RESOURCE ALLOCATION ALGORITHMS FOR EVENT-BASED ENTERPRISE SYSTEMS

by

Alex King Yeung Cheung

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

Copyright © 2011 by Alex King Yeung Cheung
Abstract

Resource Allocation Algorithms for Event-Based Enterprise Systems

Alex King Yeung Cheung
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2011

Distributed event processing systems suffer from poor scalability and inefficient resource usage caused by load distributions typical in real-world applications. The results of these shortcomings are availability issues, poor system performance, and high operating costs. This thesis proposes three remedies to solve these limitations in content-based publish/subscribe, which is a practical realization of an event processing system. First, we present a load balancing algorithm that relocates subscribers to distribute load and avoid overloads. Second, we propose publisher relocation algorithms that reduces both the load imposed onto brokers and delivery delay experienced by subscribers. Third, we present “green” resource allocation algorithms that allocate as few brokers as possible while maximizing their resource usage efficiency by reconfiguring the publishers, subscribers, and the broker topology. We implemented and evaluated all of our approaches on an open source content-based publish/subscribe system called PADRES and evaluated them on SciNet, PlanetLab, a cluster testbed, and in simulations to prove the effectiveness of our solutions. Our evaluation findings are summarized as follows. One, the proposed load balancing algorithm is effective in distributing and balancing load originating from a single server to all available servers in the network. Two, our publisher relocation algorithm reduces the average input load of the system by up to 68%, average broker message rate by up to 85%, and average delivery delay by up to 68%. Three, our resource allocation algorithm reduces the average broker message rate even further by up to 92% and the number of allocated brokers by up to 91%.
Acknowledgements

Enormous thank you to Mom (Cindy), Dad (William), sister (Annie), Mr. Biggy, Mr. Dum Dum, and Mr. Chestnut for all of their support in my Master and PhD studies even though they have no clue what I am doing. This thesis as well as my life is dedicated to them.

Thanks to my relatives and KUG friends for their constant encouragement and persistent pressure by endlessly asking "When are you graduating?" - by the way, this is one of the sensitive questions that one should never ask a PhD student. See phdcomics.com.

Thanks to everyone who graduated from or are still in MSRG for their friendly companionship and guidance.

Thanks to CA Technologies for supporting our project (especially Serge Mankovskii) and inviting us over to Las Vegas to have fun...err, I mean to demo our project in the CA tradeshow. That is a trip I will never forget: littering the busy road near Hoover Dam with all of our luggage and water (thanks V.M.), meditating over the tip of a cliff at Grand Canyon, and getting lost finding that "dog-shed" bus stop in Sausalito.

Thanks to the generous donors of the Ewing Rae, Allen Yen, and especially Viola Carless Smith scholarships for funding my PhD studies.

Last but not least, thanks to my supervisor, Professor Hans-Arno Jacobsen, for allowing me to work not only independently and remotely in all these years, but also for giving me a lot of research support, guidance, and freedom in terms of the research direction.
Contents

1 List of Terminology xii

2 Introduction 1
   2.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   2.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
      2.2.1 Problem of Availability and Scalability . . . . . . . . . . . . . . . . . . . . 4
      2.2.2 Problem of Minimizing System Load and Delivery Delay . . . . . . . . . . 5
      2.2.3 Problem of Minimal Resource Allocation . . . . . . . . . . . . . . . . . . . . 7
   2.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.4 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3 Background and Related Work 12
   3.1 Content-based Publish/Subscribe Routing . . . . . . . . . . . . . . . . . . . . . . 12
   3.2 Content-based Publish/Subscribe Language . . . . . . . . . . . . . . . . . . . . . 13
   3.3 Publish/Subscribe Architectures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   3.4 Load Balancing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   3.5 Load Balancing in Content-based Publish/Subscribe . . . . . . . . . . . . . . . . . 18
   3.6 Publisher Migration Protocols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
   3.7 Broker Overlay and Publisher Placement Reconfigurations . . . . . . . . . . . . . . 20
   3.8 Subscription Clustering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4 Dynamic Load Balancing Algorithm 23
   4.1 Load Balancing Framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.1.1 Structuring the Overlay into Clusters ........................................... 24
4.1.2 Messaging Framework ................................................................. 26
4.1.3 Load Detection Framework ......................................................... 26
4.1.4 Mediation Protocols ................................................................. 40
4.2 Load Estimation Algorithms ......................................................... 42
  4.2.1 Estimating Load Requirements of Subscriptions ............................ 43
  4.2.2 Modeling and Estimating Performance Metrics ............................. 46
4.3 Offload Algorithms ................................................................. 47
  4.3.1 Input Offload Algorithm ............................................................. 50
  4.3.2 Match Offload Algorithm ........................................................... 52
  4.3.3 Output Offload Algorithm .......................................................... 56
  4.3.4 Random Algorithm ................................................................. 57

5 Publisher Placement Algorithms .............................................. 59
  5.1 The POP Placement Algorithm .................................................... 59
    5.1.1 Phase 1: Distributed Trace Algorithm ..................................... 60
    5.1.2 Phase 2: Decentralized Broker Selection Algorithm .................. 62
    5.1.3 Phase 3: Publisher Migration Protocol .................................... 64
  5.2 The GRAPE Placement Algorithm .............................................. 65
    5.2.1 Phase 1: Distributed Publication Tracing ................................ 66
    5.2.2 Phase 2: Broker Selection Algorithm .................................... 68

6 Resource Allocation Algorithms ............................................. 72
  6.1 Phase 1: Resource Allocation Framework ..................................... 72
    6.1.1 Information Gathering .......................................................... 73
    6.1.2 Subscription and Publisher Profiles ....................................... 74
  6.2 Phase 2: Subscription Allocation Algorithms .................................. 76
    6.2.1 FBF - Fastest Broker First .................................................. 76
    6.2.2 BIN PACKING ................................................................. 77
    6.2.3 CRAM - Clustering with Resource Awareness and Minimization .... 77
6.3 Phase 3: Broker Overlay Construction 82
   6.3.1 Optimization 1 - Eliminate Pure Forwarding Brokers 83
   6.3.2 Optimization 2 - Takeover Children Broker Roles 83
   6.3.3 Optimization 3 - Best-Fit Broker Replacement 84

7 Evaluation 86
   7.1 Evaluation of the Load Balancing Algorithm 87
      7.1.1 Experiment Setup 87
      7.1.2 Local Load Balancing Results 92
      7.1.3 Global Load Balancing Results 101
   7.2 Evaluation of the Publisher Placement Algorithms 105
      7.2.1 Experiment Setup 106
      7.2.2 Evaluation Results with the Enterprise Workload 107
      7.2.3 Evaluation Results with the Random Workload 112
      7.2.4 Impact of GRAPE’s Minimization Weight 113
      7.2.5 Impact of POP and GRAPE’s Sampling Trigger 114
   7.3 Evaluation of the Resource Allocation Algorithms 115
      7.3.1 Comparison Against Baseline and Related Approaches 115
      7.3.2 Experiment Setup 117
      7.3.3 Comparison of Baseline, Related, and Proposed Approaches 118
      7.3.4 Breakdown Analysis of Performance Gains 123
      7.3.5 Evaluation of CRAM’s Closeness Metrics 124
      7.3.6 Evaluation of the Sorting Algorithms 125
      7.3.7 Evaluation of the PAIRWISE and AUTOMATIC Algorithms 126
      7.3.8 Impact of Bit Count in the Bit Vector 126

8 Conclusions 128
   8.1 Summary 128
   8.2 Future Work 131
List of Tables

4.1 Bit vector example ................................................. 44
4.2 Properties of all offload algorithms .............................. 48

7.1 Default values of load balancing parameters used in experiments ................... 90
7.2 Broker specifications in local load balancing experiment ......................... 91
7.3 Client specifications in local load balancing experiment ......................... 91
7.4 Broker specifications in global load balancing experiment ....................... 102
# List of Figures

2.1 SPECjms2007 workload scenario with a cloud deployment example ............................ 2

3.1 Example of content-based publish/subscribe routing ............................................. 13

3.2 Example of the poset data structure ................................................................. 15

4.1 Components of the load balancer ................................................................. 23

4.2 PEER architecture .................................................. 25

4.3 State transition diagram for an edge broker .................................................. 30

4.4 State transition diagram for a cluster-head broker .......................................... 30

4.5 Flowchart showing the local detection algorithm ............................................ 35

4.6 Flowchart showing the global detection algorithm ........................................... 36

5.1 Example of Publisher Profile Table ................................................................. 60

5.2 All possible outcomes of POP’s broker selection algorithm ................................. 63

5.3 String and bit vector representation of delivered publications .............................. 66

6.1 Clustering two subscriptions to form a new subscription ..................................... 74

6.2 One-to-many clustering approach of optimization 3 .......................................... 82

6.3 Broker overlay construction optimizations ....................................................... 85

7.1 POP and GRAPE in the PADRES broker .......................................................... 87

7.2 Local (left) and global (right) load balancing experiment network topologies ....... 92

7.3 Load distribution in local load balancing on various platforms ............................ 94

7.4 Output load distribution in local load balancing using the naive algorithm ........... 95
7.5 Average end-to-end delivery delay in local load balancing on various setups . . . 96
7.6 Subscriber distribution in local load balancing on various setups . . . . . . 98
7.7 Message overhead in local load balancing . . . . . . . . . . . . . . . . . . . . . 98
7.8 System scalability by adding more edge brokers into a cluster . . . . . . . 99
7.9 Load estimation accuracy on heterogeneous cluster testbed and simulation . . 100
7.10 Load distribution among clusters in global load balancing . . . . . . . . . . . . 103
7.11 System scalability by adding more clusters . . . . . . . . . . . . . . . . . . . . 104
7.12 Deployment specs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
7.13 Experiment results on PlanetLab and the cluster testbed . . . . . . . . . . . 108
7.14 Experiment results on the cluster testbed with overlay fanout of 4 . . . . . 109
7.15 Experiment results on PlanetLab (PL) and the cluster testbed (CL) . . . . 111
7.16 Micro experiments of GRAPE on the cluster testbed . . . . . . . . . . . . . 113
7.17 Micro experiments of POP on the cluster testbed . . . . . . . . . . . . . . . . 114
7.18 Experiment results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
7.19 Experiment results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
7.20 Experiments showing the breakdown of performance gains . . . . . . . . . . 123
7.21 Experiment results comparing closeness metrics . . . . . . . . . . . . . . . . . 125
7.22 Experiment results of sorting algorithms and related approaches . . . . . . 126
7.23 Experiment results showing the impact of bit vector length . . . . . . . . . . 127
# List of Algorithms

1. `calcOffloadSet(localInfo, remoteInfo, poset, PRESSProfiles)` ............................ 49
2. `calcOffloadSet(localInfo, remoteInfo, poset, PRESSProfiles)` ............................ 53
3. `calcPhase1OffloadSet(localInfo, remoteInfo, poset, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList, offloadSet, ignoreSet)` ............................ 54
4. `calcPhase2OffloadSet(localInfo, remoteInfo, PRESSProfiles, p2List, offloadSet, ignoreSet)` ............................ 55
5. `calcAvgDelay(stats, cumDelay, currBroker, prevBroker)` ............................ 68
6. `calcTotalMsgRate(currBroker, prevBroker)` ............................ 69
7. `calcDownstreamBV(currBroker, prevBroker)` ............................ 69
# Chapter 1

## List of Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIA</td>
<td>Broker Information Answer</td>
<td>Sec. 6.1.1</td>
</tr>
<tr>
<td>BIR</td>
<td>Broker Information Request</td>
<td>Sec. 6.1.1</td>
</tr>
<tr>
<td>CBC</td>
<td>CROC Back-end Component</td>
<td>Sec. 6.1</td>
</tr>
<tr>
<td>CGS</td>
<td>Covered GIF Set</td>
<td>Sec. 6.2.3</td>
</tr>
<tr>
<td>CRAM</td>
<td>Clustering with Resource Awareness and Minimization</td>
<td>Sec. 6.2</td>
</tr>
<tr>
<td>CROC</td>
<td>Coordinator for Reconfiguring the Overlay and Clients</td>
<td>Sec. 6.1</td>
</tr>
<tr>
<td>CSS</td>
<td>Covering Subscription Set</td>
<td>Sec. 3.2</td>
</tr>
<tr>
<td>FBF</td>
<td>Fastest Broker First</td>
<td>Sec. 6.2</td>
</tr>
<tr>
<td>GDSN</td>
<td>Global Data Synchronization Network</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>GIF</td>
<td>Group of Identical Filters</td>
<td>Sec. 6.2.3</td>
</tr>
<tr>
<td>GooPS</td>
<td>Google Publish/Subscribe</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>GRAPE</td>
<td>Greedy Relocation Algorithm for Publishers of Events</td>
<td>Sec. 2.2.2</td>
</tr>
<tr>
<td>IOS</td>
<td>Intersect Over Sum</td>
<td>Sec. 6.2.3</td>
</tr>
<tr>
<td>IOU</td>
<td>Intersect Over Union</td>
<td>Sec. 6.2.3</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td>Sec. 7.1.1</td>
</tr>
<tr>
<td>MCM</td>
<td>Migration Complete Message</td>
<td>Sec. 5.1.3</td>
</tr>
<tr>
<td>MSRG</td>
<td>Middleware Systems Research Group</td>
<td>Sec. 7</td>
</tr>
<tr>
<td>MUM</td>
<td>Migration Update Message</td>
<td>Sec. 5.1.3</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
<td>Reference</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>PADRES</td>
<td>Publish/Subscribe Applied to Distributed Resource Scheduling</td>
<td>Sec. 2.2.1</td>
</tr>
<tr>
<td>PEER</td>
<td>Padres Efficient Event Routing</td>
<td>Sec. 4.1.1</td>
</tr>
<tr>
<td>PIE</td>
<td>Padres Information Exchange</td>
<td>Sec. 4.1.3</td>
</tr>
<tr>
<td>POP</td>
<td>Publisher Optimistic Placement</td>
<td>Sec. 2.2.2</td>
</tr>
<tr>
<td>Poset</td>
<td>Partially-ordered set</td>
<td>Sec. 3.2</td>
</tr>
<tr>
<td>PPTTable</td>
<td>Publisher Profile Table</td>
<td>Sec. 5.1.1</td>
</tr>
<tr>
<td>PRESS</td>
<td>Padres Real-time Event-to-Subscription Spectrum</td>
<td>Sec. 4.2.1</td>
</tr>
<tr>
<td>RAM</td>
<td>Relocation Answer Message</td>
<td>Sec. 5.1.2</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio-Frequency Identification</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>RRM</td>
<td>Relocation Request Message</td>
<td>Sec. 5.1.2</td>
</tr>
<tr>
<td>RSS</td>
<td>Really Simple Syndication</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>TIA</td>
<td>Trace Information Answer</td>
<td>Sec. 5.2.1</td>
</tr>
<tr>
<td>TIR</td>
<td>Trace Information Request</td>
<td>Sec. 5.2.1</td>
</tr>
<tr>
<td>TRM</td>
<td>Trace Result Message</td>
<td>Sec. 5.1.1</td>
</tr>
<tr>
<td>TTL</td>
<td>Time-To-Live</td>
<td>Sec. 4.1.2</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
<td>Sec. 6.1.1</td>
</tr>
</tbody>
</table>
Chapter 2

Introduction

In this chapter, we outline real world use cases of event processing systems as our motivation. Then we identify the problems that this thesis addresses which include availability issues, poor system performance, and high operating costs. We describe how we address these problems by detailing our contributions. Finally, we outline the organization of this thesis.

2.1 Motivation

Distributed content-based publish/subscribe is one realization of an event processing system that is widely used in large-scale distributed applications because it allows processes to communicate asynchronously in a loosely and decoupled manner. This property gives systems higher modularity and reuse as well as easier maintainability. In publish/subscribe, loose-coupling is achieved by simply having producers publish information into the network without knowing the identity, location, and number of subscribers. Likewise, consumers subscribe to specific information without knowing the identity, location, and number of publishers. Topic-based publish/subscribe systems [64, 17, 19] too offer loose-coupling benefits, but lack the language expressiveness, finer grained in-network filtering, and complex event processing capabilities of content-based publish/subscribe systems [9, 33, 14, 71, 91, 49]. The added flexibility of content-based filtering is already realized in commercial systems such as GooPS [72], Google’s publish/subscribe system that integrates its web applications; SuperMontage [86],
Figure 2.1: SPECjms2007 workload scenario with a cloud deployment example

Tibco’s publish/subscribe distribution network for Nasdaq’s quote- and order-processing system; and GDSN (Global Data Synchronization Network) [37], a global publish/subscribe network that allows suppliers and retailers to exchange timely and accurate supply chain data. All these systems are distributed in nature, which is also the focus of this thesis. Being distributed not only gives scalability and fault tolerance properties, but also allows organizations to communicate with each other while being able to restrict local sensitive messages within their own data centers.

A motivating application scenario is a supermarket supply chain with RFID tracked goods in GDSN, as modeled by the SPECjms2007 workload [76]. Figure 2.1 illustrates the application scenario with one possible design of the publish/subscribe cloud. The cloud consists of brokers (illustrated as yellow circles) that are interconnected on an overlay network. Publish/subscribe clients (not shown) at each site connect to one of the brokers in their local network to access the publish/subscribe infrastructure. SPECjms2007 models seven interactions between the company headquarters (HQ), supermarkets (SM), distribution centers (DC), and suppliers (SP).
The interactions involve both multicast and point-to-point communications, both of which are supported by the publish/subscribe paradigm. Here, we simply present the interaction involving order/shipment handling between suppliers and distribution centers to demonstrate our point:

1. A DC sends a call for offers to all SPs that supply the types of goods that need to be ordered.
2. SPs that can deliver the goods send offers to the DC.
3. Based on the offers, the DC selects an SP and sends a purchase order to it.
4. The SP sends a confirmation to the DC and an invoice to the HQ. It then ships the ordered goods.
5. The shipment arrives at the DC and is registered by RFID readers upon entering the DC’s warehouse.
6. The DC sends a delivery confirmation to the SP.
7. The DC sends transaction statistics to the HQ.

2.2 Problem Statement

This thesis focuses on three unique problem areas in content-based publish/subscribe systems: (1) increase availability and scalability of the system through dynamic load balancing [24], (2) increase scalability by minimizing system load while decreasing message hop count and delivery delay through dynamic placement of publishers [25], and (3) further increase scalability by minimizing system load relative to (2) while simultaneously minimizing the number of allocated brokers through reconfiguring publishers, subscribers, and the broker overlay [27]. The following chapters in this thesis are divided and ordered according to these research problems stated above for clear-cut and consistent presentation, respectively. This thesis does not focus on addressing any fault tolerance concerns.

All of the approaches presented in this thesis are targeted towards enterprise-grade messaging systems consisting of hundreds to thousands of dedicated servers under administrative
control [72, 81, 35]. These servers are assumed to be connected to each other on a static overlay network. Applications of our approaches include multiplayer online games [12], decentralized business process execution [77, 44, 61, 53], workflow management [30], business activity monitoring [34], network and systems monitoring [60, 88], automated service composition [41], RSS filtering [70, 75], resource discovery [92] and the Enterprise Service Bus of Service Oriented Architecture infrastructures [43, 56].

2.2.1 Problem of Availability and Scalability

In any distributed publish/subscribe system, brokers located at different geographical areas may suffer from uneven load distribution due to different population densities, interests, and usage patterns of end-users. Just from the interaction pattern illustrated in Figure 2.1 previously, it is possible that brokers at any one of SP, DC, or HQ can be overloaded due to unexpected high order rates (such as the first business day after a long weekend), heterogeneous servers at different sites, reduced server resources due to failures, network congestion, high CPU and/or bandwidth usage from other processes on the same server, etc. Focusing on publish/subscribe alone, a broker can be overloaded in two ways. First, the broker can be overloaded if the incoming message rate into the broker exceeds the processing/matching rate of the matching engine. This effect is exacerbated if the number of subscribers is large because the matching rate is inversely proportional to the number of subscriptions in the matching engine [33]. Second, overload can also occur if the output transmission rate exceeds the total available output bandwidth. In both cases, queues accumulate with increasingly more messages waiting to be processed, resulting in increasingly higher processing and delivery delays. Worse yet, the broker may crash when it runs out of memory from queueing too many messages.

The matching rate and both the incoming and outgoing message rates determine the load of a broker. In turn, these factors depend on the number and nature of subscriptions that the broker services. Thus, load balancing is possible by offloading subscriptions from higher loaded to lesser loaded brokers. However, the use of a generic load balancing scheme with no publish/subscribe context is not a viable option. Our experiments on PlanetLab and a cluster testbed proves this point by showing that random selection of subscriptions makes the
load balancer unstable and jeopardizes the entire system even if the number of subscriptions
offloaded is meaningfully calculated. Hence, we develop a new load balancing algorithm that
distributes load by offloading strategically chosen subscriptions from heavily loaded brokers to
less loaded brokers.

2.2.2 Problem of Minimizing System Load and Delivery Delay

Many filter-based publish/subscribe systems assume that publishers and subscribers join the
broker federation by connecting to the closest broker [52, 67, 20] or to any broker with no restric-
tions [14, 30, 9, 71, 48]. The former assumption may minimize the transmission delay between
the client and broker, and the latter may provide more freedom of choice for the client. Re-
gardless, both policies introduce an unpredictable number of overlay network hops between the
publisher and subscriber that may hinder system performance and result in high delivery delays.
This problem is particularly important in commercial publish/subscribe systems that cannot
tolerate server overloads and unexpected response times such as GooPS [72], SuperMontage
[86], and GDSN [37]. Reducing in-network processing and transmission delays on publication
messages has previously been addressed by reconfiguring the broker topology [7, 46, 57], clus-
tering subscribers into multicast groups to limit publication propagation only among interested
peers [89, 2, 65, 73, 74, 15, 90], or incorporating multicast-groups with filter-based approaches
[13].

In this thesis, we show that strategic placement of publishers in a content-based pub-
lish/subscribe network can improve system scalability, robustness, and performance. We present
two different placement algorithms, POP (Publisher Optimistic Placement) and GRAPE (Greedy
Relocation Algorithm for Publishers of Events), to intelligently relocate publishers while keeping
the broker overlay intact to minimize both the average end-to-end delivery delay and system
load. Both POP and GRAPE follow a 3-Phase operational design: (1) gather publication de-
livery statistics on the publishers’ publications, (2) identify the target broker to relocate the
publisher to, and (3) transparently migrate the publisher to the target broker. Each phase
of POP and GRAPE contribute to the algorithms’ dynamic, scalable, robust, and transparent
properties. Both algorithms are dynamic by periodically making relocation decisions based on
live publication delivery patterns. Both are scalable thanks to the use of distributed design that scales with the number of brokers and clients in the network. Both are robust because an instance of POP or GRAPE runs on every broker to rule out the possibility of any single point of failure. Lastly, both are transparent to application-level publish/subscribe clients as publication statistics gathering and publisher migration all happen behind the scenes; neither require the application’s involvement nor introduce any message loss.

However, POP and GRAPE are different from each other due to design decisions that trade off simplicity for flexibility. (1) POP uses one optimization metric, the average number of publication deliveries downstream, whereas GRAPE uses two optimization metrics, the end-to-end delivery delay and total broker message rate, to compute the relocation target. (2) GRAPE allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously whereas POP does not give the user this flexibility. (3) In Phase 1, POP retrieves optimization metrics once per each traced publication message whereas GRAPE retrieves optimization metrics once per broker selection cycle, thus giving POP and GRAPE different tradeoffs between algorithm response time and message overhead. (4) In Phase 2, POP performs its broker selection in a distributed manner hop-by-hop towards the target broker, whereas GRAPE performs its broker selection in a centralized manner all locally at the publisher’s first broker. (5) GRAPE is easier to debug and test relative to POP because its core computations are centralized.

To show the effectiveness of POP and GRAPE, we derive a real business scenario from [36] that focuses on the dynamic pricing of soft drinks where retailers subscribe to product information(updates published by suppliers on GDSN. Retailers on GDSN such as Walmart, Target, SUPERVALU, Metro, Associated Grocers, and many others are likely subscribed to events published by suppliers that report moderate to large changes in the cost of soft drinks currently sold (due to peak season, supplier competition, currency fluctuations, etc.) so that shelve prices can be updated as soon as possible to maximize profit and minimize loss. However, retailers are not interested in minute price changes because price adjustment on the cent-level for all on-the-shelf products is not economically feasible. Suppliers of soft drink products on GDSN, such as Coca-Cola Enterprises and PepsiCo, are likely subscribing to all price updates
published by themselves for record keeping and by their competitor for close monitoring. If the GDSN is equipped with GRAPE set to minimize solely on average delivery delay, GRAPE will reconnect the suppliers’ publishing agents close to the hundreds of retailers’ subscriber agents that sink a subset of the publishers’ events. The result is quicker price update deliveries to the subscriber agents for more timely price adjustments. On the other hand, with GRAPE set to minimize solely on system load, GRAPE will reconnect the publishing agents close to their own and the direct competitor’s subscriber agent that sink all of the publishers’ events. The result is brokers become less loaded, which adds stability and further capacity to the GDSN backbone. If the GDSN is instead equipped with POP, the outcome can be anywhere between the two extremes of GRAPE: minimizing price update transmission delays and/or chances of system down-times, both of which are cost saving measures critical to businesses. These illustrated behaviors of GRAPE and POP are in fact what we observed in our experiments.

2.2.3 Problem of Minimal Resource Allocation

Publish/subscribe is commonly used in enterprises as a messaging substrate for event dissemination. At the same time, 97% of these enterprises are actively engaged in green computing practices to (1) reduce IT maintenance costs, (2) reduce the carbon footprint, and (3) promote an environmentally responsible brand image of the company [32]. Seeing how publish/subscribe is so intricately tied to enterprises that also have strong green IT initiatives, we present resource allocation algorithms that allocate as few brokers as possible for any given workload, while maximizing the resource utilization of allocated brokers.

In order to minimize the number of brokers, we need to minimize the amount of messages forwarded and processed in the system. Reducing the broker count also reduces the network size, which in turn improves publication hop count. To satisfy all these requirements, we develop a 3-phase scheme to reconfigure the publish/subscribe system. In Phase 1, we gather performance and workload information from the network using bit vectors. In Phase 2, we allocate the subscriptions to brokers using the information gathered from Phase 1. In Phase 3, we recursively construct the broker overlay with the subscriptions already allocated. Finally, we strategically place the publishers onto the newly built broker overlay with a publisher relocation
algorithm called GRAPE [25]. The entire process shows that we are altering three variables to meet our optimization criteria, which is proven to be an NP-complete problem [82]. In our evaluation, we compare our algorithms against three related and two baseline approaches which are representative of typical publish/subscribe deployments where the measure of a "good" topology is not easily quantifiable [7, 67, 14, 9].

To the best of our knowledge, we are among the first with Tajuddin [82] to minimize the number of brokers in content-based publish/subscribe systems. The approach and evaluation taken by Tajuddin, however, are fundamentally different from this work which we will describe in greater detail in Chapter 3. There has been prior work that focused on improving system performance (i.e., reducing the in-network processing and/or hop count) by relocating subscriptions [22], publication sources [25], and brokers [7, 46, 57]. However, these prior approaches only manipulated one variable, either brokers, publishers, or subscribers to achieve their optimization criteria. In contrast, our work manipulates all three variables to achieve the same objective while minimizing the total number of allocated brokers in the system.

