relation* strategies, despite exploiting the ring-based structure of `VtlWavenet`. This suggests that merging rings to form a distribution tree is less effective than creating larger VTs that cover the entire network. The *random* strategy performs exceptionally well in `VtlWavenet`, but falls short in `DFN` and `Litnet`.

Results for High Rules Count

Figure 6.9(b) provides a detailed breakdown of strategy performance in the form of bar charts. These bar charts correspond to the 40 rules per switch point, which is the maximum x-value in the line charts of Figure 6.9(a). Each bar represents a strategy and shows the average composition of a delivery tree, illustrating the proportions of *hop*, *stop* and *tree* labels used:

a) *Hops*: The *hops* strategy encodes the distribution tree exclusively with hop labels. Since this strategy does not use virtual trees, it produces the largest headers. In contrast, all other strategies leverage VTs to optimize the header size by combining different label types.

b) *Random*: The *random* strategy requires a significant number of hop labels in the `Litnet` and `DFN` topologies. This indicates significant gaps in VT placement that require bridging with additional hop labels. Interestingly, the *random* strategy uses an average of only one VT to encode the delivery tree across all topologies – a feature it shares with the *freq-relation* strategy.

c) *Cycles*: Despite being designed for ring-structured networks, the *cycles* strategy performs worse than all the other strategies in `VtlWavenet`. This means that ring-based trees do not pay off.

d) *Frequency-relation*: The *frequency-relation* strategy generates the smallest headers across all topologies, achieving a higher proportion of stop labels relative to hop labels. This reflects a tendency to prune VTs more frequently than to extend them. The balance between stop and hop labels is influenced by the notification frequency threshold, which is set to 30% in this scenario. This threshold results in relatively large VTs that can be efficiently pruned, leading to compact headers.

e) *Hybrid*: The *hybrid* strategy balances the strengths of the *frequency-relation* and *clusters* strategies by installing VTs generated by both approaches in a round-robin manner. This mixed approach splits switch memory between the two strategies, producing results that fall predictably between the performance levels of the individual strategies.

f) *Clusters*: The *clusters* strategy uses the highest number of tree labels and the fewest stop labels among all strategies. For instance, in `Litnet`, it constructs delivery trees using an average of between 3 and 4 tree labels. This is because the strategy focuses on densely populated client regions, resulting in smaller VTs compared to other strategies.

g) *Partitions*: The *partitions* strategy uses fewer VTs than the *clusters* strategy in `Litnet` and `DFN`, but requires more hop labels to reach all subscribers. In `VtlWavenet`, it encodes the highest number of tree labels among all strategies, but performs the worst overall. This means that small VTs are less effective in large-diameter networks with few hubs.

The results show how the stack header size and the ratio of label types vary between the topologies. While the *frequency-relation* strategy generates small headers in all topologies due to its focus on frequently contacted subscribers, other strategies such as *clusters* generate compact headers only in small-world topologies. Conversely, the *random* strategy performs well in networks with few hubs and a large diameter. These findings highlight the importance of choosing a strategy that aligns with network characteristics and switch memory constraints.

6.5.4 Structure of Distribution Trees

To better understand the functionality of our strategies, we conducted an in-depth analysis of the *distribution tree (DT) structures* as part of our experiments. This analysis aimed to evaluate the working principles of the different VT-based strategies. For this purpose, we developed a custom visualization module capable of generating graphical representations of the DT structures.

The module takes the network topology and the message-specific labels generated by the encoding algorithm (described in Sect. 5.2.7) as input. It then produces visual outputs that annotate the topology's edges with the corresponding DT labels. To enhance clarity, the visualization employs distinct colors and shapes to represent different types of labels.

The generated graphs illustrate two key versions of the distribution tree:

(i) *Initial DT:* Encoded exclusively with hop labels, representing the baseline structure before applying any VT-based optimizations.

(ii) *Optimized DT:* Refined by the selected VT strategy, where hop labels are replaced with tree and stop labels to minimize the message header size.

The visualizations show how each VT strategy transforms the initial DT into a shorter header stack, replacing a large number of hop entries with a small number of tree and stop entries. To simulate realistic conditions under practical memory constraints, we set a limit of 20 flow rules per switch for the simulations.

Exemplary Receiver Set in DFN

Figure 6.10 illustrates how four different strategies – *hops*, *freq-relation*, *clusters*, and *partitions* – encode a distribution tree (DT) in the `DFN` topology, connecting a publisher P to seven subscribers (S_1, ..., S_7).

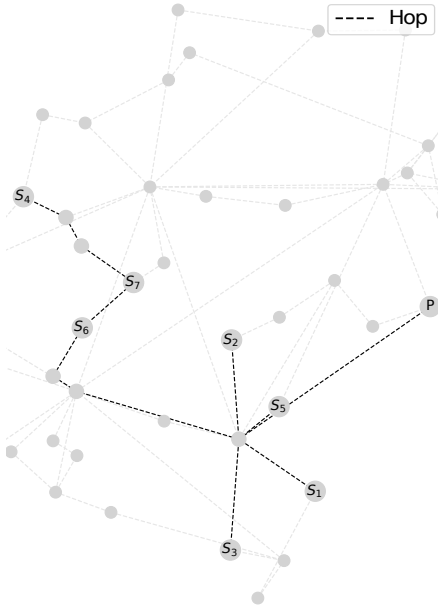
The baseline DT, depicted in Fig. 6.10(a), is produced by the *hops* strategy. This Steiner tree-based DT is encoded entirely by hop labels, represented by black dashed lines. All other strategies shorten the header stack of this *initial DT* by replacing hop labels with tree and stop labels, which can potentially restructure the DT along worthwhile VT edges.

Figures 6.10(b) to 6.10(d) show the optimized DTs for the *freq-relation*, *clusters*, and *partitions* strategies. Solid colored edges in these figures represent VT-based encoding, while black dotted lines indicate stop entries. Both *freq-relation* and *partitions* restructure the initial DT by choosing alternative network paths to better align with the installed VTs, thereby minimizing header entries.

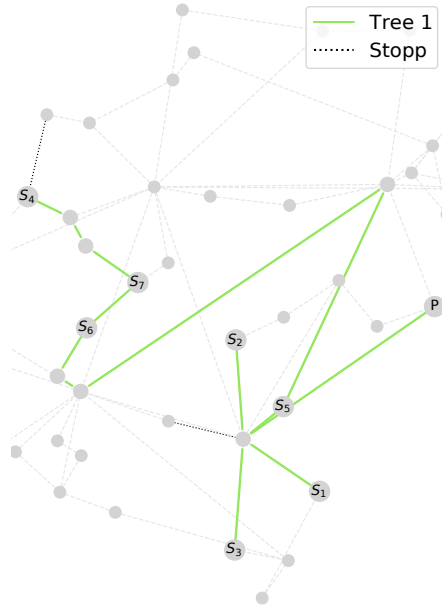
The *freq-relation* strategy (Fig. 6.10(b)) encodes the DT using a single large VT ($Tree_1$), pruned by just two stop entries to limit message propagation beyond the intended subscribers. The *freq-relation* strategy encompasses the entire DT within a single tree, achieving compact headers with minimal additional labels.

The *clusters* strategy (Fig. 6.10(c) divides the DT into three VTs ($Tree_1$, $Tree_2$, and $Tree_3$), reflecting internal VTs of hotspots or backbone VTs connecting the hotspots. The strategy also requires an additional hop entry to reach an 'outlying' subscriber. These gaps in coverage may be introduced by reliance on smaller, localised VTs, and these gaps must be filled with additional hop entries.

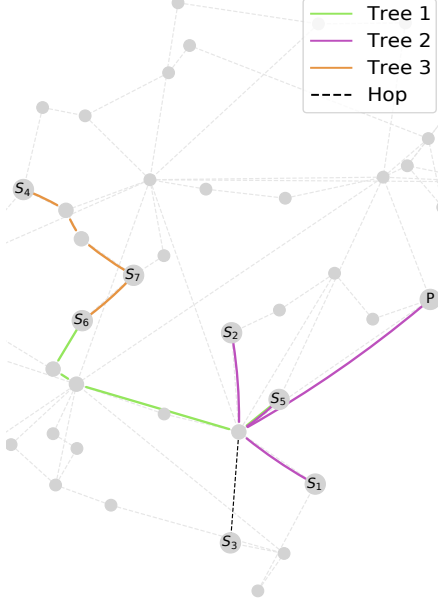
The *partitions* strategy (Fig. 6.10(d)generates the longest header stack among the optimized DTs. It encodes four tree labels ($Tree_1$ to $Tree_4$) and an additional stop label to prune a branch. The strategy splits the network into larger, non-overlapping regions. Unlike the clusters strategy, the resulting VTs cover all nodes, thus providing stored VTs with broader network coverage. However, this broader coverage may require more additional stop labels to trim the VTs.



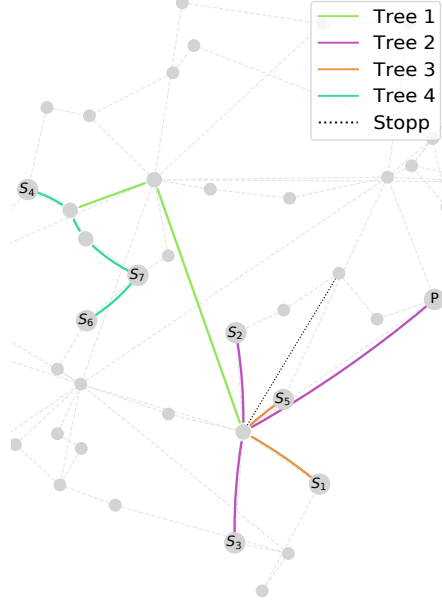
(a) Hops.



(b) Frequency-Relation.

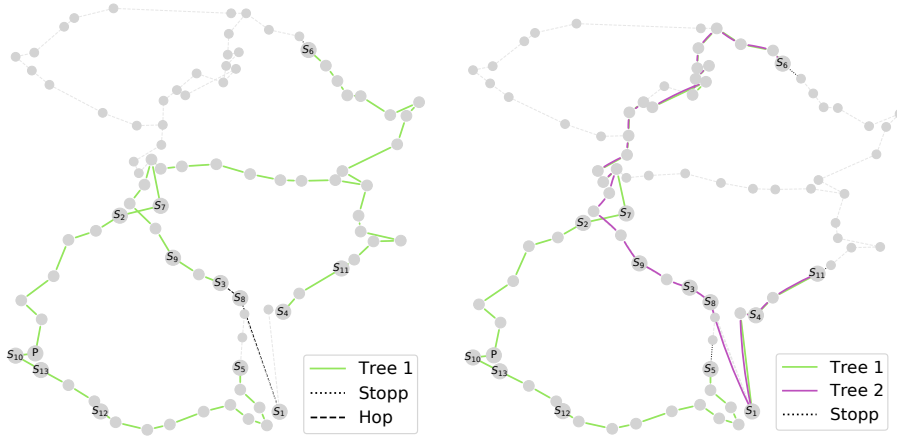


(c) Clusters.



(d) Partitions.

Figure 6.10: Exemplary distribution trees in DFN.



(a) Random. (b) Cycles.