2.3 Contributions

This thesis has three key contributions which are inline with the three research problem that this thesis focuses on. One is the dynamic load balancing algorithm that increases availability and scalability of the system [24]. Second are the publisher placement algorithms that increase scalability by minimizing system load while minimizing message hop count and delivery delay [25]. Third are the resource allocation algorithms that further improve on [25] while allocating as few resources as possible [27].

Our contributions in the load balancing algorithm include (1) a framework that facilitates load balancing by isolating subscribers to the edge brokers in the network and a set of protocols for triggering and mediating load balancing sessions; (2) a novel load estimation algorithm that profiles subscription load using bit vectors; (3) offload algorithms that load balance on multiple performance metric of a broker by selecting the appropriate subscriptions to offload based on their profiled load characteristics; and (4) experimental results that demonstrate the
Chapter 2. Introduction

performance and effectiveness of our load balancing solution. Additionally, we have significantly extended our previous work from the Master thesis [21, 22] with new algorithms that improve performance and further evaluations on real testbeds. Specifically, the new extensions include:

1. A naive offload algorithm to serve as a comparison against a generic load balancing scheme and to highlight the need and strength of our solution.

2. A more efficient load exchange protocol called Adaptive PIE that leverages the infrastructure’s publish/subscribe operations to further reduce message overhead.

3. Techniques to accurately model multiple performance metrics on real deployments, namely the input utilization ratio, matching delay, and output utilization ratio.


5. Results from live experimentation on PlanetLab, which puts our ideas under harsh real-world Internet-scale conditions, and a cluster testbed that mimics running our approach under local area network and data-center conditions.

6. Comparison and discussion of previous simulation results from [22] with real-world results on heterogeneous and homogeneous environments using the proposed and naive offload algorithms.

The contributions in our publisher placement algorithms include: (1) POP’s Phase 1 algorithm which probabilistically traces publication messages and retrieves trace information through replies with data aggregation, (2) POP’s Phase 2 algorithm which selects the target broker in a fully decentralized manner using only partial trace data, (3) GRAPE’s Phase 1 algorithm which traces publication messages and stores trace results into space-efficient bit vectors for later retrieval through replies with data aggregation, (4) GRAPE’s Phase 2 algorithm which selects the target broker in a centralized manner based on the specified prioritization metric and weight, (5) POP and GRAPE’s Phase 3 algorithm which transparently migrates the publisher from the original to the target broker while introducing minimal message overhead, and (6)
extensive experiments using real-world data on PlanetLab and a cluster testbed that quantitatively validate and compare our two approaches. Our results show that POP and GRAPE are able to reduce the average broker input utilization by up to 64% and 68%, average broker message rate by up to 85% and 85%, and average delivery delay by up to 63% and 68% on PlanetLab, respectively.

Our contributions of the resource allocation algorithms include: (1) a bit vector supported resource allocation framework, (2) three subscription allocation algorithms that account for broker capacities with one of them capable of clustering subscriptions of similar interests, (3) a recursive broker overlay construction algorithm, and (4) real world evaluation and comparison of the baseline, related, and proposed approaches.

2.4 Organization

The rest of this thesis is organized as follows. Chapter 3 presents background and related work on content-based publish/subscribe and in relation to our three research problems. Here, the fundamental aspects of content-based publish/subscribe systems are introduced, including the language and routing protocols. Different types of publish/subscribe systems are also covered. Related approaches in the area of load balancing, publisher and subscriber relocations, and broker overlay reconfiguration are presented and compared to our approaches. Chapter 4 presents the load balancing algorithm, which primarily consists of the framework, load estimation algorithm, and offload algorithms. Chapter 5 presents the publisher relocation algorithms, POP and GRAPE, and describes their three phases of operation in detail. Chapter 6 presents the resource allocation algorithms, which consists of the bit vector supported resource allocation framework, subscription allocation algorithms, and the broker overlay construction algorithm. Chapter 7 shows the macro and micro experimental results of the our load balancing algorithm, publisher placement algorithms, and resource allocation algorithms. Results from macro experiments show the performance of our approaches relative to naive and related approaches from a general point of view. Results from micro experiments show changes in behavior of our approaches by tuning each individual parameter. Chapter 8 contains the conclusion that
summarizes the major ideas and results presented in this thesis.
Chapter 3

Background and Related Work

In this chapter, we familiarize the reader on content-based publish/subscribe systems by explaining the publish/subscribe routing, language, different publish/subscribe architectures, background on load balancing in general and in relation to publish/subscribe, publisher migration, broker overlay and publisher reconfigurations, and subscription clustering.

3.1 Content-based Publish/Subscribe Routing

Figure 3.1 demonstrates an example of how advertisements, subscriptions, and publications are routed. In the first step, the publisher (represented by a triangle) advertises $a > 1$ to broker $B1$. As shown by the arrows labeled with 1, the advertisement is flooded to all brokers in the network. Subscribers never receive advertisements because advertisements are control messages and subscribers only receive publication messages that match their subscription filter. In Step 2, subscriber $sub1$ issues its subscription $a > 5$ to broker $B4$. Since $sub1$’s subscription’s space is common with the publisher’s advertisement space, $sub1$’s subscription is forwarded along the reverse path of the advertisement, as shown by the arrows labeled with 2. Notice that the publisher does not receive the subscription because subscriptions are control traffic within the publish/subscribe system and publishers never receive messages. In Step 3, subscriber $sub2$ issues its subscription $a > 9$ to broker $B3$. Similar to $sub1$’s subscription, $sub2$’s subscription travels in the reverse path of the publisher’s advertisement since it matches with a subset of
the advertisement space. However, once the subscription reaches broker \( B2 \), the subscription is purposely not forwarded to broker \( B1 \) because \( sub2 \)'s subscription is covered by \( sub1 \)'s. In other words, since \( B2 \) is already receiving publication messages that matches \( sub2 \)'s subscription space due to \( sub1 \)'s subscription, it is not necessary to forward \( sub2 \)'s subscription to \( B1 \). As illustrated in the figure, publications sent by the publisher in Step 4 will get delivered to both subscribers according to the brokers’ publication routing tables shown beside each broker (with the left and right columns representing the next-hop and subscription filter, respectively). When subscriber \( sub1 \) unsubscribes from broker \( B4 \) in the future, the unsubscription will traverse along the same path as \( sub1 \)'s original subscription up to broker \( B1 \). When the unsubscription reaches broker \( B2 \), \( B2 \) forwards \( sub2 \)'s subscription to \( B1 \) before \( sub1 \)'s unsubscription to avoid interrupting \( sub2 \)'s event delivery.

### 3.2 Content-based Publish/Subscribe Language

In publish/subscribe, clients that send publication messages into the system are referred to as *publishers*, while those that only receive messages are called *subscribers*. Publishers issue publications/events\(^1\) with information arranged in attribute key-value pairs. An example stock publication with eight key-value pairs is represented as:

\(^1\)In the publish/subscribe literature, the terms publications and events are often used synonymously.
Chapter 3. Background and Related Work

The key `class` denotes the topic of the publication; `symbol` denotes the symbol of the stock and carries a string value of `YHOO`. The stock's `open`, `high`, `low`, `close`, and `volume` values are numeric and hence unquoted.

Subscribers issue subscriptions to specify the type of publications they want to receive. In topic-based publish/subscribe, subscribers can only specify the topic of interest in their subscriptions. In content-based publish/subscribe, subscribers can specify more complex constraints in addition to just specifying the topic of interest. Depending on the implementation, subscription filters in content-based publish/subscribe can be based on attributes (which this thesis focuses on), on path expressions through the use of XML [5, 69, 40, 50], or on graph expressions [70]. Attribute-based predicates consist of an operator-value pair to specify the filtering conditions on each attribute. Some examples of operators used for string-type attributes include `equal`, `prefix`, `suffix`, and `contains` comparators, denoted as `eq`, `str-prefix`, `str-suffix`, and `str-contains`, respectively. For attributes containing numeric values, one can use the `=`, `>`, `<`, `>=`, and `<=` operators. For example, a subscription for publications regarding the `YHOO` stock whenever its volume is greater than 300,000 is indicated as follows:

```
[class,eq,`STOCK`], [symbol,eq,`YHOO`], [volume,>,300000]
```

The space of interest defined by a subscription's filtering conditions is called `subscription space`. A broker's `covering subscription set` (CSS) refers to the set of most general subscriptions whose subscription space is not covered by any other subscription managed by this broker. For example, a broker with the set of subscriptions shown in Figure 3.2a has a CSS identified by the bolded subscriptions.

For more efficient retrieval of a broker's CSS, the `partially-ordered set` (poset) [14] is used to maintain subscription relationships. The poset is a directed acyclic graph where each unique subscription is represented as a node in the graph as shown in Figure 3.2b. Nodes can have parent and children nodes where parent nodes have a subscription space that is a superset of its children nodes, while subscriptions with intersection or empty relationships will appear as
Chapter 3. Background and Related Work

siblings. As shown, the CSS is readily available as the immediate children nodes under the imaginary \( \text{ROOT} \) node.

Subscribers issue unsubscription messages whenever they disconnect from the system. Unsubscription messages do not contain any predicates, but an identifier that correlates to the subscription to be removed from the system.

In some content-based publish/subscribe systems, such as SIENA [14], REBECA [59], and PADRES [52, 53, 92, 58, 41, 24, 91, 49], advertisements are sent by publishers prior to sending the first publication message. For example, a publisher of the publication previously outlined would have an advertisement such as this:

\[
\text{[class,eq,`STOCK'],[symbol,eq,`YHOO'],[open,isPresent,0]},
\text{[high,isPresent,0]},\text{[low,isPresent,0],[close,isPresent,0]},
\text{[volume,isPresent,0],[date,isPresent,`00-00-00']}
\]

It may not always be possible to know the exact range of an attribute’s value before publishing. Hence, the \text{isPresent} operator is used to denote that the attribute merely exists with a string or numeric type. Advertisements are flooded throughout the network so that subscriptions do not have to be flooded but instead travel along the reverse path of matching advertisements. In publish/subscribe systems without advertisements, subscriptions can be flooded. It is also conceivable that publications are flooded, which under normal operating assumptions is the least desirable approach. For many publish/subscribe application scenarios, it is assumed that advertisements constitute the least number of messages, followed by subscriptions, and finally by publications constituting the largest number of messages. For example, newspaper agencies rarely change, but the number of news readers subscribing and unsubscrib-
ing from a news service is comparatively higher. For scenarios where the above assumption about message type frequencies does not hold, protocols exist that can handle these situations [62]. Unadvertisement messages are used by the publisher whenever it disconnects from the system. Our load balancing solution is applicable regardless of whether advertisements and unadvertisements are used by the publish/subscribe system.

3.3 Publish/Subscribe Architectures

Two main classes of distributed content-based publish/subscribe systems exists today: filter-based and multicast-based. In the filter-based approach, as is the focus of this thesis, subscriptions are propagated into the network to establish paths that guide publications to subscribers. Each publication is matched at every broker along the overlay to get forwarded towards neighbors with matching subscriptions. Consequently, the farther the publication travels, the higher is the delivery delay. The overlay of filter-based approaches may consist of either a set of dedicated brokers [78, 9, 14, 30, 59, 71, 67, 20, 52] or purely clients [83, 85, 87, 38, 18, 95, 8, 89, 3, 96, 16, 68]. Broker-based overlays are suitable to enterprise settings that are stable, require high availability, and offer administrators more control over the network infrastructure, such as what and where messages get routed. Client or peer-to-peer-based overlays are suitable for deployments that involve high churn and require high scalability with minimal investment cost on resources at the expense of losing network administrative control.

In the multicast-based approach [65, 90, 2, 73, 74, 15], subscribers with similar interests are clustered into the same multicast group. Each publication is matched once to determine the matching multicast group(s) to which the message should be multicasted, broadcasted, or unicast. As a result, matching and transmission of a publication message happens at most once, thus incurring minimal delivery delay. However, compared to the filter-based approach, subscribers in a multicast group may receive unwanted publications because subscribers with even slightly different interests may still be assigned to the same group. One possible solution to this problem is the introduction of filter-based functionality within each multicast group [13].

Even though our work is targeted at filter-based publish/subscribe systems consisting of an
overlay of dedicated brokers, it is also applicable to topic-based publish/subscribe systems\textsuperscript{2}. This is because brokers in both systems overload due to the same patterns, and the relationship among any two subscriptions in a topic-based system is a subset of those in a content-based system. (Topic-based can only have equal and empty relationships, whereas content-based can have equal, empty, subset, superset, and intersect relationships.) Additionally, parts of our load balancing framework can be leveraged to benefit filter-based approaches consisting of only clients as well as multicast-based approaches. For example, in peer-to-peer approaches such as [6], the idea of subscribing to load information from a partial set of nodes as used in our load exchange protocol (later referred to as PIE) can be leveraged to allow nodes to exchange load information while remaining loosely coupled with each other. In multicast-based approaches, (1) PIE and Adaptive PIE can be leveraged to give clustering algorithms awareness of server loads. (2) PRESS can also be used to accurately estimating the load of a set of subscriptions and avoid clustering too many subscriptions into a single group/server. (3) Our offload algorithms can be utilized to temporarily stop overloads before re-clustering takes place.

\section*{3.4 Load Balancing}

Load balancing has been a widely explored research topic for the past two decades since the introduction of parallel and distributed computing. The goal of all load balancing solutions is to evenly distribute workload to all available resources. Load balancing solutions can be found in four software layers: network [1, 45], operating system [55, 47, 80, 94], middleware [10, 39, 54, 4, 84], and application [11, 66]. The layer in which a load balancing strategy is implemented depends on which layer contains the logic to make effective load balancing decisions. For example, it would be ineffective to use a random DNS redirection strategy in the application layer or process migration in the OS layer to load balance a content-based publish/subscribe system that resides at the middleware layer. This is because these approaches cannot identify the relationship (whether it be intersecting, superset, subset, equal, or empty) between subscriptions nor estimate the load of a subscription imposed onto a broker already

\textsuperscript{2}Except the Adaptive PIE extension in Section 4.1.3
servicing an arbitrary set of subscriptions. We demonstrate this claim through a naive offload algorithm that randomly picks subscriptions to offload. Experiments show that the naive algorithm does lead to an unstable system on PlanetLab and the cluster testbed.

3.5 Load Balancing in Content-based Publish/Subscribe

To the best of our knowledge, [22] was among the first to address load balancing in content-based publish/subscribe systems although distributed content-based publish/subscribe systems have been widely studied. The following are various related approaches that propose load balancing techniques in other publish/subscribe approaches.

Meghdoot [15] is a peer-to-peer distributed content-based publish/subscribe system based on DHT for assignment of subscriptions onto peers and routing of events to matching subscriptions. Clients in Meghdoot form the broker overlay network, and they can be a publisher, subscriber, or both. Its load distribution algorithm relies on two load indices to make load sharing decisions: subscription load, which is based on the number of events delivered by the node to its managed subscriptions; and event load, which is the load from propagating events from an upstream neighbor node to a downstream neighbor node. Based on these load indices, a new joining peer either splits the peer’s zone in half so that each peer ends up with half of the number of subscriptions in the original zone, or replicates the highest loaded peer’s zone to evenly divide the event load in half. Such partitioning and replication schemes are common load balancing techniques used in structured peer-to-peer publish/subscribe systems [3, 96]. These schemes also assume that all peers are homogeneous, meaning they have equal resources capacities. In addition, Meghdoot’s load distribution methodology is static, which means that it cannot adapt to dynamic workloads if no new peer joins the system.

Chen et al. [20] proposed a dynamic overlay reconstruction algorithm called Opportunistic Overlay that reduces end-to-end delivery delay and also performs load distribution on the CPU utilization as a secondary requirement. Load balancing is triggered only when a client finds another broker that is closer than its home broker. It is possible that subscriber migrations may overload a broker if the load requirements of the migrated subscription exceed the load-
accepting broker’s processing capacity.

Subscription clustering is another technique to achieve load balancing in content-based publish/subscribe systems [90, 73, 74, 15]. Load balancing is done by partitioning the set of subscriptions into a predefined number of servers or groups. The partitioning clusters subscriptions of highest common interest together to minimize the total amount of network traffic. However, architecturally, this technique is not applicable to filter-based but only to multicast-based publish/subscribe systems.

Our work differs from the prior three solutions by proposing a distributed load balancing algorithm for content-based publish/subscribe systems that accounts for heterogeneous brokers and subscribers, and distributes load evenly onto all resources in the system even without new clients joining. We also present a detailed subscriber migration protocol that enforces end-user transparency and best-effort delivery to minimize message loss.

3.6 Publisher Migration Protocols

Muthusamy et al. [42, 63] proposed several publisher migration protocols with different optimization techniques to study the effects of publisher and subscriber migration on system performance. The publisher migration protocols that we designed for POP and GRAPE are different in three ways. First, instead of rebuilding the advertisement tree rooted at the new broker, we simply revise the last hop of the existing advertisement only on brokers along the migration path as in [42]. In terms of overhead message count, our approach generates $O(\log N)$ messages, whereas the approach in [63] generates $O(N)$ messages, where $N$ is the total number of brokers in a tree-overlay network with typical fan-out greater than one. Second, the advertisement/subscription tree rebuilding period is known in our approach. This allows our publishers to know precisely the earliest time to resume publishing at the new broker with assurance that those messages will be delivered to all matching subscribers in the network. Third, the objective in [63] is to analyze the impact of supporting mobile publishers on system performance, whereas here, our objective is to minimize average end-to-end delivery delay and system load by relocating publishers.
3.7 Broker Overlay and Publisher Placement Reconfigurations

A number of approaches are found in the literature that also try to reduce the overlay distance between publishers and subscribers. Baldoni et al. [7], Jaeger et al. [46], and Migliavacca et al. [57] dynamically reconfigure inter-broker overlay links to allow publications to skip over brokers with no matching subscribers, called pure forwarders, and shorten the publication delivery path. On the other hand, POP and GRAPE relocate publishers to the location of highest-rated or populated matching subscribers while preserving the broker overlay. Hence, POP and GRAPE are suited to policy-driven broker networks where inter-broker links are statically and tightly controlled by administrators. However, these approaches cannot reduce the overall system message rate if at least one subscriber subscribes to the same subscription at every broker. Our solution addresses this limitation by not only reconfiguring the broker overlay, but also relocating both publishers and subscribers. Our experiment results indeed show that under such scenario, relocating only publishers have no impact on the broker system message rate, while our approach achieves reductions of up to 92%.

Working in parallel but independently, Tajuddin also studied the problem of how to minimize the number of brokers in a publish/subscribe system [82]. Both Tajuddin and our approaches share a number of commonalities. One, both approaches reconfigure the broker overlay, publisher placement, and subscriber placement to achieve the goal of minimizing the number of brokers in the system. Two, subscription clustering is utilized in both approaches to place subscriptions of similar interests as close together as possible. Three, brokers with highest capacity are allocated first to minimize the number of allocated brokers. However, there are many differences among our approaches as well. One, our secondary goal in minimizing the number of brokers is the average broker message rate (forwarding traffic), whereas Tajuddin’s secondary goal is in minimizing the message latency. Two, our approach estimates the load of subscriptions based on past matching publications, which makes our approach adaptable to any workload distribution. Tajuddin, on the other hand, estimates the load of subscriptions based on the subscription language and assumes a uniformly random distributed publication space. Third, our load estimation scheme is independent of the publish/subscribe language, making
it applicable to any publish/subscribe system. Tajuddin’s approach, however, strictly assumes subscriptions with range-value queries. Fourth, our approach first allocates subscriptions to brokers, reuses the subscription allocation algorithm to recursively build the broker overlay, and finally relocate the publisher clients from the root of the tree. Tajuddin’s approach first assigns publishers and subscribers to brokers, then builds the broker overlay using a separate overlay construction algorithm. Fifth, the metrics proposed to evaluate the similarity between subscriptions/brokers are different among the two approaches. Sixth, our approach models the capacity of brokers more realistically by capturing the brokers’ input message processing speed (which depends on the number of subscriptions in the broker) and output network I/O bandwidth. Tajuddin’s approach models the capacity of the brokers with simply a constant message rate. Seventh, we evaluated our approach on a cluster testbed and SciNet using real stockquotes workload with a real implementation on the open source content-based publish/subscribe system called PADRES [58]. Tajuddin, on the other hand, evaluated his approach with a simulator that does not accurately model delays present in real world evaluations such as network latencies, processing delays, queueing delays, transmission delays limited by network bandwidth, and delays due to overload and congestion.

### 3.8 Subscription Clustering

One of our resource allocation approaches clusters subscriptions of similar interests to reduce the number of messages forwarded in the system. In this section, we show how our resource allocation algorithm compares with related approaches in literature.

Through epidemic-based clustering, SUB-2-SUB [89] clusters subscribers of similar interests and propagates publications only among interested peers. Similar to our work, SUB-2-SUB supports a content-based language. However, SUB-2-SUB’s peer-to-peer architecture is fundamentally different from the broker-based architecture that we focus on: each peer in SUB-2-SUB is associated with exactly one subscription, while each broker in our system is associated with any number of subscriptions, depending on the number of subscriber clients attached to the broker. This distinction allows SUB-2-SUB to route multiple events from the same publisher.
only to interested peers even if the events are delivered to different sets of peers.

Riabov et al. [73] utilize the concept of clustering to group similar subscriptions together into multicast groups to eliminate pure forwarders. SUB-2-SUB [89] and Rappel [68] cluster subscribers of similar interests and propagate publications only among interested peers. However, Riabov et al., SUB-2-SUB, Rappel, and Gryphon’s [9] cluster algorithms differ from our work in three significant ways. First, their clustering algorithms do not take resource constraints into consideration. For instance, in SUB-2-SUB and Rappel, there is no hard bound on the number of clients in a peer-to-peer group. In [73], there is no restriction on the number of clients assigned to a multicast IP address. In Gryphon, all subscriptions are stored in main memory anyway. Second, their approaches assume range or point subscription queries, whereas our approach is completely language independent. This allows our solution to be readily applicable to not only range query subscriptions, but also to queries with negation, string operators, XPath expressions [5, 69, 40, 50], or graph expressions [70]. Third, the peer-to-peer architectures of SUB-2-SUB and Rappel are fundamentally different from the dedicated broker architecture of our work. Publish/subscribe clients in SUB-2-SUB and Rappel are capable routing of messages because they assume the role of brokers as well. In our work, clients do not assume the role of brokers; therefore, only brokers are capable of routing messages. Fourth, the cluster algorithms employed in [73], such as pairwise, requires one to specify the number of clusters a priori. Our approach, on the other hand, computes the number of clusters at runtime based on the subscriptions’ interests and resource constraints of brokers. In our evaluation, we compare our approach with two extended versions of the pairwise algorithm from [73] and the clustering measurement metric from [9].
Chapter 4

Dynamic Load Balancing Algorithm

This chapter on load balancing is organized as follows. Section 4.1 presents the load balancing framework. Section 4.2 describes the load estimation algorithm. Finally, Section 4.3 describes the three offload algorithms as well as the naive offload algorithm.

4.1 Load Balancing Framework

Before we describe each component of our load balancing approach in great detail, we first give a high-level overview to show how components interact with each other. Each broker in the system runs an instance of the load balancer whose components are shown in Figure 4.1. In general, the load balancer detects overload and load imbalance, upon which specific subscriptions are

Figure 4.1: Components of the load balancer
offloaded to a broker with more available resources. Specifically, the load balancer consists of the Load Detector and Mediator as described in this section, PRESS Subscription Load Estimator as described in Section 4.2, and Offload Algorithms as described in Section 4.3. In the following, we will use the numbering in this figure as a guide to explain the control flow of a typical load balancing session. (1) The Load Detector detects if its own broker is overloaded or the load of its own broker exceeds another broker by monitoring a number of performance metrics. (2) If overload or load imbalance is detected, then the detector tells the Mediator (3) to establish a load balancing session between its own broker, namely offloading broker (broker with the higher load doing the offloading) and a load-accepting broker (broker accepting load from the offloading broker). (4) Once the load balancing session is established, (5) the offloading broker collects publication delivery statistics for each of its subscription through PRESS and passes this information to one of the three offload algorithms. (6) Each offload algorithm is specifically designed to reduce the load on one of the performance metrics, which means the performance metric that is overloaded or needs balancing determines which offload algorithm to invoke. (6) The offload algorithm strategically selects the set of subscriptions to offload and passes this set to (7) the Mediator to coordinate the subscriber migration process. The load balancing session is over once the subscriber migration is complete. The following sections will describe the load balancing framework and the operations of each component in much greater detail.

4.1.1 Structuring the Overlay into Clusters

The Publish/subscribe Efficient Event Routing (PEER) framework organizes brokers into a hierarchical structure as shown in Figure 4.2. PEER is motivated by the architecture adopted by Google’s distributed publish/subscribe system, GooPS [72], where each data center consists of a cluster of publish/subscribe brokers, and data centers in different geographical areas are connected by dedicated network links. Brokers with only one neighbor are referred to as edge brokers, while brokers with more than one neighboring broker are referred to as cluster-head brokers. A cluster-head broker with its connected set of edge brokers, if any, forms a cluster. Brokers within a cluster are assumed to be closer to each other in network proximity than brokers in other clusters. Publishers are serviced by cluster-head brokers, while subscribers are
serviced by edge brokers. **PEER** supports the notion of a multi-level hierarchical network with clusters of smaller clusters, but in this thesis we will limit our scope to one level for the sake of simplicity.

**PEER** is designed with five goals in mind. First, **PEER** allows the load balancing scheme to control the load of edge brokers simply by moving subscriptions because edge brokers have no publishers and no broker-to-broker through-traffic to route. Second, higher dissemination efficiency is achieved by having cluster-heads forward publication messages to all matching clusters almost simultaneously because cluster-heads have negligible processing delays since they do not service any subscribers. Third, cluster-head brokers may be load balanced by moving publishers and inter-broker subscriptions. Fourth, **PEER**'s organization of brokers into clusters allows for two levels of load balancing: *local-level* (referred to as *local load balancing*) where edge brokers within the same cluster load balance with each other; and *global-level* (referred to as *global load balancing*) where edge brokers from two different clusters load balance with each other. Edge brokers only need to exchange load information with edge brokers in the same cluster, and neighboring clusters can exchange aggregated load information about their own edge brokers. Fifth, local load balancing preserves subscriber locality by keeping subscribers within their original cluster, assuming that subscribers connect to the closest broker in the first place. On the other hand, global load balancing trades off locality for a better balanced system by migrating subscribers between edge brokers in neighboring clusters.
4.1.2 Messaging Framework

All coordination messaging of the load balancing framework is done via publish/subscribe. Such messages are referred to as control messages or control publications as they are not observable to the end user unless explicitly subscribed upon. There are two primary reasons why publish/subscribe is used for control messaging. One, this approach unifies the messaging protocols of all components within the publish/subscribe system, thereby assuring ease of system maintainability. Second, the publish/subscribe paradigm directly meets the needs of the load balancing algorithm. For example, as part of the detection algorithm described in the next section, an edge broker can multicast a message to all other brokers within the cluster by a simple publish operation without having to worry about new or removed edge brokers in the cluster and manage connectivity with them.

In this work, we put three unique message attributes into use. First, control messages have higher priority than normal publication traffic so that the response time of the load balancer is not affected by the load of the broker. Second, the class of control messages demonstrated in Section 4.1.3 have a Time-To-Live (TTL) field that limits the hop count of the message to neighboring clusters. Third, the payload of a message is used to carry data that does not affect the routing of the message. This provides the benefit of reduced matching time at each broker.

4.1.3 Load Detection Framework

In order for brokers to know when and which brokers are available for load balancing, they have to exchange load information with each other. That way, a detection algorithm can trigger load balancing whenever it detects an overload or a wide load difference with another broker.

Broker Performance Metrics

Our performance metrics assume a general broker architecture consisting of an input queue to buffer incoming messages to the broker’s matching engine; a matching engine which takes a message from the input queue, performs the matching (i.e., publications against subscriptions), generates and puts zero or more messages to route into the output queue; and at least one output queue to buffer messages from the matching engine to get transmitted to the next hop.
Figure 7.1 shows an example of such a broker architecture.

The load of a broker is captured by three performance metrics: input utilization ratio, average matching delay per message, and output utilization ratio. Input utilization ratio ($I_r$) captures the broker’s processing or CPU utilization as well as the amount of time that incoming messages wait at the broker’s input queue. $I_r$ is defined by the following formula, with $i_r$ representing the incoming publication rate in msg/s, and $m_r$ representing the maximum matching rate also in msg/s:

$$I_r = \frac{i_r}{m_r} \quad (4.1)$$

Maximum match rate, $m_r$, is calculated by taking the inverse of the average matching delay per message. $I_r$ can have any value greater than or equal to 0. A value of 1.0 or greater for $I_r$ signifies that the broker is input overloaded.

Matching delay is defined as the time spent by the matching engine between taking a message as input and producing zero or more messages as output. The average matching delay metric is important because it captures the average amount of processing time each message undergoes when processed by a broker.

The output utilization ratio ($O_r$) captures the broker’s output bandwidth utilization and the amount of time messages spend waiting in the output queue before being sent off. $O_r$ is defined by the following formula with $o_u$ representing the output bandwidth usage in bytes per second, and $o_t$ representing the total amount of output bandwidth with the same units:

$$O_r = \frac{o_u}{o_t} \quad (4.2)$$

Expanding on $o_u$ and $o_t$ will yield the following equation for $O_r$:

$$O_r = \left( \frac{t_{busy}}{t_{window}} \right) \left( \frac{b_{rx}}{b_{tx}} \right) \quad (4.3)$$

In Equation 4.3, $t_{window}$ is the monitoring time window, and $t_{busy}$ is the amount of time spent sending messages within $t_{window}$. The fraction of these two variables yields a value range between 0 and 1, where a value of 0 signifies that the output bandwidth is not used and 1
signifies that the resource is fully consumed. As for the last two variables, \( b_{rx} \) represents the messages (in bytes) put into the output queue in time window \( t_{window} \), and \( b_{tx} \) represents the messages (in bytes) removed from the output queue and sent successfully in time window \( t_{window} \). The fraction of these two variables yields a value range greater than or equal to 1, where a value greater than 1 signifies overload. If no messages are transmitted in the time window, then this fraction defaults to a value of 0. At times when there is no overload, the left fraction indicates the bandwidth utilization while the right fraction yields a neutral value of 1. At times of overload, the left fraction maximizes to a neutral value of 1 while the right fraction indicates the magnitude of overload by yielding a value greater than 1.

Protocol for Exchanging Load Information

Publish/subscribe Information Exchange (PIE) is a distributed hierarchical protocol for exchanging load information between brokers using publish/subscribe primitives. Brokers publish PIE messages intermittently to let other brokers in the federation know of their existence, availability for load balancing, and load levels of each performance metric. As the next section describes, the detection algorithm uses information gathered from these PIE messages to determine whether and with which broker to engage in load balancing. PIE, as well as other load balancing control messages described in later sections, has a higher routing priority than normal publish/subscribe traffic so that their delivery is not affected by the broker’s load. A PIE message contains five attributes:

1. The broker’s three performance metrics (as outlined in the previous section).
2. Load balancing states, which can be one of \( \text{OK}, \text{BUSY}, \text{N/A}, \text{or STABILIZING} \).
3. The set of edge brokers that this broker is currently balanced with (more on this in Section 4.1.4).
4. The identifier of the cluster to which the broker belongs.
5. The broker’s unique identifier.

\(^1\)With respect to the implementation, the byte count is taken by serializing the Java message object.
PEER’s hierarchical structuring naturally allows for local and global PIE messages. Local PIE messages are published and subscribed by edge brokers within the same cluster to enable local load balancing. An example of a broker subscribing to local PIE messages within cluster ID "C01" is:

```
[class, =, ‘LOCAL_PIE’], [cluster, =, ‘C01’]
```

Global PIE messages are published and subscribed by cluster-head brokers to enable global load balancing. They only propagate one cluster-hop away as enforced by a TTL of 1 and contain averaged load information from their cluster’s local PIE messages. Cluster-head brokers without any edge brokers (such as a cluster-head of many cluster-heads in a multi-level hierarchical arrangement) simply forward global PIE messages one extra hop to all of their neighbors.

PIE messages are generated at a frequency defined by a parameter called local PIE publishing period (default is 30 s) for the local-level and global PIE publishing period (default is 60 s) for the global-level. According to our micro experiments, a low value will improve data liveliness, which in turn improves load balancing response time. However, frequent publishing of PIE messages increases overhead. Moreover, a low value may yield fluctuating load values due to temporary load spikes, which triggers unnecessary load balancing. Therefore, the publishing period should be set long enough to better capture the average load of a broker in the presence of load spikes. The global PIE publishing period should be set to a value that is equal to or higher than the local PIE publishing period because the data published in global PIE messages is aggregated from the local PIE messages received by the cluster-head brokers. If the global PIE publishing period is set lower than the local PIE publishing period, then global PIE messages published in between local PIE publishing period will report the same data, which leads to unnecessary overhead. For best load balancing response time, the global PIE publishing period should be set equal to or just above the local PIE publishing period. In summary, both parameters control the tradeoff between load balancing response time and overhead.

An additional constraint is that PIE messages are only published when one of the utilization ratios or the matching delay differs by a threshold from the previously published corresponding value. The purpose of this is to avoid publishing redundant information which wastes system
resources. The thresholds are defined by the parameters *PIE ratio threshold* which is a magnitude value and *PIE delay threshold* which is a percentage value, respectively. Both parameters have a default value of 0.025. Ideally, both thresholds should be set high enough to be effective in filtering out *PIE* messages that report similar data. However, too high of a threshold can increase the staleness of *PIE* information, which can hurt the load balancing response time. Given that *PIE* messages introduce very low overhead according to our experiments, both of these threshold values can be set conservatively. In short, this parameter controls the tradeoff between overhead and load balancing response time.

**Local Detection Algorithm**

Detection allows a broker/cluster to monitor its current resource usage and also compare it with other edge brokers/clusters so that a broker/cluster can invoke load balancing if necessary. Detection runs periodically at a broker/cluster only if it has a status of *OK*, *N/A*, and *STABILIZING*. An *OK* status means that the broker is available for load balancing, *N/A* means
that it is overloaded, \textit{STABILIZING} means that it is waiting for load to stabilize after load is exchanged, and \textit{BUSY} means that it is currently in a load balancing session with another broker/cluster. A diagram showing the transition conditions between the local and global load balancing states is shown in Figures 4.3 and 4.4, respectively.

The \textit{local detection algorithm} running on an edge broker is composed of two steps. Step 1 identifies whether the local broker is overloaded, and if so triggers load balancing. Step 2 is only invoked if Step 1 identifies no overload and no other observable edge brokers are overloaded. The latter condition gives overloaded brokers higher priority in establishing load balancing over non-overloaded brokers that simply want to balance load. The purpose of Step 2 is to identify other edge brokers whose load difference with the local broker is greater by a threshold. Figure 4.5 summarizes the two phases of the local detection algorithm.

Specifically, Step 1 examines three utilization ratios, namely input, output, and CPU utilization ratio to determine if the broker itself is overloaded. The parameter \textit{lower overload threshold} is introduced to prevent the broker from accepting further load by updating the broker’s status to \textit{N/A} when one of its utilization ratios exceeds 0.9. This parameter should be adjusted to reflect the maximum load under which the broker can operate reliably. If a utilization ratio exceeds the \textit{higher overload threshold} at 0.95, then the broker is deemed close to overloading and load balancing is invoked immediately to start offloading subscriptions.

Both overload thresholds dictate the safe bounds under which a broker can consume the underlying physical resources. Very high overload threshold values will make more full use of the underlying resources, but the broker may enter the overloaded state before it has time to do offloading, which can have a significant impact on the delivery delay of messages. The magnitude of the difference between the two threshold parameters controls the efficiency of load balancing. The smaller the magnitude is, the more frequent the broker engages in load balancing sessions as a result of the following endless load balancing cycle: accept load from other brokers until reaching the \textit{lower overload threshold}, trigger offloading due to reaching the \textit{higher overload threshold}, accept load from other brokers until reaching the \textit{lower overload threshold} and so on. However, a high magnitude in the difference will cause more load imbalance among the brokers. In summary, the \textit{higher overload threshold} controls the tradeoff between
the utility of the underlying physical resources and broker stability. The difference between the lower overload threshold and higher overload threshold controls the tradeoff between load balancing efficiency and load imbalance.

By the end of this step, a broker-action list of \(<\text{load-accepting broker}, \text{offload action}>\) is generated. The list is sorted in descending order by the difference between the offloading broker’s overloaded metric and load-accepting broker’s corresponding metric. The offload action is “input” to denote triggering the input offload algorithm if the input utilization ratio is overloaded, “output” if the output utilization is overloaded, or “match” if the CPU utilization is overloaded. The reasoning behind the offload actions is that the input offload algorithm targets specifically at reducing the input utilization ratio, the output offload algorithm at reducing the output utilization ratio, and the match offload algorithm at reducing the matching/processing delay.

Step 2 checks to see if any one of the input utilization ratio, output utilization ratio, or matching delay differ from another edge broker by more than a threshold. The threshold for utilization ratios is called the local ratio trigger threshold, and for matching delay the local delay trigger threshold. The local ratio trigger threshold defines the minimum input or output utilization ratio difference before load balancing can be initiated between two brokers. Similarly, the local delay trigger threshold applies to the matching delay. All trigger thresholds have a value greater than 0. A low trigger value will initiate load balancing on even the smallest load difference between two brokers. However, the trigger threshold should not be set lower than the load estimation error as that can lead to endless load balancing sessions trying to correct the imbalance arising from previous sessions. A high trigger threshold will yield more load imbalance among brokers, but since load balancing is rarely taking place, overhead is also lower as well. When the trigger threshold is equal to or higher than the higher overload threshold, load balancing only starts when a broker’s input or output utilization is past the higher overload threshold. In general, the triggering thresholds control the tradeoff between the amount of overhead and the load imbalance tolerated by the algorithm. By default, both thresholds are set to 0.1.

The difference for utilization ratio is just the magnitude of the difference, while for matching
delay, the following formula is used:

\[
d_{\%Diff} = \left| \frac{d_1 - d_2}{N_f} \right|
\]

where \(d_1\) and \(d_2\) are the two delay values used in the comparison. \(N_f\) represents the normalization factor and is set to 0.1 by default so that delay differences much less than 0.1 s do not yield high percentage differences and trigger unwanted load balancing. At the end of this step, a broker-action list of \(<\text{load-accepting broker}, \text{performance metric/offload action}>\) is generated that is sorted in descending order of greatest performance metric difference. The purpose of sorting is to favor load balancing with load-accepting brokers that have the largest load difference. Not only does this potentially reduce the number of load balancing sessions to balance load across edge brokers, but also help to bring an overloaded broker out of its overloaded state in as few load balancing sessions as possible, thereby minimizing response time of the load balancing algorithm. The list is passed to the local mediator which establishes a load balancing session with the top-most available load-accepting broker in the list.

Immediately after a broker finishes a load balancing session, its load information may mislead the broker into making an incorrect load balancing decision. For example, brokers accepting load may not experience an increase in utilization immediately. This may cause the broker to accept more load balancing sessions, which may cause its resource consumption to overshoot. To prevent this from occurring, both the offloading and load-accepting brokers inherit a status of \textit{STABILIZING} for a stabilize duration period before setting their status back to \textit{OK} (see Figure 4.3). When a broker has a \textit{STABILIZING} status, it cannot accept load balancing requests nor invoke load balancing unless the broker is overloaded. If all performance metrics do not fluctuate by more than the stabilize percentage after a stabilize duration, then the broker sets its status back to \textit{OK}.

The broker may wait for multiple stabilize duration periods if the stabilize percentage is not satisfied after each period. If the stabilize duration is set to a low value, then stability evaluation is done more frequently, which may help reduce waiting time and improve response time. However, a longer stabilize duration will capture a more accurate view of the broker’s load, which leads to more reliable stabilization detection. In summary, this parameter controls
the tradeoff between response time and load balancing effectiveness. By default, the stabilize duration is set to 30 s, but it can be set to any other value as long as it allows sufficient time for the broker to sense its final load level.

As for the stabilize percentage, it should be set as low as possible to be certain that the broker’s load has stabilized. However, in real deployments, the load of brokers is never constant even if the set of subscribers do not change and the publication rate of publishers stay constant. Some possible reasons for this include different publication-to-subscription matching patterns, shared server resources, network congestion, etc. Therefore, stabilize percentage should at the same time be set high enough so that noise does not prevent a broker from coming out of the stabilization phase. In summary, the stabilize percentage controls the tradeoff between effectiveness of load balancing and response time. By default, this parameter is set to 0.05 such that it is low enough to realize that load is no longer dropping drastically while ignoring noise.

Alternatively, in place of their utilization ratio counterparts, it is also possible for the load balancer to use input queuing delay and output queuing delay as performance metrics. However, queuing delays cannot show the utilization level of a resource but only indicate that the resource is overloaded when queueing delays rise rapidly. Moreover, queuing delay measurements do not accurately indicate the load of a broker at the instant the metric is measured because it is obtained after the message gets dequeued. Therefore, the measurement is lagging by the delay measured.

Global Detection Algorithm

In the global detection algorithm, a cluster-head uses a subset of the status indicators in local load balancing (see Figure 4.4) to indicate its cluster’s load balancing status. The only difference here is that a cluster is N/A if one or more edge brokers are N/A. This allows the overloaded edge brokers to offload subscribers to brokers within the same cluster first to preserve locality. The global detection algorithm is almost the same as the local detection algorithm, except that the global detector uses different threshold values (namely, global ratio triggering threshold and global delay triggering threshold, both default to 0.15) and works with averaged values of each performance metric calculated from the edge brokers in the local cluster. As a rule of thumb,
Chapter 4. Dynamic Load Balancing Algorithm

Figure 4.5: Flowchart showing the local detection algorithm

Start

Overload? Y

Recommend to set status to N/A

N

Recommend to set status to OK

Status was STABILIZING? Y

Compute broker-action list according to thresholds

N

Stabilized? Y

Recommend to set status to STABILIZING

N

Empty list? N

Pass broker-action list to mediator

N

Empty list? Y

End
Chapter 4. Dynamic Load Balancing Algorithm

Figure 4.6: Flowchart showing the global detection algorithm
the global trigger thresholds should be set higher than the local trigger thresholds to prioritize load balancing among the edge brokers within the same cluster. The motivation behind this is to maintain the clients’ connection within the same cluster because the client is closest to that group of brokers in network vicinity (i.e., same network cluster or corporate network). Figure 4.6 summarizes the global detection algorithm.

Another way of invoking global load balancing is on receiving a number of global load balancing requests from one or more edge broker. An edge broker sends a global load balancing request to its cluster-head broker if it is in the N/A state and cannot find another edge broker to load balancing with because none have a status of OK. The threshold for invoking global load balancing is defined by the parameter *global load balancing request limit*, whose default is set to 3. Setting this parameter to a low number will make the algorithm more responsive and minimize the amount of time an edge broker stays overloaded. However, because global load balancing is more easily triggered with a lower value, client locality is sacrificed. As well, message overhead increases due to increased frequency of global load balancing. In summary, the *global load balancing request limit* controls the tradeoff between responsiveness to overload, client locality, and overhead.

Both the local and global detector components run periodically to decide whether or not to trigger load balancing. Experiments show that the periodic interval can be set very low to make the detector more responsive to overloads and load imbalance. However, since the detector makes decisions on the average load information gathered between each detection run, short detection intervals in the presence of temporary load spikes can lead to unnecessary load balancing, which may prevent convergence of the load balancing algorithm. Therefore, the detection interval should be set long enough so that the detector uses load information that better represents the broker’s average load to make load balancing decisions. Instead of setting a fixed detection interval which may fix the load balancing order of edge brokers, we set the minimum and maximum interval values and let the brokers wait a random value between those bounds before subsequent detection runs. The respective minimum and maximum detection intervals for local load balancing are the *local detection minimum interval* and *local detection maximum interval*. The respective minimum and maximum detection intervals for global load
balancing are the *global detection minimum interval* and *global detection maximum interval*. Generally, the minimum bound should not be set lower than half the maximum bound to ensure fairness among the brokers or clusters. As well, the global-level parameters can be set much higher than the local-level parameters to better preserve client locality and minimize overhead while trading off load imbalance among clusters. In summary, the detection interval parameters control the tradeoff between response time and load balancing effectiveness.

**Adaptive PIE**

From our experiments in running PIE at the default setting, the total message overhead is 0.2\%. However, if the number of edge brokers or the PIE publishing frequency is increased (i.e., to decrease the response time of the load balancing algorithm), then the amount of message overhead will increase, which in turn will decrease the capacity of the system. To reduce the overhead of PIE messages at both local and global levels, we introduce *Adaptive PIE* that is an extension to the original PIE protocol outlined in Section 4.1.3. Adaptive PIE exploits the fine-grain filtering capability of content-based publish/subscribe to have brokers subscribe to only PIE messages that report a lighter load than the broker itself. This is achieved by dynamically adjusting the range of subscribed values for each of the three performance metrics. Instead of sending one subscription for local PIE messages, each broker sends three subscriptions, one for each of input utilization ratio, matching delay, and output utilization ratio. As well, each subscription has its own unique subscribed range. Below is an example of a local PIE subscription of a broker in cluster $C_1$ interested in input utilization ratio less than 0.5:

```
[class,=,‘LOCAL_PIE’],[cluster,=,‘C1’],[input,<,0.5]
```

Equation 4.5 below shows how to compute the value to subscribe for any of the performance metrics.

$$S_{new}^x = L_{current}^x - T_L^x + L_{stddev}^x \quad (0 \leq S_{new}^x \leq 1) \quad (4.5)$$

Ideally, the upper bound value to subscribe on performance metric $x$, $S_{new}^x$, should always be less than or equal to the broker’s current load value for metric $x$ ($L_{current}^x$) minus the
metric’s detection threshold \( T^x_L \) as only those brokers can become potential load balancing candidates. However, since the load of the broker constantly changes over time, \( S^x_{\text{new}} \) will have to be constantly updated through subscribe and unsubscribe operations, resulting in even more overhead than conventional PIE.

In order to avoid high overheads from frequent subscription updates due to broker load changes, we introduce some leeway into \( S^x_{\text{new}} \), called \( L^x_{\text{stddev}} \), which is the standard deviation of the broker’s performance metric \( x \) in the last \( N \) published PIE messages. The use of a standard deviation function offers several benefits. First, if the broker’s load is static, then \( L^x_{\text{stddev}} \) approaches zero and the function \( S^x_{\text{new}} \) is just \( L^x_{\text{current}} - T^x_L \), which filters out unnecessary PIE messages and no subscription updates are required. Second, if a broker’s load intermittently changes, then PIE subscription will only be updated intermittently at worst. Last, if the broker’s load is very erratic, then \( L^x_{\text{stddev}} \) dominates the equation, forcing \( S^x_{\text{new}} \) to have an upper bound value of 1.0 over time, which also means no PIE subscription updates are necessary.

By default, \( N \) has a value of 10. \( N \) should be set high enough to capture past load fluctuation patterns at the broker, which will minimize re-subscribing of PIE messages. However, if the history is too long, the standard deviation may not be representative of the current load patterns, which may make the PIE subscription attract useless PIE messages from other edge brokers. Thus, the history should be set small enough to capture only recent load patterns. A shorter history also has the benefit of lower memory consumption and computational overhead. However, if the history is set too small, periodic load spikes in the past may not be captured by the standard deviation, which leads to unnecessary re-subscribing. To sum up, the history size controls the tradeoff between the number of PIE re-subscribing operations and the number of irrelevant PIE messages received.

A validity check for each performance metric’s subscription is run at every PIE publication interval. If \( S^x_{\text{new}} \) is greater than the current subscribed value \( S^x_{\text{current}} \) by the detection threshold for this metric, then the subscription for this performance metric needs to be updated. On the other hand, if \( S^x_{\text{new}} \) is less than \( S^x_{\text{current}} \), then the subscription for this performance metric is only updated if \( S^x_{\text{current}} \) is greater than at least one of the other edge broker’s corresponding performance metric and \( S^x_{\text{new}} \) is less than that same figure. In other words, do not update the
subscription unless the new subscription can filter out PIE messages from at least one other broker. On bootstrap, the broker uses a value of 0 for $S_{\text{new}}$. Our experiments in Section 7.1.2 show that Adaptive PIE indeed reduces the amount of PIE-related message overhead by up to 65%.

### 4.1.4 Mediation Protocols

All load balancing activities are coordinated by exchanging messages using the underlying publish/subscribe infrastructure for simplicity and efficiency. Specifically, request-reply and one-way protocols are implemented in publish/subscribe to coordinate broker and subscriber activities.

**Mediating Load Balancing Sessions**

A local load balancing session consists of a pair of brokers: an offloading broker and a load-accepting broker. Once a broker is engaged in a local load balancing session, it cannot participate in another local load balancing session until the current one is finished. This is to allow the offloading broker to accurately estimate the load impact of offloading subscription to the load-accepting broker. An alternative design choice may involve one offload broker with multiple load-accepting brokers. However, such a design can introduce deadlocks if two or more offloading brokers attempt to "reserve" the same set of load-accepting brokers. Even with deadlock prevention, offloading brokers may be required to serialize their local load balancing sessions, which slows down the response time of the algorithm. With our pairwise load balancing approach, pairs of brokers can load balance concurrently without deadlocking, making the algorithm efficient and simple, which is inline with the goals that commercial systems strive to achieve.

A local load balancing session is composed as follows. Once the local detection algorithm composes the broker-action list of candidate brokers for load balancing, the local mediator sends a load balancing request sequentially to brokers in the sorted list. When a load-accepting broker gets this request, its local mediator replies back with its current status. If the status is OK, the request is accepted and both brokers update their status to BUSY. In the OK reply, the
load-accepting broker appends its load information in the message so that the requesting broker can use this information for offload algorithms to compute which subscribers are suitable for offload. This load information includes the load-accepting broker’s CSS, input publication rate, matching delay equation, number of subscriptions, and total output bandwidth. For all other states, the load-accepting broker rejects the load balancing request.

A global load balancing session consists of a pair of clusters where edge brokers from each cluster load balance with each other. A pairwise design follows the same efficient and simple design theme as the local load balancing approach. The global mediator running at the cluster-head broker uses the same protocol as the local mediator to set up global load balancing. The difference here is that after a successful handshake, both cluster-heads have to tell all edge brokers in their own cluster to subscribe to the other cluster’s local PIE messages. This allows edge brokers from one cluster to load balance with edge brokers in the other cluster.

Global load balancing ends when all edge brokers are balanced with each other as indicated by the balanced set field in local PIE messages. On terminating a global load balancing session, edge brokers unsubscribe from the local PIE publications of the other cluster. This action has the effect of blocking edge brokers in one cluster from seeing edge brokers in another cluster through PIE messages, which in turn stops local load balancing sessions between two clusters from initiating.

**Mediating Subscriber Migration**

Once the offloading algorithm is done with its computation, it returns back to the mediator a list of subscribers to offload. The mediator has to migrate the indicated subscribers to the new broker in the most efficient and timely manner with minimal delivery loss. First, the mediator sends a control publication message to each subscriber in the offload list telling them to issue their subscription to the new load-accepting broker. Subscribers issue a subscription to the load-accepting broker containing the ID of the load balancing session and the total number of migrating subscribers. These two pieces of information allow the load-accepting broker to know when it has received all migrating subscribers in the current load balancing session. For efficiency and best-effort guarantee of minimal delivery loss, the receiving broker waits for \( N \times \)
migration timeout seconds for all migrating subscribers to connect, where \( N \) is the total number of migrating subscribers, and migration timeout is set to a default value of 5 s. As long as a subscriber connects to the load-accepting broker within the timeout period, then there will be no message loss for that subscriber. However, connecting to the load-accepting beyond the timeout will have no guarantees on message loss. A high timeout value will ensure that subscribers that are slow to connect to the load-accepting broker will not lose any publication messages throughout the migration process. However, a high timeout value will make the load-accepting broker wait unnecessarily long for clients that disconnected from the system or crashed during migration.

While subscribers are connected to the two brokers, they need to detect and drop duplicate publications (by using a short message history) because they are subscribing to the same subscription at two different endpoints. When all subscribers have connected to the load-accepting broker or when the timeout occurs, the receiving broker sends a \texttt{DONE} control publication message back to the offloading broker to terminate the load balancing session. This message ensures that the publication paths for all migrated subscribers have been set up to flow to the load-accepting broker. When the offloading broker receives the \texttt{DONE} message, it tells the migrating subscribers to wait for all the messages currently in the offloading broker’s input queue to be matched and delivered from the output queues before sending an unsubscribe message. This waiting period corresponds to the sum of the offloading broker’s input queuing delay, matching delay, and output queuing delay. Once the migrating subscribers unsubscribe from the offloading broker, the migration process is complete. Note that all control and duplicate messages are handled transparently by a thin software layer on the client side that hides the intricate details of load balancing from the end-user application.

### 4.2 Load Estimation Algorithms

Load estimation is used by the offload algorithms to estimate a subscription’s load contribution in the form of additional input publication rate, matching delay, and output publication rate on the load-accepting broker as well as the load reduced at the offloading broker.
4.2.1 Estimating Load Requirements of Subscriptions

*Publish/subscribe Real-time Event-to-Subscription Spectrum*\(^2\) (PRESS) is a space and time-efficient technique for estimating the bandwidth requirements and common set of publication messages attracted by two or more subscriptions based on current events. It uses bit vectors to record the matching pattern of subscriptions, hence the term *event-to-subscription*. It does not require the publish/subscribe system to use advertisements, nor does it assume that publications are in any sort of distribution. The operation of PRESS is best explained as part of the local load balancing algorithm after the mediation step as described in Section 4.1.4 where two brokers have agreed to load balance with each other.