Figure 6.11: Exemplary distribution trees in `VTlWavenet`.

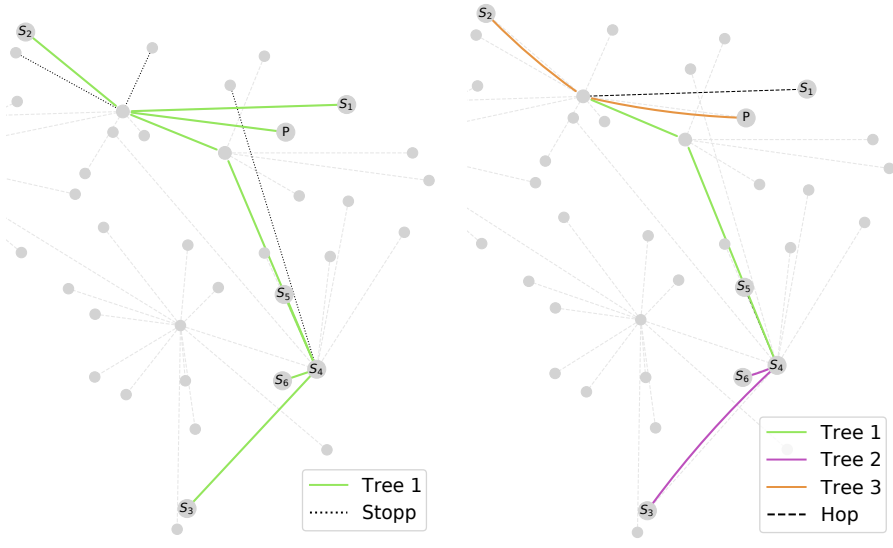
Exemplary Receiver Set in VtlWavenet

Figure 6.11 illustrates the distribution trees (DTs) created using the *random* and *cycles* strategies, applied to a sample receiver set of 13 subscribers (S_1, ..., S_{13}) spread across multiple rings in the `VtlWavenet` topology.

In Fig. 6.11(a), the *random* strategy uses a single large virtual tree. The different branches of the VT lead up to S_3, S_4, S_5 and S_6. The branch leading to S_3 is extended by two additional hop entries to connect neighboring subscribers S_8 and S_1. Furthermore, the VT is pruned after S_6 using a stop entry. Notably, the VT already terminates at S_4 and S_5, so there is no need for stop labels at these nodes. In total, the DT encoding for the *random* strategy requires one VT and two stop entries.

In contrast, Fig. 6.11(b) shows the DT produced by the *cycles* strategy, which divides the distribution into two VTs ($Tree_1$ and $Tree_2$). These VTs overlap on two branches: one extending to S_6 and the other to S_{11}. The overlapping sections are depicted using a dual-color representation. Two stop entries are used to prune the branches of the overlapping trees after S_6 and S_{11}, as well as an additional stop entry that truncates $Tree_1$ after S_5. Overall, the *cycles* strategy uses two VT entries and three stop entries to encode the DT.

The comparison highlights that the *random* strategy relies on a single, large VT to minimize the number of header stack entries, whereas the *cycles* strategy employs multiple smaller VTs to leverage ring structures within the topology.



(a) Frequency-Relation. (b) Partitions.

Figure 6.12: Exemplary distribution trees in `Litnet`.

Exemplary Receiver Set in Litnet

Figure 6.12 illustrates the DTs produced by the *frequency-relation* and *partitions* strategies, applied to a receiver set of six subscribers (S_1, ..., S_6) in the `Litnet` topology. Both strategies demonstrate distinct approaches to encode DTs.

Fig. 6.12(a) shows that the *freq-relation* strategy uses a single, publisher-centric VT that spans all intended subscribers as well as three uninterested nodes. To adapt the VT to the specific receiver set, three stop entries are employed to prune the three unnecessary branches, ensuring that the DT delivers messages only to the targeted nodes. This preserves the core structure of the VT while cutting off unused branches.

In contrast, Fig. 6.12(b) depicts the DT created by the *partitions* strategy, which splits the receiver set across three hierarchical VTs ($Tree_1$ to $Tree_3$). $Tree_1$ acts as backbone, connecting network hubs and facilitating inter-partition communication. $Tree_2$ and $Tree_3$ are smaller, localized VTs confined to their respective partitions. Each of these VTs spans the minimum number of edges required to reach their designated subscribers. $Tree_2$ perfectly covers its partition, while $Tree_3$ requires an additional hop entry to attach S_1.

The *freq-relation* strategy relies on a single, expansive VT, minimizing the number of tree entries. However, this comes at the cost of including uninterested nodes, which requires pruning via stop labels. In contrast, the *partitions* strategy

builds a more segmented DT by combining multiple smaller VTs. This reduces the number of stop labels, but requires more tree and hop labels.

Summary

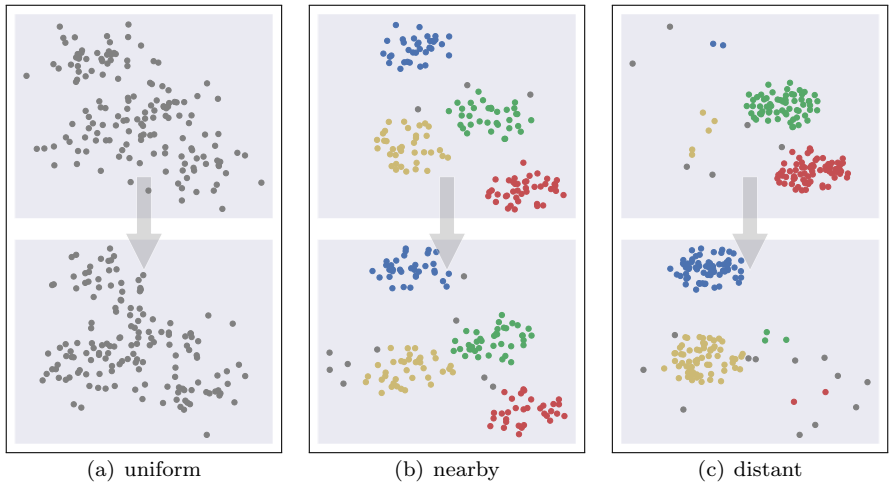
The visualizations provide valuable insights into how different VT strategies adapt initial DTs to encode compact headers. The *partitions* strategy and *clusters* strategy build DTs from smaller, localized VTs, whereas the *frequency-relation* strategy uses a single large VT that is pruned to match the receiver set. The *cycles* strategy uses structural patterns (e.g., rings) for localized placement, whereas the *random* strategy installs large VTs; both disregard client positions. Each strategy's effectiveness depends on VT placement, network topology, and subscriber distribution, making them suitable for different scenarios.

Note that the overhead taken to install or update VTs was not evaluated, as updates are not required after initial deployment – even with complete client redistribution. Thus, although the effort required to (re-)calculate VTs depends on a Steiner tree computation, it plays only a minor role.

6.6 Dynamic Scenarios with Churn

Communication flows within a network are heavily influenced by the geographic placement of publishers and subscribers (clients) as well as their notification frequencies. The virtual tree strategies introduced in this chapter account for these parameters to varying extents. Strategies that rely on detailed network knowledge – such as the *frequency-relation* strategy – can construct distribution-sensitive VTs when precise information about client locations and notification patterns is available. However, dynamic changes in client distribution during runtime pose a significant challenge. In the worst-case, all clients could migrate to different network locations. For the *freq-relation* strategy, this would result in a loss of VT coverage, necessitating either updates to the rule base or causing the strategy to degenerate into hop-based source routing. In contrast, distribution-independent strategies – such as the *random strategy* – do not require the rule base to be updated, even as client locations and subscriptions evolve.

To evaluate the robustness and efficiency of VT-based strategies under dynamic conditions, we simulate client migrations using *three distinct migration scenarios*. At the beginning of each simulation, VTs are constructed based on an initial, fixed client distribution. As the simulation progresses, clients gradually migrate and reconnect to different network nodes, following the specific client migration behaviors defined by the scenario. This incremental migration continues until the client distribution is fully redistributed. These changes in client placement alter communication patterns, directly impacting the relevance of the precomputed VTs. The results show how each virtual tree installation strategy performs and adapts under dynamic conditions and evolving networks.



(a) uniform (b) nearby (c) distant

Figure 6.13: Client migration scenarios.

6.6.1 Migration Scenarios

We simulate the dynamic and fluctuating nature of pub/sub systems through following three different migration scenarios depicted in Fig. 6.13:

1. **Uniform distribution.** In the scenario shown in Figure 6.13(a), clients are evenly scattered across the network. Each node has an equal chance of gaining or losing a client. The network load is the same across all regions.

2. **Nearby distribution.** In the scenario shown in Figure 6.13(b), the initial clients are divided into (four) different clusters. New clients predominantly join within one of these clusters and rarely outside of them. This ensures that clients remain tightly packed and traffic remains concentrated around stable cluster centers.

3. **Distant distribution.** The scenario shown in Figure 6.13(c) starts with (two) initial clusters containing the most clients. Unlike the nearby migration scenario, new clients tend to join at network locations that are distant from the original clusters, leading to the formation of (two) new clusters. Over time, old clusters disappear and new clusters become dominant, shifting message traffic from the old to the new clusters.

By simulating these migration patterns, the performance of VT strategies can be evaluated under dynamic conditions. This provides valuable insights into how usable and combinable the strategies' virtual trees are for changing and fluctuating subscriptions.

6.6.2 Evaluation

We conducted an extensive series of experiments to evaluate our VT-based strategies across various real-world network topologies and probability-driven client placements, as defined by different migration scenarios. These scenarios simulated gradual shifts in client distribution, ranging from 0% to 100% churn, with increments of 10%. Here, 0% churn reflects a static client distribution with no migration, while 100% churn represents a complete redistribution of publishers and subscribers. To enable comparison of the strategies under the same conditions, we maintained a constant total number of clients throughout the simulation. Each time a client was removed from a location, a new client was immediately placed at a different node. This process ensured continuous client turnover until full redistribution was achieved.

As with the static scenario experiments (see Sect. 6.5.2), the simulations for the dynamic scenarios were conducted using *Mininet* [73] and *BMv2* [10]. Each node in the topology was configured as a switch connected to a single host with the ability to store up to 40 rules. A randomized selection process assigned 30% of the hosts as publishers, with each publisher having 30% of the hosts designated as its subscribers. These selections were weighted by node-specific probabilities, which were defined by the migration scenarios (cf. Fig. 6.13). Notably, the `nearby` and `distant` migration scenarios modeled densely populated hotspot clusters, where clients tend to reside close to the clusters' centers.

The experiments varied across four key dimensions: (i) *strategy*, (ii) *topology*, (iii) *migration scenario*, and (iv) *churn rate*. During each simulation run, the number of *tree*, *stop*, and *hop* labels generated for each computed distribution tree was recorded. This allowed for a detailed analysis of the performance and adaptability of the VT strategies in diverse and dynamic network conditions.

Strategy Results

Figure 6.11 presents the evaluation results of the VT strategies using a 3×3 matrix chart layout. The columns correspond to the three candidate network topologies (`Litnet`, `Dfn`, `VtlWavenet2008`), while the rows represent the tested migration scenarios (`uniform`, `nearby`, `distant`). For a broader context, evaluation results for the 25 networks listed in Table 6.1 from the *Internet Topology Zoo* (featuring networks with more than 40 nodes) are publicly accessible at [118].