First, the offloading broker *locally subscribes* to the CSS of the load-accepting broker (as supplied in the *OK* reply message from the replying broker). Locally subscribe means that subscriptions are sent to the matching engine, but never get forwarded to neighboring brokers because their message TTL is set to 0. The purpose of this operation is for the offloading broker to identify which publications it currently sinks are also received by the load-accepting broker. Next, all client subscriptions in the matching engine are allocated a bit vector of length \(N_p\) initialized to 0, where \(N_p\) represents the number of samples. Sampling starts immediately after getting the load-accepting broker’s *OK* reply message and ends after \(N_p\) publications have been received or a timeout \(T\) is met, whichever comes first. According to our micro experiments, a low sample limit will make PRESS more responsive, but the load estimation accuracy will suffer below 50 samples. As well, a high sample limit beyond 100 not only makes PRESS slow, but also hurts estimation accuracy. By default, \(N_p\) is set to 50. \(T\), on the other hand, is used to limit the time spent on profiling load under the condition where the broker’s incoming publication rate is very low. If one favors estimation accuracy without regard for response time, then \(T\) should be set to an extremely high value. \(T\) has a default value of 30 s.

The algorithm starts at the right-most position of the bit vector for all subscriptions. A ‘1’ is set if the subscription matched the incoming publication, ‘0’ otherwise, before moving onto the next bit on the left. During the sampling period, the total incoming publication

\(^2\) *Real-time* refers to sampling using live incoming publications to the broker
Table 4.1: Bit vector example

<table>
<thead>
<tr>
<th>Candidate Subscriptions</th>
<th>Bit Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[class,,=,,'STOCK']</td>
<td>110111</td>
</tr>
<tr>
<td>[class,,=,,'STOCK'],[volume,&gt;,15]</td>
<td>110100</td>
</tr>
<tr>
<td>[class,,=,,'STOCK'],[volume,&gt;,150]</td>
<td>100000</td>
</tr>
<tr>
<td>[class,,=,,'SPORTS']</td>
<td>001000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-Accepting Broker’s CSS</th>
<th>Bit Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[class,,=,,'STOCK'],[volume,&gt;,50]</td>
<td>110000</td>
</tr>
<tr>
<td>[class,,=,,'STOCK'],[volume,&lt;,5]</td>
<td>000001</td>
</tr>
<tr>
<td>[class,,=,,'MOVIES']</td>
<td>000000</td>
</tr>
<tr>
<td>CSS bit vector</td>
<td>110001</td>
</tr>
</tbody>
</table>

Equation 4.6 shows the formula to calculate the publication rate matching a particular subscription represented by \( s_{PR} \), where \( i_r \) represents the total input publication rate of the offloading broker, \( n_{BS} \) represents the number of bits set in the subscription’s bit vector, and \( N \) represents the number of samples taken in PRESS.

\[
s_{PR} = i_r \left( \frac{n_{BS}}{N} \right)
\]  

(4.6)

For example, if the total input publication rate \( i_r \) at the offloading broker is assumed to be 3 msg/s, and if the subscription \([class,\,=,\,'STOCK']\) is assumed to have 5 out of the 6 bits
set, then the publication rate for \([\text{class}=\text{'STOCK'}]\) comes out to 2.5 msg/s. Moreover, the additional incoming publication rate introduced at the load-accepting broker for each candidate subscription can be calculated by using Equation 4.6 with \(n_{BS}\) obtained from the following function:

\[
n_{BS} = |s_{BV} \land \overline{l_{cssBV}}|,
\]

where \(s_{BV}\) is the candidate subscription’s bit vector and \(l_{cssBV}\) is the aggregated load-accepting broker’s CSS bit vector. The intuition behind Equation 4.7 is that we are interested in the set publications that are received by the offloading broker and, at the same time, not received by the load-accepting broker. For example, to calculate the additional incoming publication rate imposed by the subscription \([\text{class}=\text{'STOCK'}]\) on the load-accepting broker, we first compute the value of \(n_{BS}\).

\[
n_{BS} = |110111 \land 110001| = |000110| = 2
\] (4.8)

With \(n_{BS}\) of 2, and reusing 3 msg/s for \(i_r\), the additional incoming publication rate on the load-accepting broker for this subscription is 1 msg/s. In some cases, offloading a subscription may alter the CSS of the load-accepting broker. With \text{PRESS}, it is not necessary to resample all subscriptions again because the aggregated CSS bit vector can be updated by merging it with the offloaded subscription’s bit vector using the OR bit operator. For example, if \([\text{class}=\text{'STOCK'}]\) was chosen for offloading, then the load-accepting broker’s CSS is updated to 110111.

Given \(S\) candidate subscribers, \(C\) subscriptions in the load-accepting broker’s CSS, and \(N\) publications to sample, the memory overhead of \text{PRESS} is \(O((S + C)N)\) bits and the matching overhead is \(O(S + C)\). Under typical deployments where there are 10,000 subscribers with \(N\) set to at most 100, \text{PRESS} only uses 1Mb of memory. Given that the load-accepting broker’s CSS is usually small (it is just one in the case of \([\text{class}=\text{'*'}]\)), an increase in the matching delay is negligible.
4.2.2 Modeling and Estimating Performance Metrics

In this section, we describe an approach to model the matching delay and how to estimate gains or reductions in matching delay, input utilization ratio, and output utilization ratio at the offloading and load-accepting brokers.

Depending on the matching algorithm and workload that is used, the matching delay may grow linearly or exponentially with the number of subscriptions stored in the matching engine. To decouple our load balancing algorithm with the implementation of the matching algorithm, we chose to use a linear equation to model the matching delay, $d$:

$$d = mn + b$$

where $m$ is the slope of the matching delay function, $n$ is the number of subscriptions, and $b$ is the y-intercept. The advantages of using a linear equation is in its simplicity and its ability to accurately capture the slope of any matching delay function given a specific number of subscriptions in the matching engine. The disadvantage is that it is unable to accurately model a non-linear matching delay function over a wide range of $n$. To remedy this shortcoming, multiple linear functions across different ranges of $n$ will be used to model an arbitrary matching delay function. This is done by calculating values for $m$ and $b$ using Equation 4.9 over different number of subscriptions in the past. The number of subscriptions is rounded to the nearest ten to capture delays over wider range in subscription count. In other words, matching delays for subscription counts from 5-14 are assigned to bucket “10”, 15-24 are assigned to bucket “20”, etc. To avoid storing too many readings, the maximum number of buckets is limited to 10, where each bucket stores a running average of the observed matching delay for a specific subscription count. A bucket is only created when the first matching delay value is observed and no buckets exist for that specific subscription count. Since the number of subscriptions tend to increase over time, the bucket with the least subscription count is booted if the total number of buckets exceed 10. A maximum of 2 consecutive delay values that surpass 150% or undercut 50% of the last observed 5 or more values are ignored to filter out spikes due to uncontrollable external disturbances such as Java’s garbage collector or external processes on
PlanetLab. With two or more buckets filled, the variable $m$ is calculated by averaging the slope values from taking the last updated bucket with all other buckets. With $m$, the variable $b$ is calculated by substituting $d$ and $n$ from the last updated bucket. The reason for choosing the last updated bucket is so that the final equation contains that point, which is also the best indication of the broker's current performance. With both variables $m$ and $b$ known, the matching delay can easily be estimated by substituting a different number for $n$.

Input utilization ratio ($I_r$) is estimated by substituting estimated values into the variables of the input utilization ratio equation (Equation 4.1):

$$I_r = \frac{i'_r}{m'_r}$$

(4.10)

where $i'_r$ is the new rate of incoming publications estimated using PRESS, and $m'_r$ is the new maximum message match rate calculated by taking the inverse of the estimated matching delay. Since CPU utilization ratio is the equivalent of this load index when the value is less than 1, this formula is also used for predicting CPU load.

Output utilization ratio ($O_r$) is estimated by substituting estimated values into the variables of the output utilization ratio equation (Equation 4.2):

$$O_r = \frac{o'_u}{o_t}$$

(4.11)

Since the total output bandwidth is fixed, the only missing variable here is the estimated output bandwidth usage ($o'_u$), which is given by:

$$o'_u = o_u + \Delta o_u$$

(4.12)

where $\Delta o_u$ is the change in output traffic imposed by the offloaded subscribers estimated using PRESS.

## 4.3 Offload Algorithms

After profiling all subscriptions using PRESS, the offloading broker will feed the profiled data along with the load-accepting broker's load information to the offload algorithm to compute
the set of subscribers to offload. The offload algorithm to choose depends on what performance metric to balance, which is decided initially by the detector in the broker-action list as mentioned in Section 4.1.3. Table 4.2 summarizes the key properties of all offload algorithms.

For baseline comparison against the three offload algorithms, we introduce a naive offload algorithm called the Random Algorithm to load balance the three performance metrics. The purpose of introducing this algorithm is to show the effectiveness of load balancing a content-based publish/subscribe system with neither load estimation nor subscription space awareness. Section 4.3.4 will describe more details about this algorithm.

<table>
<thead>
<tr>
<th>Offload Algorithm</th>
<th>Performance Metric Being Balanced</th>
<th>Methodology</th>
<th>Side Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Input utilization ratio</td>
<td>Offload subscriptions in the CSS</td>
<td>Output utilization ratio is also decreased at offloading broker and increased at load-accepting broker</td>
</tr>
<tr>
<td>Match</td>
<td>Matching delay Overloaded CPU utilization ratio Overloaded memory utilization ratio</td>
<td>Offload subscriptions with least traffic</td>
<td>None</td>
</tr>
<tr>
<td>Output</td>
<td>Output utilization ratio</td>
<td>Offload subscriptions with highest traffic in Phase-I</td>
<td>None</td>
</tr>
<tr>
<td>Random</td>
<td>All</td>
<td>Offload randomly chosen subscriptions</td>
<td>Unpredictable, oscillations may occur</td>
</tr>
</tbody>
</table>

Table 4.2: Properties of all offload algorithms
Algorithm 1  \text{calcOffloadSet}(\text{localInfo}, \text{remoteInfo}, \text{poset}, \text{PRESSProfiles})

\begin{verbatim}
  if doneOffloading(localInfo, remoteInfo) then
    return null

  offloadSet ← {}  
  prevBalanceDiff ← getInitialBalanceDiff(localInfo, remoteInfo)  
  reportCardList ← computeReports(localCSS, PRESSProfiles, localInfo, remoteInfo)  

  while !localCSS.isEmpty() and !reportCardList.isEmpty() do  
    // Quit if no promising report is found  
    if bestReport is null then
      break
    
    if bestReport.numberOffloaded is 0 then
      continue  
      // Update the load percentage difference  
      prevBalanceDiff ← bestReport.balancePercentageDiff  
      // Add the required number of subscription IDs into the offloadSet  
      offloadSet.addAll(bestReport.getOffloadSubIDs())  
      // Update load information of both brokers  
      updateBothBrokerInfos(bestReport, localInfo, remoteInfo)  
      poset.remove(offloadSubIDSet)  
      // Terminate when load balancing is done  
      if doneOffloading(localInfo, remoteInfo) then
        return offloadSet
      
      reportCardList ← computeReports(localCSS, PRESSProfiles, localInfo, remoteInfo)
  
  return offloadSet
\end{verbatim}
4.3.1 Input Offload Algorithm

This algorithm is invoked by the offloading broker when the input utilization ratio needs load balancing. The aim here is to reduce the offloading broker’s input utilization ratio and increase the same metric on the load-accepting broker with minimal effect on the other performance metrics.

There are two strategies to reduce the offloading broker’s input utilization ratio: increase the rate at which messages are matched, or reduce the rate of incoming publication messages. Increasing the rate of matching is achieved by reducing the number of subscriptions in the matching engine. However, this action conflicts with the match offload algorithm that is trying to balance the matching delay and therefore is not applied here. Hence, the incoming publication rate can only be reduced by offloading subscriptions in the CSS because their subscription space is a superset of all subscriptions not in the CSS. With the poset [14], CSS lookup will take $O(1)$ time. Once the subscriptions in the CSS are identified, a report card is calculated for each of them. A report card consists of the following fields:

1. **Number of subscribers** of this subscription to offload.

2. Resulting **load percentage difference** between the two brokers by offloading this subscription, where a negative value indicates that the offloading broker will become less loaded than the load-accepting broker. This value is calculated using the estimated input utilization ratios of the two brokers in the input offload algorithm, matching delays in the match offload algorithm, and output utilization ratios in the output offload algorithm.

3. Boolean value indicating if this **subscription is covered** by the load-accepting broker’s CSS.

4. **Publication rate reduced** at the offloading broker estimated using PRESS.

5. **Output bandwidth required** per subscriber estimated using PRESS.

The number of subscribers to offload per unique subscription is restrained by two conditions. First, the offload should not overload any of the load-accepting broker’s resources.
Second, the performance metric of interest of the two brokers should be balanced within the *balanced threshold*, which is 0.005 by default; or bring the offloading broker’s metric below the load-accepting broker’s. The performance metric of interest for the input, match, and output offload algorithms are the input utilization ratio, matching delay, and output utilization ratio, respectively. The formulas for estimating the load impact of each subscriber is summarized previously in Section 4.2.

Specifically, the *balance threshold* parameter controls how closely an offload algorithm will attempt to balance the performance metric of two brokers in a local load balancing session. The value of this parameter can be any real number equal to or greater than 0. By setting the parameter to zero, the offload algorithm will offload subscriptions until the difference between the two brokers’ performance metric is zero or until any further offload from the offloading broker will make the load-accepting broker’s performance metric become higher than the offloading broker’s. If the parameter is a non-zero value, then the offload algorithm will offload subscriptions until the difference between the two brokers’ performance metric is within the defined threshold. It is possible for the load-accepting broker to end up having a higher load than the offloading broker in this case. Therefore, setting the *balance threshold* too high may cause the load balancing algorithm to diverge. In summary, this parameter controls the tradeoff between load balancing effectiveness and load balancing stability.

After calculating the report cards to determine the number of subscribers to offload for each subscription in the CSS, the subscription that results in the two brokers’ input utilization ratio difference closest to zero is chosen for offloading. This selection scheme ensures that subscriptions with the highest input publication rate are chosen first, which helps to reduce the number of subscriptions offloaded, and in so doing, reduces the impact on the load-accepting broker’s matching delay. If any additional offload will result in a higher load difference than before, then the selection process terminates. This guarantees that all load balancing actions will always converge to a state where the brokers have smaller load differences.

Subscriptions chosen to be offloaded are removed from the poset to prevent future consideration for offloading. Load information of both brokers (the one obtained in the mediation process) is updated with estimated values according to the offloaded subscription’s report card.
Updated load information of both brokers are used on the next iteration of the subscription selection algorithm. The selection process ends when no more subscriptions are available for offloading, the offloading broker's input utilization ratio is below that of the load-accepting broker, or the absolute difference between the two brokers' input utilization ratios fall within the balance threshold.

The computational complexity of the input offload algorithm is $O(n^2 \log n)$ where $n$ is the number of subscriptions at the offloading broker. A summary of the input offload algorithm is given in Algorithm 1 in the form of pseudocode.

### 4.3.2 Match Offload Algorithm

Although the input utilization ratio varies directly with the matching delay, balancing the input utilization ratio does not balance the matching delay. The objective of this offload algorithm is to balance the matching delays without affecting the input and output utilization ratios of the two brokers. Intuitively, subscriptions with the lowest publication traffic are most suited to this criterion. Furthermore, subscriptions that introduce the smallest amount of additional incoming traffic into the load-accepting broker are most favorable. In this algorithm, report cards are computed for all subscriptions in the offloading broker and they are sorted by ascending output bandwidth. The algorithm to compute number of subscribers to offload for each unique subscription is almost identical to the input offload algorithm. The only difference is that input utilization ratios are replaced by matching delays.

If the match offload algorithm is invoked because the broker is overloaded and wants to reduce its CPU utilization ratio, input utilization ratio, or memory utilization ratio, then subscriptions should continue to be offloaded until the CPU utilization ratio, input utilization ratio, and memory utilization ratio drops below the lower overload threshold. After a subscription is chosen to be offloaded, load information about both brokers are updated. The same criterion used in the input offload algorithm applies here for terminating the match offload process. The computational complexity of the match offload algorithm is $O(n^2 \log n)$ where $n$ is the number of subscriptions at the offloading broker. This offload algorithm follows the same framework as the input offload algorithm summarized in Algorithm 1.
Algorithm 2 calcOffloadSet(localInfo, remoteInfo, poset, PRESSProfiles)

if doneOffloading(localInfo, remoteInfo) then

    return null

// List variables classifying types of subscriptions in each phase
// p1 = Phase-I, p2 = Phase-II
p1typeIList ← {}
p1typeIIList ← {}
p1typeIIIList ← {}
p2List ← {}
offloadSet ← {}
ignoreSet ← {}

// Calculates report cards for subscription yet to be offloaded and
// puts Phase-I subscriptions into one of the three categorized lists
classifyPhase1Subs(localInfo, remoteInfo, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList, ignoreSet)

// Engage Phase-I offload first. If that doesn’t balance the load (i.e., returns
// false) then engage Phase-II offloading. If the latter have not balanced the
// load yet, then repeat Phase-I, etc.
while !calcPhase1OffloadSet(localInfo, remoteInfo, poset, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList, offloadSet, ignoreSet) do
    if calcPhase2OffloadSet(localInfo, remoteInfo, PRESSProfiles, p2List, offloadSet, ignoreSet) then
        break
    classifyPhase1Subs(localInfo, remoteInfo, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList)

return offloadSet
Algorithm 3 calcPhase1OffloadSet(localInfo, remoteInfo, poset, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList, offloadSet, ignoreSet)

\[
\begin{align*}
&\text{bestReport} \leftarrow \{} \\
&\text{prevBalanceDiff} \leftarrow \text{getInitialBalanceDiff(localInfo, remoteInfo)} \\
\text{while true do} \\
&\quad \text{if !p1typeIList.isEmpty() then} \\
&\hspace{1em} \text{bestReport} \leftarrow \text{pickBestReport(p1typeIList, prevBalanceDiff)} \\
&\quad \text{else if !coveredNonreducibleSubList.isEmpty() then} \\
&\hspace{1em} \text{bestReport} \leftarrow \text{pickBestReport(p1typeIIList, prevBalanceDiff)} \\
&\quad \text{else if !p1typeIIIList.isEmpty() then} \\
&\hspace{1em} \text{bestReport} \leftarrow \text{pickBestReport(p1typeIIIList, prevBalanceDiff)} \\
&\quad \text{else} \\
&\hspace{2em} \text{break} \\
&\hspace{1em} // \text{Quit if no promising report is found} \\
&\quad \text{if bestReport is null then} \\
&\hspace{1em} \text{break} \\
&\hspace{2em} \text{ignoreSet.add(bestReport)} \\
&\quad \text{if bestReport.numberOffloaded is 0 then} \\
&\hspace{1em} \text{continue} \\
&\hspace{1em} // \text{Update the load percentage difference} \\
&\hspace{2em} \text{prevBalanceDiff} \leftarrow \text{bestReport.balancePercentageDiff} \\
&\hspace{1em} // \text{Add the required number of subscription IDs into the offloadSet} \\
&\hspace{2em} \text{offloadSet.addAll(bestReport.getOffloadSubIDs())} \\
&\hspace{1em} // \text{Update load information of both brokers} \\
&\hspace{2em} \text{updateBothBrokerInfos(bestReport, localInfo, remoteInfo)} \\
&\hspace{2em} \text{poset.remove(offloadSubIDSet)} \\
&\hspace{1em} // \text{Terminate when load balancing is done} \\
&\hspace{2em} \text{if doneOffloading(localInfo, remoteInfo) then} \\
&\hspace{3em} \text{return true} \\
&\hspace{1em} \text{classifyPhase1Subs(localInfo, remoteInfo, PRESSProfiles, p1typeIList, p1typeIIList, p1typeIIIList, ignoreSet)} \\
\end{align*}
\]

\text{return false}
Algorithm 4 calcPhase2OffloadSet(localInfo, remoteInfo, PRESSProfiles, p2List, offloadSet, ignoreSet)

bestReport ← {}

prevBalanceDiff ← getInitialBalanceDiff(localInfo, remoteInfo)

classifyPhase2Subs(localInfo, remoteInfo, PRESSProfiles, p2List, ignoreSet)

while true do

if p2List.isEmpty() then

    return true

end if

bestReport ← pickBestReport(p2List, prevBalanceDiff)

// Quit if no promising report is found

if bestReport is null then

    break

end if

ignoreSet.add(bestReport)

if bestReport.numberOffloaded is 0 then

    continue

end if

// Update the load percentage difference

prevBalanceDiff ← bestReport.balancePercentageDiff

// Add the required number of subscription IDs into the offloadSet

offloadSet.addAll(bestReport.getOffloadSubIDs())

// Update load information of both brokers

updateBothBrokerInfos(bestReport, localInfo, remoteInfo)

poset.remove(offloadSubIDSet)

// Terminate when load balancing is done

if doneOffloading(localInfo, remoteInfo) then

    return true

end if

if bestReport has children nodes in poset then

    break

else

    p2List.clear()

classifyPhase2Subs(localInfo, remoteInfo, PRESSProfiles, p2List, ignoreSet)

end if

return false
4.3.3 Output Offload Algorithm

This algorithm attempts to balance the output utilization ratios of two brokers by manipulating the amount of output bandwidth used at each broker. Prioritizing subscriptions for the offload process is divided into two phases. In Phase-I, subscriptions that are covered by or equal to the load-accepting broker’s CSS are considered. These subscriptions are further classified into three types by using the fields computed for every subscription’s report card. Offloading Type-I subscriptions will reduce the input publication rate of the offloading broker. These should be offloaded first because they reduce the overall input load of the system. Type-II subscriptions are similar to Type-I, except that they do not reduce the input publication rate because all subscribers for a subscription cannot be offloaded to produce a more balanced state. Type-III subscriptions are considered last in Phase-I because they do not reduce the input publication rate of the offloading broker even if all subscribers for that particular subscription are offloaded.

The algorithm for calculating the number of subscribers to offload for each unique subscription is similar to the input offload algorithm shown previously, except that input utilization ratios are now replaced by output utilization ratios.

After a subscription is chosen to be offloaded, load information about both brokers is updated. If both brokers are balanced, then the algorithm stops and forwards the subscriber migration list to the mediator. Otherwise, Phase-II is invoked to further balance the output utilization ratio with some side-effects. All subscriptions considered in Phase-II are not contained in the CSS of the load-accepting broker. Therefore, these subscriptions may have the side-effect of significantly increasing the incoming publication rate of the load-accepting broker. What may happen is that there will be an oscillation between the input offload algorithm trying to balance the input utilization ratio disrupted by Phase-II of the output offload algorithm, and Phase-II of the output offload algorithm trying to balance the output utilization ratio disrupted by the input offload algorithm. To prevent this unstable situation from happening, Phase-II terminates when the input utilization ratios of both brokers are balanced, even if the output utilization ratios are not. An exception applies if the offloading broker is output overloaded, in which case the offloading broker will stop offloading once its output utilization ratio is below the lower overload threshold. With this exception, no oscillation occurs because the offloading
broker cannot take back any subscriptions since it has a status of N/A at the lower overload
threshold.

The sorting and selection scheme in Phase-II is exactly the same as in the input offload
algorithm with the use of load differences. If the subscription offloaded in Phase-II covers other
local subscriptions, then Phase-I is invoked to offload those covered subscriptions because they
are now covered by the load-accepting broker’s CSS. Otherwise, if the subscription offloaded
in Phase-II does not cover any other subscriptions, then Phase-II continues to run. The com-
putational complexity of the output offload algorithm is $O(n^2 \log n)$ where $n$ is the number of
subscriptions at the offloading broker. The output offload algorithm is summarized by pseu-
docode in Algorithm 2 with details of Phase-I and Phase-II summarized in Algorithms 3 and
4, respectively.

### 4.3.4 Random Algorithm

The Random Algorithm is a naive approach to load balance any of the three performance
metrics. Since the purpose of this algorithm is to represent a generic load balancer, it has no
load estimation and no awareness of subscription space. Instead, this algorithm uses simple
calculations to determine the number of subscriptions to offload, $c$, then randomly picks $c$
subscriptions to offload. This effectively gives the algorithm a runtime complexity of $O(c)$. It
is possible to make the subscription count random as well, but this will make the algorithm too
naive and inefficient to draw a meaningful comparison with the three offload algorithms.

To compute the offload subscription count, the offloading broker first computes the target
value to reach for the chosen performance metric $x$. The target value, $L_{x}^{avg}$, is simply the average
of the offloading broker’s value of the performance metric, $L_{x}^{off}$, and the load-accepting broker’s
value of the same performance metric, $L_{x}^{acc}$. Then, Equation 4.13 below is used to calculate $c_1$,
the number of subscriptions to offload, with $n_{off}$ representing the number of subscriptions at
the offloading broker itself.

$$
c_1 = n_{off} - L_{x}^{avg} \left( \frac{n_{off}}{L_{x}^{off}} \right)
$$

(4.13)

However, using $n_{acc}$ (the number of subscriptions residing at the load-accepting broker)
yields a different result:

\[ c_2 = L_{avg} \left( \frac{n_{acc}}{L_{acc}} \right) - n_{acc} \quad (4.14) \]

A simple solution is to simply take the average of \( c_1 \) and \( c_2 \) to arrive at \( c \). Or set \( c \) to be \( c_1 \) if \( c_2 \) is greater than the maximum number of subscriptions at the offloading broker.

Because the random algorithm is not aware of the potential oscillating effect of offloading certain subscriptions (such as those in Phase-II of the output offload algorithm as described above) and because it has no load estimation mechanism, the effect of each offload is unpredictable and may lead to load oscillations. Oscillations in turn cause the algorithm to diverge which cause instability in the system as witnessed in our experiments on heterogeneous platforms.
Chapter 5

Publisher Placement Algorithms

In this chapter, we present two different placement algorithms, POP (Publisher Optimistic Placement) in Section 5.1, and GRAPE (Greedy Relocation Algorithm for Publishers of Events) in Section 5.2, to intelligently relocate publishers while keeping the broker overlay intact to minimize both the average end-to-end delivery delay and system load. Both sections describe POP and GRAPE’s 3-Phase operation in detail: (1) gather publication delivery statistics on the publishers’ publications, (2) identify the target broker to relocate the publisher to, and (3) transparently migrate the publisher to the target broker.

5.1 The POP Placement Algorithm

The rest of this thesis makes use of the terms downstream and upstream to identify other brokers relative to an arbitrarily referenced broker and a publisher. Downstream brokers are those that receive publication messages from the referenced broker, directly or indirectly over multiple neighbors. In other words, downstream brokers are those farther away from the publisher compared to the referenced broker. The opposite definition holds for upstream brokers. Using Figure 5.1 as an example, if the referenced broker is B6 and the publisher is at B1, then B7 and B8 are downstream brokers while B5 and B1 are upstream brokers.

The following sections show POP’s 3-Phase operation in detail. Section 5.1.1 presents Phase 1, where POP probabilistically traces each publisher’s live publications to discover the location
and number of matching subscribers in the network. Section 5.1.2 describes Phase 2, where POP uses trace information obtained from Phase 1 to pinpoint the broker closest to the highest number of matching subscribers that the publisher should connect to. Section 5.1.3 presents Phase 3, which involves transparently migrating the publisher to the broker identified in Phase 2 with minimal routing table updates. Message sequence diagrams detailing each phase under Figure 5.1’s scenario are included in our online Appendix [23] for further clarification. Our evaluation shows that POP’s data structures use no more than 34% (or 19 MB) of additional memory, and message overhead varies between 6% and 57% at two most extreme POP configurations.

### 5.1.1 Phase 1: Distributed Trace Algorithm

The goal of Phase 1 is to gather the average number of subscribers downstream of each brokers’ neighbor links for each publisher client. To realize this goal, we developed (1) an algorithm to tag publication messages to trace where they got delivered, (2) a reply protocol to notify upstream brokers of the number of subscribers to which the publication was delivered at downstream brokers, and (3) a data structure to store and aggregate results from the traces.
Probabilistic Publication Tagging

POP utilizes a special publication tagging technique to reduce both message and computation overhead from publication tracing. Whenever the publisher’s first broker handles a publication message from the client, it can choose to trace the message by tagging/setting the trace header field to true, or disable tracing by leaving trace at its default value of false. Tagging is based on $P_{\text{trace}}$, which is defined by the function: $P_{\text{trace}} = 1 - \frac{T}{N}$. Here, $T$ is the number of messages already tagged for tracing in the current time window $W$, and $N$ is a configurable parameter that limits the maximum number of publication messages traced in time window $W$. By default, $N$ is set to 50 and $W$ to 60 s. Each publisher is associated with its own value of $T$. The advantages of using the $P_{\text{trace}}$ function over a constant function are: (1) the number of publications traced within $W$ is bounded by $N$, (2) for extremely low-rated publishers, at least one publication message is tagged with 100% probability in each time window, and (3) for high-rated publishers, this equation offers a higher chance of tagging publication messages sent near the end of each time window.

Trace Result Notification

On handling a publication message with a true value in the trace header field, the broker has to send back to the upstream broker a Trace Result Message (TRM). A TRM contains two fields: (1) publisher’s advertisement ID obtained from the publication’s header and (2) cumulative subscriber count, which is the total number of subscribers at and downstream of the reporting broker. A broker can only send a TRM to the upstream broker if any of the following two conditions are satisfied: (1) the publication message is only delivered to subscribers or (2) a corresponding TRM is received from each neighbor to which the publication message is sent.

Publisher Profile Table

POP stores trace results for each publisher into a Publisher Profile Table (PPTable) which has two columns: (1) downstream broker and (2) the average number of subscribers. A running average is used to maintain the average number of subscribers because it has the benefit of efficiently aggregating multiple values to conserve space. By default, the running average gives
a weight of 0.25 to the newest value and 0.75 to the last average value. Figure 5.1 shows an example of the PPTable at each broker after tracing one publication message that got delivered to all illustrated subscribers.

5.1.2 Phase 2: Decentralized Broker Selection Algorithm

The goal of Phase 2 is to use the PPTables gathered in Phase 1 to incrementally pinpoint the broker closest to the highest number of matching subscribers, which we refer to as the closest broker from here on. POP’s Phase 2 algorithm is initiated after two conditions are met: (1) the number of publications traced meets the threshold $P_{threshold}$ and (2) the publisher does not have any outstanding publication trace results (so as to prevent trace data inconsistency among brokers). By default, $P_{threshold}$ is set to 100. On Phase 2 initiation, POP in the publisher’s first broker creates a Relocation Request Message (RRM). The RRM contains three values: (1) publisher’s advertisement ID, (2) total number of subscribers down the link from which this request is sent, and (3) list of brokers traversed by this RRM in decreasing order of closest location. The latter field identifies brokers on the migration path that need routing table updates in Phase 3.

When a broker creates or receives a RRM, it has to determine the next closest neighboring broker to forward the message to. The next closest neighboring broker is:

\[ \text{the one whose number of downstream subscribers is greater than the sum of all other neighbors’ downstream subscribers plus the local broker’s subscribers.} \]

If no neighbor broker satisfies the closest condition, then the closest broker is itself. Note that each broker handling the RRM has its own definition of downstream as will be shown through an example in the next paragraph. The closest condition can be extended to include a threshold parameter to dampen any potential ping-pong effect when the difference is just one. However, we will leave this extension for future work and just focus on POP with minimal optimizations in this thesis. Figure 5.2 summarizes all three possible outcomes of the broker selection algorithm. If the closest broker is not the originator of the RRM, then a Relocation Answer Message (RAM) is sent back to the originator with the publisher’s advertisement ID
Chapter 5. Publisher Placement Algorithms

Figure 5.2: All possible outcomes of POP’s broker selection algorithm

and the list of brokers traversed by the RRM including the closest broker itself. Otherwise, the publisher is already at the closest broker, in which case Phase 3 is aborted and Phase 1 will be initiated again after getting \( P_{\text{threshold}} \) new trace results.

Using broker \( B_1 \) in Figure 5.1 as an example, since \( B_5 \)'s sum of 15 from the PPTable is greater than the sum of \( B_2, B_4, \) and \( B_1, 6 + 3 + 1 = 10, B_5 \) is the next closest broker. As a result, \( B_1 \) updates and forwards the RRM to \( B_5 \). Specifically, \( B_1 \) adds itself to the head of the broker list and increments the total number of subscribers field by 10, which is the number of subscribers at the local broker \( B_1 \) plus the number of subscribers at and downstream of all non-closest neighbors, namely \( B_2 \) and \( B_4 \). Upon receiving the RRM from \( B_1, B_5 \) finds that the number of subscribers downstream to \( B_6 \) (according to the PPTable) is greater than the
number of subscribers downstream to $B_1$ (according to the RRM). Therefore, $B_5$ updates the RRM’s broker list to $[B_5, B_1]$ and forwards the message to $B_6$. Upon receiving the RRM from $B_5$, $B_6$ discovers that there are 10 subscribers downstream to $B_5$. Since no neighbor is able to satisfy the closest condition, $B_6$ determines itself to be the closest broker and sends a RAM back to $B_1$ to initiate Phase 3.

5.1.3 Phase 3: Publisher Migration Protocol

On receiving a RAM from the closest (or target) broker, the publisher’s first (or source) broker initiates the migration by informing the designated publisher to (1) temporarily pause publishing or buffer its messages locally and (2) submit a migration advertisement, which is an advertisement with the RAM as payload, to the target broker. POP at the target broker intercepts the special advertisement message from entering the matching engine and sends a Migration Update Message (MUM) to itself carrying the list of brokers on the migration path and the publisher’s advertisement ID obtained from the advertisement’s payload. Each broker handling the MUM updates its own routing tables to reflect the publisher’s new location, clears the PPTable entry for this publisher, and forwards the MUM to the next broker along the migration path. Once the MUM reaches the source broker and finishes updating the routing tables, the source broker sends a Migration Complete Message (MCM) to the target broker to end the migration. The purpose of sending the MCM to the target broker over the migration path instead of the publisher directly is because the arrival of this message there guarantees that all subscriptions forwarded by any brokers on the migration path will have reached the target broker. At that point, the target broker completes the migration process by notifying the publisher to resume publishing and disconnect from the source broker.

Notice that our publisher migration protocol limits the amount of computational and message overhead to the set of brokers along the migration path. In a tree network consisting of $N$ brokers with typical fanout greater than one, there exists only one migration path and the overhead complexity is bounded by $O(\log N)$. Brokers outside of the migration path do not participate because the state of those brokers before and after the migration remains the same. The routing table update operations at the individual brokers in Phase 3 include: (1) updating
the last hop of the publisher’s advertisement to reflect the migrated position, (2) removing subscriptions that no longer match any advertisement, and (3) forwarding subscriptions that match the updated advertisement. To reduce the amount of matching overhead in operation #2, only subscriptions with a last hop equal to the advertisement’s new last hop need to be checked for removal. The entire migration session is transparent to the application as all migration activities are handled by a thin software layer built into the publish/subscribe client. Subscribers are also isolated from the migration as the migration protocol is completely lossless, though subscribers may notice a short delivery interruption while the publisher migrates. Our evaluation shows that a 10 hop migration takes 5 s on PlanetLab and 1.5 s on the cluster testbed.

For clarification, the following explains how each of the above update operations apply to the scenario given in Figure 5.1. Operation #1 applies to all brokers along the migration path where broker B6 updates publisher P’s advertisement last hop to a local destination, broker B5 updates P’s advertisement last hop to B6, and broker B1 updates P’s advertisement last hop to B5. Operation #2 applies to brokers B1 and B5 where subscription(s) from the 15 subscribers that reside on the right of broker B5 are removed\(^1\). Operation #3 applies to brokers B1 and B5 where subscription(s) from the 10 subscribers that reside on the left of B5 to broker B6 are forwarded\(^1\).

5.2 The GRAPE Placement Algorithm

Like POP, GRAPE follows the same 3-Phase operational design and uses the publisher migration protocol presented in Section 5.1.3. However, the data structures and algorithms that GRAPE uses in Phases 1 and 2 are completely different. As we will show in our evaluation, GRAPE uses up to an additional 58\% (or 31 MB) of memory with message overhead ranging between 0\% and 46\% at two extreme GRAPE configurations. Compared to POP, that is 24\% (13 MB) more memory but 20\% less message overhead in the extreme worst case.

\(^1\)With subscription covering, the number of subscriptions removed/forwarded may not equal the number of subscribers.
5.2.1 Phase 1: Distributed Publication Tracing

Logging Publication Delivery Statistics

GRAPE tracks publications from publishers only within trace sessions. In a trace session, \( G_{\text{threshold}} \) publications are traced. By default, \( G_{\text{threshold}} \) is 100. Each publisher is associated with its own trace session as managed by its first broker. Trace sessions are identified by the message ID of the first trace-enabled publication in that session. Message IDs are uniquely generated by prefixing the value of an incrementing counter with the ID of the publisher’s first broker. Publications published within a trace session carry the same trace session ID in the \text{traceID} header field. A publication that is not trace-enabled has \text{traceID} set to null.

During a trace session, brokers handling a trace-enabled publication capture two pieces of information. (1) The total number of local subscribers that matched this publication, or simply the total number of local deliveries. This value is used in Phase 2 to estimate the average end-to-end delivery delay of all matching subscribers when GRAPE tries to place the publisher at different brokers. (2) The set of publication messages delivered to local subscribers. This information allows Phase 2 to accurately estimate the amount of traffic that flows through each broker when GRAPE tries to place the publisher at other brokers. Instead of storing a set of publication messages, we developed a novel scheme that utilized one \text{String} and one bit vector variable. The \text{String} variable records the trace session ID, whose suffix signifies the starting index of the bit vector. On delivering a trace-enabled publication with message ID \( M + \Delta \) for a trace session with identifier \( M \), GRAPE will set the \( \Delta \)-th bit of the bit vector. An example is demonstrated in Figure 5.3 with \( M = 212 \). Use of the bit vector comes with many advantages, including space efficiency, ease of aggregating multiple bit vectors with the \text{OR} bit operator,
and the direct proportional relationship between cardinality and message rate. Unlike POP where publications are probabilistically selected for tracing, GRAPE has to trace consecutive publications in a trace session to minimize the size of the bit vector.

Retrieval of Delivery Statistics

When the required number of publication messages are traced as governed by $G_{\text{threshold}}$, GRAPE sends a Trace Information Request (TIR) message to all downstream neighbors that have received at least one publication from this publisher within this trace session. Brokers receiving a TIR message will (1) forward the TIR message to downstream neighbors that satisfy the previously stated condition, (2) wait to receive a Trace Information Answer (TIA) reply message from the same set of downstream neighbors, and (3) send an aggregated TIA reply message containing their own and all downstream brokers’ trace information in the payload. Brokers with no downstream neighbors immediately reply back with a TIA message to the upstream neighbor with a payload containing the following information about themselves: (1) broker ID, (2) neighbor broker ID(s), (3) bit vector capturing the delivery pattern, (4) total number of local deliveries, (5) input queuing delay, (6) average matching delay, and (7) output queuing delay to each neighbor broker and the client binding. The latter three figures can be measured outside of GRAPE by a monitor module as is the case in our implementation. For additional clarification, please see our online Appendix [23] for a message sequence diagram that shows GRAPE’s trace retrieval protocol under the scenario illustrated in Figure 5.1. If we assume default GRAPE settings with each broker’s ID to be around 10 characters, each broker having on average three neighbors, then the size of one broker’s payload is only about 100 bytes. After sending a TIA message, the broker clears all data structures related to that trace session to free up memory. Compared to POP, GRAPE’s TIA messages are very similar to POP’s TRM messages. What is different, however, is that GRAPE sends a reply message after each trace session whereas POP sends a reply message after each traced publication.
5.2.2 Phase 2: Broker Selection Algorithm

With the statistical information from Phase 1, GRAPE in Phase 2 can estimate the average end-to-end delivery delay and system load if the publisher is moved to any one of the candidate brokers. The candidate brokers are the downstream brokers that replied with a TIA message in Phase 1. In POP, where the broker selection algorithm is distributed, the broker selection algorithm in GRAPE is entirely centralized at the publisher’s first broker. Some may argue that the amount of processing will overwhelm a node or there exists a single point of failure, but both arguments are not quite true. The total processing time on PlanetLab never exceeded 70 ms in the worse case with subscribers residing on all 63 brokers in the network. As well, each publisher is managed by GRAPE running at the publisher’s first broker. Therefore, if the first broker fails, the publisher can reconnect to any other broker and continue to be managed by another instance of GRAPE. The major benefits of adopting a centralized approach are the ease of design, implementation, and verification.

Algorithm 5 calcAvgDelay(stats, cumDelay, currBroker, prevBroker)

// Get delivery statistics from clients on the current broker
stats.totalDelay+ = currBroker.totalDeliveries ×

(cumDelay + currBroker.queueAndMatchDelaysTo(client))
stats.totalDeliveries += currBroker.totalDeliveries

for neighbor in currBroker.neighborSet do

// Skip the upstream broker
if neighbor equals prevBroker then
   continue

// Accumulate this broker’s processing and queuing delays
newCumDelay = cumDelay +
   currBroker.queueAndMatchDelaysTo(neighbor)

// Get delivery statistics of clients at downstream brokers
   calcAvgDelay(stats, newCumDelay, neighbor, currBroker)

return
Algorithm 6 calcTotalMsgRate(currBroker, prevBroker)
// Get the message rate going through this broker
localMsgRate =
calcDownstreamBV(currBroker, prevBroker).cardinality
// Get the total message rate at all downstream brokers
dsMsgRate = 0
for neighbor in currBroker:neighborSet do
    if neighbor equals prevBroker then
        continue
    dsMsgRate += calcTotalMsgRate(neighbor, currBroker)
// Return the sum of all brokers’ message rates
return localMsgRate + dsMsgRate

Algorithm 7 calcDownstreamBV(currBroker, prevBroker)
aggregatedBV = currBroker.bitVector
// Take local deliveries and OR with downstream deliveries
for neighbor in currBroker:neighborSet do
    // Skip over the upstream broker
    if neighbor equals prevBroker then
        continue
    // Try to retrieve the downstream bit vector to given neighbor
dsBV = currBroker:getDownstreamBVTo(neighbor)
    // If the downstream bit vector has not been calculated before, then...
    if dsBV is NULL then
        // Calculate and store the downstream bit vector at each downstream broker recursively
        dsBV = calcDownstreamBV(neighbor, currBroker)
        currBroker:setDownstreamBVTo(neighbor, dsBV)
    // Aggregate all downstream bit vectors (if any) with the local bit vector
    aggregatedBV |= dsBV
return aggregatedBV
GRAPE allows the user to prioritize which of the two metrics to minimize: average end-to-end delivery delay or system load (in the form of message rate), and also specify a minimization weight from 0 to 100% to indicate how much of that metric to minimize. If the primary metric to minimize is delay, and the minimization weight is $P$, then GRAPE first asks the publisher to reply back with a set of ping times to each candidate broker. Due to fluctuating network conditions and unavailability of ping on PlanetLab, publishers instead invoke an API on each candidate broker five times to measure the round trip times. On our cluster testbed, publishers invoke the API once with multiple candidate brokers simultaneously. Landmark and multidimensional-scaling based latency estimation techniques such as Netvigator [79] and Vivaldi [31] can be substituted in place of ping, but they trade off faster turnaround time for less accuracy.

The ping times together with the queuing and matching delays from each broker allow GRAPE to estimate the average end-to-end delivery delay with the publisher located at any downstream candidate broker by using the \texttt{calcAvgDelay()} function as shown in Algorithm 5. This function recursively calculates the average end-to-end delivery delay from the publisher to each subscriber according to the number of deliveries made in the past and using actual queuing and matching delay measurements at each broker. After invoking \texttt{calcAvgDelay()} on each candidate, GRAPE normalizes the candidates’ delivery delays and drops those candidates with delivery delays greater than $100 - P$. GRAPE then calculates the total system message rate with the publisher positioned at each remaining candidate by using \texttt{calcTotalMsgRate()} as shown in Algorithm 6. The total system message rate is the sum of the input message rates introduced by this publisher into every broker. Note that the input message rate at each broker may be different as each broker’s message rate depends on both local and downstream subscriptions. To aid in this calculation, \texttt{calcTotalMsgRate()} uses the helper function \texttt{calcDownstreamBV()} (shown in Algorithm 7) to aggregate downstream broker bit vectors to accurately compute the input message rate at each broker. The candidate that offers the lowest total system message rate is the selected broker.

On the other hand, if the primary metric to minimize is load, then the algorithm is reversed. GRAPE first calculates the total system message rate with the publisher placed at each candidate broker by using \texttt{calcTotalMsgRate()}, drops candidates with normalized message rates past
100 − P, fetches the set of publisher ping times, calculates the average delivery delay with the publisher positioned at each candidate broker by using \texttt{calcAvgDelay()}, and finally selects the broker that offers the least average delivery delay. At the very extreme case, if \textsc{GRAPE} is set to minimize load at 100%, then \textsc{GRAPE} will select the candidate where the publisher introduces minimal amount of traffic in the system without regards to the average delivery delay. If \textsc{GRAPE} is set to minimize delay at 100%, then \textsc{GRAPE} will select the candidate that offers the lowest average delivery delay without regards to the system load. The worst case runtime complexity of this algorithm is $O(N^2)$ where $N$ is the number of brokers in the system. Our experiments on PlanetLab show that even when $N$ is 63, \textsc{GRAPE}'s broker selection algorithm took less than 70 ms.
Chapter 6

Resource Allocation Algorithms

In this chapter, we present resource allocation algorithms that allocate as few brokers as possible for any given workload, while maximizing the resource utilization of allocated brokers. Our algorithms follow a 3-phase scheme to reconfigure the publish/subscribe system. Section 6.1 describes Phase 1 where we gather performance and workload information from the network using bit vectors. Section 6.2 describes Phase 2 where we allocate the subscriptions to brokers using the information gathered from Phase 1. Section 6.3 describes Phase 3 where we recursively construct the broker overlay with the subscriptions already allocated. Finally, publishers are placed strategically onto the newly built broker overlay with the GRAPE publisher relocation algorithm.

6.1 Phase 1: Resource Allocation Framework

Our strategy to minimize the number of active brokers consists of three phases. Phase 1 gathers performance and workload information from the brokers in order to carry out computations in Phases 2 and 3. Phase 2 assigns subscriptions to brokers using a subscription allocation algorithm. Phase 3 recursively constructs the broker overlay using the subscription allocation strategy from Phase 2. After Phase 3, GRAPE relocates the publishers from the center of the network to where the matching subscribers reside.

The components required to support Phases 1 to 3 are:
• An external publish/subscribe client called *Coordinator for Reconfiguring the Overlay and Clients* (CROC) that connects to any broker in the overlay to collect information about the currently deployed system, executes Phases 2 and 3, and orchestrates the reconfiguration.

• Integrated into each broker is the CROC Back-end Component (CBC) that responds to commands sent by CROC, such as responding to information requests, profiling of subscribers, etc.

### 6.1.1 Information Gathering

The information gathering protocol can be implemented as an out-of-band messaging protocol or using publish/subscribe that is already supported by the system. We chose the latter methodology to avoid the additional complexity of a new messaging layer. When CROC connects to a broker on the overlay, it sends a *Broker Information Request* (BIR) message to the first broker. Whenever a broker receives a BIR message, it broadcasts the BIR message to all of its neighbors. Brokers reply to the BIR message with a *Broker Information Answer* (BIA) message only if it has no neighbors to forward the BIR message or have received the BIA messages of all neighbors to which it forwarded the BIR message. The latter condition enables the aggregation of received BIA messages with the current broker’s into one BIA message to reduce overhead. The BIA message contains the following information about the broker:

• **URL** - This is needed for reassigning subscribers and broker neighbors in Phases 2 and 3

• **Matching delay function** - A linear function that models the matching delay as a function of the number of subscriptions. This enables CROC to predict the input load of the broker during subscription assignment and overlay construction in Phases 2 and 3

• **Total output bandwidth** - CROC uses this to predict the output load of the broker during subscription assignment and overlay construction in Phases 2 and 3

• **Set of local subscriptions and their profiles** - CROC will relocate these subscriptions in Phase 2 based on their information profile
Chapter 6. Resource Allocation Algorithms

• Set of local publishers and their profiles - CROC uses this information to predict the load imposed by each subscription

Once CROC obtains all BIA messages from every broker in the system, it executes Phases 2 and 3 to reassign subscriptions to brokers and reconfigures the broker overlay. The results of the reassignment is in the form of publications directed to each broker controlling where publishers and subscribers should migrate, and which neighbors brokers should connect with.

6.1.2 Subscription and Publisher Profiles

A subscription profile captures the publications sinksed by a subscription in the recent past. This allows CROC to accurately estimate the load requirements of subscriptions without any assumptions on the workload distribution (i.e., Gaussian, Zipf, etc.) and cluster subscriptions independent of the publish/subscribe language. Profiles for subscriptions are generated and maintained by the CBC at the subscribers' immediate brokers.

A subscription profile consists of one or more bit vectors each associated with a counter variable. The bit vector is a medium to record the publications that this subscription received from a specific publisher using as little memory space as possible. Thus, there is one bit vector for every unique publisher. Using the left side of Figure 6.1 as an example, subscription $S_1$ has two bit vectors because it received publications from two publishers, $Adv1$ and $Adv2$. Each
publisher appends a message ID, which is just an integer counter, as well as its globally unique
advertisement ID into its publication messages, which serves to identify the publisher of every
publication. An integer counter is associated with each bit vector to indicate the ID of the
first bit in the bit vector. Naturally, a set bit in the bit vector corresponds to the subscription
having received that particular publication. Using Figure 6.1 as an example, subscription $S_1$
received publications with IDs 75, 76, and 77 from publisher $Adv1$ and publications with IDs
144 to 148 from publisher $Adv2$.

Bit vectors have bounded size, whose default value is 1,280. A larger size will improve
the accuracy of estimating the anticipated load of a subscription, but will lengthen the time
required to profile subscriptions (i.e., fill up the bit vector). If on receiving a publication, the
bit to set in the bit vector exceeds the size of the bit vector length, then the bit vector is shifted
just enough to record the publication in the last bit of the bit vector while updating the integer
variable by the number of bits shifted. Depending on the implementation, if the first bit starts
at the most significant bit in memory, then one would shift the bit vector to the left, or vice
versa. For example, if the bit vector length is 10 while the counter representing the first bit is
100, and an incoming publication has a publication ID of 119, then shift the bit vector by 10
bits, set the bit at index 9, and update the counter to 110.

A publisher profile contains the publisher’s advertisement ID, publication rate, bandwidth
consumption, and the message ID of the last publication message sent. The second and third
pieces of information are used for estimating the load requirements of subscriptions. The last
piece of data is used for synchronizing the message ID counter in all bit vectors that correspond
to the same publisher. As an example, to estimate the bandwidth requirement of a subscription
with 10 out of 100 bits set in a bit vector corresponding to a publisher whose publication rate
is 50 msg/s and bandwidth is 50 kB/s, the publication rate induced by this subscription is
5 msg/s and the output bandwidth requirement of this subscription on a broker is 5 kB/s.
6.2 Phase 2: Subscription Allocation Algorithms

We developed three subscription allocation algorithms of different complexities to allocate subscriptions to a minimal set of brokers: Fastest Broker First (FBF), BIN PACKING, and Clustering with Resource Awareness and Minimization (CRAM). We will refer to the former two approaches as sorting algorithms. The set of subscriptions to allocate consists of all the subscriptions reported in the BIA messages in Phase 1. We call this set of subscriptions the subscription pool. The set of brokers on which to allocate the subscriptions include all the brokers that sent a BIA message back to CROC. We call this set of brokers the broker pool. The outcome at the end of Phase 2 are a set of non-connected brokers where some have subscriptions allocated to them and some do not.

6.2.1 FBF - Fastest Broker First

In FBF, brokers in the broker pool are first sorted in descending resource capacity. From our experiences in working with an open source publish/subscribe system [58] on cluster and Internet scale testbeds, the bottleneck of a broker is not the processing but the forwarding of messages, that is, the network I/O. Thus, we first sort the brokers in the broker pool in descending order of total available output bandwidth. Next, a subscription is randomly removed from the subscription pool and is assigned to the next most resourceful broker that has the capacity to handle the subscription. A broker is deemed to have enough capacity to handle a subscription only if by accepting this subscription, its remaining available output bandwidth is greater than 0 and its incoming publication rate is less than or equal to its maximum matching rate. The maximum matching rate is calculated by taking the inverse of the matching delay computed using the matching delay function supplied in the BIA message. The algorithm ends when all subscriptions from the subscription pool are allocated to the brokers or if at least one subscription cannot be allocated to any broker. Assuming that the number of subscriptions is much larger than the number of brokers, the complexity of this algorithm is O(S) where S is the total number of subscriptions in the system.
6.2.2 BIN PACKING

BIN PACKING is similar to FBF except that instead of randomly picking subscriptions and assigning them to brokers, subscriptions in the subscription pool are first sorted in descending order of bandwidth requirement. Then, the algorithm repeatedly allocates the next subscription with the highest bandwidth requirement to the next most resourceful broker that has the capacity to handle the subscription. The algorithm ends when all subscriptions from the subscription pool are allocated to the brokers or if at least one subscription cannot be allocated to any broker. Due to the sorting of the subscriptions, the complexity of this algorithm is \(O(S\log(S))\). Our experiments show that BIN PACKING consistently allocates one less broker than FBF, which is inline with theoretical expectations [28].

6.2.3 CRAM - Clustering with Resource Awareness and Minimization

CRAM is significantly different from the two prior sorting algorithms because it clusters the next closest pair of subscriptions before allocating them to brokers. The closeness between two subscriptions can be measured by the following four possible closeness metrics, where \(S_1\) and \(S_2\) represent the bit vector profiles of two arbitrary subscriptions.

- **INTERSECT**: \(|S_1 \cap S_2|\) - cardinality of the intersection

- **XOR**: \(|S_1 \oplus S_2|\)\(^{-1}\) - inverse of the xor’ed cardinality with a capped maximum value to handle division by zero. This is derived from the closeness metric in Gryphon [9] to make it consistent with the other metrics (i.e., higher magnitude is more favorable).