Each chart displays groups of bars, where each group corresponds to a specific VT strategy. Within each group, individual bars represent results at various churn rates, with distinct shading to indicate the intensity of the churn rate. The x-axis displays the average number of encoded entries per strategy, while the y-axis distinguishes the different strategies. Each bar represents the average distribution tree of a strategy and is color-coded to show the average proportions of *hops*, *stops* and *trees* in a stacked manner. These bars clearly show how each strategy balances label usage across different topologies and migration scenarios.

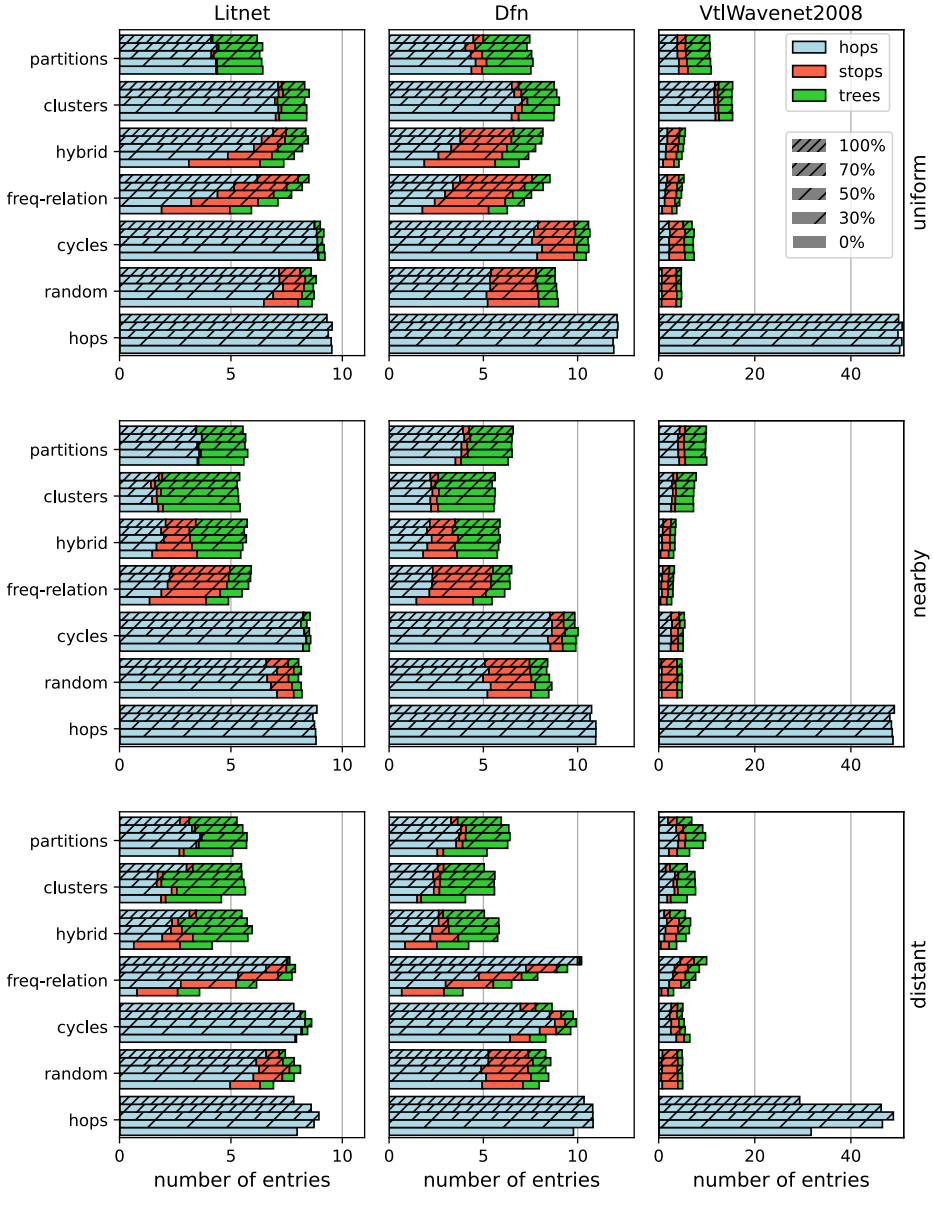


Figure 6.14: Entries per notification by different churn rates (40 rules per switch, 30% subscribers per publisher).

The performance results show that both *topology* and *client distribution* significantly influence the number of notification header stack entries. Below, we provide a detailed comparison of the header stacks generated by each strategy.

a) *Hops*: The bottommost bar group of each chart shows the results of the *hops* strategy, which does not use stored trees and encodes the entire distribution tree with hop entries. Comparing the *hops* results across the three different client distributions reveals that the `uniform` distribution requires more hop entries than both clustered distributions (`nearby` and `distant`). This is intuitive, as in clustered distributions most clients are within clusters, and delivery trees often span only combinations of nodes within those clusters, rather than the entire network, resulting in smaller delivery trees. Conversely, in the `uniform` distribution, clients are spread across the network, resulting in larger delivery trees. Furthermore, *hops* shows relatively constant results across all churn rates, except for the `distant` distribution, where a 'wave crest' in header stack size is shown. This peak occurs because clients shift from original clusters to newly formed ones, causing the distribution trees to temporarily span both old and new client clusters. The peak in header stack size is highest when 50% of clients are distributed between old and new clusters. At this peak, publishers are forced to address subscribers in both halves equally, resulting in larger delivery trees and consequently the largest header stacks.

b) *Random*: The second set of bars represents the performance of the *random* strategy, which leverages VTs and employs all three label types. For the `Litnet` topology, which has a high hub density and a small network diameter, *random* shows only modest improvement over *hops*. However, it performs notably better on the `VtlWavenet` topology, which has fewer hubs and a larger diameter. Such networks provide fewer alternative routes between nodes, reducing the likelihood of VTs interfering with each other. This makes randomly generated VTs, each covering significant parts of the network, more applicable. The results also indicate that, across all topologies and scenarios, *random* primarily relies on a single VT for delivery tree construction. In `VtlWavenet`, the selected VT is pruned exclusively by stop labels, indicating a strong fit. Conversely, in `Litnet` and `DFN`, both stop and hop labels are required to adapt the VT, indicating a poorer fit. The larger number of hop labels is due to the fact that a single stop entry can significantly truncate large parts of a VT. In fact, a single stop can prune multiple overlapping VTs, which are encoded together in the header stack of the notification, simultaneously. Conversely, a hop label adds only one edge to the delivery tree, ideally connecting a destination node directly. In most cases, however, the destination node is several hops away, requiring multiple hop labels. This basic strategy serves as a reference for quantifying the benefits of more advanced VT strategies that leverage additional network knowledge.

c) *Cycles*: The *cycles* strategy generates the largest headers among all VT strategies in all scenarios, indicating that this strategy cannot use topological knowledge to generate better trees than the *random* strategy. In *cycles* VTs are not installed for nodes outside network rings, such as the satellite nodes in `Litnet`. This results in single-hop connections for the outlying nodes, increasing the number of hop labels and making them even higher on average than in the *random* strategy.

d) *Frequency-Relation*: The *frequency-relation* strategy demonstrates its robustness in stable scenarios without migration (0% churn). When the migration changes are marginal and clients stay in their clusters, as modeled by the `nearby` migration scenario, few adjustments are required and the original VTs remain reasonably reusable. In contrast, in the `distant` migration scenario, where clients move to different network regions, the number of necessary header entries rises rapidly. This is especially apparent in `DFN` and `Litnet`, where the originally installed VTs become less and less useful because the virtual trees cover the original clients, but not the new ones. Consequently, a significant number of additional hops are required to adapt the delivery tree to the new client distribution. As migration approaches 100%, the *freq-relation* strategy degenerates into source routing, where only hop labels are encoded and the VTs become useless.

e) *Clusters*: The *clusters* strategy generates significantly more VT entries compared to the *frequency-relation* strategy, due to its different encoding principles. Unlike *freq-relation*, which modifies publisher-centric VTs by extending or pruning them with stop and hop labels, *clusters* combines multiple hierarchical VTs to construct a distribution tree. This strategy works well in `nearby` and `distant` migration scenarios, where clients are concentrated within specific clusters. In these cases, the hierarchical structure of *clusters* minimizes the need for additional labels, resulting in compact headers. However, the *clusters* strategy becomes inefficient in the `uniform` distribution scenario. This is to be expected, as the strategy relies on preinstalled VTs in specific, predefined regions. When clients are evenly distributed across the network, many fall outside these regions and require a sequence of hop labels to connect them to the distribution tree. As a result, header stacks in `uniform` scenarios are dominated by hop entries, significantly increasing the overall header size.

f) *Partitions*: The *partitions* strategy outperforms the *clusters* strategy in `uniform` distribution scenarios by providing comprehensive VT coverage across the entire network, thereby minimizing the need for bridging through hop labels. Unlike *clusters*, which relies on predefined regions and is sensitive to client placement, *partitions* operates independently of client distribution, making it suitable for environments where client locations are unknown or unpredictable. However, in scenarios where clients are grouped (`nearby` or `distant` distributions), the *clusters* strategy prevails because it is designed for localized placements.

g) *Hybrid*: The *hybrid* strategy combines the strengths of the *frequency-relation* and *clusters* strategies by installing VTs from both strategies in a round-robin manner. This balanced approach deploys both publisher-centric and cluster-based VTs. It allows optimized *freq-relation* trees for static scenarios, and prevents complete degradation to hop-based source routing in the `distant` migration scenario with increasing dynamics. The results show that *hybrid* performs comparably to the best performing strategy for low and high migration rates, corresponding to *freq-relation* and *clusters* respectively. Even with half the rules for each strategy installed, *hybrid* remains remarkably close to the best performing of the two. As the migration rate increases, the cluster-based rules increasingly serve the migrated clients, while the *freq-relation* rules continue to serve the original clients. This adaptability makes the *hybrid* strategy particularly effective for dynamic scenarios with intermediate churn rates, bridging the gap between static and fully migrated client distributions. An alternative implementation could be to combine the *freq-relation* and *partitions* strategies to create a hybrid strategy that combines publisher-centric VTs with network-wide (rather than localized) hierarchical VTs.

In summary, the performance of the strategies is influenced by both the *network topology* and the *client migration scenario*. In terms of topology, the smaller hierarchical VTs generated by the *clusters* strategy or *partitions* strategy are particularly effective in `Litnet`, a scale-free network with many hubs. Conversely, in `VtlWavenet`, a network with fewer central hubs, larger VTs generated by the *random* strategy or *frequency-relation* strategy, which rely more on tree pruning and less on hop extension, perform better.

Churn has a significant impact on the *frequency-relation* strategy. In particular, as clients migrate to `distant` network regions and communication patterns evolve, the original *freq-relation* VTs become less usable. Meanwhile, all other strategies maintain relatively stable header stacks because they do not consider the exact client location for VT installation. For instance, the *clusters* strategy is more resilient to client migration because it leverages broader location data to identify densely populated hotspot clusters. As long as clients stay within these clusters, the VTs do not need to be adjusted and the resulting header stacks remain compact. The *hybrid* strategy, which alternates between installing VTs from the *clusters* and *freq-relation* strategies, mitigates the performance degradation observed with *freq-relation* and prevents a fallback to source routing.

6.6.3 Best Strategies

For each of the selected networks from the *Internet Topology Zoo* [70] Figure 6.15, presents the simulation results of the strategies that encode the most compact notification message headers in a scattergram in a 3×3 matrix of scatterplots. Each row in the matrix represents different client distributions (`uniform`,

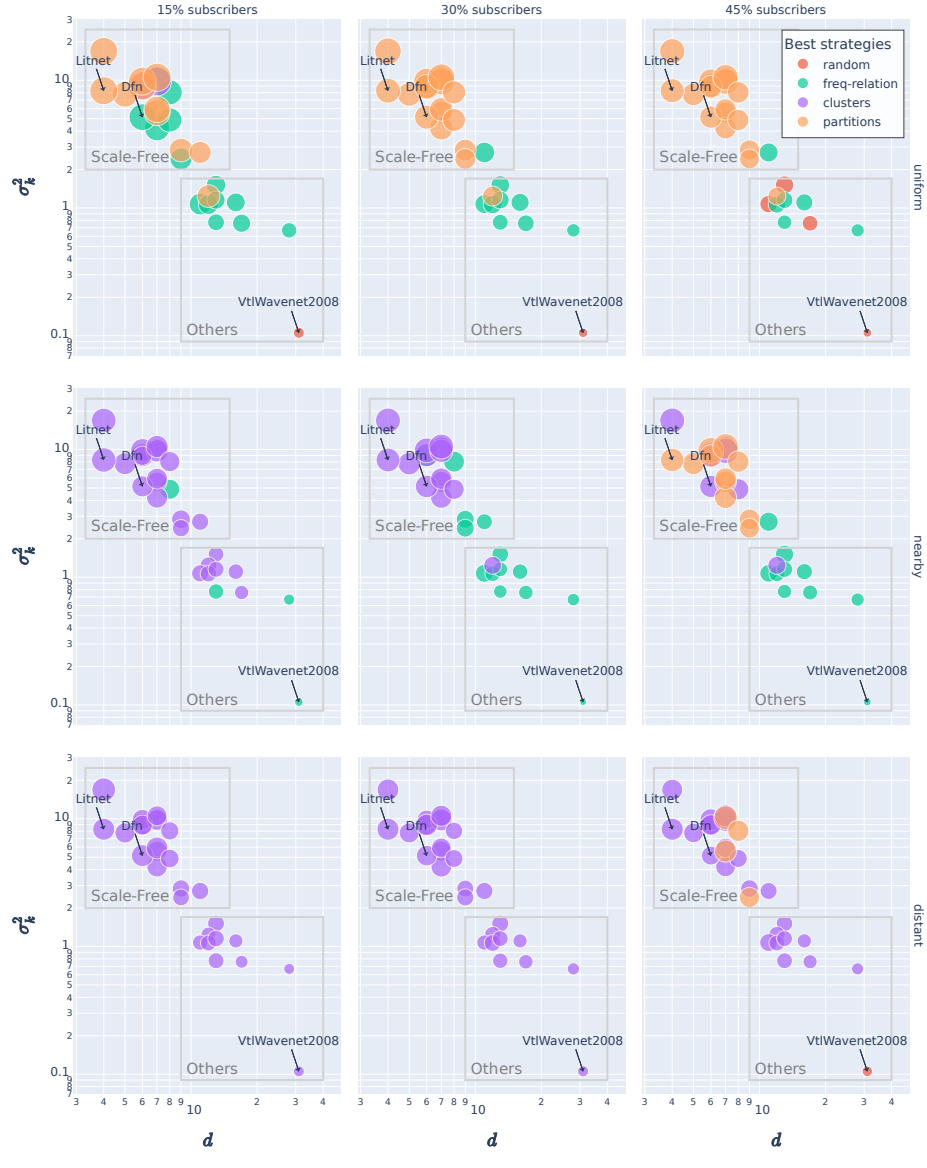


Figure 6.15: Best strategies (after 100% client migration).

`nearby`, or `distant`), while the columns represent various subscriber quantities (15%, 30%, and 45%). In each diagram, the x-axis represents the network diameter (d), and the y-axis displays the fluctuation of network node degrees (σ_k^2) on a double logarithmic scale. Simulations were carried out for a fixed churn rate of 100% for all three migration scenarios. Each dot stands for a topology and shows via its color and size two features: (i) the color identifies the best VT installation strategy for the network, and (ii) the size represents the average number of header stack entries encoded by this strategy. Smaller dots represent better, more compact delivery tree encodings.

The scatter plots generally fall into two main quadrants:

- *Scale-Free Networks (SFNs):* Located in the upper-left quadrant with small diameters and significant node degree deviations (e.g. `Litnet`).
- *Other Networks (RNs):* Located in the lower right quadrant with larger diameters and node degrees distributed around the mean (e.g. `VtlWavenet`).

Dot sizes, representing header lengths, reveal that SFNs generally produce larger headers than RNs – despite having shorter average path lengths. This initially counterintuitive result stems from the high hub density in SFNs, which enables many possible DT configurations. However, limited switch memory (40 rules per switch) restricts the number of mappable VT permutations. Consequently, DT adaptation often requires more frequent extension, trimming, or combination of existing VTs. While increasing the number of rules can reduce header size, it also raises header stack computation time, which grows with the number of installed virtual trees (see Section 5.2.7).

Dot colors, which distinguish strategies, indicate that optimal performance largely depends on the *client migration scenario* and *network type*, with subscriber count playing a lesser role.

In *scale-free networks (SFNs)*, the *partitions* strategy and *clusters* strategy often generate the shortest headers across all client distributions and subscriber quantities. These strategies leverage hierarchical VTs to exploit the densely connected hub nodes characteristic of SFNs. In the `uniform` client migration scenario, *partitions* outperforms *clusters* by installing VTs across the entire network, rather than concentrating only on predefined regions.

In *other networks (RNs)*, the *frequency-relation* strategy and *clusters* strategy generate often the shortest headers. The *freq-relation* strategy is particularly robust in the `nearby` migration scenario where clients remain close to the original distribution trees. However, when clients migrate to `distant` regions, the *clusters* strategy prevails as it offers usable VTs for both old and new client clusters, providing better adaptability to changing network conditions.

The following provides a deeper analysis of the results achieved by the most effective strategies.

a) *Random*: The *random* strategy only gives the best results for a few RNs with a high number of subscribers (45%). It also performs best for every client migration scenario in `VtlWavenet`, exploiting the fact that this topology has few hubs, limiting the number of possible branch permutations for a DT. However, the use case for *random* is quite limited compared to the next three strategies.

b) *Frequency-Relation*: The *freq-relation* strategy shows its strength particularly for RNs in `uniform` or `nearby` migration scenarios. Despite complete client redistribution, this strategy effectively adapts original trees to new scenarios with minimal additional routing information. However, in the `distant` migration scenario, the original VTs become progressively inefficient as churn rate increases, making the *clusters* strategy the best performing one for this distribution.

c) *Clusters*: The *clusters* strategy often works well for `nearby` and `distant` migration scenarios. However, it is less suitable for the `uniform` distribution because clients outside the predefined clusters must be attached to the DT via individual hop labels, making this strategy less efficient.

d) *Partitions*: The *partitions* strategy is best suited to the `uniform` distribution as it covers the entire network with VTs. It also performs best when addressing a significant number of subscribers (45%) in clustered SFN topologies (e.g. Litnet). This means that full coverage of the network with hub-and-spoke VTs is efficient either for addressing a large number of subscribers or for a uniform distribution of clients in SFNs.

These simulation results emphasize that the network type and client migration scenario are the primary factors for identifying the best strategy, with subscriber quantity playing a minor role. Scale-free networks benefit most from hierarchical VTs generated by the *partitions* and *clusters* strategies. In other networks with low hub density and large network diameter, the *frequency-relation* strategy is optimal for `nearby` migration scenarios, while the *clusters* strategy is best for `distant` migrations. This variation in results highlights the importance of carefully selecting VT strategies for specific network conditions to achieve compact header stack encoding with minimal switch memory consumption.

6.7 Related Work

To the best of our knowledge, our forwarding scheme offers unparalleled flexibility in combining stateful and stateless routing for dynamic multicast, as required by content-based pub/sub systems. Specifically, our approach enables the use of multiple preinstalled distribution trees, with branches that can be dynamically extended or pruned as needed. This unique capability sets our approach apart, making direct comparisons with other systems challenging.

Nevertheless, several existing approaches share certain similarities with our *virtual trees*. These approaches also distinguish between various types of labels or orchestrate the creation of multiple multicast trees to achieve specific optimization goals. While there are parallels in certain aspects of our methodology, the comprehensive integration of stateful and stateless routing, along with the dynamic adaptability of our VTs, remains a distinctive feature of our approach.

6.7.1 MTRSA

The authors of *Multi-Tree Routing and State Assignment Algorithm* (MTRSA) [54] address the *Scalable Multicast Traffic Engineering* problem by optimizing multicast tree configurations to reduce state complexity. This is achieved through flexible resource allocation and the joint minimization of the number of branch nodes and edges across multiple multicast trees. A branch node – defined as having at least three edges – is responsible for maintaining and managing, whereas other nodes remain stateless and are connected through unicast tunnels.

The primary challenge is to satisfy (i) *link capacity* constraints and (ii) *node capacity* constraints. The link capacity limits the total rate of all multicast trees on each link, while the *node capacity* ensures that the group table of each node has sufficient capacity to support the required multicast trees.

When a node's group table reaches its maximum capacity, unicast tunneling shifts the resource demand from the node to its incident links. This process reroutes the multicast tree and leverages the resources of nearby nodes and links, optimizing overall network resource usage.

Incorporating a similar tree installation strategy into our framework for virtual trees could help minimize the total bandwidth cost for all virtual trees while respecting link and node capacity constraints. However, implementing unicast tunneling within our system would require modifications to our header structure.

Concentrating the state at branch nodes would shorten the header stack and increase the available payload space. However, implementing tunneling through stateless nodes would require an additional (unicast protocol) header, which would, in turn, reduce the available payload space. Therefore, incorporating this approach into our virtual tree framework involves balancing the advantages of smaller header stack entries with the disadvantages of an additional header.

6.7.2 Segment Routing

Segment Routing (SR) [112] is a stateless forwarding method for unicast traffic, where a packet's route is encoded as a list of segment labels in its header. Each *segment label (SID)* represents either a node or a network edge. SR provides two source routing options: *strict*, specifying a complete sequence of hops from source to destination, and *loose*, specifying key waypoints to be traversed.

SR can be implemented using MPLS or IPv6. *MPLS-based SR (SR-MPLS)* simplifies the MPLS control plane and is suitable for networks with existing MPLS infrastructure. In contrast, *IPv6-based SR (SRv6)* relies entirely on IPv6 and offers greater operational flexibility.