- **IOS**: \(\frac{|S_1 \cap S_2|^2}{|S_1| + |S_2|}\) - cardinality of the intersection squared over the sum of the cardinalities

- **IOU**: \(\frac{|S_1 \cap S_2|^2}{|S_1 \cup S_2|}\) - cardinality of the intersection squared over the cardinality of the union

Ideally, the two subscriptions that share the highest overlap of publication traffic while introducing the least amount of non-overlapping traffic is desired. The INTERSECT metric is the simplest and it accounts for the former but misses the latter property. The XOR metric accounts for the latter but misses the former property. In addition, the XOR metric cannot identify whether there is an empty or non-empty relationship among two subscriptions. Hence
we develop two additional closeness metrics, *Intersect-Over-Sum* (IOS) and *Intersect-Over-Union* (IOU), that not only account for both conditions and yield zero when two subscriptions have empty relationship, but also favor clustering higher traffic subscriptions (by taking the square of \(|S_1 \cap S_2|\)) since their impact on load is more significant than lower traffic subscriptions. In our evaluation, we test and compare the effectiveness of all four metrics.

We will now explain the foundation of the CRAM algorithm and then present optimizations to improve its runtime efficiency and clustering effectiveness. On initialization, CRAM allocates the subscriptions without any clustering using the BIN PACKING algorithm. If this fails, (i.e., due to insufficient broker resources) then the algorithm terminates. Otherwise, record the subscription allocation scheme. Then, CRAM repeatedly executes the following steps:

1. Find and cluster the pair of subscriptions having the next highest non-zero closeness value.
   When two subscriptions are clustered, their bit vectors are aggregated together using the OR bit operation to form a new subscription (see Figure 6.1). The subscription pool is updated of the new pairing. If no pairing can be found, then the algorithm ends and returns the last successful allocation scheme.

2. Allocate the subscriptions in the subscription pool using the BIN PACKING algorithm. If allocation fails, then undo and note the clustering (so that this subscription pair will not be paired again) and revert the changes made to the subscription pool. Otherwise, record the successful subscription allocation scheme.

3. Repeat step 1.

The purpose of recording the last successful allocation scheme upon initialization and at the end of step 2 is so that the algorithm can return an allocation scheme if all subsequent clustering results are not allocatable. In terms of computational complexity, the CRAM algorithm as we have described so far is \(O(S^3 \log(S))\) because step 1 finds up to \(O(S^2)\) pairs of subscriptions, takes \(O(S^2 \log(S))\) time to sort the subscription pairs in descending closeness, and repeats up to \(S\) iterations.
Optimization 1 - Grouping of Equal Subscriptions

In order to reduce the computation complexity of step 1, we need to reduce $S$, the number of subscriptions, by putting subscriptions with equal bit vectors into the same Group of Identical Filters (GIF). In our experiments using 8,000 subscriptions, $S$ is reduced by up to 61%. Therefore, instead of clustering pairs of individual subscriptions, this optimization facilitates clustering pair of GIFs. However, there can be multiple subscriptions in a GIF, and clustering all subscriptions in the two GIFs may exceed the broker capacity. To overcome this problem, we have developed new clustering techniques based on the relationship among the two GIFs. The relationship between a pair of GIFs is determined by the publications that they receive, which is captured by the bit vectors. If their relationship is equal, then the GIF is paired with itself. In this case, we use binary search to repeatedly find the largest allocatable cluster(s) of the GIF’s subscriptions. The result is one or more units of clustered subscriptions within the GIF. If their relationship is intersect, such as the relationship of $S_1$ and $S_2$ illustrated in Figure 6.2, then we cluster the smallest subscription unit from each GIF since the larger subscription units within each GIF are clustered to their largest allocatable size already. If one GIF’s bit vector is a superset or subset of another, then we cluster the lightest subscription unit from the covering GIF with as many subscriptions from the covered GIF (by doing binary search on the set of covered subscriptions sorted in ascending order of output bandwidth requirement).

Optimization 2 - Search Pruning

Our second optimization is to limit the search space for finding the closeness between GIFs. Specifically, we want to avoid spending any CPU cycles on calculating the closeness between GIF pairs that have empty relationship or lower closeness than what is already found. At the same time, we also want to reduce the number of candidate GIF pairs to sort in step 1 from $S^2$ to $S$ by keeping track of only the closest partnering GIF pair for each GIF. This effectively reduces the sorting complexity on each iteration of the algorithm from $O(S^2\log(S))$ to $O(S\log(S))$, which reduces the complexity of the overall algorithm from $O(S^3\log(S))$ to $O(S^2\log(S))$.

In order to support our second optimization, we utilize the poset data structure [14] because it allows us to exploit an important property related to the closeness metrics, which we explain
shortly. The overhead of using the poset data structure is well worth the tradeoff too because the computational complexity for insertions and deletions is $O(S)$. However we expect insertions and deletions to take on average $O(\log(S))$ for a balanced poset structure. Our experiments support this claim as inserting 3,200 GIFs takes only 2 s.

Upon invoking the CRAM algorithm, each GIF is inserted as a node in the poset. In this work, we use the bit vectors from the subscription profiles to identify the relationship among GIFs instead of the subscription language as is commonly used in literature. The algorithm for identifying the relationship among subscriptions with one or more bit vectors is explained in our online Appendix [26]. With the poset initialized, CRAM iteratively takes a GIF from the GIF pool and calculates the GIF’s closeness with each poset node, starting from the root node and iterating over the poset using breadth-first enumeration. Depending on which closeness metric is used, the enumeration ending criteria is different.

For the INTERSECT, IOS, and IOU metrics, the closeness value will either be zero (if GIF pair has empty relation) or a non-zero value (if GIF pair has non-empty relation). For GIF pairs with zero closeness, all of their descendent nodes are guaranteed to have zero closeness as well, which enables CRAM to prune the search process. For GIF pairs with non-zero closeness, the closeness value either remains constant (INTERSECT) or increases (IOS and IOU) traversing down the poset. The closeness value then starts to decrease when traversing past the GIF’s own poset node where the number of intersecting bits start to decrease. Therefore, the search for a GIF’s closest pair can be pruned once the closeness value starts to decrease, which is at the closest pair’s children nodes. For GIF pairs that fail the allocation test, they need to be noted so that they will be skipped over in subsequent searches. Our experiment results demonstrate that finding the closest pair for every GIF using this optimization reduces the number of closeness computations from approximately 5,000,000 (assuming that the first optimization reduced 8,000 subscriptions to 3,200 GIFs) to 280,000. This means that in the average case, the computational complexity to search for the closest pair for each GIF is reduced from $O(S)$ to $O(\log^2(S))$.

The XOR metric, however, does not have the advantage of pruning the search over GIFs having empty relationships because the value of this metric is non-zero regardless of whether the relationship is empty or not. In fact, experiment results show that GIFs with empty relationships
do actually get clustered together. Compared to the other three closeness metrics, XOR requires at least 75% longer computation time.

**Optimization 3 - One-to-Many Clustering**

The CRAM algorithm as we have described so far tries to cluster two GIFs on each iteration of the loop. This is similar with the pairwise subscription clustering algorithm in [73]. However, we develop an additional subscription clustering strategy that further improves on the effectiveness of pairwise clustering using the INTERSECT, IOS, and IOU metrics.

Consider the scenario illustrated in Figure 6.2. If we take each block in the grid as one bit, then subscription $S_1$ on the left has 36 bits and subscription $S_2$ on the right has 16 bits. The shaded 8 bits in the middle is $S_1 \cap S_2$, which is the set of publications that both $S_1$ and $S_2$ sink. Using the IOS metric as an example (INTERSECT and IOU will yield the same verdict), the closeness between $S_1$ and $S_2$ is $8^2 \div 60 \approx 1.07$, between $S_1$ and any one of its covered subscriptions (2x2 shaded blocks) is $4^2 \div 40 = 0.4$, and between $S_2$ and any one of its covered subscriptions (1x1 shaded blocks) is $1^2 \div 25 \approx 0.04$. Since the closeness between $S_1$ and $S_2$ is highest, the pairwise algorithm will cluster $S_1$ and $S_2$ over clustering either $S_1$ or $S_2$ with their covered subscriptions. However, it is actually more favorable to first cluster $S_1$ (likewise for $S_2$) with its set of covered subscriptions before clustering $S_1$ with $S_2$ together because there is higher overlap between each of $S_1$ and $S_2$ with their covered subscriptions. The IOS metric also supports this claim as the closeness between $S_1$ ($S_2$) and all of its covered subscriptions is $12^2 \div 48 = 3$ ($8^2 \div 32 = 2$), which are greater than the closeness between $S_1$ and $S_2$ themselves.

Therefore, our 3rd optimization to CRAM is to first attempt to cluster each GIF with their covered GIFs if a candidate GIF pair has an intersect relationship. The lookup for the set of covered GIFs takes $O(1)$ with the poset. The set of covered GIFs to cluster with the candidate GIF, called **Covered GIF Set (CGS)**, should be chosen such that the GIFs in CGS have least overlapping set bits with each other to minimizing the number of GIFs to cluster (which increases the chance of passing the allocation test) while maximizing the total bit coverage (which increases the closeness value of the CGS with its parent GIF). Naturally, the greedy solution to the set cover problem fits in nicely with these requirements by mapping the “1” bits within the
bit vector(s) of each GIF in our approach to the elements of each set in the generic solution. That is, we maintain a list of the covered GIFs sorted by the number of bits not already in the CGS in descending order. On each iteration of the loop, include the next GIF in the head of the list to the CGS, update and re-sort each GIF remaining in the list according to the new CGS, and terminate when clustering any further GIF will force the CGS-parent cluster to exceed the load requirements of the original GIF pair. The intuition behind this terminating condition is to ensure a fair closeness value comparison. Once the CGS is finalized, the CGS is valid if it passes the allocation test and its closeness with its parent GIF is higher than the closeness between the original candidate GIF pair.

### 6.3 Phase 3: Broker Overlay Construction

In Phase 2, we have described three subscription allocation algorithms which allocate subscriptions to a set of brokers. In Phase 3, we design a tree overlay to connect the brokers allocated in Phase 2 with the assumption that the publishers will initially relocate to the root of the tree. Once the overlay is designed, CROC informs each broker to execute the reconfiguration, after which GRAPE strategically relocates the publisher clients on the final overlay.

Our goal in designing the broker overlay construction algorithm is to make it as least complex
as possible while still being able to make systematic and intelligent decisions. To do so, we simply map each broker allocated in the previous run of the subscription allocation algorithm to a subscription profile by aggregating all subscription bit vectors serviced by the broker using the OR bit operator (similar to building a new subscription from clustering two subscriptions as illustrated in Figure 6.1) and then recursively invoke the subscription allocation algorithm. Pictorially, we are building the tree overlay layer-by-layer, starting with subscriptions at the bottom layer, with lesser-and-lesser brokers allocated at each higher layer until only one broker is allocated. That last allocated broker is the root of the tree and is also where all the publishers connect to in the new topology.

The benefits of our broker overlay construction algorithm is two-folds. One, we are reusing the subscription allocation algorithm in Phase 2, which means we are reusing as much code and logic as possible. This not only helps minimize code development time but also minimizes additional complexity into the algorithm. Two, the recursive nature makes the entire allocation scheme consistent among subscriptions and brokers. For example, if CRAM is used to allocate subscriptions to brokers, then CRAM is also used to build the broker overlay. Additionally, we introduce three new optimizations to the broker overlay construction algorithm that further reduce the number of brokers allocated in Phase 3. These optimizations are applied in the order that they are presented below after allocating each layer of brokers, which is just prior to the recursive invocation.

6.3.1 Optimization 1 - Eliminate Pure Forwarding Brokers

Figure 6.3a shows an example of a pure forwarding broker, B5, that has only one neighbor to forward publication traffic. Since pure forwarding brokers are not even multicasting and just purely forwarding incoming traffic, they can be safely deallocated to avoid that extra broker hop of unnecessary filtering and forwarding.

6.3.2 Optimization 2 - Takeover Children Broker Roles

Since our overlay construction algorithm allocates brokers layer-by-layer, it is possible that the load assigned to the last broker at each layer be very low. In this case, brokers allocated on
the next higher level can potentially have the capacity to directly handle the load of the under-utilized child broker. Figure 6.3b demonstrates an example of such a scenario. To handle this anomaly, we check if any allocated broker in the last allocation run can directly handle the load of its children brokers in order of least-to-highest utilization. This sorting order maximizes the number of children brokers that can be taken over by the parent broker.

6.3.3 Optimization 3 - Best-Fit Broker Replacement

Also due to the same condition as described in the above section, allocated brokers may be under-utilized. Under-utilized resources are not only inefficient but can also lead to more brokers allocated than is necessary. To deal with this problem, this optimization replaces each allocated broker with a broker whose resource capacity has the best-fit. Figure 6.3c illustrates an example of this optimization scheme.
Figure 6.3: Broker overlay construction optimizations

(a) Opt. 1: Deallocate B5 since it is a pure forwarding broker

(b) Opt. 2: B3 has enough capacity to serve B2’s subscriptions directly along with B1

(c) Opt. 3: Replace under-utilized high resource brokers with low resource brokers
Chapter 7

Evaluation

For experimentation, we implemented all approaches presented in this thesis on PADRES [52, 53, 92, 58, 41, 24, 51]. PADRES is an open source distributed content-based publish/subscribe system developed by the Middleware Systems Research Group (MSRG) at the University of Toronto. The back-end component of each approach is integrated into the PADRES broker as an additional internal module as illustrated in Figure 7.1. Using the flow of a message as a guide to describe the broker’s architecture, a message from a neighbor broker or client comes into the broker through an input binding. The message is enqueued into the broker’s input queue to wait for processing. The matching engine dequeues messages one by one from the input queue to perform the publish/subscribe match operation. Output from the matching engine is a set of messages with preset next hops, which can be a neighbor, a client, or one of the broker’s internal modules. Each module in the PADRES broker has an associated output queue to receive messages from the matching engine. Messages sent by a module are enqueued back into the broker’s input queue to let the matching engine decide how to route those messages. Each broker that is neighbor to this broker has an associated output queue to receive messages from this broker and a neighbor binding to transmit messages from this broker’s output queue to the neighbor. Each client is attached to one of the broker’s client bindings to receive messages from the broker. The purpose of bindings is to allow the broker to support multiple network communication protocols.
7.1 Evaluation of the Load Balancing Algorithm

We ran experiments on homogeneous and heterogeneous platforms including PlanetLab and a cluster testbed to compare how the load balancing algorithm performs in a real setting versus simulation setting. Our results examine the effectiveness and efficiency of load balancing at both the local and global levels.

7.1.1 Experiment Setup

We ran experiments on a cluster testbed and PlanetLab to evaluate our load balancing algorithm under real-world networking conditions and compare the results with those obtained previously from simulation. The cluster testbed we use consists of 22 identical machines with 4 GB memory, Intel Xeon 1.86 GHz dual core CPUs, and 1 Gbps network links. On PlanetLab, random nodes around the world are selected. Different experiments on PlanetLab may use a different set of nodes because nodes unpredictably become either extremely slow or unreachable. On both real platforms, each broker process runs on a separate machine, while all publisher and subscriber processes run on the same machine to accurately measure end-to-end publication delivery delay.

Deployments on both the cluster and PlanetLab are aided by the use of a tool developed by the MSRG called PANDA (Padres Automated Node Deployer and Administrator). This tool
allows us to specify the experiment setup within a text formatted topology file such as the
time and nodes to run brokers and clients, as well as any process specific runtime parameters
such as the publication rate and stock quote publications for the publisher, subscription for the
subscriber, neighbors for brokers, etc. The topology file is fed into PANDA which then deploys
the processes automatically.

In simulation experiments, all brokers and clients are instantiated within the same JVM.
Network connections between brokers and clients are implemented as direct function calls.
Hence, to get network performance measurements, bandwidth delays are simulated. Input
and output queuing delays are simulated on the broker-level as brokers have input and output
queues. Any other delays beneath the broker overlay, such as network latency, are not simulated
because they are not optimized by the load balancing algorithm and hence have no effect
other than being shown as extra overhead in the experimental results. Matching delays and
matching engine memory consumptions are modeled as linear functions because each of these
measurements cannot be isolated per broker instance within the Java Virtual Machine (JVM).

Deployments in the simulator do not use PANDA. Rather, the simulator itself allows one to
specify a set of brokers with specific resource capacities, and a set of publishers and subscribers
with their stockquote publications and subscriptions, respectively. Brokers can have custom
CPU speeds, memory sizes, and network bandwidths. CPU speed affects the matching delay,
and hence the input capacity of brokers. CPU speeds do not have any units as they are used
for relativity sake, although they can be interpreted as MHz. Memory sizes limit the amount of
memory used by the matching engine and all message queues of a broker. Network bandwidth
affects how fast messages can be sent. Brokers and overlay links are verified to be up and
running before clients are deployed.

In this work, we conducted experiments under three unique scenarios with both the random
and proposed offload algorithms:

1. Heterogeneous environment on PlanetLab

2. Heterogeneous environment through bandwidth throttling and matching delay scaling on
   a cluster testbed
3. Homogeneous environment on a cluster testbed

These conditions range from a heterogeneous environment with unstable resources to a purely homogeneous environment with minimal outside disturbance. We also draw a comparison to show how our real-world findings are inline with our previous simulation results from [21].

As shown in Table 7.1, default values for the load balancing parameters are used unless otherwise specified.

Publishers on creation are assigned to publish stock quote publications of a particular company at a defined rate. Publishers can be configured to change publication rates at any point in time in the experiment. Stock quote publications are real-world values obtained from Yahoo! Finance containing a stock’s daily closing prices\(^1\). A typical publication looks like this:

```
[class, 'STOCK'], [symbol, 'YHOO'], [open, 18.37],
   [high, 18.6], [low, 18.37], [close, 18.37],
   [volume, 6200], [date, '5-Sep-96'],
   [openClose%Diff, 0.0], [highLow%Diff, 0.014],
   [closeEqualsLow, 'true'], [closeEqualsHigh, 'false']
```

Subscribers are assigned to a fixed subscription based on one of the templates with the probabilities shown below. `SUB_SYMBOL` is randomly chosen out of the known stock symbols, with `SUB_HIGH`, `SUB_LOW`, and `SUB_VOLUME` replaced by a randomly chosen value of the same attribute from the stock’s publication set.

\begin{itemize}
\item 20% [class, =, 'STOCK'], [symbol, =, 'SUB_SYMBOL'], [high, >, SUB_HIGH]
\item 20% [class, =, 'STOCK'], [symbol, =, 'SUB_SYMBOL'], [low, <, SUB_LOW]
\item 20% [class, =, 'STOCK'], [symbol, =, 'SUB_SYMBOL'], [volume, >, SUB_VOLUME]
\item 34% [class, =, 'STOCK'], [symbol, =, 'SUB_SYMBOL']
\item 5% [class, =, 'STOCK'], [volume, >, SUB_VOLUME]
\item 1% [class, =, 'STOCK']
\end{itemize}
<table>
<thead>
<tr>
<th>Parameter (reference)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local PIE publishing period (Section 4.1.3)</td>
<td>30 s</td>
</tr>
<tr>
<td>Global PIE publishing period (Section 4.1.3)</td>
<td>60 s</td>
</tr>
<tr>
<td>PIE ratio threshold (Section 4.1.3)</td>
<td>0.025</td>
</tr>
<tr>
<td>PIE delay threshold (Section 4.1.3)</td>
<td>0.025</td>
</tr>
<tr>
<td>Adaptive PIE history (Section 4.1.3)</td>
<td>10</td>
</tr>
<tr>
<td>Lower overload threshold (Section 4.1.3)</td>
<td>0.9</td>
</tr>
<tr>
<td>Higher overload threshold (Section 4.1.3)</td>
<td>0.95</td>
</tr>
<tr>
<td>Local ratio triggering threshold (Section 4.1.3)</td>
<td>0.1</td>
</tr>
<tr>
<td>Local delay triggering threshold (Section 4.1.3)</td>
<td>0.1</td>
</tr>
<tr>
<td>Global ratio triggering threshold (Section 4.1.3)</td>
<td>0.15</td>
</tr>
<tr>
<td>Global delay triggering threshold (Section 4.1.3)</td>
<td>0.15</td>
</tr>
<tr>
<td>Delay normalization factor (Section 4.1.3)</td>
<td>0.1 s</td>
</tr>
<tr>
<td>Stabilize duration (Section 4.1.3)</td>
<td>30 s</td>
</tr>
<tr>
<td>Stabilize percentage (Section 4.1.3)</td>
<td>0.05</td>
</tr>
<tr>
<td>Local detection minimum interval (Section 4.1.3)</td>
<td>20 s</td>
</tr>
<tr>
<td>Local detection maximum interval (Section 4.1.3)</td>
<td>40 s</td>
</tr>
<tr>
<td>Global detection minimum interval (Section 4.1.3)</td>
<td>50 s</td>
</tr>
<tr>
<td>Global detection maximum interval (Section 4.1.3)</td>
<td>70 s</td>
</tr>
<tr>
<td>Global load balancing request limit (Section 4.1.3)</td>
<td>3</td>
</tr>
<tr>
<td>Migration timeout (Section 4.1.4)</td>
<td>5 s</td>
</tr>
<tr>
<td>PRESS samples (Section 4.2.1)</td>
<td>50</td>
</tr>
<tr>
<td>PRESS timeout (Section 4.2.1)</td>
<td>30 s</td>
</tr>
<tr>
<td>Balance threshold (Section 4.3.1)</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Table 7.1: Default values of load balancing parameters used in experiments
Chapter 7. Evaluation

### Table 7.2: Broker specifications in local load balancing experiment

<table>
<thead>
<tr>
<th>Broker ID</th>
<th>CPU (MHz)</th>
<th>Memory (MB)</th>
<th>Bandwidth (Mbps)</th>
<th>Delay Factor</th>
<th>Memory (MB)</th>
<th>Bandwidth (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>2000</td>
<td>32</td>
<td>10</td>
<td>0</td>
<td>2000</td>
<td>100</td>
</tr>
<tr>
<td>B1</td>
<td>100</td>
<td>16</td>
<td>0.5</td>
<td>1×</td>
<td>256</td>
<td>12</td>
</tr>
<tr>
<td>B2</td>
<td>200</td>
<td>16</td>
<td>1</td>
<td>1.5×</td>
<td>256</td>
<td>9</td>
</tr>
<tr>
<td>B3</td>
<td>400</td>
<td>32</td>
<td>2</td>
<td>2.5×</td>
<td>256</td>
<td>6</td>
</tr>
<tr>
<td>B4</td>
<td>1000</td>
<td>32</td>
<td>5</td>
<td>10×</td>
<td>256</td>
<td>0.85</td>
</tr>
</tbody>
</table>

### Table 7.3: Client specifications in local load balancing experiment

<table>
<thead>
<tr>
<th>Setup</th>
<th>Number of Publishers</th>
<th>Avg Pub. Rate (msg/min)</th>
<th>Time of +50% Pub. Rate (s)</th>
<th>Number of Subscribers</th>
<th>0-Traffic Subscribers (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlanetLab</td>
<td>15</td>
<td>30</td>
<td>1000</td>
<td>120</td>
<td>50</td>
</tr>
<tr>
<td>Hetero. Cluster</td>
<td>60</td>
<td>30</td>
<td>1500</td>
<td>600</td>
<td>50</td>
</tr>
<tr>
<td>Homo. Cluster</td>
<td>60</td>
<td>30</td>
<td>1500</td>
<td>600</td>
<td>50</td>
</tr>
<tr>
<td>Simulation</td>
<td>40</td>
<td>30</td>
<td>3000</td>
<td>2000</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 7.2: Broker specifications in local load balancing experiment

Table 7.3: Client specifications in local load balancing experiment
7.1.2 Local Load Balancing Results

In these experiments, we show how load and subscribers are distributed among the edge brokers in local load balancing. We also show the impact of the load balancing algorithm on the end-to-end delivery delay and amount of message overhead introduced. Results on the scalability of the local load balancing algorithm and the accuracy of load estimations are also presented.

The setup used for the local load balancing experiment involves four edge brokers connected to one cluster-head broker to form a star topology as shown on the left of Figure 7.2. This setup mimics a cluster at one data center as found in practice. Specifications on both heterogeneous cluster and simulation setups for the brokers are summarized in Table 7.2 and for clients in Table 7.3. Because each testbed has different levels of computation power and I/O speeds, the same workload cannot be used on all three testbeds. CPU speeds in the simulation setup do not have any units as it is used for relativity sake, although it can be interpreted as MHz. On experiment startup, brokers $B_0$, $B_1$, $B_2$, $B_3$, and $B_4$ are instantiated simultaneously. After all broker processes and links are established, all publishers connect to broker $B_0$. Immediately following, all subscribers connect to broker $B_1$ at a time chosen randomly within the first 500 s on PlanetLab, 600 s on the cluster testbed, and 1010 s in the simulator using a uniform random distribution. The greatest number of publishers and subscribers are chosen to the best of our ability without overloading the entire system. The reasoning behind connecting all

\footnote{The complete data set is available at \url{http://www.msrg.org/datasets/acDataSet}}
subscribers at $B1$ is to try overloading this broker and observe how the load balancing algorithm prevents overload by dynamically distributing the subscribers to other edge brokers. For the experiments on churn, subscribers join the system at a rate of 60 per minute continuously. Each subscriber remains connected to the system for 7 to 14 minutes (chosen randomly) until which the subscriber unsubscribes and disconnects from the broker. We experimented with two joining patterns: (1) *grouped-random* where all subscribers within each minute join the same random edge broker, and (2) *individually-random* where each subscriber joins a random edge broker. Of all subscribers, a certain percentage of them sink zero traffic, which means their subscriptions do not match any publications in the system. Sometime into the experiment (see Table 7.3), 50% of the publishers are randomly chosen to have their publication rates increased by 100%. This shows the dynamic behavior of the load balancer under changing load conditions. For the experiment on edge broker scalability, all edge brokers are homogeneous except when running on PlanetLab.

**Broker Load Distribution**

Figure 7.3 shows the load distribution among the four edge brokers in local load balancing on the heterogeneous cluster, the homogeneous cluster, PlanetLab, and the simulator. Graphs for matching delay are not shown as their trend looks similar to the input utilization ratio graphs. Input utilization ratio graphs for the homogeneous cluster and PlanetLab are omitted because they closely resemble Figure 7.3c where their values never exceed the threshold to trigger load balancing, indicating that the output resource is the only bottleneck. Comparing the graphs for the three performance metrics on the heterogeneous cluster and simulation setups in Figures 7.3a-d:

- Even though all subscribers join broker $B1$, those subscribers get offloaded to other edge brokers once a sizeable load difference or overload occurs.

- All performance metrics converge until the 50% increase publication rate disturbance

- All performance metrics converge and stabilize within the configured thresholds
Figure 7.3: Load distribution in local load balancing on various platforms
Figure 7.4: Output load distribution in local load balancing using the naive algorithm

The graphs from the simulations have less noise as there is absolutely no outside activity disturbances such as operating system background processes, user processes, and Java’s garbage collector. The workload used in simulation have much more subscribers joining B1 than the workload in real experiments, which causes B1 to become temporarily overload within the first 1000 s in the simulation but not in the real experiment.

Figure 7.3e shows that the output utilization ratio does not converge on PlanetLab. This is due to the stability restriction of the output offload algorithm. Also worth noting is that load fluctuations are more pronounced on PlanetLab than on the cluster testbed. We believe this is due to the greater resource contention as a result of shared nodes among world wide users and added dynamics of the Internet links. Figure 7.3f shows that our proposed load balancing algorithm balances load effectively on a homogeneous platform as well.