In both SR-MPLS and SRv6, a segment label is encoded as either an MPLS label or an IPv6 address to enable switch forwarding. The three basic SR data plane operations are: (i) adding a label at the beginning of the segment list as active segment label (*push*), (ii) removing an active element to activate the segment label behind it (*next*) or (iii) tunneling a network device using regular IP routing (*continue*). Unlike our P4-based strategies, SR always processes the top segment label and does not allow random access to list elements.

Segments in SR are categorized as either *global* or *local*. Global segments are known to all nodes in the SR domain and direct packets to a specific destination node or network. Local segments, on the other hand, are specific to individual nodes and direct packets through a particular output interface, similar to our source-based strategies. An SR list can combine global and local segment labels to define any path.

Recent SR enhancements [26, 27] include multicast support, where each segment denotes a distribution tree and the segment label contains the tree segment id (Tree SID). Similar to our virtual trees, nodes along these trees must maintain mappings between segment labels and corresponding egress ports.

6.7.3 POLKA

Dominicini et. al [37] combine the *Residual Number System* with binary polynomial arithmetic to perform stateless segment routing. This method leverages the *Chinese Remainder Theorem (CRT)* and assumes that the IDs of nodes along a path are pairwise co-prime numbers.

For a packet to be sent, a route identifier is computed using the CRT for polynomials and embedded in the packet header. This packet-specific route identifier is used to calculate the output port at each node along the path by performing a modulo operation. Each node calculates the remainder of a binary polynomial division using the route identifier and its own node identifier. At each node, the modulo operation determines the node-specific output port, ensuring the correct forwarding of the packet.

However, commercial network hardware and the P4 language do not natively support polynomial or integer modulo operations with non-constant operands. To address this challenge, the authors repurpose *Cyclic Redundancy Check (CRC)* hardware, typically used for error detection, to perform the modulo operation. This involves pre- and post-processing using binary arithmetic to transform the route ID so that it can be processed by the CRC hardware. The modulo-2 arithmetic capabilities of the CRC hardware are then used to perform the necessary calculations.

While the approach could potentially be extended to encode entire distribution trees, there are several practical hurdles to consider related to hardware limitations and scalability. First, CRC operations are constrained when dealing with user-defined polynomials, particularly with respect to the bit width of the CRC polynomials, which typically cannot exceed 32 bits. Secondly, for large distribution trees with many nodes, the route ID can become excessively large. The required field width for these IDs depends on the number of nodes and their associated primes, while the size of a prime number is related to the number of adjacent edges. Finally, the support for CRC polynomials varies between different architectural models and their implementations.

6.7.4 KYRA

KYRA [21] leverages knowledge of the event space to organize network participants into clusters and to assign responsibility for specific event content within each cluster. This approach uses a two-stage distribution strategy based on *cliques* and *content zones*.

KYRA initially groups servers into cliques based on network proximity, ensuring that servers within a clique are closely connected. Multiple routing trees are established across these cliques, each dedicated to specific event subsets represented by *content zones*. In subdividing the notification space, KYRA distinguishes between *local* and *global* management:

- *Local management*: Within each clique, the event notification space is divided among the servers. Each server is assigned a unique content zone and acts as a proxy for overlapping subscriptions in that zone.
- *Global management*: The overall content space is partitioned between the routing trees, with each tree designated to manage a specific content zone. These trees interconnect servers from different cliques that share the same content zone.

When a subscription is submitted, it is first directed to a server close to the subscriber. This server then redirects the subscription to one or more proxy servers based on the overlapping content zones. Published notifications are first routed to the server within the same clique whose content zone covers the notification, and then propagated through the appropriate routing tree for that zone.

While KYRA's hub-and-spoke approach shares similarities with our *clusters* and *partitions* strategies, there are notable differences. First, KYRA uses unicast transmission within a clique, directly between clients (publishers or subscribers) and proxy servers. In contrast, our approach uses additional hop labels in the header stack to attach off-tree clients to the delivery tree. Furthermore, while KYRA improves load balancing and reduces memory requirements for brokers compared to flat filter-based systems, the indirection through the fixed proxy servers can lengthen the delivery path and introduce delays in message delivery.

6.8 Summary

This chapter evaluated the design and deployment of *virtual trees (VTs)*, exploring various strategies that embed VT identifiers within notification message headers along with supplementary data for VT extension or pruning. The primary objective of these strategies is to create streamlined and efficient headers, thereby minimizing network load and reducing processing overhead on network switches.

Our strategies are based on varying degrees of network knowledge. The simplest implementations use topological insights, whereas more advanced strategies leverage pub/sub-specific knowledge. This specialized knowledge includes the locality of publishers and subscribers, their interrelationships, and the notification rates of subscribers.

To evaluate the performance of these strategies across different real-world network topologies, we conducted evaluations in multiple scenarios, varying publisher/subscriber distributions and churn rates. The results indicate that each scenario benefits from a specific strategy. For instance, scale-free networks perform well with hub-and-spoke paradigms, which install hub-centric trees with limited sprawl, as seen in our *clusters* strategy and *partitions* strategy. Conversely, other topologies benefit from our *random* strategy, which does not consider any knowledge of the network and only generates uniformly distributed trees.

The findings highlight the significance of pub/sub-specific knowledge. As the level of knowledge integration in the virtual tree construction increases, the VT edges become more closely aligned with the actual notification flows. Even in scenarios with churn – where clients leave or join the pub/sub network, or change their relationships due to changes in advertisements or subscriptions – the original VTs remain largely functional.

This adaptability is due to the ability to augment message headers with additional information, often eliminating the need to update the rule base. However, strategies that take into account the specific locations of publishers and subscribers, such as in the *clusters* strategy and *frequency-relation* strategy, are less effective at encoding compact headers in churn scenarios, as misalignment between VTs and actual communication flows results in more header stack labels.

In summary, the effectiveness of VT strategies is closely related to the level of network knowledge they incorporate and their ability to adapt to dynamic network conditions. Through the selection of appropriate strategies on the basis of the network topology and the distribution of publishers and subscribers, it is possible to maintain efficient communication flows and generate compact notification message headers even in dynamic environments.

Chapter 7

Conclusion

This chapter concludes by emphasizing the two cornerstone attributes for an efficient content-based pub/sub notification service: *flexibility* and *minimal delivery times*. Our research has focused on these key principles to develop an efficient middleware system that operates through SDN-enabled switches and P4-based network protocols. The protocols, implemented by P4 programs, are adaptable to a variety of scenarios. They allow for the delivery of notifications via individually encoded distribution trees, which use routing information stored both in packet headers and within the switch infrastructure.

In the following, we summarize the thesis, highlight the key contributions and outcomes, critically review the developed notification distribution strategies with respect to our technical goals, and compare them. Lastly, we address remaining open questions, discuss approaches for future research, and outline the next steps needed to address these approaches.

7.1 Summary

Our primary objective was to develop a *fast* and *flexible* pub/sub middleware capable of delivering notifications to arbitrary receiver sets with significantly less worst-case delay than broker-based approaches. To achieve this, we took advantage of programmable network devices that can take over notification distribution tasks traditionally handled by application-layer brokers. Our approach aimed to overcome the inherent delays and network load associated with broker-based routing.

This work represents a shift away from *application-layer filtering and forwarding*, and a strategic departure from the limitations of current IP-based protocols widely used in content-based pub/sub systems. We have designed a scalable middleware solution that avoids the bottleneck of mapping filter expressions

to multicast addresses, thus enhancing system scalability and preserving the expressiveness of advertisements and subscriptions.

The cornerstone of our approach is the use of the *P4 SDN programming language*, which enables the implementation of a low-latency and scalable pub/sub middleware. This middleware bridges the gap between *application-layer brokers* and *network-layer switches*. With P4, we achieve complete control over the notification distribution process. Unlike traditional network-layer protocols, P4's protocol-independent nature grants the flexibility to define custom pub/sub schemes, allowing us to fine-tune packet-processing behaviors to the specific requirements of a pub/sub application. As a result, our protocols provide precise control over forwarding operations, while allowing real-time adjustments to precomputed and stored distribution trees. This ensures adaptability to dynamic changes in subscriptions, publications, and network conditions.

By offloading flexible multicast routing functionality to the *data plane*, we leverage network devices equipped with *ternary content-addressable memory (TCAM)* that support wildcard bits to efficiently perform complex matching and forwarding operations at the network layer. This approach enables network switches to take over many of the functions traditionally performed by pub/sub brokers, but at *wire speed*, thereby overcoming the limitations of application-layer multicast and existing network protocols. This represents a significant advancement in the field of programmable networking, as it ensures fast, efficient, and reliable routing of messages from publishers to interested subscribers.

In the following, we summarize our contributions for each chapter.

Chapter 3: Source-routed delivery trees. In Chapter 3, we introduced innovative notification distribution strategies for source routing in content-based pub/sub systems. We presented three different strategies, each encoding the entire delivery tree of a notification directly in the message header, thus eliminating the need to store routing information in switches. These strategies are particularly beneficial in dynamic pub/sub environments, where frequent subscription changes would otherwise require constant updates to forwarding rules. Through comprehensive evaluations, we compared the three different P4 distribution strategies and assessed their performance. We furthermore conducted a comparison of the *switch/port* strategy, which serves as a representative of *P4 pub/sub*, with *OpenFlow multicast*, three discrete application layer multicast variants, *unicast* and *broadcast*. The results show that P4 pub/sub is almost as efficient as *OpenFlow multicast* in terms of network bandwidth, while it does not rely on stored forwarding trees. Additionally, it significantly reduces worst-case delays compared to *unicast* and *application-layer multicast* approaches.

Chapter 4: Stored forwarding trees and hybrid approach. Chapter 4 introduced routing schemes for content-based pub/sub systems that store stable distribution information in switch memory. We presented three strategies based on

stored multicast trees, each focusing on different approaches for storing distribution information. We also proposed a *hybrid* strategy that strikes a balance between static and dynamic routing. This strategy maintains routing trees for stable subscribers, while encoding additional routing paths for dynamic subscribers directly in the message header. Only when subscribers within the stored tree migrate is an update to the rule base required to avoid false positives. Meanwhile, dynamic subscribers enlarge the message header through individual source-based routing with additional hop labels. Our evaluations confirm the importance of partitioning subscribers in a meaningful way to determine a stable subset for which it is worth to maintain a stored multicast routing tree.

Chapter 5: Virtual trees with greedy encoding scheme. In Chapter 5, we introduced more advanced routing schemes that forward notifications along multiple partially stored delivery trees called *virtual trees (VTs)*. The approach uses a greedy encoding algorithm that optimizes the initial delivery tree and reduces the amount of forwarding information embedded in the notification header. By using a stack of different label types in the header, the algorithm dynamically references VTs, trims unnecessary branches, or adds extra paths to form a complete delivery tree. This reduces the need for frequent updates when subscriptions change. We applied this approach in a datacenter network and developed strategies that install VTs based on topological features, knowledge about the relationships between publishers and subscribers, and runtime statistics. The proposed strategies exploit the hierarchical datacenter structure to derive worthwhile and combinable VTs. Evaluation results show that integrating VTs with additional distribution information in the message header significantly reduces header size and network traffic. The results also show that pruning larger VTs is more beneficial than extending smaller ones by adding individual delivery paths to form the complete delivery tree.

Chapter 6: Virtual tree construction schemes. In Chapter 6, we explored different strategies for constructing VTs, taking into account various levels of knowledge about the pub/sub network – ranging from topological knowledge to pub/sub-specific insights. We developed seven distinct VT construction schemes and evaluated them in 25 real-world networks, considering different client distribution and migration patterns. The constructed VTs enable adaptable notification delivery, allowing publishers to prune or extend trees by embedding additional routing information within the message headers. This flexibility ensures compact notification headers without requiring frequent updates to the stored rule base in the switches, even in dynamic networks with frequent subscription changes or migrating clients. Our evaluations show that the VT construction strategies perform differently depending on the network scenario. For example, scale-free networks benefit from *hub-and-spoke*-based strategies using small hierarchical trees, whereas topologies with large diameters and few hubs benefit from larger trees. Considering client distributions, as in the *clusters* strategy,

significantly reduces message header size as long as clients stay within the strategy's predefined densely populated regions. Stable sender-receiver sets benefit from the *frequency-relation* strategy, resulting in minimal notification message header size.

In conclusion, this work represents a significant advancement in the design of content-based pub/sub systems. We developed an efficient and flexible solution that leverages P4 programmability, source-based routing, and composable partial delivery trees. By addressing the limitations of application-layer multicast and traditional IP-based protocols, we provide a novel framework for content distribution. Our approach paves the way for future research in programmable networks and content-based pub/sub systems, offering improved scalability, reduced latency, and greater adaptability to dynamic environments.

7.2 Future Work

While this thesis has made significant progress in developing an efficient and flexible publish/subscribe middleware using programmable network devices, there are still several areas that offer potential for further research and development. The following aspects are particularly noteworthy:

Automated strategy selection. The evaluation in Chapter 6 demonstrates that different strategies are effective depending on the characteristics of the network topology, as well as the placement and actual interaction of publishers and subscribers. Although the evaluation provides detailed information on the most effective strategies under specific conditions, selecting the optimal strategy remains a complex process. Informed decisions require consideration of multiple factors, such as network diameter, node degree fluctuation, client distribution and churn rate. An *automated selection mechanism* could significantly simplify this process by automatically collecting the relevant parameters, identifying the current network scenario, and selecting the most appropriate strategy. This mechanism would remove the need for manual analysis of the pub/sub network and significantly reduce the technical barrier to deploying our strategies. Such a system could dynamically adjust to real-time conditions, ensuring that the most efficient strategy is always in use.

Dynamic flow rule adaptation. The presented encoding schemes for distribution trees based on stored trees significantly improve the adaptiveness of pub/sub middleware. However, further research could explore *dynamic flow rule adaptation* in more detail. As network conditions evolve – due to factors such as subscriber churn or changing communication patterns – there is an increasing need for flow rules to be dynamically adapted in real-time. Future work could focus on developing *adaptive algorithms* that can dynamically reconfigure the

flow rules of already established trees. These algorithms would need to adjust and optimize tree structures in response to network changes without disrupting service or losing messages during the adaptation process. A flexible mechanism for dynamic tree reconfiguration would guarantee a consistently short header length while ensuring reliable delivery.

Aggregating multicast traffic. Although our *virtual tree* strategies make use of precomputed multicast trees to optimize message delivery, the challenge of effectively aggregating multicast traffic remains a complex and promising area of research. The NP-hard nature of the *Branch-aware Steiner Tree (BST)* problem [48] highlights the inherent difficulty of efficiently aggregating multicast traffic while minimizing resource consumption. In this thesis, we adopted a Steiner heuristic [103] to approximate solutions. However, there are other algorithms that have potential in similar contexts, such as the *Locality-Aware Multicast Approach* [76], the *Branch-Aware Edge Reduction Algorithm* [55], and the *Extended Dijkstra's Shortest Path Algorithm* [65]. Future work could involve conducting comparative evaluations of these algorithms within diverse topologies and under varying multicast traffic patterns. A systematic comparison would reveal the relative advantages and trade-offs associated with each algorithm, including their ability to be applied to virtual trees. These studies could also show how specific network topologies influence the performance of these BST algorithms. Such analyzes would identify the most appropriate algorithmic solutions for optimizing multicast traffic in real-world networks.

Decoupling publishers and subscribers. With the current implementation, publishers need to know their subscribers directly in order to send messages to the right recipients. This contradicts the concept of decoupling senders and receivers in pub/sub. For situations where anonymity between publishers and subscribers is required, this dependency can be problematic. A key area for future research is to explore methods of decoupling the relationship between publishers and subscribers while maintaining the accurate delivery of notifications. One promising direction is to investigate how advertisement and subscription filter management can be implemented using SDN controllers. In such an approach, the SDN controller could receive and manage filter expressions from both publishers and subscribers, and identify filter overlaps to dynamically derive and preinstall virtual trees. These trees would enable efficient routing of notifications without requiring direct knowledge of subscribers by publishers. This research would require the development of mechanisms that balance low latency, scalability, and precise targeting of notifications with participant anonymity, ensuring that publishers and subscribers remain fully decoupled.

Bibliography

[1] A. Adams, J. Nicholas, and W. Siadak. *Protocol Independent Multicast - Dense Mode (PIM-DM) – Protocol Specification*. RFC 3973. Jan. 2005. DOI: 10.17487/rfc3973.

[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient pattern matching over event streams". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, June 2008, pp. 147–160. ISBN: 9781605581026. DOI: 10.1145/1376616.1376634.

[3] Toyokazu Akiyama, Yuuichi Teranishi, Ryohei Banno, Katsuyoshi Iida, and Yukiko Kawai. "Scalable Pub/Sub System Using OpenFlow Control". In: *Journal of Information Processing* 24 (July 2016), pp. 635–646. DOI: 10.2197/ipsjjip.24.635.

[4] Alexey Andreyev. *Introducing data center fabric, the next-generation Facebook data center network*. Mar. 2014.

[5] David Arthur and Sergei Vassilvitskii. *k-means++: The Advantages of Careful Seeding*. Tech. rep. Stanford InfoLab, June 2006.

[6] Anthony J. Ballardie. *Core Based Trees (CBT version 2) Multicast Routing – Protocol Specification*. RFC 2189. Sept. 1997. DOI: 10.17487/RFC2189.

[7] Tony Ballardie, Paul Francis, and Jon Crowcroft. "Core based trees (CBT)". In: *ACM SIGCOMM Computer Communication Review* 23 (4 Oct. 1993), pp. 85–95. ISSN: 0146-4833. DOI: 10.1145/167954.166246.

[8] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. "An efficient multicast protocol for content-based publish-subscribe systems". In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. IEEE Comput. Soc, 1999, pp. 262–272. ISBN: 0-7695-0222-9. DOI: 10.1109/ICDCS.1999.776528.

[9] Guruduth S Banavar and Lukasz Opyrchal. *Method and system for applying cluster-based group multicast to content-based publish-subscribe system*. US Patent 6,336,119. Jan. 2002.

[10] *Behavioral model version 2 (BMv2)*. URL: https://github.com/p4lang/behavioral-model (visited on 02/01/2025).

[11] Michele Benzi and Christine Klymko. "On the Limiting Behavior of Parameter-Dependent Network Centrality Measures". In: *SIAM Journal on Matrix Analysis and Applications* 36 (2 Jan. 2015), pp. 686–706. ISSN: 0895-4798. DOI: 10.1137/130950550.

[12] Supratik Bhattacharyya. *An Overview of Source-Specific Multicast (SSM)*. RFC 3569. July 2003. DOI: 10.17487/rfc3569.

[13] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, Frank Durr, Thomas Kohler, and Kurt Rothermel. "High Performance Publish/Subscribe Middleware in Software-Defined Networks". In: *IEEE/ACM Transactions on Networking* 25 (3 June 2017), pp. 1501–1516. ISSN: 1063-6692. DOI: 10.1109/TNET.2016.2632970.

[14] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13 (7 July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.

[15] R. Boivie, N. Feldman, Y. Imai, W. Livens, and D. Ooms. *Explicit Multicast (Xcast) Concepts and Options*. RFC 5058. Nov. 2007. DOI: 10.17487/rfc5058.

[16] Béla Bollobás and W. Fernandez de la Vega. "The diameter of random regular graphs". In: *Combinatorica* 2 (2 June 1982), pp. 125–134. ISSN: 0209-9683. DOI: 10.1007/BF02579310.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44 (3 July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding metamorphosis". In: *ACM SIGCOMM Computer Communication Review* 43 (4 Sept. 2013), pp. 99–110. ISSN: 0146-4833. DOI: 10.1145/2534169.2486011.

[19] Alejandro Buchmann and Boris Koldehofe. "Complex Event Processing". In: *IT - Information Technology* 51 (5 Sept. 2009), pp. 241–242. ISSN: 2196-7032. DOI: 10.1524/itit.2009.9058.

[20] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. *Internet Group Management Protocol, Version 3*. RFC 3376. Oct. 2002. DOI: 10.17487/rfc3376.

[21] Fengyun Cao and J.P. Singh. "Efficient event routing in content-based publish-subscribe service networks". In: *IEEE INFOCOM 2004*. Vol. 2. IEEE, 2004, pp. 929–940. ISBN: 978-0-7803-8355-5. DOI: 10.1109/INFCOM.2004.1356980.

[22] Antonio Carzaniga. "Architectures for an Event Notification Service Scalable to Wide-Area Networks". PhD thesis. Politecnico di Milano, 1998.

[23] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. "Design and evaluation of a wide-area event notification service". In: *ACM Transactions on Computer Systems* 19 (3 Aug. 2001), pp. 332–383. ISSN: 0734-2071. DOI: 10.1145/380749.380767.

[24] R. Chand and P. Felber. "XNET: a reliable content-based publish/subscribe system". In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* IEEE, 2004, pp. 264–273. ISBN: 0-7695-2239-4. DOI: 10.1109/RELDIS.2004.1353027.

[25] Raphaël Chand and Pascal Felber. "Semantic Peer-to-Peer Overlays for Publish/Subscribe Networks". In: *Euro-Par 2005 Parallel Processing.* Ed. by José C. Cunha and Pedro D. Medeiros. 2005, pp. 1194–1204. ISBN: 978-3-540-31925-2. DOI: 10.1007/11549468_130.

[26] Hyunseok Chang, Fang Hao, Murali Kodialam, T. V. Lakshman, Sarit Mukherjee, and Matteo Varvello. "SNAPS: Seamless Network-Assisted Publish-Subscribe". In: *2022 IFIP Networking Conference (IFIP Networking).* IEEE, June 2022, pp. 1–9. ISBN: 978-3-903176-48-5. DOI: 10.23919/IFIPNetworking55013.2022.9829774.

[27] Hyunseok Chang, Fang Hao, Murali Kodialam, T.V. Lakshman, Sarit Mukherjee, and Matteo Varvello. "Optimized SRv6 Multicasting for Network-Assisted Publish-Subscribe Systems". In: *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR).* IEEE, June 5, 2023, pp. 1–6. ISBN: 978-1-6654-7640-9. DOI: 10.1109/HPSR57248.2023.10147988.