In the presence of churn, the load balancing algorithm continues to operate in a stable manner while trying to keep the load among brokers balanced. Results for grouped-random joins are shown in Figures 7.3g and 7.3h, while the graphs for individually-random joins have the same trend but with load fluctuations of lesser magnitude. Though not shown, the naive approach performed similar to the proposed approach under churn on the homogeneous cluster testbed.

Figure 7.4a shows that the output load converges on the homogeneous setup when using the naive approach under the workload where all subscribers join broker B1. However, there are three significant differences with Figure 7.3f. First, a heavy concentration of sharp spikes in the first 2000 s indicate that more offload sessions each involving a greater number of subscrip-
tion swaps are needed to find a stable subscription distribution. This indicates that the naive algorithm is not effective in choosing subscriptions in balancing load, resulting in more offload sessions than is necessary, which in turn leads to inefficiency. Second, due to the unpredictable trial-and-error offload sessions, the naive algorithm takes 500 s longer to converge than the proposed approach. Third, intermittent small noise disturbances trigger unnecessary large offloads that result in large spikes in load. On both PlanetLab (Figure 7.4b) and the heterogeneous cluster setup (not shown), the naive approach actually causes the output utilization ratio on brokers $B_2$, $B_3$, and $B_4$ to diverge, creating system-wide overload.

In short, our results on the cluster testbed and PlanetLab show that local load balancing is effective on both homogeneous and heterogeneous environments. As well, a naive offload algorithm that has no subscription load and space awareness is unable to load balance effectively and efficiently in a content-based publish/subscribe system.
Client Perceived Delivery Delay

Figure 7.5 shows the average end-to-end delivery delay on all testbeds. On both the cluster and PlanetLab, the load balancing algorithm prevents overload from occurring at \( B1 \) by dynamically distributing incoming subscribers to other edge brokers. As a result, the delivery delay stays constant at around 0.2 s throughout the experiment rather than increase without bound as in an overload condition. Two large spikes in both graphs are due to a slight period of output overload at broker \( B4 \) in both cases.

In the presence of churn, the delivery delay remained constant and never exceeded 0.1 s. If the rate of subscriber joins far exceeds the offloading rate, as shown in Figure 7.5b, then the delivery delay grows until the broker is no longer overloaded and processes all messages in its queues. With the naive approach running on PlanetLab as shown in Figure 7.5d, not only does overload yield high delivery delays but also lead to zero message delivery as brokers behave erratically from insufficient memory at 5000 s and finally crash at 13000 s. In summary, the proposed load balancing algorithm is able to minimize disruptions to the delivery delay by preventing overload in the system.

Subscriber Distribution Among Brokers

Figures 7.6a and 7.6c show that the load balancing algorithm can account for heterogeneous brokers by assigning more subscribers to more powerful brokers. Figure 7.6d shows that the naive algorithm needs to offload a much larger number of subscribers and perform many trial and error offload sessions before converging to a stable state. This is because the algorithm has no sense of the traffic associated with each subscription. Even for slight load differences, the naive algorithm swaps as much as hundreds of subscribers whereas the proposed algorithm offloads less than ten even under unstable conditions such as on PlanetLab (Figure 7.6c).

Load Balancing Message Overhead

Figures 7.7a and 7.7b show that message overhead, which includes subscription messages and PIE messages, across all brokers is less than 50% during subscriber joins and less than 0.2% and 1.1% after load balancing has converged in simulation and real experiments, respectively.
Figure 7.6: Subscriber distribution in local load balancing on various setups

Figure 7.7: Message overhead in local load balancing
If we examine the overhead message rates with conventional and Adaptive PIE as is shown in Figures 7.7c and 7.7d, respectively, the highest source of message overhead is the migrating subscribers’ subscriptions. While load balancing, there are 0.4 msg/s of load balancing coordination messages, which consist of load balancing request and reply and subscriber migration coordination messages, and 0.85 msg/s of PIE messages. With Adaptive PIE however, there are 40% less PIE messages. Once the system stabilizes with no load balancing activity, there are no load balancing coordination messages and Adaptive PIE reduces the amount of PIE messages by 65% compared to conventional PIE. In the churn experiments where the load of brokers constantly fluctuates, Adaptive PIE reduces the amount of PIE overhead messages, including PIE advertisements, subscriptions, and unsubscriptions, by up to 30%. In summary, the added cost of running the load balancing algorithm is almost negligible and does not hinder the scalability of the existing publish/subscribe system.
Chapter 7. Evaluation

Figure 7.9: Load estimation accuracy on heterogeneous cluster testbed and simulation

Edge Broker Scalability

To measure how the publish/subscribe system scales with the number of brokers in a cluster, we focus on the load reduced on each edge broker as well as any gains in delivery delay. Figures 7.8a and 7.8b indicate that the output utilization ratio, which is the limiting resource on both cluster and Planetlab setups, is reduced as the number of edge brokers increase. This indicates that the proposed load balancing algorithm indeed allows the system to scale with the number of brokers. However, gains in delivery delay are insignificant past adding five edge brokers as witnessed on the cluster testbed and simulation. On PlanetLab, where Internet link delays dominate over broker processing delay, there is no observable gain in delivery delay.

Load Estimation Accuracy

Load estimation accuracy determines the effectiveness of each local load balancing session. Figure 7.9 shows the estimation accuracy for all performance metrics on the heterogeneous cluster setup and the input utilization ratio obtained from simulation. Graphs for the matching
delay and output utilization ratio estimation in the simulation are not shown here because they have the same trends and exception cases as the input utilization ratio. Dots on the $y = x$ line denote 100% accuracy. In the experiment on the cluster testbed, a majority of the data points lie close to the $y = x$ line as shown in Figures 7.9a-c. Specifically, the mean average percentage error of both input and output utilization ratios is 17%, and matching delay is 13%. Compared to the simulation, the input and output utilizations are 8% and 3%, respectively, while there is no error in matching delay estimation as matching delays are simulated by a predefined linear function. The amount of error represented in the simulation results indicates the prediction error of future traffic based on the present data. We believe the additional error in the real experiments are due to external disturbances such as CPU and network contention on the shared cluster nodes, interruptions in processing from Java’s garbage collector, estimation of number of bytes transmitted over RMI by taking the length of serialized message objects, and modeling matching delay with a linear function. With 90% of load estimations having a mean average percentage error of less than 11% on PlanetLab, this means that at least 90% of local load balancing sessions are effective, that is, they do not need a followup session to further balance load. Followup load balancing sessions slow down the overall response time of the algorithm and increase overhead compared to a single effective load balancing session.

7.1.3 Global Load Balancing Results

In these experiments, we will show how global load balancing sessions are coordinated and how load is transferred from one cluster to another in global load balancing. We also show the scalability of global load balancing by scaling up the number of clusters in the system.

The setup used for the global load balancing experiment involves 12 brokers organized into 4 clusters, with 2 edge brokers per cluster as shown on the right of Figure 7.2. This setup mimics multiple data centers (clusters) connected together to form the complete publish/subscribe system [29, 72]. Specifications on both heterogeneous cluster and simulation setups regarding the brokers are summarized in Table 7.4. CPU speeds in the simulation setup do not have any units as it is used for relativity sake, although it can be interpreted as MHz. Client configurations are the same as in the local load balancing setup as shown in Table 7.3 except
### Table 7.4: Broker specifications in global load balancing experiment

<table>
<thead>
<tr>
<th>Broker ID</th>
<th>Simulation</th>
<th>Heterogeneous Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU (MHz)</td>
<td>Memory (MB)</td>
</tr>
<tr>
<td>B11</td>
<td>300</td>
<td>128</td>
</tr>
<tr>
<td>B12</td>
<td>200</td>
<td>128</td>
</tr>
<tr>
<td>B21</td>
<td>350</td>
<td>64</td>
</tr>
<tr>
<td>B22</td>
<td>400</td>
<td>64</td>
</tr>
<tr>
<td>B31</td>
<td>266</td>
<td>64</td>
</tr>
<tr>
<td>B32</td>
<td>700</td>
<td>64</td>
</tr>
<tr>
<td>B41</td>
<td>166</td>
<td>64</td>
</tr>
<tr>
<td>B42</td>
<td>233</td>
<td>128</td>
</tr>
</tbody>
</table>

that half the publishers in the simulation increase their publication rate by 100% at 8000 s. On experiment startup, all brokers are instantiated simultaneously. After all broker processes and links are established, all publishers connect to broker $B_{10}$. Immediately following, all subscribers connect to broker $B_{11}$ at a time chosen randomly within the first 500 s on PlanetLab, 600 s on the cluster testbed, and 1010 s in the simulation using a uniform random distribution. The experiment on churn uses the same join rates, durations, and patterns as described in Section 7.1.2. The greatest number of publishers and subscribers are chosen to the best of our ability without overloading the entire system. For the experiment on the cluster scalability, there are two edge brokers in each cluster, with all edge brokers having equal resource capacity except on PlanetLab.

**Cluster Load Distribution**

The average load of each cluster engaged in global load balancing is shown in Figure 7.10. Figures 7.10a and 7.10c show the results of running on the heterogeneous cluster testbed setup, while 7.10b and 7.10d show the results of running on the simulator. On PlanetLab, only the output utilization ratio graph is shown (Figure 7.10e) because only the output resource is constrained enough to trigger load balancing. Whenever a cluster performs global load balancing with another cluster, the two clusters’ loads appear merged on the graph. Compared
Figure 7.10: Load distribution among clusters in global load balancing
to local load balancing, global load balancing takes longer to converge because the clusters are arranged in a chain topology in this experiment, which limits parallelizing load balancing sessions. For example, when cluster $B2x$ load balances with $B3x$, cluster $B1x$ stays idle. Global load balancing converges 2000 s after the load balancing algorithm starts distributing load to the last cluster ($B4x$) on all setups. When global load balancing converges, clusters further away from cluster $B1x$ carries a smaller amount of load. This is desired so that subscribers do not get relocated too far into the network away from the publisher. Figure 7.10e shows that the load balancing algorithm works even in the presence of system overload until load is evenly distributed among all clusters to stop overload. Under churn conditions, both the proposed (Figure 7.10f) and naive approaches continually perform load balancing without diverging. In summary, global load balancing is able to effectively distribute load originating from one broker to others brokers in surrounding clusters.
Chapter 7. Evaluation

Cluster Scalability

When clusters are organized in a chain-like topology, there is a load diminishing effect on clusters further away from the source of load, namely cluster $B1x$ in our experiment. This is primarily due to the global detection threshold setting where clusters only engage in load balancing if one of their performance metrics exceeds a certain value. On both the cluster testbed (Figure 7.11a) and in simulations (Figure 7.11b), clusters more than 3 cluster-hops away from cluster $B1x$ no longer reduce the overall delivery delay. Instead, the delivery delay gradually increases as the publications travel over more broker hops to reach the subscribers. This indicates a tradeoff between load distribution and client locality. Perfect load distribution onto all nodes will result in a more stable system, but because the clients are scattered throughout the entire network, end-to-end delivery delay is sacrificed. However, this trend is not observable on PlanetLab (Figure 7.11c) because the magnitude of delivery delay is dominated by Internet link delays, which heavily depend on the distance between the brokers and subscribers in the world. Yet, on both real life platforms, the overall system output utilization ratio decreases with additional clusters.

7.2 Evaluation of the Publisher Placement Algorithms

We evaluated POP and GRAPE with real-world data sets on PlanetLab and a cluster testbed while varying the subscriber distributions, GRAPE’s minimization metric (which is either the end-to-end delivery delay or system load), GRAPE’s minimization weight, and number of samples for triggering broker selection. For graphs without time as the x-axis, the plotted values are obtained at the end of the experiment, which is 18 minutes after all clients are deployed. All graphs use average values across all brokers or clients in the system. For concise presentation, we include a subset of the graphs here with the full set in an online Appendix [23]. Nevertheless, we summarize all of our results here in this section of the thesis. From this point on, we will use the notation load 75% (delay 25%) to denote GRAPE’s configuration to prioritize on minimizing average system load (delivery delay) with 75% (25%) weight.
7.2.1 Experiment Setup

We ran experiments on PlanetLab to show how POP and GRAPE behave under real-world networking conditions and on a cluster testbed to validate our PlanetLab results under controlled networking conditions. The cluster testbed and deployment tools used are the same as those in evaluating the load balancing algorithm. Our evaluation uses two different subscriber traffic distributions. One is random where 70% of the subscribers are low-rated, meaning they sink about 10% of their publishers’ traffic; 25% are medium-rated, meaning they sink about 50% of their publishers’ traffic; and 5% are high-rated, meaning they sink all of their publishers’ traffic. Subscribers of each category are randomly assigned to $X$ number of brokers, where $X$ is varied in our experiments. The other distribution is referred to as enterprise, where 95% of the subscribers are low-rated and 5% are high-rated. All high-rated subscribers connect to one broker, while low-rated subscribers are randomly assigned to the other $X-1$ brokers. The random workload represents a generic scenario, while the enterprise workload mimics an enterprise deployment consisting of a database sinking all traffic at the main office and many end-users subscribing to selected traffic at different branch office locations. An example of low, medium, and high-rated subscriptions to YHOO stockquotes are shown below:

low - [class,=,'STOCK'],[symbol,=,'YHOO'],
      [highLow%Diff,>,0.15]
medium - [class,=,'STOCK'],[symbol,=,'YHOO'],
         [volume,>,1000000],[openClose%Diff,>,0.025]
high  - [class,=,'STOCK'],[symbol,=,'YHOO']

The overlay topology we used for all evaluations is a balanced tree with a fan-out of 2 or 4. The number of publishers, subscribers, brokers and all other settings we used on PlanetLab and the cluster testbed are shown in Figure 7.12. Unless otherwise stated, default POP and GRAPE settings are used. A publisher and its set of matching subscribers run on the same machine to accurately measure end-to-end publication delivery delay. Different publishers run on randomly chosen machines that also run broker processes. Each publisher publishes stock quote publications of a particular stock that are real-world values obtained from Yahoo! Finance containing a stock’s daily closing prices.
### 7.2.2 Evaluation Results with the Enterprise Workload

Under the enterprise scenario, Figures 7.13a and 7.13d show that GRAPE’s load 100% yields the lowest average broker message rate compared to GRAPE’s delay 100%, POP, and static (with neither GRAPE nor POP enabled). This is true on both testbeds and across all subscriber distributions except when all subscribers to each publisher are concentrated at one broker. In which case, both POP and GRAPE make the same relocation decision to migrate the publishers to the broker where all the matching subscribers reside. Lower average broker message rate translate directly to lower average broker input (Figures 7.13b and 7.13e) and output utilizations (Appendix [23]). However, because GRAPE’s load 100% setting moves the publishers closest to the subscribers that subscribe to all of the publishers’ traffic, the publishers are farther away from the majority of subscribers who sink a subset of the traffic. As a result, Figures 7.13c, 7.13f, and 7.13g show that average delivery delay and hop count are sacrificed, especially in the experiment on the cluster testbed where the delivery delay of load 100% is higher than static.

We believe the latter behavior is due to the increased number of subscribers and maximum hop count in the cluster testbed setup over the PlanetLab setup. Figures 7.14a, 7.14b, and the Appendix [23] show that the performance trends of GRAPE and POP still hold as we increase the fanout of the network from 2 to 4 while keeping the number of brokers constant. However, since the maximum length of the network is now decreased, publishers start off closer to the subscribers before their relocation. Hence, the broker selection time, publisher wait time, and reductions in delivery delay and broker message rate of GRAPE and POP decrease as fanout.

<table>
<thead>
<tr>
<th>Setting</th>
<th>PlanetLab</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brokers</td>
<td>63</td>
<td>127</td>
</tr>
<tr>
<td>Publishers</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>_msgs per min per Publisher</td>
<td>10-40</td>
<td>30-300</td>
</tr>
<tr>
<td>Subscribers per Publisher</td>
<td>80</td>
<td>200</td>
</tr>
<tr>
<td>Total Subscribers</td>
<td>1600</td>
<td>6000</td>
</tr>
<tr>
<td>P\text{threshold}</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>G\text{threshold}</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 7.12: Deployment specs
Figure 7.13: Experiment results on PlanetLab and the cluster testbed
Figure 7.14: Experiment results on the cluster testbed with overlay fanout of 4 increases.

With GRAPE’s delay 100% setting, the system achieves lowest average delivery delay across almost all subscriber distributions as shown in Figures 7.13c and 7.13f. This is due to placing the publishers closest to the subscribers with the highest number of publication deliveries, which dominates what is shown in Figure 7.13g. However, because the publishers are further away from the high-rated subscribers, the system has to transmit more messages overall (Figures 7.13a and 7.13d) which leads to higher input (Figures 7.13b and 7.13e) and output utilizations (Appendix [23]). On platforms where latency and processing delay are minimal already such as the cluster testbed, Figure 7.13f shows that there is not much more that POP and GRAPE can reduce.

Figure 7.13c also demonstrate another very interesting property. The delivery delay is high when all subscribers are placed at only one broker due to high simultaneous contention for network I/O. The delivery delay is even higher when subscribers are placed at only two brokers because the subscribers not at the broker where the publisher connects with experience high delivery delay while contention is still a major factor. As subscribers become more spread out in the network, the delivery delay becomes much lower because bandwidth contention is significantly reduced. However, the delivery delay rises slightly again when subscribers are located at every broker in the network because a majority of the subscribers are far away from the publisher.

POP minimizes both average system load and delivery delay simultaneously without offering a choice of which metric to prioritize and by how much. According to Figures 7.13b, 7.13c,
7.13e, 7.13f, and 7.13g, POP's average input utilization, delivery delay, and hop count is between GRAPE's load 100% and delay 100%. However, the average broker message rate of POP as shown in Figures 7.13a and 7.13d exceeds that of GRAPE due to the increased message overhead from TRM's (Figure 7.13h).

As expected, the advantages of both POP and GRAPE come at a cost, as Figure 7.13h and the Appendix [23] reveal that both approaches introduce additional message and memory overhead, respectively. GRAPE introduces far less control messages into the system than POP because, in GRAPE, a reply-like message is only generated after each trace session, whereas in POP, a reply-like message is generated after each traced publication message. The result is that maximum message overhead for POP is 32% and GRAPE is 16%. Note that the results in Figure 7.13h are specific to the publishers' publication rates in the experiment and trigger settings of each algorithm. The latter parameter is further analyzed below. In terms of memory overhead, both GRAPE settings use up to an additional 58% (or 31 MB) of memory compared to static. POP uses at most 34% (or 19 MB) more memory, thanks to the running average function used in PPTables to aggregate values from multiple traces.

Figure 7.15a shows the average time to obtain pings to 63 brokers for GRAPE with delay 1% setting. This setting is the slowest since it requires the publisher to obtain ping times to all relevant downstream brokers, whereas load 100% is the fastest setting since it requires no pings to be sent. The ping time should be part of GRAPE's broker selection time as shown in Figure 7.15b, but we have separated them into two graphs for a clearer analysis. Given the large network performance fluctuations on PlanetLab, each publisher takes the average of five pings to each relevant downstream broker with all pings done in serial to obtain the most accurate measurement. Using this approach, pinging 63 brokers required around 30 s on PlanetLab and 7 s on the cluster testbed. For more stable networks such as the cluster testbed, it is possible to ping each broker once with multiple ping threads to bring down the waiting time to less than 0.4 s without any performance degradation as shown in the Appendix [23]. Examining the average broker selection times without the ping times on Figure 7.15b, it takes GRAPE around 5 s on PlanetLab and 1 s on the cluster to fetch data from all 63 downstream brokers and perform the localized computation. This is higher than POP on both testbeds because GRAPE
Figure 7.15: Experiment results on PlanetLab (PL) and the cluster testbed (CL)
has to obtain delivery statistics from all downstream brokers. Thus, the subscriber distribution affects GRAPE but not POP. Conversely, the length of the migration path has negligible impact on GRAPE's selection time but has a linear effect on POP's selection time.

Figure 7.15c illustrates that the average time a publisher waits while migrating to the target broker is directly proportional to two variables: (1) the number of brokers on the migration path and (2) the distribution of subscribers. This observation makes sense because the longer is the migration path, the higher is the number of brokers that need updating. Likewise, the greater is the number of brokers with matching subscribers, the more routing table update operations are needed.

### 7.2.3 Evaluation Results with the Random Workload

With high-rated subscribers randomly assigned to different brokers, there are virtually no visible differences between the average broker message rate, input and output utilizations, delivery delay, hop count, message overhead, and memory utilization of GRAPE’s load 100% and delay 100%. Only the average delivery delay graphs for the cluster testbed and PlanetLab are included here in Figures 7.15d and 7.15e, respectively, with the rest in the Appendix [23]. GRAPE’s average broker message rate and input utilization matches that of static, with the output utilization of GRAPE surpassing the static case. Message overhead of GRAPE is at most 5%, and is 91% lower than POP. Summarizing POP’s results, POP achieves the same average hop count, and input and output utilizations as GRAPE, but introduces approximately 50% higher broker message rate than GRAPE due to the message overhead from TRMs. Figure 7.15d also illustrates that POP’s metric for broker selection is more effective than GRAPE’s delay 100% for minimizing delay under the random workload. On PlanetLab, we experienced overloads at about 1-3 internal brokers in our static, POP, and GRAPE’s delay 100% experiments. During overloads, subscriber clients experience orders of magnitude higher delivery delays on publication messages as demonstrated by the spikes in Figure 7.15e. However, no overloads ever happened to GRAPE’s load 100% experiments. This shows that GRAPE’s load 100% setting is effective in preventing the chances of overloading brokers by minimizing broker load in an unstable environment.
7.2.4 Impact of GRAPE’s Minimization Weight

We ran experiments with GRAPE set to prioritize minimization of both load and delay from 1% to 100% at increments of 25%. Similar to our previous observations with the random workload, GRAPE behaves indifferently regardless of the prioritization metric and minimization weight settings. However, the reverse is true with the enterprise workload. Interpreting from the graphs in our Appendix [23] and Figure 7.15f, as the weight on load minimization decreases from 100% to 1%, the average delivery delay and hop count decreases, putting more emphasis on minimizing delay. All the while, load minimization is sacrificed with increased average broker message rate, and input and output utilizations. Moreover, the results of load 1% are virtually identical to delay 100%. Similarly, as GRAPE is varied from delay 100% to delay 1%, we observe an increase in average delivery delay and hop count, and decrease in system load and average broker message rate. Likewise, delay 1% is virtually equivalent to load 100%. No notable differences in memory usage and message overhead are observed over different minimization weights.
Figure 7.17: Micro experiments of POP on the cluster testbed

7.2.5 Impact of POP and GRAPE’s Sampling Trigger

Before broker selection can occur in POP, or delivery statistics are retrieved in GRAPE, a required number of publications must be traced. Recall, that this number is $P_{\text{threshold}}$ for POP and $G_{\text{threshold}}$ for GRAPE. We experimented with altering the thresholds using the enterprise scenario on the cluster testbed with subscribers distributed among 16 brokers. For POP, we set the maximum number of traceable publications per time window to be half of $P_{\text{threshold}}$. Our results from Figure 7.16 indicate that increasing $G_{\text{threshold}}$ will increase the algorithm’s response time and decrease the amount of message overhead. For example, at 12800 samples the overhead is lowest because GRAPE never performed a single relocation throughout the experiment. However, at 10 samples, not only is the message overhead high but also the average delivery delay and system loads are higher than if $G_{\text{threshold}}$ is set at 400 samples. Therefore, GRAPE performs best when $G_{\text{threshold}}$ is set within the hundreds range for best tradeoff between message overhead, response time, and performance gains.

On the contrary, POP’s threshold setting behaves different from GRAPE as increasing
$P_{\text{threshold}}$ will increase both the algorithm’s response time and amount of message overhead according to Figure 7.17. Our results indicate that POP achieves best performance gains with fast response time and minimal overhead when $P_{\text{threshold}}$ is set below 100.

### 7.3 Evaluation of the Resource Allocation Algorithms

We ran experiments on a local cluster and SciNet, a high performance computing cluster, with homogeneous and heterogeneous setups to mimic data center environments. Our results examine the effectiveness and efficiency of each resource allocation algorithm.

#### 7.3.1 Comparison Against Baseline and Related Approaches

The main objective of our evaluation is to examine the performance gains and tradeoffs of our following algorithms under homogeneous and heterogeneous environments: (1) FBF, BIN PACKING, and CRAM subscription allocation algorithms, (2) INTERSECT, IOS, and IOU closeness metrics in our CRAM algorithm, (3) recursive broker overlay construction algorithm, and (4) effectiveness of the bit vector based resource allocation framework. We also compare our work with the following two baseline and three related approaches: (1) two baseline subscription allocation and broker overlay construction approaches called MANUAL and AUTOMATIC (described below), (2) two derivatives of the pairwise clustering approach proposed in [73] which we refer here as PAIRWISE-K and PAIRWISE-N (described below), and (3) the Gryphon-derived closeness metric, XOR [9], in our CRAM algorithm.

One of our baseline approaches is called MANUAL, which forms the initial overlay topology that we use for all evaluations. In this approach, the broker overlay has a fan-out of 2 to minimize the chance of overloading internal brokers in the tree, and publishers are randomly placed on the overlay. Under the homogeneous scenario, subscribers too are randomly placed on the overlay. However, under the heterogeneous scenario, the most resourceful brokers are manually placed at the top of the tree and the number of subscribers allocated to each broker is proportional to the brokers’ resource levels. Our other baseline algorithm is called AUTOMATIC which positions the clients and builds the broker overlay randomly. We chose to evaluate these two baseline
approaches because they are representative of typical publish/subscribe deployments where the measure of a "good" topology is not easily quantifiable [52, 7, 67, 14, 9, 22, 78].

Moreover, we compare our work against two derivatives of the pairwise cluster algorithm both using the XOR closeness metric from [73]. The pairwise algorithms are derived because of two reasons. One, they originally do not allocate subscriptions to brokers nor build the broker overlay. Thus, we extend both pairwise algorithms to build the broker overlay using the AUTOMATIC approach. Two, we extend the pairwise algorithms to use bit vectors instead of the subscription language due to limited development time. Doing so actually benefits the related approaches because the workload that we use are stockquotes, which do not follow any well-defined distribution pattern. Thus, the use of bit vectors actually enable the pairwise algorithms to make better clustering decisions.

One of the derivatives is called PAIRWISE-K where we set the number of clusters to that computed by CRAM with the XOR closeness metric and then randomly assign subscription clusters to brokers. The reason for choosing the XOR closeness metric is because that is the metric used in [73]. The other derivative is called PAIRWISE-N where we set the number of clusters to the number of brokers in the system and assign each subscription cluster to a broker.

Our evaluation first shows the results of our macro experiments which compares the performance among the best of the baseline approaches (MANUAL), related approaches (PAIRWISE-N), sorting algorithms (BIN PACKING), and closeness metrics (IOU) with the CRAM algorithm. Then, we analyze the gains by applying publisher relocation and subscriber relocation with overlay reconfiguration separately. Following that, we examine the results of our micro experiments comparing the two baseline approaches, two related approaches, two sorting algorithms, and four closeness metrics in the CRAM algorithm. Finally, we study the performance implications on varying the length of the bit vector. Our evaluation focuses on the following metrics: number of allocated brokers, resource utilizations, broker message rates, publication delivery delays and hop counts, and computation time of the algorithm.