[28] Luciano Jerez Chaves, Islene Calciolari Garcia, and Edmundo Roberto Mauro Madeira. "OFSwitch13: Enhancing ns-3 with OpenFlow 1.3 support". In: *Proceedings of the Workshop on ns-3 - WNS3 '16.* ACM Press, 2016, pp. 33–40. ISBN: 978-1-4503-4216-2. DOI: 10.1145/2915371.2915381.

[29] M. Cilia. "Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach". In: *"International Workshop on Distributed Event-based Systems (DEBS 2004)" W18L Workshop - 26th International Conference on Software Engineering.* Vol. 2004. IEE, 2004, pp. 26–31. ISBN: 978-0-86341-433-6. DOI: 10.1049/ic:20040378.

[30] Mariano Cilia, Christof Bornhövd, and Alejandro P. Buchmann. "Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments". In: *Cooperative Information Systems – 9th International Conference, CoopIS 2001.* Ed. by Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella. 2001, pp. 195–211. ISBN: 978-3-540-44751-1. DOI: 10.1007/3-540-44751-2_16.

[31] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. "Finding community structure in very large networks". In: *Physical Review E* 70 (6 Dec. 2004). ISSN: 1539-3755. DOI: 10.1103/PhysRevE.70.066111.

[32] P4 Language Consortium. *The P4 Language Specification - version 1.1.0*. Jan. 2016.

[33] P4 Language Consortium. *P4 16 Language Specification version 1.2.4*. May 2023.

[34] Gianpaolo Cugola and Alessandro Margara. "Processing flows of information: From data stream to complex event processing". In: *ACM Computing Surveys* 44 (3 June 2012), pp. 1–62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677.

[35] Nigel Deakin, Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service*. Version 2.0. Oracle, Mar. 2013.

[36] S. Deering, W. Fenner, and B. Haberman. *Multicast Listener Discovery (MLD) for IPv6*. RFC 2710. Oct. 1999. DOI: 10.17487/rfc2710.

[37] Cristina Dominicini, Diego Mafioletti, Ana C. Locateli, Rodolfo Villaca, Magnos Martinello, Moises Ribeiro, and Alexander Gorodnik. "PolKA: Polynomial Key-based Architecture for Source Routing in Network Fabrics". In: *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, June 2020, pp. 326–334. ISBN: 978-1-7281-5684-2. DOI: 10.1109/NetSoft48620.2020.9165501.

[38] T. Eckert, M. Menth, and G. Cauchie. *Tree Engineering for Bit Index Explicit Replication (BIER-TE)*. RFC 9262. Oct. 2022. DOI: 10.17487/RFC9262.

[39] Patrick Th. Eugster. "Type-Based Publish/Subscribe: Concepts and Experiences". In: *ACM Transactions on Programming Languages and Systems* 29 (1 Jan. 2007), p. 6. ISSN: 0164-0925. DOI: 10.1145/1180475.1180481.

[40] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. "The Many Faces of Publish/Subscribe". In: *ACM Computing Surveys* 35 (2 June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.

[41] Patrick Th. Eugster and Rachid Guerraoui. "Distributed Programming with Typed Events". In: *IEEE Software* 21 (2 Mar. 2004), pp. 56–64. ISSN: 0740-7459. DOI: 10.1109/MS.2004.1270763.

[42] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. "Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction". In: *ECOOP 2000 — Object-Oriented Programming*. Ed. by Elisa Bertino. 2000, pp. 252–276. ISBN: 978-3-540-45102-0. DOI: 10.1007/3-540-45102-1_13.

[43] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. "Type-Based Publish/Subscribe". PhD thesis. Università della Svizzera Italiana (USI), 2000.

[44] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A scalable, commodity data center network architecture". In: *SIGCOMM Comput. Commun. Rev.* 38 (4 Aug. 2008), pp. 63–74. ISSN: 0146-4833. DOI: 10.1145/1402958.1402967.

[45] B. Fenner, M. Handley, H. Holbrook, I. Kouvelas, R. Parekh, Z. Zhang, and L. Zheng. *Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised).* RFC 7761. Mar. 2016. DOI: 10.17487/RFC7761.

[46] Open Networking Foundation. *OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06).* Mar. 2015.

[47] M. R. Garey, R. L. Graham, and D. S. Johnson. "The Complexity of Computing Steiner Minimal Trees". In: *SIAM Journal on Applied Mathematics* 32 (4 June 1977), pp. 835–859. ISSN: 0036-1399. DOI: 10.1137/0132072.

[48] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979. ISBN: 0-7167-1045-5.

[49] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. "SPADE". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, June 2008, pp. 1123–1134. ISBN: 978-1-6055-8102-6. DOI: 10.1145/1376616.1376729.

[50] Narain H Gehani, Hosagrahar V Jagadish, and Oded Shmueli. "Composite event specification in active databases: Model & implementation". In: *Proceedings of the International Conference on Very Large Data Bases* 92 (1992), pp. 327–338.

[51] The P4.org API Working Group. *P4 Runtime Specification Version 1.3.0.* July 2, 2021. URL: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html (visited on 02/01/2025).

[52] Aric Hagberg, Daniel Schult, and Pieter Swart. "Proceedings of the Python in Science Conference (SciPy): Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference (SciPy2008)* (2008).

[53] Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. "Toward Scalable Systems for Big Data Analytics: A Technology Tutorial". In: *IEEE Access* 2 (2014), pp. 652–687. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2332453.

[54] Liang-Hao Huang, Hsiang-Chun Hsu, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. "Multicast traffic engineering for software-defined networks". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications.* IEEE, Apr. 2016, pp. 1–9. ISBN: 978-1-4673-9953-1. DOI: 10.1109/INFOCOM.2016.7524383.

[55] Liang-Hao Huang, Hui-Ju Hung, Chih-Chung Lin, and De-Nian Yang. "Scalable and bandwidth-efficient multicast for software-defined networks". In: *2014 IEEE Global Communications Conference*. IEEE, Dec. 2014, pp. 1890–1896. ISBN: 978-1-4799-3512-3. DOI: 10.1109/GLOCOM.2014.7037084.

[56] J.Y. Hui and T. Renner. "Queueing analysis for multicast packet switching". In: *IEEE Transactions on Communications* 42 (2/3/4 Feb. 1994), pp. 723–731. ISSN: 0090-6778. DOI: 10.1109/TCOMM.1994.577101.

[57] Misha Hungyo and Mayank Pandey. "SDN based implementation of publish/subscribe paradigm using OpenFlow multicast". In: *2016 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. IEEE, Nov. 2016, pp. 1–6. ISBN: 978-1-5090-2193-2. DOI: 10.1109/ANTS.2016.7947820.

[58] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2022* (2022), pp. 1–7025. DOI: 10.1109/IEEESTD.2022.9844436.

[59] Salekul Islam, Nasif Muslim, and J. William Atwood. "A Survey on Multicasting in Software-Defined Networking". In: *IEEE Communications Surveys & Tutorials* 20 (1 2018), pp. 355–387. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2776213.

[60] Aakash Iyer, Praveen Kumar, and Vijay Mann. "Avalanche: Data center Multicast using software defined networking". In: *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, Jan. 2014, pp. 1–8. ISBN: 978-1-4799-3635-9. DOI: 10.1109/COMSNETS.2014.6734903.

[61] Angur Jarman. *Hierarchical cluster analysis: Comparison of single linkage, complete linkage, average linkage and centroid linkage method*. Feb. 2020. DOI: 10.13140/RG.2.2.11388.90240.

[62] Zbigniew Jerzak and Christof Fetzer. "Bloom filter based routing for content-based publish/subscribe". In: *Proceedings of the second international conference on Distributed event-based systems*. ACM, July 2008, pp. 71–81. ISBN: 9781605580906. DOI: 10.1145/1385989.1385999.

[63] Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. "Big Data Processing in Cloud Computing Environments". In: *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*. IEEE, Dec. 2012, pp. 17–23. ISBN: 978-1-4673-5064-8. DOI: 10.1109/I-SPAN.2012.9.

[64] Wen-Kang Jia. "A Scalable Multicast Source Routing Architecture for Data Center Networks". In: *IEEE Journal on Selected Areas in Communications* 32 (1 Jan. 2014), pp. 116–123. ISSN: 0733-8716. DOI: 10.1109/JSAC.2014.140111.

[65] Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen. "Extending Dijkstra's shortest path algorithm for software defined networking". In: *The 16th Asia-Pacific Network Operations and Management Symposium*. IEEE, Sept. 2014, pp. 1–4. ISBN: 978-4-88552-288-8. DOI: 10.1109/APNOMS.2014.6996609.

[66] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. "LIPSIN: Line speed publish/subscribe inter-networking". In: *ACM SIGCOMM Computer Communication Review* 39 (4 Aug. 2009), pp. 195–206. ISSN: 0146-4833. DOI: 10.1145/1594977.1592592.

[67] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. "Compiling Packet Programs to Reconfigurable Switches". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. 2015, pp. 103–115.

[68] Sheetal S. Joshi and Ketki R. Kulkarni. "Internet of Things: An Overview". In: *IOSR Journal of Computer Engineering* 18 (04 Apr. 2016), pp. 117–121. ISSN: 2278-8727. DOI: 10.9790/0661-180405117121.

[69] Keith Kirkpatrick. "Software-defined networking". In: *Communications of the ACM* 56 (9 Sept. 2013), pp. 16–19. ISSN: 0001-0782. DOI: 10.1145/2500468.2500473.

[70] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. "The Internet Topology Zoo". In: *IEEE Journal on Selected Areas in Communications* 29 (9 Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[71] Blaine Kohl. *Ethernet Jumbo Frames*. Tech. rep. Ethernet Alliance, 2009.

[72] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103 (1 Jan. 2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[73] Bob Lantz, Brandon Heller, and Nick McKeown. "A network in a laptop: Rapid prototyping for software-defined networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, Oct. 2010, pp. 1–6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466.

[74] Rhys Lewis. *Advanced Message Applications with MSMQ and MQSeries*. Que Professional, Dec. 1999. ISBN: 978-0-7897-2023-8.

[75] Xin Li, Martina Eckert, José-Fernán Martinez, and Gregorio Rubio. "Context Aware Middleware Architectures: Survey and Challenges". In: *Sensors* 15 (8 Aug. 2015), pp. 20570–20607. ISSN: 1424-8220. DOI: 10.3390/s150820570.

[76] Ying-Dar Lin, Yuan-Cheng Lai, Hung-Yi Teng, Chun-Chieh Liao, and Yi-Chih Kao. "Scalable multicasting with multiple shared trees in software defined networking". In: *Journal of Network and Computer Applications* 78 (Jan. 2017), pp. 125–133. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.11.014.

[77] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. "Survey of real-time processing systems for big data". In: *Proceedings of the 18th International Database Engineering & Applications Symposium on - IDEAS '14*. ACM Press, 2014, pp. 356–361. ISBN: 978-1-4503-2627-8. DOI: 10.1145/2628194.2628251.

[78] Carlos A.B. Macapuna, Christian Esteve Rothenberg, and Magalhaes F. Mauricio. "In-packet Bloom filter based data center networking with distributed OpenFlow controllers". In: *2010 IEEE Globecom Workshops*. IEEE, Dec. 2010, pp. 584–588. ISBN: 978-1-4244-8863-6. DOI: 10.1109/GLOCOMW.2010.5700387.