The plotted values are an average over ten 1-minute intervals of the system at its converged state. All graphs use average values across all clients and only allocated brokers in the system. Most of our results are shown in this thesis, with the full set available in an online Appendix.
Nevertheless, we summarize all of our results here in this section of the thesis. Algorithms labeled in the graphs with an asterisk denote that those algorithms lead to overloaded brokers and anomalies for them are expected due to erratic behavior of the JVM running out of memory.

### 7.3.2 Experiment Setup

The CBC is implemented into the **PADRES** broker as an additional internal component with 1,600 lines of Java code. **CROC** is implemented as a separate publish/subscribe client with 4,200 lines of Java code. Regarding the actual reconfiguration at the end of Phase 3, we re-instantiate every broker in the system and have the original clients connect to the new broker instances. We chose this method over reusing the original broker instances because this is the simplest and surest way to start brokers from a clean state.

We evaluated all baseline, related, and proposed approaches on two testbeds which mimic enterprise-scale data center environments [72]. One is a cluster testbed consisting of 21 nodes each with two Intel Xeon 1.86 GHz dual core CPUs connected by 1 Gbps network links. The other is SciNet, a high performance computing cluster consisting of 3,780 nodes each with two Intel Xeon 2.53 Ghz quad core CPUs connected by 1 Gbps network links. A publisher and its set of matching subscribers run on the same machine to accurately measure end-to-end publication delivery delay. Different publishers run on randomly chosen machines that also run broker processes. Each publisher publishes stock quote publications of a particular stock that are real-world values obtained from Yahoo! Finance containing a stock’s daily closing prices. We chose stockquotes as the workload not only because it is real world data but it also highlights how our solution can cope with workloads of arbitrary distribution.

We evaluated all approaches under both homogeneous and heterogeneous scenarios on the cluster testbed. Both scenarios consist of 80 brokers with 40 publishers each publishing a unique stockquote at 70 msg/min. Brokers in the homogeneous scenario all have equal processing and bandwidth capacities. As well, each publisher has an equal number of subscriptions. To see the impact of the number of subscriptions, we vary this number from 50 to 200 per publisher in increments of 50 on the cluster testbed. Thus, the total number of subscriptions ranges from 2,000 to 8,000 in increments of 2,000. Furthermore, we conducted large-scale deployments
with 400 and 1,000 brokers with 225 subscribers per publisher on SciNet under a homogeneous setting. The total number of publishers are set to 72 and 100 for networks of sizes 400 and 1,000, respectively, to initially saturate the system which is represented by the baseline MANUAL case.

In the heterogeneous scenario, 15 brokers are configured to have 100\% of the network capacity of the brokers in the homogeneous scenario, 25 brokers are configured to have 50\% of the network capacity, and 40 brokers are configured to have 25\% of the network capacity. We achieve bandwidth throttling through the use of a bandwidth limiter in each broker. The number of subscriptions for the $i_{th}$ publisher is $N_s \div i$ with $N_s$ ranging from 50 to 200 in increments of 50 per experiment. For example, in an experiment with $N_s$ set to 200, the total number of subscriptions is 4,100, and the lowest and highest number of subscribers for a publisher are 5 and 200, respectively.

Using the YHOO stock as an example to describe our subscription workload, 40\% of the subscriptions subscribe to the template $[\text{class,}=,\text{STOCK}], [\text{symbol,}=,\text{YHOO}]$, while the other 60\% also subscribe to that same subscription but with an additional inequality attribute, such as $[\text{class,}=,\text{STOCK}], [\text{symbol,}=,\text{YHOO}], [\text{low,<},\text{REPLACE_LOW}]$. The value for that additional attribute is chosen randomly from the same field in that stock’s publication set.

Since one of the main objectives of this work is to reduce the overall broker message rate, we configure GRAPE to focus 100\% on minimizing the system load rather than the delivery delay. GRAPE’s publication sample count was set to 1,000 to allow us enough time to capture the stabilized system state in between publisher relocations.

### 7.3.3 Comparison of Baseline, Related, and Proposed Approaches

Figure 7.18a shows the number of brokers allocated over a range of subscription workloads on the cluster testbed under the heterogeneous scenario. Figure 7.18b shows the number of brokers allocated on the cluster (8000 subscriptions) and SciNet (beyond 16200 subscriptions) testbeds under the homogeneous scenario. In these graphs, lesser brokers allocated denote higher resource usage efficiency. Algorithms aware of broker capacities including BIN PACKING, FBF, and CRAM allocate less than half the total set of brokers. The number of brokers allocated
also scales with the number of subscriptions in the workload. Both the baseline and related approaches always allocate all of the brokers because they lack awareness of broker capacities. Yet, both AUTOMATIC and PAIRWISE-K cannot avoid overloading at most one broker in the system. This emphasizes the need for resource awareness in allocation algorithms. CRAM-IOU consistently allocates the fewest number of brokers, with a reduction of up to 91% and 80% in the homogeneous and heterogeneous scenarios, respectively. Compared to BIN PACKING, CRAM-IOU allocates up to 47% less brokers than BIN PACKING. Yet, BIN PACKING always allocates one less broker than FBF which is inline with theoretical expectations [28]. A similar trend is observed in the homogeneous scenario.

Figure 7.18c shows the average, standard deviation, and maximum broker message rates under the heterogeneous scenario. Figure 7.18d shows the same metrics but on the cluster (8000 subscriptions) and SciNet (beyond 16200 subscriptions) testbeds under the homogeneous scenario. The lower is the broker message rate, the more efficient are resources being used. Compared to MANUAL, PAIRWISE-N shows that clustering subscriptions significantly reduces
both the average and maximum broker message rates while still allocating all 80 brokers. However, \texttt{PAIRWISE-N} suffers from broker overloads in the heterogeneous scenario because it has no resource awareness. \texttt{BIN PACKING}, a simple sorting algorithm with resource awareness, allocates less than 88\% of the brokers in the network. However, without clustering, brokers sink almost all of the publications sent by all publishers in the system, which is represented by the horizontal dotted line at 46.7 msg/s in 7.18c. With subscription clustering and broker capacity awareness, \texttt{CRAM-IOU} combines the benefits of low message rates from \texttt{PAIRWISE-N} and low number of allocated brokers from \texttt{BIN PACKING} without overloading any brokers. Figure 7.18d shows that \texttt{CRAM-IOU} reduces the average and maximum broker message rates by up to 92\% and 85\%, respectively. As well, \texttt{PAIRWISE-N} did not experience any overloads in the homogeneous scenario.

Figure 7.19a shows the average, standard deviation, and maximum broker input utilization ratios. Input utilization ratio captures the brokers’ matching rate versus the flow of incoming traffic\(^1\). Because the input utilization ratio varies directly with the broker message rate, the trend in Figure 7.18c is carried over to this graph as well. \texttt{BIN PACKING}, with highest broker message rate, imposes highest average and maximum input utilization on the brokers. \texttt{CRAM-IOU}’s input utilization is higher than \texttt{PAIRWISE-N}’s because \texttt{CRAM-IOU} allocates significantly less brokers and every allocated broker in \texttt{CRAM-IOU} is fully utilized in terms of output bandwidth, whereas the brokers in \texttt{PAIRWISE-N} are not.

Figure 7.19b shows the average, standard deviation, and maximum broker output utilization ratios. Output utilization ratio captures the output bandwidth usage of a broker\(^1\). Since the broker’s output resource is the bottleneck, the goal is to saturate the output bandwidth as much as possible to achieve maximum resource usage efficiency without overloading the broker. Algorithms that are aware of broker capacities, namely \texttt{BIN PACKING} and \texttt{CRAM-IOU} in this graph, are effective at using up on average 80\% of the output resource regardless of the workload and scenario. Also notice that all allocated brokers in \texttt{BIN PACKING} and \texttt{CRAM} are much better utilized as demonstrated by the small standard deviation and difference between the maximum and average values as compared to the baseline and related approaches. The output utilization

\(^1\)Please see [24] for definitions of these metrics
ratio of \textsc{Pairwise-N} is lower than \textsc{Manual} because of the significant reduction in broker message rates as shown previously in Figure 7.18c. Results from the heterogeneous scenario follow a similar trend except that \textsc{Pairwise-N} suffers from output overload as we will show later in Figure 7.22b.

Given that both \textsc{Cram-IoU} and \textsc{Bin Packing} allocate fewer brokers and dynamically construct the broker overlay, the network size is expected to be much smaller. This fact is supported by Figure 7.19c which shows the average publication hop count. Lower hop count is favorable because the publication delivery path is shorter, which reduces the amount of resources used in processing publications. In this graph, \textsc{Manual} is expected to be the highest because the maximum length of the network is 11 with 80 brokers and a fanout of 2. Also using 80 brokers but with a higher fanout, \textsc{Automatic} achieves lower hop count than \textsc{Manual}. \textsc{Bin Packing} yields even lower hop count thanks to significantly smaller network size made up of much less brokers. \textsc{Cram-IoU}'s hop count is lowest of all approaches thanks to both subscription clustering and small network size. A similar trend is observed in the heterogeneous scenario where \textsc{Cram} reduces the hop count to 0 across all workloads.

Contrary to common belief, reduction to hop count does not always translate to lower delivery delay as shown in Figure 7.19d. In a real system, the delivery delay also depends on the contention for the broker's output bandwidth. Which means, the more spread out are the subscribers of common interests, the lower is the bandwidth contention at the broker during publication delivery. This observation is supported by the delivery delay shown for the \textsc{Manual} case under both scenarios. Using the same number of brokers as \textsc{Manual} but with subscription clustering, publications matching all \textit{N} subscriptions have to be queued up at one broker to be delivered to the \textit{N} subscribers, which increases the delivery delay for \textsc{Pairwise-N} by as much as 56\% compared to \textsc{Manual}. This also explains why the delivery delay for \textsc{Bin Packing} is higher than \textsc{Fbf} because all heavy traffic subscriptions in \textsc{Bin Packing} are allocated to the same brokers. Still, the delivery delay of \textsc{Cram-IoU} is below that of \textsc{Bin Packing}, which we believe is due to the significant reduction in broker message rate as shown previously in Figure 7.18c. The delivery delays for \textsc{Automatic} and \textsc{Pairwise-K} are well beyond the range of this graph due to overloaded brokers. In the heterogeneous scenario, the delivery delays among all
Figure 7.19: Experiment results
Figure 7.20: Experiments showing the breakdown of performance gains

of the approaches are smaller because of fewer number of subscriptions per publisher, which reduces the bandwidth contention. However, PAIRWISE-N suffers from high delivery delays in the heterogeneous scenario due to overloaded brokers.

Figure 7.19e shows the time required to compute the subscription allocation and broker overlay, where lower is obviously better. The computation time for AUTOMATIC and the sorting algorithms is the shortest and grows linear with the number of subscriptions. CRAM-IOU takes the longest to run because it has to invoke the BIN PACKING algorithm after clustering every candidate GIF pair. The pairwise algorithms take approximately 15% and 25% less time than CRAM-IOU in the homogeneous and heterogeneous scenarios, respectively, mainly because these algorithms do not recursively cluster brokers nor invoke the BIN PACKING algorithm after clustering each GIF pair. Both CRAM and the pairwise algorithms’ computation times are shown to vary directly with the squared of the number of subscriptions. Later in this paper, our micro experiments show that the computation for CRAM-IOU depends linearly on the length of the bit vector.

7.3.4 Breakdown Analysis of Performance Gains

Figure 7.20a shows the broker message rates of MANUAL and CRAM with and without publisher relocation. Starting off with MANUAL + GRAPE, results show that in a system where subscriptions are randomly placed in the network, the message rate cannot be reduced by simply relocating publishers. We believe that strategies that only reconfigure the broker overlay [7, 46, 57] also have the same effect under such a scenario. Clustering subscriptions and allocat-
ing them on a new broker overlay reduces the average broker message rate by up to 90% in the homogeneous scenario. However, without adaptively relocating the publisher, the maximum message rate is unchanged from \textsc{manual + grape}. It is only until after we manipulate all three variables, subscription placement, broker overlay connections, and publisher placement, that we are able to reduce the maximum message rate by up to 85% in the heterogeneous scenario. Not shown, the maximum message rate of \textsc{cram-IOU} in the homogeneous scenario is higher than the heterogeneous scenario because not all subscriptions for a publisher can be serviced at the same broker. Additionally, publisher relocation slightly reduces the input utilization ratio of \textsc{cram-IOU}, but has no impact on the output utilization ratio.

From the client’s perspective, relocating publishers reduces the hop count on average by 40% in both scenarios [26]. In the heterogeneous scenario, publisher relocation actually reduces the hop count of \textsc{cram-IOU} to zero. According to Figure 7.20b, relocating publishers reduces the delivery delay on average by 16% in the homogeneous case and 24% in the heterogeneous case. Therefore, one can view publisher relocation as a remedy to reduce the delivery delay penalty for using \textsc{cram} to maximize the efficiency and minimize the allocation of resources.

7.3.5 Evaluation of CRAM’s Closeness Metrics

Figures 7.21a to 7.21d show the broker message rate, delivery delay, publication hop count, and computation time, respectively. According to these graphs, the average and maximum message rates for \textsc{xor} are on average 55% and 93% higher than \textsc{IOU}, respectively. This is because the \textsc{xor} metric clusters subscriptions with empty relationships together. The fact that \textsc{xor} produces 74% (41%) fewer clusters than \textsc{IOU} in the homogeneous (heterogeneous) scenario supports this claim. Another notable difference is that \textsc{xor} yields 2,200% higher hop count and 18% higher delivery delay compared to the \textsc{IOU} metric. \textsc{xor} also requires 105% (75%) more computation time than all other closeness metrics in the homogeneous (heterogeneous) scenario. The primary reason for this is because of its inability to identify subscriptions with empty relation which the other closeness metrics have that enable them to prune the search for potential clustering candidates.

\textsc{intersect}, \textsc{ios}, and \textsc{IOU} metrics show almost no differences in their input utilization
Figure 7.21: Experiment results comparing closeness metrics

ratios, output utilization ratios, number of allocated brokers, computation times, and number of subscription clusters. Initially, we expected the performance of the INTERSECT closeness metric to be inferior to IOS and IOU because of its inability to capture the extra traffic introduced in GIF pairs having intersecting relationships. However, the use of poset data structure to search for the next closest GIF pair eliminated this deficiency and enabled it to perform equally well with IOS and IOU.

7.3.6 Evaluation of the Sorting Algorithms

In most cases, the average input (Figure 7.22a) and output utilization ratios of BIN PACKING exceed FBF, which indicates that BIN PACKING is more effective in allocating more subscriptions into the brokers. We believe this is because BIN PACKING allocates subscriptions in the order of highest-to-lowest bandwidth requirement. As well, the maximum input utilization is higher in BIN PACKING because the sorting of subscriptions in descending traffic allocates more, if not all, high traffic subscriptions matching each publisher into the same broker. There are no
significant differences in terms of the broker message rates among the two sorting algorithms.

### 7.3.7 Evaluation of the PAIRWISE and AUTOMATIC Algorithms

Figure 7.22b shows that there exists overloaded brokers in AUTOMATIC and PAIRWISE-K under all workloads tested, and PAIRWISE-N when the number of subscriptions exceed 1,025 in the heterogeneous scenario. The main reason for the overload is allocating too many subscriptions to a broker and/or broker neighbors which exceed the broker’s output bandwidth capacity. The take-away message here is the necessity of resource awareness in the subscription assignment and broker overlay construction algorithms.

### 7.3.8 Impact of Bit Count in the Bit Vector

Use of the bit vector scheme to predict traffic based on past data is very powerful in that it can adapt to workloads of any distribution. At the same time, because it is impossible to get 100% accurate estimation based on past data, we discover that a scaling factor is required to accommodate for potential additional traffic to prevent overloading brokers. With a bit vector length of 1,024 bits, we have to fix the minimum bits set to 20% and increase the bit count for subscriptions with less than 40% bits set by 25%. For bit vector lengths of 512 or less, we have to fix the minimum bits set to 35% and scale subscriptions with less than 25% and 40% traffic by 30% and 45%, respectively.

Figures 7.23a to 7.23c show the graphs for computation time, broker message rate, and allocated brokers over bit vector lengths ranging from 128 up to 1,024, respectively. The results
Figure 7.23: Experiment results showing the impact of bit vector length

show that the computation time varies linearly with the bit vector length. However, because the computation time for CRAM is much longer, we can see the linear effect more prominently with CRAM-IOU than BIN PACKING. For CRAM-IOU, increasing the bit vector length also increases the subscription load estimation accuracy, which in turn decreases the maximum broker message rates, increases the average input utilization ratio, and decreases the number of brokers allocated. This translates to better distribution of load and more efficient use of broker resources. As for BIN PACKING, the only notable difference by increasing the bit vector is fewer allocated brokers. Each reduction in an allocated broker increases the delivery delay, which is inline with our earlier observation in Figure 7.19d.
Chapter 8

Conclusions

In this chapter, we conclude by summarizing our contributions and evaluation results, as well as identifying future research directions.

8.1 Summary

In Chapter 4, we presented a load balancing solution consisting of a load balancing framework, load estimation methodologies, and three offload algorithms. The load balancing framework consists of the PEER architecture, a distributed load exchange protocol called PIE, and detection and mediation mechanisms at the local and global load balancing levels. The core of the load estimation is PRESS, which uses an efficient bit vector approach to estimate the input and output publication loads of a subscription. Each of the three offload algorithms are designed to load balance on a particular performance metric with minimal side-effects and proven stability. Both the load estimation and offload algorithms are independent of the load balancing framework. Our solution inherits all of the most desirable properties that make a load balancing algorithm flexible. PIE contributes to the distributed and dynamic nature of our load balancing solution by allowing each broker to invoke load balancing whenever necessary. Adaptiveness is provided by the three offload algorithms that load balance on a unique performance metric. The local mediator gives transparency to the subscribers throughout the offload process. Finally, load estimation with PRESS allows the offload algorithms to account for broker and
subscription heterogeneity.

In Chapter 5, we proposed two publisher placement strategies that improve system scalability, robustness, and performance. We have demonstrated this with two unique algorithms, POP and GRAPE, that relocate publishers in the network to minimize average delivery delay and system load. POP is a simple yet effective algorithm that uses only one optimization metric in its calculation, which is the number of average matching downstream subscribers. GRAPE is a more flexible algorithm that allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously while taking a completely different computational approach than POP. For example, all of POP’s computations are distributed while GRAPE’s core computations are done in a robust yet centralized manner. Nevertheless, both algorithms adopt a 3-Phase architecture where in Phase 1 the algorithms efficiently trace and store publication delivery information, in Phase 2 the algorithms precisely select the target broker to which the publisher should relocate, and in Phase 3 the algorithms orchestrate the publisher migration in a transparent fashion.

In Chapter 6, we described three resource allocation algorithms that find a subscription assignment, broker overlay topology, and publisher placement to use as few broker resources as possible given an arbitrary workload and a set of broker resources. The motivation behind our goal is inline with the current green IT initiatives in using resources as efficiently as possible, which not only decreases IT operational costs but also increases the capacity of the system by allowing it to handle more load. Our key contributions in this work include developing a bit vector supported resource allocation framework, designing and comparing four different classes with a total of ten variations of subscription allocation algorithms, developing a recursive overlay construction algorithm, and comparing our work against two baseline and three related approaches. A compelling feature of our approach is that it works under any arbitrary workload distribution and is independent of the publish/subscribe language, which makes it easily applicable to any topic and content-based publish/subscribe system.

In Chapter 7, we showed evaluation results of our three key contributions on real world testbeds including PlanetLab and a cluster testbed. All of our contributions are implemented on the open-source publish/subscribe system called PADRES.
In regards to our first contribution, results show that our load balancing algorithm prevents overload by distributing subscribers while simultaneously balancing three performance metrics among edge brokers. Our algorithm enables the publish/subscribe system to scale with the amount of resources available to it and is effective in both homogeneous and heterogeneous environments. By adaptively subscribing to load information, message overhead of the load balancing infrastructure is reduced by an additional 65% to 0.2% in our experiments on the cluster testbed. Our results also show that a naive load balancing solution that cannot identify subscription space and load is not only inefficient but can also lead to system instability.

In regards to our second contribution, our experimental results confirm that both POP and GRAPE are effective under enterprise and random workloads on both PlanetLab and a cluster testbed. GRAPE’s load 100% setting reduced the average input load of the system by up to 68% and average broker message rate by up to 84%, while GRAPE’s delay 100% setting reduced the average delivery delay by up to 68%. GRAPE was able to minimize both average delivery delay and system load simultaneously according to the specified priority and weight. POP consistently reduced both average delivery delay and system load on PlanetLab, but the reductions fell in between the two extreme settings of GRAPE, load 100% and delay 100%, except for the random workload where POP produced the lowest average delivery delay. POP’s broker selection time is lower and is dependent on the length of the migration path. GRAPE’s broker selection time is higher and is dependent on the number of brokers with matching subscribers. The amount of message overhead from both approaches depended upon the number of publications traced per trace session, which in turn controlled the response time of both approaches. Taking all results and the unique features of POP and GRAPE into account, we recommend POP for publish/subscribe systems that strive for simplicity (such as GooPS [72]) and expect unpredictable subscription and traffic patterns. On the other hand, we recommend GRAPE for systems that strive to achieve minimal delivery delay (such as Tibco’s Supermontage [86]), load usage (such as sensor networks), or require the flexibility to minimize delivery delay when there are available resources (i.e., during off-peak Internet hours) and minimize network traffic when servers are about to overload (i.e., during on-peak Internet hours).

In regards to our third contribution, experiment results on a cluster and high performance
computing testbed show that CRAM excels over both baseline and related approaches in many ways. One, CRAM reduces the average broker message rate by up to 92% whereas prior work on only relocating publishers [25] achieved no reduction whatsoever. Two, CRAM overcomes limitations in earlier work on subscription clustering [73] by not only overcoming broker overload issues and preventing excessive delivery delays but also reducing the number of brokers allocated by up to 91% and bringing the publication hop count down to zero. Three, our proposed clustering closeness metric, IOU, is capable of reducing the computation time of the CRAM algorithm by up to 105% over the Gryphon-derived closeness metric, XOR [9]. At the same time, we found tradeoffs in the CRAM algorithm including slightly longer computation time compared to prior subscription clustering approaches and higher delivery delays compared to baseline approaches. However, we also discover remedies to these tradeoffs through adjustments to the bit vector length to decrease computation time and strategically relocate publishers to reduce the delivery delay penalty.

8.2 Future Work

In regards to the load balancing work, the runtime complexity of the offload algorithms can be further optimized to run faster to improve the response time of the overall solution. For example, instead of computing the report cards for all candidate subscriptions on every iteration, only compute the report cards for the next most promising subscription. This should effectively reduce the computational complexity from the current $O(n^2 \log n)$ to $O(n \log n)$. Parameters used by the load balancing algorithm can be adjusted in real-time by a machine learning algorithm that self-adapts the algorithm according to the network conditions. This will alleviate any need for human intervention in the system, which is the goal of this work in the first place. The challenge here is that the self-tuning algorithm will also need to be aware of settings that may conflict each other. For example, the lower overload threshold should always be set lower than the higher overload threshold. Responsiveness of the global load balancing algorithm can be further improved by, for example, load balancing directly with clusters more than one hop away. In our global load experiments with a chain of clusters, we witnessed that the first cluster,
B1x, had to wait while the second and third clusters, B2x and B3x respectively, were engaged in global load balancing. This may be detrimental to cluster B1x if it gets overloads while waiting and it has no other cluster to load balancing with.

Estimation accuracy can be further improved by the use of stochastic differential equations that can better predict future publication traffic of subscriptions based on past data. This concept also applies to the publisher placement algorithms and the resource allocation algorithms that also use bit vectors to estimate future network traffic patterns. Improving the load estimation accuracy has a significant impact in each of the approaches. In the load balancing solution, improving the load estimation accuracy reduces the number of offload sessions required, which lead to reduced overhead and faster convergence time. In the publisher placement approaches, better estimation accuracy will lead to less publisher migrations, which reduces the publisher downtime frequency. In the resource allocation algorithms that are aware of broker resource capacities, such as FBF, BIN PACKING, and CRAM, better load estimation accuracy will eliminate the need for the compensation factors used in this thesis. This improvement will enable the algorithm to not only allocate less brokers because resources are utilized to a higher extent, but also extend the time in which the final deployment stays optimal (i.e. no over- and under-utilizations at the brokers).

Future work for the resource allocation algorithms falls into two categories. One is to further enhance the current static approach either by parallelizing or reducing the time complexity of the CRAM algorithm. The cause for the current inefficiency is that the BIN PACKING algorithm is invoked after clustering every pair of GIF to test for allocatability. This design also makes it hard to do any parallelization because the algorithm always has to cluster and allocate the next closest subscriptions. To address this challenge, one may want to first gain insights from the theoretical domain on any parallelization techniques currently used for the bin packing algorithm. The benefits of parallelization will be tremendous even today as each SciNet server has two quad-core Intel CPUs capable of running up to eight threads simultaneously. Another enhancement is to replace the current greedy algorithm, which allocates the next most resourceful broker first, by a smarter approach that can backtrack and reserve more powerful brokers higher up in the tree in cases where allocation fails. This research will be very interesting
because the complexity of the algorithm will increase due to adoption of dynamic programming concepts. Yet, the benefit is that the allocation failure rate will decrease. Moreover, instead of invoking GRAPE as a separate process at the end, the publisher’s new location can also be computed within CRAM. This has two benefits. One, it avoids overloading the root broker as all publishers initially connect to that broker in the new overlay network. Two, it may potentially further reduce the number of internal brokers allocated in the tree because publication traffic may no longer have to originate from the root of the tree and propagated downwards if all matching subscribers are located at one single broker at the bottom of the tree.

This thesis have focused on developing a static approach to resource allocation because it is simpler to try out as a proof of concept. With the significant gains that have been demonstrated in our experiments, the next logical step is to develop a dynamic approach that swiftly allocate and deallocate brokers according to changing workload conditions. A dynamic approach has the benefit of incrementally adapting the system without incurring significant service interruptions present in the static approach. There is already work on dynamic network expansion by Young et al. [93]. However, dynamic network shrinkage has not been explored and is a new research topic in resource efficient content-based publish/subscribe systems. It is likely that dynamic solutions may only have local information at its disposal, which may yield only suboptimal results. A static approach, on the other hand, has global information and can make more optimal decisions. Therefore, the dynamic approach can work alongside with the static approach to complement each other’s strengths and weaknesses. For example, the static approach can be invoked to completely overhaul the system whenever there is a drastic change in the workload pattern. The dynamic approach can be invoked frequently after a complete overhaul by the static approach to incrementally adapt the system near its optimal setting for small changes in workload patterns.

An orthogonal future research direction is to extend the current work to be fault tolerant, which has been mildly addressed in some of the approaches presented in this thesis. This is a very challenging topic because the algorithms will have to handle broker and client failures at any phase of the algorithm. For example, the load balancing algorithm as is currently presented will put a broker at its BUSY state endlessly if the load-accepting broker crashes while accepting
new subscribers. Addressing fault tolerance will definitely make each algorithm more complex, but will also make them more attractive to real world applications where reliability is crucial.
Bibliography