[79] J. Legatheaux Martins and Sergio Duarte. "Routing algorithms for content-based publish/subscribe systems". In: *IEEE Communications Surveys & Tutorials* 12 (2010), pp. 39–58. ISSN: 1553-877X. DOI: 10.1109/SURV.2010.020110.00065.

[80] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38 (2 Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[81] D. McPherson and B. Dykes. *VLAN Aggregation for Efficient IP Address Allocation*. RFC 3069. Feb. 2001. DOI: 10.17487/rfc3069.

[82] Daniel Merling, Steffen Lindner, and Michael Menth. "Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4". In: *IEEE Access* 9 (2021), pp. 34500–34514. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3061763.

[83] C. Metz and M. Tatipamula. "A look at native IPv6 multicast". In: *IEEE Internet Computing* 8 (4 July 2004), pp. 48–53. ISSN: 1089-7801. DOI: 10.1109/MIC.2004.1.

[84] Brenda Michelson. *Event-Driven Architecture Overview*. Patricia Seybold Group, Feb. 2006. DOI: 10.1571/bda2-2-06cc.

[85] J. Moy. *MOSPF: Analysis and Experience*. RFC 1585. Mar. 1994. DOI: 10.17487/rfc1585.

[86] Gero Mühl, Ludger Fiege, and Alejandro Buchmann. "Filter Similarities in Content-Based Publish/Subscribe Systems". In: *Trends in Network and Pervasive Computing — ARCS 2002*. Ed. by Hartmut Schmeck, Theo Ungerer, and Lars Wolf. Mar. 2002, pp. 224–238. ISBN: 978-3-540-45997-2. DOI: 10.1007/3-540-45997-9_17.

[87] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. "Exploiting IP Multicast in Content-Based Publish-Subscribe Systems". In: *Middleware 2000 – IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Ed. by Joseph Sventek and Geoffrey Coulson. 2000, pp. 185–207. ISBN: 978-3-540-67352-1. DOI: 10.1007/3-540-45559-0_10.

[88] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. "Big Data technologies: A survey". In: *Journal of King Saud University - Computer and Information Sciences* 30 (4 Oct. 2018), pp. 431–448. ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2017.06.001.

[89] *P4 16 Portable Switch Architecture (PSA)*. URL: https://p4.org/p4-spec/docs/PSA.html (visited on 02/01/2025).

[90] *P4 Architecture Working Group Charter*. URL: https://github.com/p4lang/p4-spec/blob/main/p4-16/psa/charter/P4_Arch_Charter.adoc (visited on 02/01/2025).

[91] *P4 compiler for behavioral model (p4c-behavioral)*. URL: https://github.com/p4lang/p4c-behavioral/tree/master (visited on 02/01/2025).

[92] *P4 Reference Compiler (p4c)*. URL: https://github.com/p4lang/p4c (visited on 02/01/2025).

[93] *P4 v1model*. URL: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4 (visited on 02/01/2025).

[94] Helge Parzyjegla, Christian Wernecke, and Gero Mühl. "Content-based Publish/Subscribe in Software-defined Networks". In: *2. GI/ITG KuVS-Fachgespräch Network Softwarization* (May 8, 2020). DOI: 10.15496/publikation-41784.

[95] Helge Parzyjegla, Christian Wernecke, Gero Mühl, Eike Schweissguth, and Dirk Timmermann. "Implementing content-based publish/subscribe with OpenFlow". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, Apr. 2019, pp. 1392–1395. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297589.

[96] *Performance of BMv2*. URL: https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md (visited on 02/01/2025).

[97] J. Postel. *The TCP Maximum Segment Size and Related Topics*. RFC 0879. Nov. 1983. DOI: 10.17487/rfc0879.

[98] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031. Jan. 2001. DOI: 10.17487/rfc3031.

[99] Christian Esteve Rothenberg, Carlos Alberto Braz Macapuna, Maurício Ferreira Magalhães, Fábio Luciano Verdi, and Alexander Wiesmaier. "In-packet Bloom filters: Design and networking applications". In: *Computer Networks* 55 (6 Apr. 2011), pp. 1364–1378. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.12.005.

[100] Julius Rückert, Jeremias Blendin, Rhaban Hark, and David Hausheer. "Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With DynSdm". In: *IEEE Transactions on Network and Service Management* 13 (4 Dec. 2016), pp. 754–767. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2607281.

[101] Stuart Sechrest. *An introductory 4.4 BSD interprocess communication tutorial*. Tech. rep. Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Apr. 1986.

[102] Chuck Semeria and Tom Maufer. *Introduction to IP Multicast Routing*. Internet Draft. 1997.

[103] A. Shaikh and Kang Shin. "Destination-driven routing for low-cost multicast". In: *IEEE Journal on Selected Areas in Communications* 15 (3 Apr. 1997), pp. 373–381. ISSN: 0733-8716. DOI: 10.1109/49.564135.

[104] Jeferson Santiago da Silva, Francois-Raymond Boyer, Laurent-Olivier Chiquette, and J.M. Pierre Langlois. "Extern Objects in P4: an ROHC Header Compression Scheme Case Study". In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, June 2018, pp. 517–522. ISBN: 978-1-5386-4633-5. DOI: 10.1109/NETSOFT.2018.8460108.

[105] *Simple Switch Thrift API*. URL: https://nsg-ethz.github.io/p4-utils/p4utils.utils.sswitch_thrift_API.html (visited on 02/01/2025).

[106] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. "Jellyfish: Networking data centers randomly". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 225–238.

[107] Thirunavukkarasu Sivaharan, Gordon Blair, and Geoff Coulson. "GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing". In: *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*. Ed. by Robert Meersman and Zahir Tari. 2005, pp. 732–749. ISBN: 978-3-540-32116-3. DOI: 10.1007/11575771_46.

[108] Henning Stubbe. "P4 Compiler & Interpreter: A Survey". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)* 47 (2017). DOI: 10.2313/NET-2017-05-1_07.

[109] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. "PLEROMA: A SDN-based high performance publish/subscribe middleware". In: *Proceedings of the 15th International Middleware Conference on - Middleware '14*. ACM Press, 2014, pp. 217–228. ISBN: 978-1-4503-2785-5. DOI: 10.1145/2663165.2663338.

[110] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and Practice of Bloom Filters for Distributed Systems". In: *IEEE Communications Surveys & Tutorials* 14 (2012), pp. 131–155. ISSN: 1553-877X. DOI: 10.1109/SURV.2011.031611.00024.

[111] Rex S Toh and Richard G Higgins. "The Impact of Hub and Spoke Network Centralization and Route Monopoly on Domestic Airline Profitability". In: *Transportation Journal* 24 (1985), pp. 16–27. ISSN: 0041-1612.

[112] Pier Luigi Ventre, Stefano Salsano, Marco Polverini, Antonio Cianfrani, Ahmed Abdelsalam, Clarence Filsfils, Pablo Camarillo, and Francois Clad. "Segment Routing: A Comprehensive Survey of Research Activities, Standardization Efforts, and Implementation Results". In: *IEEE Communications Surveys & Tutorials* 23 (1 2021), pp. 182–221. ISSN: 1553-877X. DOI: 10.1109/COMST.2020.3036826.

[113] D. Waitzman, C. Partridge, and S.E. Deering. *Distance Vector Multicast Routing Protocol*. RFC 1075. Nov. 1988. DOI: 10.17487/rfc1075.

[114] Christian Wernecke, Helge Parzyjegla, and Gero Mühl. "Implementing Content-based Publish/Subscribe on the Network Layer with P4". In: *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2020, pp. 144–149. ISBN: 978-1-7281-8159-2. DOI: 10.1109/NFV-SDN50289.2020.9289860.

[115] Christian Wernecke, Helge Parzyjegla, and Gero Mühl. "P4-programmable Data Plane for Content-based Publish/Subscribe". In: *3. GI/ITG KuVS-Fachgespräch Network Softwarization* (Apr. 7, 2022). DOI: 10.15496/publikation-67450.

[116] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, Eike Schweissguth, and Dirk Timmermann. "Stitching Notification Distribution Trees for Content-based Publish/Subscribe with P4". In: *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2020, pp. 100–104. ISBN: 978-1-7281-8159-2. DOI: 10.1109/NFV-SDN50289.2020.9289916.

[117] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, Eike Schweissguth, and Dirk Timmermann. "Evaluating P4-based Virtual Delivery Trees for Content-based Publish/Subscribe". In: *2022 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2022, pp. 78–84. ISBN: 978-1-6654-7334-7. DOI: 10.1109/NFV-SDN56302.2022.9974746.

[118] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, Eike Schweissguth, and Dirk Timmermann. *Virtual Delivery Trees Evaluation Results*. Sept. 2022. DOI: 10.5281/zenodo.7129010.

[119] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, and Dirk Timmermann. "Realizing Content-Based Publish/Subscribe with P4". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2018, pp. 1–7. ISBN: 978-1-5386-8281-4. DOI: 10.1109/NFV-SDN.2018.8725641.

[120] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Eike Schweissguth, and Dirk Timmermann. "Flexible Notification Forwarding for Content-Based Publish/Subscribe Using P4". In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2019, pp. 1–5. ISBN: 978-1-7281-4545-7. DOI: 10.1109/NFV-SDN47374.2019.9040048.

[121] Michael Weyrich, Jan-Philipp Schmidt, and Christof Ebert. "Machine-to-Machine Communication". In: *IEEE Software* 31 (4 July 2014), pp. 19–23. ISSN: 0740-7459. DOI: 10.1109/MS.2014.87.

[122] I. Wijnands, E. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin. *Multicast Using Bit Index Explicit Replication (BIER)*. Ed. by I. Wijnands and E. Rosen. RFC 8279. Nov. 2017. DOI: 10.17487/RFC8279.

[123] R. Xu and D. Wunsch. "Survey of Clustering Algorithms". In: *IEEE Transactions on Neural Networks* 16 (3 May 2005), pp. 645–678. ISSN: 1045-9227. DOI: 10.1109/TNN.2005.845141.

[124] Zhonghua Yang and Keith Duddy. "CORBA: A Platform for Distributed Object Computing". In: *ACM SIGOPS Operating Systems Review* 30 (2 Apr. 1996), pp. 4–31. ISSN: 0163-5980. DOI: 10.1145/232302.232303.

Eigenständigkeitserklärung

Ich versichere durch eigenhändige Unterschrift, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen bzw. von anderen Autoren/Urhebern übernommen worden sind, habe ich als solche kenntlich gemacht. Dies gilt auch für Zeichnungen, Kartenskizzen und Darstellungen.

Die Arbeit ist noch nicht veröffentlicht und in gleicher oder ähnlicher Weise weder gleichzeitig noch zu einem anderen Zeitpunkt als Studienleistung zur Anerkennung oder Bewertung vorgelegt worden. Ich weiß, dass bei Abgabe einer falschen Versicherung die Prüfung als nicht bestanden zu gelten hat und als Täuschungsversuch zu werten ist.

Rostock, 13 Juni 2024 Christian Wernecke